Forord

Dette er en hovedoppgave i databehandling til graden cand. scient. ved Institutt for informatikk, Universitetet i Oslo.

Det er mange som fortjener en takk i forbindelse med denne oppgaven. Først og fremst vil jeg takke min veileder Olaf Owe for mange gode forslag til oppgaven og en rekke interessante diskusjoner. Oppgaven bygger på hans arbeid, og uten dette fundamentet ville det ikke vært noen oppgave.

Oppgaven har blitt støttet økonomisk av NAVF (nå NFR), noe jeg selvsagt er takknemlig for.

Takk går også til Ole-Johan Dahl som på tross av stort arbeidspress tok seg tid til å lese gjennom oppgaven og komme med nyttige kommentarer.

Videre vil jeg takke Olav Asheim som har prøvd å lære meg logikk et uavgjørbart antall ganger. Han har også sett på en tidlig versjon av oppgaven og foreslått noen forbedringer.

Sigbjørn Næss fortjener en takk for sine forslag til språklige forbedringer av oppgaven og for sitt bidrag til det sosiale miljøet på Ifi.

Det siste er det også mange andre som burde takkes for. Takk til mine sosiale medstudenter. Dere vet hvem dere er.

Takk til min mor som har oppmuntret meg til å ta utdannelse, og en takk for at hun har passet Siri i innspurten med oppgaven.

Aller mest vil jeg takke Dag som har oppmuntret meg til å arbeide med oppgaven og som har gjort dette mulig ved å passe barn og hjem. Jeg vil også takke ham for at han har kommet med mange konstruktive kommentarer under arbeidet, og for at han har lest oppgaven og kommet med forslag til forbedringer.

Oslo, mai 1994

Kari Asheim

Innhold

Kapittel 1

Innledning

Når vi lager programmer er det viktig at vi kan forsikre oss om at de fungerer som ønsket. Å teste programmer vil kunne luke ut en del feil, men for at vi skal kunne være helt overbevist om at et program oppfyller en spesifikasjon, kan det være nyttig å bruke mer formelle resonnementer.

Spesielt gjelder dette for parallelle programmer. Da har vi flere prosesser som kan utføre sine instruksjoner samtidig. Vi kan som regel ikke vite hvor fort disse prosessene kjører i forhold til hverandre, og dermed kan vi ikke vite i hvilken rekkefølge operasjonene i programmet utføres. Dette kan også være forskjellig for to eksekveringer av et program og dermed har vi ikke-determinisme.

Dette gjør at antall tilstander for et parallelt program vil være av samme orden som produktet av antall tilstander i hver prosess. Det blir altså så mange muligheter at vi i praksis ikke klarer å teste alle. Dermed er det ikke tilstrekkelig å bare teste programmet, men også nødvendig å kunne resonnere formelt om det. Til dette formålet er det foreslått en rekke bevissystemer.

For at flest mulig korrekt spesifiserte programmer skal kunne bevises i et bevissystem, må systemet være relativt komplett[1]. Et bevissystem for parallelle programmer må ha med mytiske variable for at det skal være relativt komplett. I [Owe92] er det gitt et bevissystem der det brukes historiesekvenser som mytiske variable. I disse tar vi vare på informasjon om hva hver enkelt prosess har gjort med felles variable. Dette bevissystemet er relativt komplett.

Det er interessant å se hvor lite informasjon som kan tas vare på i mytiske variable uten at vi mister kompletthet. I [Owe92] foreslås det å fjerne en del informasjon om rekkefølge av operasjoner. For hver variabel skal det tas vare på en sekvens av tilordnete verdier og en sekvens av leste verdier. Det argumenteres for at det er mulig å lage et slikt bevissystem som er relativt komplett.

I denne oppgaven vil jeg gjøre som foreslått i [Owe92], men i tillegg vil jeg også fjerne sekvensene av leste verdier. Det eneste som tas vare på av historieinformasjon er sekvensene av verdier som er tilordnet til felles variable. Jeg lager et bevissystem med redusert mengde av mytiske variable og viser at dette er relativt komplett.

[1] Uttrykket *relativt komplett* og andre begreper defineres i neste kapittel.

Det gis også et par eksempler på hvordan vi kan lage beviser med så lite historieinformasjon som beskrevet ovenfor. Spesielt vil jeg se på et produsent-konsument-eksempel som illustrerer hvordan vi må gå frem for å bevise synkronisering av prosesser.

Videre vil jeg ta for meg hvordan vranglås/fravær av vranglås kan bevises i et av bevissystemene fra [Owe92].

Leseren av denne oppgaven bør ha god kjennskap til hvordan programmer verifiseres. Dette grunnlaget kan komme fra kurset i programspesifikasjon- og verifikasjon som undervises ved instituttet [Dah92]. Videre er det en fordel å kjenne til verifikasjon av parallelle programmer, noe som også undervises ved instituttet [Dah93]. Det kan også være nyttig med noe mer kunnskaper om logikk, slik at begreper som sunnhet og kompletthet er godt kjent.

Oppgaven er bygget opp slik:

I kapittel 2 gis bakgrunnen for oppgaven. Her finnes syntaks for programmeringsspråket vi ser på, Hoare-logikk for sekvensielle programmer og en beskrivelse av bevissystemene fra [Owe92]. Videre forklares notasjonen som brukes i disse bevissystemene og noen sentrale begreper som f.eks. relativ kompletthet. Jeg ser også litt på hva andre har gjort for å bevise parallelle programmer.

Kapittel 3 inneholder bevissystemet der kun verdier av variable og historiesekvenser av tilordnete verdier kan brukes i spesifikasjonen. Det blir gitt noen eksempler på hvordan systemet brukes. Videre vises det at bevissystemet er komplett, og det argumenteres for at det (opplagt) må være sunt.

I kapittel 4 forklares det hvordan vi i et bevissystem fra [Owe92] kan vise vranglås/fravær av vranglås. Det bevises at ulike programmeringsstrategier for "Dining Philosophers" er fri fra vranglås/kan gå i vranglås.

Kapittel 5 inneholder en kort oppsummering.

Kapittel 2

Bakgrunn

I dette kapitlet beskrives først det programmeringsspråket jeg behandler. Videre gis en kort oppsummering av hvilke regler jeg bruker fra Hoare-logikk for sekvensielle programmer. Dernest følger en kort diskusjon om relativ kompletthet, som gjelder for Hoare-logikk (både for sekvensielle og parallelle programmer). Jeg tar også for meg hvordan historiesekvenser brukes som mytiske variable i bevissystemene. Bevissystemene i [Owe92] som denne oppgaven bygger på presenteres, og til slutt ser jeg på en del andre arbeider som behandler parallelle programmer med felles variable.

2.1 Programmeringsspråket

Det programmeringsspråket jeg ser på er et enkelt ALGOL-aktig programmeringsspråk utvidet med parallell sammensetning, **cobegin** $P_1 \parallel \cdots \parallel P_n$ **coend**, der $P_1, \ldots, P_n$ er prosesser som utføres i parallell. Disse prosessene kan referere til felles variable, men bare slik at hver tilordningssetning i en prosess kan ses på som en udelelig hendelse. Vi kan ikke anta noe om den relative hastigheten til de ulike prosessene, så programmereren må selv sørge for eventuell synkronisering ved hjelp av **await**-konstruksjonen, som beskrives nedenfor. BNF-syntaksen for språket ser slik ut:

⟨stmt⟩ ::= ⟨assign⟩ | ⟨skip⟩ | ⟨if⟩ | ⟨while⟩ | ⟨comp⟩ | ⟨cobegin⟩
⟨assign⟩ ::= ⟨*var*⟩:=⟨*exp*⟩
⟨skip⟩ ::= **skip**
⟨if⟩ ::= **if** ⟨*exp*⟩ **then** ⟨stmt⟩ **else** ⟨stmt⟩ **fi**
⟨while⟩ ::= **while** ⟨*exp*⟩ **do** ⟨stmt⟩ **od**
⟨comp⟩ ::= ⟨stmt⟩ ; ⟨stmt⟩
⟨cobegin⟩ ::= **cobegin** ⟨process⟩ ∥ ⋯ ∥ ⟨process⟩ **coend**
⟨process⟩ ::= ⟨pstmt⟩
⟨pstmt⟩ ::= ⟨passign⟩ | ⟨skip⟩ | ⟨pif⟩ | ⟨pwhile⟩ | ⟨pcomp⟩ | ⟨await⟩
⟨passign⟩ ::= ⟨*local var*⟩:=⟨*shared exp*⟩ | ⟨*var*⟩:=⟨*local exp*⟩
⟨pif⟩ ::= **if** ⟨*local exp*⟩ **then** ⟨pstmt⟩ **else** ⟨pstmt⟩ **fi**
⟨pwhile⟩ ::= **while** ⟨*local exp*⟩ **do** ⟨pstmt⟩ **od**

⟨pcomp⟩ ::= ⟨pstmt⟩ ; ⟨pstmt⟩
⟨await⟩ ::= **await** ⟨*exp*⟩ **do** ⟨cstmt⟩ **od**
⟨cstmt⟩ ::= ⟨assign⟩ | ⟨skip⟩ | ⟨cif⟩ | ⟨cwhile⟩ | ⟨ccomp⟩
⟨cif⟩ ::= **if** ⟨*exp*⟩ **then** ⟨cstmt⟩ **else** ⟨cstmt⟩ **fi**
⟨cwhile⟩ ::= **while** ⟨*exp*⟩ **do** ⟨cstmt⟩ **od**
⟨ccomp⟩ ::= ⟨cstmt⟩ ; ⟨cstmt⟩

Som vanlig i sekvensielle programmer brukes ⟨*var*⟩ for en variabel og ⟨*exp*⟩ for et uttrykk med konstanter, variable og operasjoner på disse. Datatypene som brukes er vanlige, statiske typer som heltall, boolske og arrayer av disse. Deklarasjoner gjøres ikke eksplisitt i programmeringsspråket, men det er enkelt å ta dette med om ønskelig. Variable som refereres til utenfor **cobegin**...**coend** eller i flere prosesser er felles, mens variable som bare refereres til i én prosess er lokale.

Vi antar at alle uttrykk er *veldefinerte*. Et eksempel på et uttrykk som ikke er veldefinert, er a/b når $b = 0$. Videre antar vi at at alle funksjoner er *totale*, dvs. funksjoner som gir veldefinerte verdier for alle veldefinerte argumenter av riktig type.

Når det bare står ⟨*var*⟩ eller ⟨*exp*⟩ er det ingen begrensning på om variable som brukes er lokale eller felles. Inne i prosessene ønsker vi å betrakte hver programsetning som en udelelig operasjon. Dermed kan det bare tilordnes til én felles variabel, eller leses én felles variabel. Uttrykket ⟨*shared exp*⟩ i BNF-syntaksen inneholder én referanse til felles variable. Resten av tilordningssetningen og tester i **if**/**while**-setninger skal ikke inneholde noen referanse til felles variable, og dette uttrykkes som ⟨*local var*⟩/⟨*local exp*⟩. Vi kan tillate flere referanser til felles variable ved å legge på en implisitt kritisk region.

await ⟨*exp*⟩ **do** ⟨cstmt⟩ **od** er kritisk region der ⟨*exp*⟩ er ventebetingelse og ⟨cstmt⟩ er "innmat". En kritisk region implementeres som en udelelig operasjon, så her kan vi referere til så mange felles variable vi ønsker på en gang. Det er vanskelig å lage en effektiv implementasjon av denne formen for kritisk region, men reglene for denne konstruksjonen kan lett tilpasses til den måten kritisk region er implementert på i praksis. Nesting av kritiske regioner er ikke tillatt, men dette er heller ikke noe poeng, siden vi allerede har eksklusiv aksess til samtlige felles variable.

Nestede **cobegin**...**coend**-setninger er heller ikke tillatt, fordi det kompliserer bevissystemet. Det burde imidlertid være mulig å utvide systemet slik at dette kan tillates.

2.2 Hoare-logikk

De bevissystemene jeg ser på i denne oppgaven bygger på Hoare-logikk slik Hoare definerte den i [Hoa69]. Jeg bruker samme notasjon som i [Dah92], og denne avviker litt fra den Hoare opprinnelig brukte. Dette er ikke noen innføring i Hoare-logikk, bare en rask oppsummering av hvilke regler jeg skal bruke i denne oppgaven. Se [Dah92] for en grundig innføring i emnet.

Følgende notasjon brukes:

- P_e^x representerer utsagnet vi får ved å erstatte alle frie forekomster av variabelen x i P med uttrykket e.

- $\mathcal{V}[P]$ er mengden av frie variable i P hvis P er et utsagn eller mengden av variable det refereres til i P hvis P er en programsetning.
- $\mathcal{W}[S]$ er mengden av variable som tilordnes i programsetningen S.

I Hoare-logikk ser vi på utsagn på formen

$$\{P\}S\{Q\}$$

der S er en program-setning og P og Q er utsagn i første ordens logikk. Dette utsagnet kan vi tolke som

> Hvis forbetingelsen P holder før en eksekvering av programbiten S, og eksekveringen terminerer normalt, så vil bakbetingelsen Q holde etter at eksekveringen har terminert.

La x være vektoren av variable i programmet, og la σ_1 og σ_2 være tilstander som gir verdier til variablene. Et program S kan tolkes som en relasjon over to tilstander i et på forhånd definert tilstandsrom. La $\sigma_1 S \sigma_2$ bety at en eksekvering av S som starter i tilstand σ_1 kan terminere normalt i tilstand σ_2. Da kan meningen av et program uttrykkes mer formelt som

$$\{P\}S\{Q\} \triangleq \forall\sigma_1,\sigma_2 | P^x_{\sigma_1} \wedge \sigma_1 S \sigma_2 \Rightarrow Q^x_{\sigma_2}$$

I denne fremstillingen er det tatt med flere regler enn Hoare hadde i sin opprinnelige artikkel. Følgende aksiomskjemaer er tatt fra [Dah92]:

AS: $$\{P^x_e\}x := e\{P\}$$

CONS: $$\{P\}S\{P\} \qquad \text{for } \mathcal{V}[P] \cap \mathcal{W}[S] = \emptyset$$

SKIP: $$\{P\}\mathbf{skip}\{P\}$$

Fra samme sted har jeg tatt med disse reglene:

SEQ: $$\frac{\{P\}S_1\{Q\} \quad , \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

TDIF: $$\frac{\{P \wedge e\}S_1\{Q\} \quad , \quad \{P \wedge \neg e\}S_2\{Q\}}{\{P\}\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\{Q\}}$$

WDO: $$\frac{\{I \wedge e\}S\{I\}}{\{I\}\mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{od}\{I \wedge \neg e\}}$$

CQL (Left Consequence): $$\frac{P \Rightarrow P' \quad , \quad \{P'\}S\{Q\}}{\{P\}S\{Q\}}$$

CQR (Right Consequence): $$\frac{\{P\}S\{Q'\} \quad , \quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

CJ (Conjunction): $$\frac{\{P\}S\{Q_1\} \quad , \quad \{P\}S\{Q_2\}}{\{P\}S\{Q_1 \wedge Q_2\}}$$

DJ (Disjunction): $$\frac{\{P_1\}S\{Q\} \quad , \quad \{P_2\}S\{Q\}}{\{P_1 \vee P_2\}S\{Q\}}$$

EXST: $$\frac{\{P\}S\{Q\}}{\{\exists z|P\}S\{Q\}} \qquad \text{for } \{z\} \cap (\mathcal{V}[S] \cup \mathcal{V}[Q]) = \emptyset$$

UNIV: $$\frac{\{P\}S\{Q\}}{\{P\}S\{\forall z|Q\}} \qquad \text{for } \{z\} \cap (\mathcal{V}[P] \cup \mathcal{V}[S]) = \emptyset$$

Disse reglene brukes utenfor **cobegin**...**coend** og når betingelsene ikke refererer til felles variable. Reglene brukes også i de delene av en prosess som ikke refererer til felles variable, og der denne prosessen har eksklusiv aksess til felles variable. Siden tester i **if/while**-setninger ikke kan referere til felles variable, kan de tilhørende reglene brukes direkte i bevissystemene.

Andre regler enn de som er gitt her, f.eks. for andre konstruksjoner i sekvensielle programmer kan selvfølgelig også brukes. Det er da viktig å passe på at kravet om atomiske operasjoner overholdes.

Reglene som er gitt her er bare for konstruksjoner i sekvensielle programmer, så i tillegg må det gis regler for de programkonstruksjonene som finnes for parallelle programmer. Jeg vil senere i denne oppgaven se på flere slike regelsett.

2.3 Egenskaper ved bevissystemer

I dette avsnittet vil vi se på følgende begreper: partiell korrekthet, total korrekthet, vranglås, sunnhet, kompletthet, relativ kompletthet og hierarkisk oppbygging av bevis.

2.3.1 Partiell korrekthet, total korrekthet og vranglås

Å bevise en spesifikasjon av et sekvensielt program som definert i avsnitt 2.2, kalles å bevise *partiell korrekthet*. Dersom vi også viser at et program terminerer, kalles det *total korrekthet*.

Det kan være flere årsaker til at et parallelt program ikke terminerer. I tillegg til de samme problemene som vi vil ha for sekvensielle programmer kan et parallelt program gå i *vranglås*. Vi bruker følgende definisjon av *total* vranglås:

> Alle prosesser som ikke har terminert er i venting.

I tillegg kan vi ha *partiell* vranglås, der bare en del av prosessene er blokkert. I dette tilfellet vil hver blokkerte prosess vente på en annen prosess som også er blokkert.

Begrepene partiell og total korrekthet kan generaliseres til også å gjelde for parallelle programmer. Ofte brukes begrepene *safety* og *liveness*. Safety-egenskaper brukes for å uttrykke at ikke noe galt kan skje, mens liveness-egenskaper brukes for å uttrykke at noe (riktig) vil skje.

I følge denne definisjonen er fravær av vranglås en safety-egenskap, og denne egenskapen ved et program bør kunne bevises i et bevissystem for partiell korrekthet.

2.3.2 Sunnhet

For å verifisere programmer er det ikke tilstrekkelig å bruke bare Hoare-logikk. Vi trenger i tillegg et deduktivt system, og da er det vanlig å bruke første ordens logikk slik vi gjør i konsekvensreglene, LCQ og RCQ. Videre er det nødvendig med et aksiomsystem for hver datatype vi bruker. For enkelhets skyld er det vanlig å anta at vi bare har med datatypen naturlige tall og at aksiomsystemet er Peano-aritmetikk. Disse resonnementene kan utvides til også å gjelde for andre datatyper.

Litt upresist kan vi si at en *tolkning* består av et *domene*, dvs. en mengde av termer, og en tilordning av verdier til funksjoner/relasjoner over disse. Følgende notasjon benyttes for sann, gyldig og bevisbar:

- $\models_T P$ — utsagnet P er sant i tolkningen T.
- $\models P$ — utsagnet P er gyldig, dvs. sant i alle tolkninger.
- $\vdash_B P$ — utsagnet P kan vises i bevissystemet B.

Sunnhet vil si at alt vi kan vise med bevissystemet er gyldig. Formelt kan sunnhet av et bevissystem for spesifikasjon av programmer uttrykkes som

> *Hvis* $A \vdash_{(H,D)} \{P\}S\{Q\}$, *så* $\models_T \{P\}S\{Q\}$

hvor (H, D) er bevissystemet (Hoare-logikk og første ordens logikk), T står for en tolkning og A er mengden av utsagn som er sanne i T. Siden vi ser på naturlige tall vil A inneholde alle sanne utsagn fra Peano-aritmetikk. Sunnhet bør være et minimumskrav til et bevissystem, selv om det er lett å lage gale regler. I litteraturen er det flere eksempler på slike feil, spesielt gjelder dette reglene for prosedyrekall [Coo78].

2.3.3 Kompletthet

Kompletthet er det motsatte av sunnhet, dvs. at alt som er gyldig, kan bevises. Dette kan formelt uttrykkes slik

$$Hvis \models_T \{P\}S\{Q\},\ så\ A \vdash_{(H,D)} \{P\}S\{Q\}$$

Dessverre er det umulig å finne et bevissystem for verifikasjon av programmer som er komplett, også selv om det deduktive systemet i bunnen er komplett.

Vi lar det deduktive systemet bestå av første ordens logikk og Presburger-aritmetikk, og dette systemet er komplett. Hvis vi ser på dette systemet i sammenheng med stoppeproblemet, vil vi finne ut hvorfor et komplett bevissystem for verifikasjon av programmer ikke kan finnes.

Presburger-aritmetikk er det samme som Peano-aritmetikk uten multiplikasjon. Mengden av utsagn som kan vises fra aksiomene i Presburger-aritmetikk er rekursivt enumererbar, dvs. det er mulig å lage en algoritme som lister opp mengden, og alle elementene vil før eller senere komme med[1].

På den annen side har vi stoppeproblemet som er uavgjørbart. Dette innebærer at mengden av programmer på formen $\{P\}S\{\mathbf{false}\}$ ikke er rekursivt enumererbar. Hvis denne mengden var rekursivt enumererbar, ville vi ha en løsning på stoppeproblemet. For å finne ut om et gitt program ikke stopper, kunne vi bare liste opp alle programmer som ikke stopper, og før eller senere ville vi komme til det spesielle programmet.

Siden denne mengden ikke er rekursivt enumererbar vil det ikke være mulig å lage et bevissystem basert på første ordens logikk og Presburger-aritmetikk, som kan bevise alle programmer på formen $\{P\}S\{\mathbf{false}\}$. Vi kan altså ikke finne et komplett bevissystem basert på dette deduktive systemet.

2.3.4 Relativ kompletthet

Heldigvis kan vi likevel si noe om kompletthet. Cook [Coo78] tar utgangspunkt i en interpretiv modell og definerer kompletthet relativt til denne. Dette kaller han *relativ kompletthet*, men denne formen for kompletthet kalles også Cook-kompletthet i litteraturen.

En betingelse for relativ kompletthet hos Cook er at vi kan uttrykke sterke nok betingelser. Dersom det for en gitt tolkning og en gitt klasse programmer for enhver forbetingelse, P, og enhver programbit, S, er mulig å formulere bakbetingelsen, Q, i språket vårt, sier vi at språket er *uttrykksfullt* (expressive) relativt til tolkning og klassen av programmer.

[1] Mengden av naturlige tall er rekursivt enumererbar.

Anta at vi har en tolkning, en klasse av programmer og et språk som er uttrykksfullt relativt til tolkningen og klassen av programmer. Et bevissystem er relativt komplett hvis vi for hver slik tolkning kan bevise mengden av programmer som er sanne i tolkningen, med utsagn som kan bevises fra teorien som antagelse.

De tolkninger som gir uttrykksfullhet for det språket vi ser på vil for Peano-aritmetikk være standardmodellen[2] eller en tolkning der domenet er en endelig delmengde av de naturlige tallene.

Relativ kompletthet kan dermed uttrykkes som

$$Hvis \models_{T_0} \{P\}S\{Q\},\ så\ A \vdash_{(H,D)} \{P\}S\{Q\}$$

der T_0 er standardmodellen for Peano-aritmetikk, eventuelt en annen tolkning som gir uttrykksfullhet.

Denne fremstillingen er svært kort. I [Apt81] er det gitt en mer detaljert fremstilling, og der bevises sunnhet/kompletthet for flere programkonstruksjoner for sekvensielle programmer.

2.3.5 Hierarkisk oppbygging av bevis

Et program i det språket vi ser på vil være bygget opp av delprogrammer. Det er ønskelig å kunne benytte den strukturen dette gir i beviset av programmet. Dette ønsket kan formuleres som prinsippet om *hierarkisk oppbygging* av bevis (principle of compositional program verification), se f.eks. [Zwi89]:

> "Spesifikasjonen av et program bør kunne bevises utfra spesifikasjonene av de syntaktiske delene av programmet, uten at det skal være nødvendig å kjenne innmaten i disse delene."

Dette innebærer at vi kan bevise en isolert del av programmet av gangen, og det blir enklere å lage et bevis som er hierarkisk oppbygget enn et som ikke er det. Spesielt tydelig blir dette hvis vi ser på parallelle programmer der kompleksiteten av bevis kan bli svært stor.

I beviset av en prosess kan vi bruke betingelser om globale tilstander, men vi kan ikke bruke lokale betingelser fra en annen prosess, for at kravet om hierarkisk oppbygging skal være overholdt.

All spesifikasjon kan være i en global invariant, som inneholder *all* informasjon om alle prosesser. Dette vil ikke gi noen reduksjon i kompleksiteten i spesifikasjonen av delene i forhold til spesifikasjonen av hele programmet, så kravet ovenfor kan med fordel forsterkes.

Ideelt sett ønsker vi oss at kompleksiteten av beviset av hele programmet skal være av samme orden som summen av kompleksitetene av bevisene for hver prosess, ikke av samme orden som produktet. Det kan virke som dette kravet er noe sterkt, men vi bør kunne ha noe reduksjon i kompleksitet av delbevisene for en prosess i forhold til beviset av hele programmet.

[2] Standardmodellen for de naturlige tallene skiller seg fra en ikke-standardmodell ved måten ∞, uendelig, behandles på.

2.3.6 Mytiske variable

Mytiske variable er variable som tas med i et program bare for å lette gjennomføringen av et bevis. Slike variable vil ikke bidra noe til resultatet av programutførelsen, og som et siste skritt i beviset av et program kan de derfor fjernes. Mytiske variable kan ikke brukes på høyre side i en tilordning til programvariable, siden dette vil ha effekt for resultat av programmet. I en tilordning til mytiske variable trenger vi imidlertid ikke å ta slike hensyn.

For å få relativ kompletthet av et bevissystem for parallelle programmer må vi kunne uttrykke tilstrekkelig sterke betingelser. Inne i parallelle prosesser er det nødvendig med mytiske variable for å få til dette. En av de første som brukte dette i et bevis var Brinch Hansen [Bri73]. I [Owi75] vises det at det er nødvendig å ta med mytiske variable for å få kompletthet av et bevissystem.

Det er flere muligheter for hva slags mytiske variable vi kan bruke. F.eks. er det i [Owi75] opp til den som verifiserer et program å velge riktige mytiske variable selv, og det er gitt en generell regel for håndtering av mytiske variable. Dersom disse mytiske variablene skal endre verdi i løpet av programmet, må brukeren selv skrive mytiske programsetninger for å oppdatere variablene.

Ofte er det imidlertid vanlig at de mytiske variablene er en del av det formelle systemet, og at det ikke er nødvendig med mere mytisk informasjon enn det som er innbakt i systemet. Dette er tilfelle i de bevissystemene jeg ser på i denne oppgaven. Her brukes historiesekvenser som tar vare på informasjon om hva som er skjedd med de felles variablene.

2.4 Historiesekvenser brukt i System 1, 2 og 3

I dette avsnittet presenteres de mytiske variablene som brukes i bevissystemene i denne oppgaven. De er en del av bevissystemet, og en bruker trenger ikke tenke på å skrive kode for å oppdatere dem.

2.4.1 Sekvenser

Siden sekvenser er viet så mye oppmerksomhet i denne fremstillingen, vil jeg kort oppsummere de funksjonene for sekvenser som benyttes i denne oppgaven:

- ε — den tomme sekvensen
- $\langle x_1, \ldots, x_n \rangle$ — sekvensen med elementene x_1,...,x_n
- $q \vdash x$ — legg til et element til høyre i en sekvens
- $x \dashv q$ — legg til et element til venstre i en sekvens
- $q \vdash\!\!\!\vdash r$ — konkatener to sekvenser
- $\#q$ — lengden av en sekvens

- $rt(q)$ — right term, høyre element av en sekvens
- $q[i]$ — indeksering, element nr. i fra en sekvens
- $q[i{:}j]$ — delsekvens, f.o.m. element nr. i t.o.m. element nr. j[3]
- q/m — projeksjon på en mengde m. Resultatet av denne er en sekvens med kun de elementene som også finnes i m

En fullstendig definisjon av disse funksjonene finnes i [Dah92], og der finnes også definisjonen av følgende predikat:

- q **head** r — q er en headsekvens (prefiks) av r

og et predikat for fletting av to sekvenser, som jeg bruker denne generelle varianten av:

- q **ismerge** $q_1, \ldots, q_n$ — q er en *fletting* av sekvensene $q_1, \ldots, q_n$

En fletting av sekvenser vil si at i uttrykket over vil q inneholde alle elementene fra sekvensene $q_1, \ldots, q_n$. Rekkefølgen av elementene i sekvensen q_i, $(i = 1, \ldots, n)$, vil være den samme som i q når vi ser bort fra elementer de andre sekvensene er opphav til.

2.4.2 Lokale historiesekvenser

Vi skal se hvordan vi kan ta vare på all informasjon om manipulering med felles variable sett fra én prosess. En slik historiesekvens h_i kalles lokal fordi den hører sammen med prosess i og inneholder det vi kan vite lokalt i denne prosessen. Den kan bestå av følgende typer av elementer:

- $(!j, e)$ — tilordnet verdien e til variabelen r_j
- $(?j, e)$ — lest verdien e fra variabelen r_j
- $(??r)$ — inngang til kritisk region, lest vektoren av verdier r fra variablene
- $(!!r)$ — utgang av kritisk region, tilordnet vektoren av verdier r til variablene

$!j$ og $?j$ er navn som konstrueres utfra indeksen til variabel r_j i vektoren av felles variable. De brukes bare som konstanter i en historiesekvens for å angi hvilken variabel som er tilordnet/lest, og de vil ikke bli endret ved anvendelse av Hoare-regler.

En kritisk region kan vi tenke på som at alle variable leses ved starten av den. Deretter kan alle felles variable leses/tilordnes fritt uten at dette avleires i historiesekvensen, og til slutt har vi en tilordning til alle variablene.

La x og y være felles variable og v lokal variabel i prosess P_1 i følgende eksempel:

[3] I [Dah92] er denne definert slik at $i{:}j$ kan erstattes av en vilkårlig sekvens av naturlige tall.

$x := 0;$
$y := 0;$
cobegin
P_1 :: $x := 3;$
$v := y$
||
P_2 :: **await** $x = 3$ **do**
$x := y + 1$
od
coend

Det vil alltid være en definert ordning av variablene. I sekvensene i dette eksemplet lar vi x ha indeks 1 og y indeks 2 i vektoren av felles variable.

Vi vet følgende om hvordan de lokale historiesekvensene h_1 og h_2 ser ut på slutten av hver prosess (men før **coend**) i en eksekvering av dette programmet:

- $h_1 = \langle(!1, 3), (?2, y)\rangle$ — Vi kan ikke vite hvilken verdi som ble lest i andre element (uten å se på resten av programmet). I praksis vil det by på problemer å bruke y inne i en historiesekvens. Hvis y får en ny verdi senere i prosessen, vil historiesekvensen være gal.
- $h_2 = \langle(??(3, y)), (!!(y + 1, y))\rangle$ — Her vet vi at verdien av y i andre element er den samme som den som ble lest i første element, og verdien av x i andre element er verdien av y fra første element pluss 1.

Som sagt kan vi ikke vite lokalt i en prosess hvilke verdier som blir lest fra de felles variablene i dette eksemplet, selv om vi lett ser hvilke verdier det dreier seg om ved å se på hele programmet. Informasjonen i lokale historiesekvenser skal bare inneholde det vi vet lokalt i en prosess. Etter **coend** kan vi imidlertid se på alle historiesekvensene og finne ut hvordan en fletting av dem må være. Da kan vi også finne de manglende verdiene.

2.4.3 Globale historiesekvenser

Istedenfor å se på informasjonen lokalt, kan vi se på alle prosessene og konstruere en global historiesekvens, som tar med all nødvendig informasjon fra alle prosessene. Følgende elementer kan finnes i en global historiesekvens h:

- $(i!j, e)$ — prosess i har tilordnet verdien e til variabelen r_j
- $(i?j)$ — prosess i har lest variabelen r_j. Verdien blir ikke tatt vare på
- $(i!!r)$ — utgang fra kritisk region, prosess i har tilordnet vektoren av verdier r til alle variablene

Merk at vi her tar med i, prosessidentifikasjon for prosess P_i, slik at vi får med informasjon om hvilken prosess som gjorde hva. Det er ingen verdi-del i ?-elementet, bare angivelse av

hvilken variabel som ble lest. Dette er fordi vi kan finne verdien ved å se tilbake til siste tilordning, eventuelt initialverdien, i en global historiesekvens. ??-elementet er også fjernet, fordi verdiene som ble lest kan finnes ved å se på resten av sekvensen. Slik kritisk region er definert vet vi at et ??-element skal etterfølges av et tilhørende !!-element.

For å finne verdien av en variabel i en historiesekvens kan vi bruke følgende funksjon:

$$final_j(q) = rt((!!r0 \dashv q).values_j)$$

der $values_j$ plukker ut verdidelen av de elementene som refererer til r_j, og $r0$ er vektoren av variablenes initialverdier. Hvis vi bruker $final_j$ på en global historiesekvens, vil vi få den verdien som sist er tilordnet til r_j av en av prosessene, altså den riktige verdien. Brukes den på en lokal sekvens får vi den siste verdien som er lest eller tilordnet av denne prosessen. Dette er ikke nødvendigvis den verdien variabelen har i øyeblikket.

All informasjonen som bare er implisitt i h, kan gjøres eksplisitt ved å bruke funksjonen $expl$, definert som følger:

$$\begin{aligned}
&expl(\varepsilon) = \varepsilon \\
&expl(q \vdash (i?j)) = expl(q) \vdash (i?j, final_j(q)) \\
&expl(q \vdash (i!j, a)) = expl(q) \vdash (i!j, a) \\
&expl(q \vdash (i!!..a_j..)) = expl(q) \vdash (i??..final_j(q)..) \vdash (i!!..a_j..)
\end{aligned}$$

For å konstruere en lokal historiesekvens h_i, fra en global kan vi bruke en funksjon

$local_i(q)$, som er $expl(q)/i$ med prosessidentifikasjonen i fjernet fra hvert element.

der $expl(q)/i$ betyr sekvensen av elementer fra $expl(q)$ som har prosessidentifikasjon i.

Eksemplet over vil gi følgende to muligheter for den globale historiesekvensen, slik at $h_1 = local_1(h)$ og $h_2 = local_2(h)$:

- $h = \langle (1!1,3), (1?2), (2!!(1,0)) \rangle$
- $h = \langle (1!1,3), (2!!(1,0)), (1?2) \rangle$

Merk at sekvensen må begynne med elementet $(1!1,3)$ på grunn av ventebetingelsen i **await**-setningen.

Fra disse mulige h'ene, initialverdiene til variablene og funksjonene definert over kan vi konstruere lokale historiesekvenser, og i disse vil vi vite hvilke verdier som kan være lest (og dermed tilordnet etterpå) i hver prosess.

All informasjon som er tilgjengelig kan samles i en global historiesekvens, som beskrevet over. Imidlertid er dette svært mye å holde rede på. Ofte har vi ikke så mye informasjon som i eksemplet over, og ofte kan vi ønske å ikke være nødt til å spesifisere hele historien.

Følgende kortnotasjon brukes for å projisere historiesekvensene h eller h_i på en passende mengde av elementer:

- $q/!$ — projeksjon på elementer som har med tilordning å gjøre, $!j$- og !!-elementer

- $q/?$ — projeksjon på $?j$- og $??$-elementer
- $q/!j$ — er en forkortelse for $(q/!).values_j$, dvs. verdidelen av de elementene som har som effekt tilordning til variabelen r_j
- $q/?j$ — forkortelse for $(q/?).values_j$, dvs. verdidelen av elementene for lesing av variabelen r_j
- q/i — projeksjon på elementer som tilsvarer en operasjon utført av prosess i

2.4.4 Oppsplittede historiesekvenser

Vi kan splitte historiesekvensene h_i og h opp i flere sekvenser som hver inneholder mindre informasjon, men som tilsammen er ekvivalent med de historiesekvensene som er introdusert så langt.

For hver variabel kan vi lage én sekvens av tilordnete verdier og én sekvens av leste verdier. For ikke å miste informasjon under oppsplittingen må vi også ha en sekvens som holder rede på rekkefølgen de ulike operasjonene er utført i. Globalt må denne også inneholde prosessidentifikasjon for at all informasjon skal være tilgjengelig.

For m felles variable får vi følgende $2 * m + 1$ sekvenser som en lokal historiesekvens, h_i, kan splittes opp i:

- $!r_j$ — $h_i/!j$, en sekvens for hver variabel r_j, som inneholder de verdiene som er tilordnet til denne variabelen
- $?r_j$ — $h_i/?j$, en sekvens for hver variabel r_j med verdier som er lest
- α — en sekvens for hele prosessen, som inneholder rekkefølgen de ulike operasjonene er utført i, dvs. elementer av typen $!j_1$, $?j_3$, $??$ og $!!$

Den globale historiesekvensen med ?-informasjonen eksplisitt, $expl(h)$, deles opp i tilsvarende $2 * m + 1$ sekvenser. For å bevare all informasjon må α inneholde prosessidentifikasjon også, dvs. elementer av typen $3!j_4$, $2?j_1$, $2??$ og $1!!$.

Dette gir to sett med sekvensvariable med identiske navn i de to settene. Vi kan tillate å snakke om de lokale variablene globalt, og for å unngå navnekonflikt, prefikses de med prosessnavn, P_i.

Vi ser nok en gang på det samme eksemplet. Dette vil gi følgende lokale, oppsplittede sekvenser for prosess P_1:

- $!x = \langle 3 \rangle$
- $?x = \varepsilon$
- $!y = \varepsilon$

- $?y = \langle y \rangle$ — lokalt kan vi ikke vite hvilken verdi som ble lest, og som nevnt i eksemplet med h_i kan det være dumt å bruke y i spesifikasjonen av historiesekvensen. Derfor vil vi ofte bare si $\#?y = 1$.
- $\alpha = \langle !1, ?2 \rangle$

og vi vet følgende om sekvensene for prosess P_2:

- $!x = \langle y \rangle$ — vi vet ikke hvilken verdi som ble tilordnet, bare at denne er lik y som ble lest ved inngang til kritisk region, og ikke er endret ennå.
- $?x = \langle 3 \rangle$
- $!y = \langle y + 1 \rangle$
- $?y = \langle y \rangle$
- $\alpha = \langle ??, !! \rangle$

Som når vi hadde hele historien samlet i en sekvens kan vi vite sammenhengen mellom verdiene som ble lest og tilordnet i **await**-setningen: $!y[1] = ?y[1]$ og $!x[1] = ?y[1] + 1$. Det er bedre å bruke dette i spesifikasjonen sammen med informasjon om lengden av sekvensene enn å referere til y.

Vi får følgende globale sekvenser:

- $!x = \langle 3, 1 \rangle$
- $?x = \langle 3 \rangle$
- $!y = \langle 0 \rangle$
- $?y = \langle 0, 0 \rangle$
- $\alpha = \langle 1!1, 1?2, 2??, 2!! \rangle$ eller $\alpha = \langle 1!1, 2??, 2!!, 1?2 \rangle$

Legg merke til at elementet 2?? skal matche med et element fra *begge* ?-sekvenser, siden inngang til kritisk region kan ses på som lesing av alle felles variable. For 2!! må vi ha et matchende element i hver !-sekvens.

Vi vet også følgende om de lokale sekvensene, sett fra et globalt synspunkt:

- $P_1!x = \langle 3 \rangle$
- $P_1?x = \varepsilon$
- $P_1!y = \varepsilon$
- $P_1?y = \langle 0 \rangle$
- $P_1.\alpha = \langle !1, ?2 \rangle$

- $P_2!x = \langle 1 \rangle$
- $P_2?x = \langle 3 \rangle$
- $P_2!y = \langle 0 \rangle$
- $P_2?y = \langle 0 \rangle$
- $P_2.\alpha = \langle ??, !! \rangle$

De historiesekvensene som er presentert i dette avsnittet inneholder svært mye informasjon. Faktisk er mye av denne informasjonen overflødig. Som vi skal se senere i oppgaven klarer vi oss med en delmengde av de oppsplittede sekvensene som eneste mytiske variable, uten at vi mister kompletthet av bevissystemet.

2.5 System 1: Bevissystem uten global invariant

Det første bevissystemet fra [Owe92] bygger på en modell for parallelle programmer. I denne modellen vil operasjoner på felles variable ha sideeffekt på historievariable, h_i, som beskrevet i forrige avsnitt. I tillegg er det i modellen tatt hensyn til ikke-determinismen vi har ved lesing og inngang til kritisk region. Ved **coend** flettes de lokale historiesekvensene sammen til en global sekvens.

Bevissystemet er sunt og komplett relativt til denne modellen. Med den regelen for parallell sammensetning som vi har i dette bevissystemet trenger vi i beviset av en prosess ikke å ta hensyn til hva en annen prosess gjør. Dermed tillater systemet hierarkisk oppbygging av bevis.

2.5.1 Regler

Tilordning til felles variabel:

$$\text{T1:} \qquad \{Q^{r_j, h_i}_{e,\ h_i \vdash (!j,e)}\} r_j := e \{Q\}$$

Vi ser at i denne regelen endres r_j som vanlig for tilordningsregler. I tillegg er sideeffekten at h_i utvides med elementet $(!j, e)$ tatt med.

Lesing av felles variabel:

$$\text{L1:} \qquad \{\forall r_j | Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)}\} v := e_j \{Q\}$$

der r_j er den felles variabelen som nevnes i uttrykket e_j. I denne regelen oppdateres h_i, og i tillegg er ikke-determinismen ved at vi ikke kan vite hvilken verdi som er lest, fanget inn med en $\forall$-kvantor. Dette innebærer at vi lokalt ikke kan anta noe om verdien av en variabel vi er i ferd med å lese. Siden dette er en felles variabel kan den være endret av en annen prosess like før denne prosessen leser den.

Kritisk region:

$$\text{K1:} \qquad \frac{\{P\}S\{Q^{h_i}_{h_i \vdash (!!r)}\}}{\{\forall r | e \Rightarrow P^{h_i}_{h_i \vdash (??r)}\}\textbf{await } e \textbf{ do } S \textbf{ od}\{Q\}}$$

Ventebetingelsen e må være sann i det vi går inn i kritisk region, og dermed kan vi anta denne i beviset. Siden alle variable leses, må vi ha en $\forall$-kvantor for hele vektoren av felles variable.

Parallell sammensetning:

$$\text{P1:} \qquad \frac{\{P_i\}S_i\{Q_i\}}{\{r = r0 \bigwedge_i P_i^{h_i}{}_{\varepsilon}\}\textbf{cobegin } S_1 \parallel \cdots \parallel S_n \textbf{ coend}\{\exists h | \bigwedge_i \overline{Q_i} \bigwedge_j r_j = final_j(h)\}}$$

for alle $i = 1, \ldots, n$. $\overline{Q_i}$ er $(Q_{i\,..final_j(h_i)..}^{\cdots r_j \cdots})^{h_i}_{local_i(h)}$, og $r0$ er vektoren av startverdiene til variablene. Historiesekvensene skal være tomme ved **cobegin**. Samme sted kan vi anta at variablene har initialverdien $r0$. Vektoren av initialverdier brukes som tidligere nevnt i funksjonen $final_j$ for å finne verdien av en variabel gitt en historiesekvens. Denne funksjonen brukes etter **coend** for å gi verdien av de felles variablene. Etter **coend** er alle de lokale forekomster av felles variable erstattet av $final_j(h_i)$ siden det ikke gir mening å snakke om lokale verdier for felles variable på dette stedet. Videre flettes historiesekvensene sammen til en global historiesekvens h, og denne globale sekvensen skjules med en $\exists$-kvantor, siden vi ikke ønsker mytiske variable utenfor **cobegin**...**coend**.

Variable som er lokale i en prosess, skulle egentlig ha vært deklarert eksplisitt. Vi antar isteden at vi har en implisitt deklarasjonssetning i starten av S_i og en konstruksjon som terminerer skopet til de lokale variablene på slutten. Ved å bruke en Hoare-regel for variabeldeklarasjoner, ser vi at lokale variable vil skjules av en $\forall$-kvantor i P_i og av en $\exists$-kvantor i Q_i.

2.6 System 2: Bevissystem med global invariant

I bevissystemet som ble presentert i forrige avsnitt ble beviset for hver prosess utført kun med kunnskap om denne prosessen vi for øyeblikket så på. Først på slutten ble de ulike sekvensen koblet sammen, og vi så der på hele den globale historien.

Isteden kan det være ønskelig å kunne anta noe om de andre prosessene også under beviset lokalt for en prosess. Dette kan gjøres som i [Owe92], ved å innføre en global invariant, som bestandig skal være sann. Denne kan da antas et hvilket som helst sted i prosessene, men samtidig må vi forsikre oss om at det ikke utføres noen programsetninger som ødelegger invarianten.

Dette bevissystemet er også sunt og komplett relativt til modellen, og det tillater at bevis bygges opp hierarkisk. Systemet har den fordelen fremfor System 1 at det blir flere muligheter for hvordan vi kan utføre beviset av et program. Vi kan fortsatt la all informasjonen ligge i de lokale betingelsene, men i stedet lønner det seg ofte å bruke den globale invarianten for å spesifisere hvordan de ulike prosessene samhandler.

Vi bruker følgende notasjon

$$(I)\{P\}S\{Q\}$$

som betyr

> Hvis I og P holder før S, og S terminerer, vil I og Q holde etter at S har terminert, såfremt alle andre prosesser sørger for å vedlikeholde I.

I de lokale betingelsene kan vi bare se på den lokale historiesekvensen h_i for denne prosessen og lokale utgaver av felles variable.

I den globale invarianten ser vi imidlertid på alle prosessenes historie h og global verdi for felles variable. Her kan vi selvsagt ikke referere til lokale variable.

For å kunne se på både lokale og globale betingelser definerer vi først noen utsagn om sammenhengen mellom lokale og globale sekvenser/verdier.

Den lokale verdien av en variabel er den siste som er tilordnet eller lest av denne prosessen. Dette vil være avleiret i h_i:

$$R_i \triangleq \bigwedge_j r_j = final_j(h_i)$$

Den globale verdien av en variabel er den siste verdien som en av prosessene har gitt den, og denne vil finnes i h:

$$R \triangleq \bigwedge_j r_j = final_j(h)$$

Det er naturligvis en sammenheng mellom hvordan lokale og globale historiesekvenser kan se ut på et gitt sted i programmet.

$$H_i \triangleq h_i = local_i(h)$$

Disse sammenhengene bruker vi for å få en global betingelse, gitt en lokal:

$$G_i[P] \triangleq R \wedge \exists r, h_i | R_i \wedge H_i \wedge P$$

og motsatt for å få en lokal betingelse fra en global:

$$L_i[P] \triangleq R_i \wedge \exists r, h | R \wedge H_i \wedge P$$

I noen tilfeller kan vi postulere at h og h_i gir samme verdi for r_j. Da kan vi bruke en sterkere variant av disse operatorene, G_i^j og L_i^j der j betyr at $\exists$-kvantoren kan sløyfes for r_j.

For hver regel må vi nå, som i forrige avsnitt, vise de lokale betingelsene. Nå får vi lov å anta en lokalisert utgave av den globale invarianten også.

I tillegg må det vises at den globale invarianten vedlikeholdes. Siden denne bare inneholder variable som har relevans for de felles variable, er det bare nødvendig å vise dette for programsetninger som faktisk manipulerer de felles variable. I et slikt bevis kan vi anta både den globale invarianten og en globalisert utgave av den lokale forbetingelsen.

2.6.1 Regler

Tilordning til felles variabel:

$$\text{T2:} \qquad \frac{P \wedge L_i[I] \Rightarrow Q^{r_j,h_i}_{e,\ h_i \vdash (!j,e)} \quad , \quad I \wedge G_i[P] \Rightarrow I^{r_j,h}_{e,\ h \vdash (i!j,e)}}{(I)\{P\} r_j := e \{Q\}}$$

Første premiss i denne regelen tilsvarer aksiomskjemaet i System 1. Beviset for at den globale invarianten vedlikeholdes svarer til andre premiss.

Lesing av felles variabel:

$$\text{L2:} \qquad \frac{P \wedge L^j_i[I] \Rightarrow Q^{v,\ h_i}_{e_j, h_i \vdash (?j,r_j)} \quad , \quad I \wedge G^j_i[P] \Rightarrow I^h_{h \vdash (i?j)}}{(I)\{\forall r_j | P\} v := e_j \{Q\}}$$

der r_j er den felles variabelen som nevnes i uttrykket e_j. Her har vi også en premiss for lokale betingelser og en for globale som i regelen for tilordning til felles variabel.

Kritisk region:

$$\text{K2:} \qquad \frac{\{e \wedge I \wedge P \wedge R \wedge R_i\} S \{(H_i \Rightarrow Q \wedge I)^{h,\ h_i}_{h \vdash (i!!r), h_i \vdash (!!r)}\}}{(I)\{\forall r | P^{h_i}_{h_i \vdash (??r)}\} \textbf{await } e \textbf{ do } S \textbf{ od} \{Q\}}$$

Siden inngang til kritisk region tilsvarer at denne prosessen leser alle variablene, vil vi ha samme verdier lokalt og globalt for alle variablene. Vi kan se på lokale og globale betingelser på samme nivå uten å bruke L_i og G_i, og derfor trenger vi bare én premiss i denne regelen. Fordi vi ser på kritisk region som en atomisk operasjon er det ikke nødvendig at invarianten er sann hele tiden i S.

Parallell sammensetning:

$$\text{P2:} \qquad \frac{(I)\{P_i\} S_i \{Q_i\}}{\{I^h_\varepsilon \wedge r = r0 \bigwedge_i P_i{}^{h_i}_\varepsilon\} \textbf{cobegin } S_1 \parallel \cdots \parallel S_n \textbf{ coend} \{\exists h | I \bigwedge_i G_i[Q_i]\}}$$

for alle $i = 1, \ldots, n$. Vi ser at historiesekvensene skal være tomme i alle utsagn før **cobegin** og at vi etter **coend** ser på globale betingelser. Også i denne regelen vil forekomster av lokale variable i P_i og Q_i være implisitt kvantorisert vekk. Utenfor **cobegin**...**coend** vil det dermed bare være frie forekomster av felles variable, siden de mytiske variablene også fjernes.

Vi har også disse reglene:

$$\frac{\{P\} S \{Q\}}{(I)\{P\} S \{Q\}} \qquad \text{for } \{r\} \cap \mathcal{V}[S] = \emptyset$$

Dersom S ikke refererer til felles variable vil den ikke ødelegge den globale invarianten.

$$\frac{(I)\{P\}S1\{Q\} \quad , \quad (I)\{Q\}S2\{R\}}{(I)\{P\}S1; S2\{R\}}$$

Dersom invarianten vedlikeholdes over to programsetninger, vil den også vedlikeholdes over den sekvensielle sammensetningen av dem.

$$\frac{(I)\{P\}S\{Q\}}{(I)\{L_i[I] \Rightarrow P\}S\{L_i[I] \wedge Q\}}$$

Vi kan alltid anta en lokal utgave av den globale invarianten i den lokale delen av et bevis.

2.7 Andre arbeider

Svært mange bevissystemer er foreslått for parallelle programmer. Felles for mange av dem er at de bygger på Hoare-logikk [Hoa69]. Et slikt bevissystem vil aldri kunne være komplett, så komplett vil i dette avsnittet bety *relativt komplett*. Først ser vi på bevissystemer der det ikke nødvendigvis er gitt hvordan fravær av vranglås skal bevises.

2.7.1 Felles variable

Et av de første forslagene kom fra Hoare [Hoa72]. I dette systemet gis en regel for parallell sammensetning, der for- og bakbetingelsen til hele programmet er konjunksjonen av betingelsene for hver enkelt prosess og en *ressursinvariant*. Det forutsettes at endring av felles variable foregår inne i en kritisk region med hensyn på denne variabelen, og felles variable kan bare nevnes i ressursinvarianten. Disse reglene er dermed ikke sterke nok i mange tilfeller.

I [Owi75] finner vi et komplett bevissystem for parallelle programmer med felles variable. Det er ikke nødvendig at prosessene er disjunkte, men det kreves et bevis for at hver setning i en prosess ikke ødelegger beviset for de andre prosessene. Dette kalles bevis av *ikke-interferens*. For å gjennomføre et slikt bevis må vi ved beviset av en prosess vite hvordan de andre prosessene implementeres, og dette bevissystemet er altså ikke egnet til hierarkisk oppbygging av bevis. Det vises at mytiske variable eller en lignende utvidelse er nødvendig for å ha et komplett bevissystem. Derfor gis det en generell regel for håndtering av mytiske variable, men det er ikke spesifisert hva slags mytiske variable som skal brukes.

[Ash75] bygger på Floyds metode for sekvensielle programmer [Flo67], der programmer representeres som flytdiagrammer, og det settes en betingelse på hver pil i diagrammet. I dette systemet brukes en global betingelse som må overholdes av hver programsetning, og det må vises at denne impliserer hver lokale betingelse i programmet. I motsetning til Hoare-logikk er ikke Floyds metode egnet til hierarkisk oppbygging av bevis selv for sekvensielle programmer. Dermed vil heller ikke dette systemet for parallelle programmer [Ash75] være egnet til det.

I [Lam80] gis et bevissystem som ligner svært på [Owi75]. Regelen for parallell sammensetning settes opp slik at det må vises at en betingelse holder for hele programmet. I dette

beviset vises også implisitt ikke-interferens via denne betingelsen. Bevissystemet kan brukes til hierarkisk oppbygging av beviser hvis vi sier at det da er tilstrekkelig å få spesifikasjonen av hele programmet fra spesifikasjonen av prosessene. Imidlertid blir dette svært tunge bevis siden spesifikasjonen er den samme for hele programmet, som for hver enkelt prosess. Vi får ingen reduksjon i kompleksiteten av betingelsene i delbevisene. I dette bevissystemet brukes det kontrollpredikater (*at, in, after*) istedenfor mytiske variable. Dette gjør at bevissystemet blir nokså komplisert.

I [Jon83] presenteres et bevissystem som i tillegg til for- og bakbetingelser har *rely*- og *guarantee*-betingelser. Disse uttrykker hhv. hva en prosess kan stole på at omgivelsene overholder og hva prosessen garanterer at den vil overholde. Systemet er egnet til hierarkisk oppbygging av bevis. Det brukes ikke mytiske variable i dette systemet, men man kan spesifisere at en variabel har hatt en bestemt verdi. Det er ikke vist at bevissystemet er komplett.

[Sti88] bygger på [Jon83], men istedenfor enkle rely- og guarantee-betingelser brukes mengder av betingelser i denne delen av spesifikasjonen. Det brukes mytiske variable som i [Owi75], og det vises sunnhet og kompletthet av systemet.

I [Stø91] utvides Jones' metode med en *wait*-betingelse i tillegg til for-, bak-, rely- og guarantee-betingelsene. Denne wait-betingelsen uttrykker når prosessen kan blokkeres. Det er også tatt med vilkårlige mytiske variable i dette bevissystemet. Systemet er sunt og komplett.

I [Sou84] brukes historievariable for hver prosess, og i disse variablene tas det vare på informasjon om hvilke operasjoner prosessen har utført. Dette er mytiske variable, men de er en del av bevissystemet. Det er ikke nødvendig med flere mytiske variable i konstruksjonen av et bevis av et program, så man slipper å måtte implementere mytiske variable som en del av programmet. Som en del av regelen for parallell sammensetning flettes de lokale historiene sammen. De må da oppfylle et *kompatibilitetspredikat* som kan være noe tungt å vise. Videre er det ikke tillatt å referere til felles variable i betingelsene i et bevis. Dette kan virke som en unødvendig begrensning. Systemet er egnet til hierarkisk oppbygging av bevis siden endring i implementasjonen av en prosess bare vil ha effekt for beviset av denne prosessen og spesifikasjonen av hele programmet.

Som nevnt tidligere bygger denne oppgaven på [Owe92], som igjen bygger på [Sou84]. I [Owe92] er ikke kompatibilitetspredikatet eksplisitt brukt. Imidlertid kan det også i dette systemet bli nokså kompliserte resonnementer om flettinger av historiesekvenser. Det som er gjort her ligner mer på [Mel86a] som beskrives i neste avsnitt. I [Owe92] tillates også referanser til felles variable i betingelsene.

I [Dæh87] introduseres globale historiesekvenser og global invariant. Kompatibilitetspredikatet for historiesekvensene ved **coend** er med i dette bevissystemet også. Systemet utvides til også å omfatte progresjon. Dette arbeidet bygger på [Sou84, Mel86a].

I [GMK89] inkluderes predikater i historiene. Fortsatt brukes kompatibilitetspredikat, men det er enklere å få det til å implisere ønsket bakbetingelse for programmet fordi informasjon om verdien av predikater er tatt med.

Ikke alle bevissystemer bygger så direkte på Hoare-logikk. I [MP84] brukes *temporal logikk* for å vise total korrekthet av programmer. Det ses på sekvenser av tilstander, og betingelser

kan referere til andre tilstander i programmet. Til dette innføres det temporale operatorer. Av de viktigste kan nevnes *always* som innebærer at det etterfølgende utsagn alltid skal være sant og *eventually* der det etterfølgende utsagn skal være sant for en tilstand senere i programmet. I tillegg brukes også et kontrollpredikat *at* som holder rede på hvor programkontrollen er. Den delen av systemet som brukes for å vise partiell korrekthet ligner på [Lam80] der det må vises at alle deler av programmet vedlikeholder en invariant.

Av andre bevissystemer kan også nevnes [FS81], der et parallelt program transformeres til et ikke-deterministisk program som bevises på lignende vis som i [Dij76]. Imidlertid gis det ikke generelle regler for en slik transformasjon, så systemet blir vanskelig å bruke.

2.7.2 Kommunikasjon via kanaler

Parallelt med utviklingen av bevissystemer for programmer med felles variable, har det også blitt laget systemer for å bevise programmer der prosessene kommuniserer via kanaler. I f.eks. CSP (Communicating Sequential Processes) [Hoa78] foregår samhandlinger mellom prosessene på denne måten. Utviklingen av disse bevissystemene har blitt påvirket av eksisterende systemer for felles variable og omvendt, uten at det nevnes eksplisitt for hvert system.

I [AFdR80] er det ikke gitt i bevissystemet som egen regel hvordan betingelsene skal være i forbindelse med en kommuniksajonssetning. For at systemet skal være sunt må det derfor bevises at prosessene *cooperates*, og til dette formål introduseres en global invariant, som må vedlikeholdes. Et bevis av cooperation vil tilsvare Owickis bevis av ikke-interferens. Dermed støtter ikke bevissystemet hierarkisk oppbygging av bevis. Som i Owickis system brukes mytiske variable for å få kompletthet, og disse må implementeres av brukeren av systemet. At bevissystemet er sunt og komplett vises i [Apt83], og der indikeres det hvordan systemet kan utvides til å håndtere total korrekthet.

Et bevissystem for total korrekthet finnes i [LG81]. Som i [AFdR80] kan hva som helst gjelde etter en kommunikasjon. Derfor må det gis et *satisfaction*-bevis som sikrer at disse bakbetingelsene stemmer med hverandre. Det tillates at mytiske variable er globale, og dermed er det nødvendig med et bevis av *ikke-interferens*. For å minimere den sjekkingen som må utføres her innføres *synchronously altered*-variable og *universal*-betingelser, noe som gjør bevissystemet ganske komplisert. Det kan virke som det er greiere å bruke en global invariant. Systemet er ikke egnet til hierarkisk oppbygging av bevis, fordi det må gjennomføres en rekke resonnementer utfra hvordan de andre prosessene er implementert.

I [MC81] spesifiseres hver prosess med betingelser som inneholder historier av sekvenser som er overført via kommunikasjonskanaler. Denne typen bevissystemer kalles gjerne *trace-basert*. Systemet støtter hierarkisk oppbygging av bevis, og det håndterer nestet parallellitet. Imidlertid blir systemet komplisert når sekvensielle konstruksjoner inkluderes.

Kommunikasjonshistorier brukes også i [Hoa81]. Betingelser er invariant for en prosess, og systemet kan brukes til hierarkisk oppbygging av bevis.

Dessverre er ikke disse to trace-baserte bevissystemene komplette, som vist i [Ngu85, WGS92]. [Ngu85] tar spesielt for seg Misra og Chandys bevissystem og foreslår en ny regel som vil gjøre dette systemet komplett. Denne regelen er imidlertid svært uformell. I [WGS92] er utvidelsen mer formell. Her gis et *temporal ordning*- og et *prefiks*-aksiom som

sammen med *always*-operatoren fra temporal logikk er tilstrekkelig for å få kompletthet i følge [WGS92]. I [Sou91] vises det at heller ikke dette er tilstrekkelig. Det argumenteres for at det er nødvendig med temporal logikk eller en historie for hver prosess istedenfor kanal.

Soundararajan [Sou86] har laget et bevissystem for CSP som er svært likt systemet for felles variable [Sou84].

I [Mel86a] brukes en global historiesekvens for hele systemet av prosesser, men det brukes ikke global invariant, bare lokale betingelser, som kan referere til den globale historien. Det gis en funksjon for å fjerne intern kommunikasjon fra den globale historien, og dermed kan bevissystemet brukes til å bevise programmer med nestet parallellitet. Systemet er egnet til hierarkisk oppbygging av bevis, og det har den fordelen at det ikke er nødvendig med kompatibilitetspredikat ved parallell sammensetning, som i [Sou84, Sou86, Dæh87, GMK89]. Systemet er utledet fra en modell, og det argumenteres for at det er komplett, fordi det er bevist at et lignende system er komplett [Mel86b].

2.7.3 Vranglås

Å bevise fravær av vranglås gjøres ofte som et ekstra bevis, i tillegg til et bevis for partiell korrekthet av et program. Imidlertid er det også noen bevissystemer hvor dette glir inn som en del av resten av beviset. Her skal vi se eksempler på hvordan begge metoder er brukt.

Ressurser kan gis en ordning som i [Hoa72]. Dersom alle prosesser først tar ressurser av lavest orden når de skal ta flere ressurser, vil det aldri bli vranglås. Dette er en enkel, men dessverre ikke generell metode for å unngå vranglås.

I [Owi75] vises det at dersom forbetingelsen til en kritisk region er sann, må den tilhørende ventebetingelsen også være sann for at vi ikke skal få vranglås. Dette kravet er sterkere enn nødvendig, men det blir ikke så mange muligheter som må sjekkes, som det fort kan bli i andre metoder.

I [AFdR80] ses det på tupler av tilstander der prosessene er blokkert. Dersom ikke alle lokale betingelser, som tilhører et slikt tuppel, og den globale invarianten kan være sanne samtidig, kan det ikke være vranglås. Dette må vises for hvert tuppel, og dette blir et bevis av eksponensiell orden.

En lignende metode som [AFdR80] er gitt i [LG81]. Der ses det på mulige konfigurasjoner, dvs. tilstander som kan gi opphav til vranglås. For hver slik konfigurasjon, må det vises at det enten er umulig å komme dit, eller at det finnes guard som er sann, slik at vi får progresjon.

I [Hoa81] er en del av spesifikasjonen av en kanal mengden av verdier denne kanalen er klar til å kommunisere. Dersom denne mengden er tom for alle kanaler har vi vranglås. Som en del av spesifikasjonen for hele programmet, må vi altså vise at ikke alle kanaler kan ha tom mengde av verdier den er klar til å sende.

I [FS81] gis det egne *predicate transformers* for en rekke korrekthetsegenskaper, deriblant fravær av vranglås. Imidlertid tar disse utgangspunkt i et ikke-deterministisk program, og det er vanskelig å se hvordan de skal brukes.

I bevissystemer som bygger på historiesekvenser, kan det legges begrensninger på hvilke historiesekvenser som gir lovlige eksekveringer av et program, som i [Sou86]. Her må historiesekvensene fra prosessenes bakbetingelser (før **coend**) oppfylle et konsistenspredikat. Som en del av dette predikatet sikres det at historiesekvensene ikke vil kunne gi eksekveringer som gir opphav til vranglås.

Historiesekvenser benyttes i [Dæh87] også. Der er det en global historiesekvens, og fravær av vranglås sikres ved å utføre et bevis for at det alltid må finnes et element som den globale historiesekvensen kan forlenges med. En slik betingelse må impliseres av den globale invarianten.

En annen måte å gjøre det på er som i [Stø91] med en egen wait-betingelse, som uttrykker når prosessen kan blokkeres. I dette bevissystemet vil vranglåsfrihet ofte bevises samtidig som resten av beviset.

Kapittel 3

System 3: Redusert mengde av mytiske variable

Owe [Owe92] foreslår å ta utgangspunkt i System 2, splitte opp historiesekvensene og fjerne informasjonen om rekkefølgen av operasjoner, α-sekvensene. Dermed står vi igjen med en !-sekvens (verdiene som er tilordnet) og en ?-sekvens (verdiene som er lest) for hver variabel. I følge [Owe92] bør dette være en tilstrekkelig mengde med mytiske variable for at et bevissystem skal være komplett, fordi flettinger kan uttrykkes i invarianten som informasjon om !- og ?-sekvensenes relative lengder.

I bevissystemet som kalles System 3, går jeg et skritt lenger enn foreslått. Jeg tar også vekk ?-sekvenser og dermed muligheten for å huske hvilke verdier som er lest. Informasjonen som lå i α-sekvensen og ?-sekvensene kan uttrykkes ved å si eksplisitt i den globale invarianten hvordan de lokale !-sekvensene kan flettes og hvor lange sekvensene kan være relativt til hverandre.

3.1 Notasjon

Som definert i avsnitt 2.4.4 er sekvenser/variable i en global betingelse lokale hvis de har prosessidentifikasjon foran, ellers er de globale. I lokale betingelser spesifiserer vi bare lokal informasjon, så derfor brukes ikke prosessidentifikasjon foran sekvenser/variable. Siden vi ofte vil ha behov for å referere til alle felles variable eller alle !-sekvenser benyttes følgende forkortelser:

- r betyr $r_1, \ldots, r_m$, vektoren av alle de felles variablene
- $i.r$ betyr $i.r_1, \ldots, i.r_m$, vektoren av alle lokale felles variable i en global betingelse
- $!r$ betyr $!r_1, \ldots, !r_m$, alle tilordningssekvenser
- $i!r$ betyr $i!r_1, \ldots, i!r_m$, alle lokale tilordningssekvenser i en global betingelse
- $!r \vdash r$ er forkortelse for $!r_1 \vdash r_1, \ldots, !r_m \vdash r_m$

- $i!r \vdash r$ er forkortelse for $i!r_1 \vdash r_1, \ldots, i!r_m \vdash r_m$
- ε brukes som forkortelse for $\varepsilon, \ldots, \varepsilon$, der det ikke skaper forvirring

3.2 Modell

Vi tar utgangspunkt i modellen for System 2, som beskrives i [Owe92], s. 329, og modifiserer denne slik at vi får en skisse av en modell for System 3. Sammenhengen mellom globale og lokale tilstander er i System 2 gitt ved $h_i = local_i(h)$. I System 3 tilsvarer dette at globale !-sekvenser må være en fletting av de lokale sekvensene for samme variabel, som uttrykt formelt i første del av $\mathcal{M}$, definert nedenfor.

Videre har vi i System 2 at den globale verdien av en variabel kan finnes ved $r_j = final_j(h)$. Slik $final_j$ er definert vil dette tilsvare at verdien av r_j i System 3 er siste element i $!r_j$, og dette gir andre del av $\mathcal{M}$. Tilsammen får vi at følgende alltid vil gjelde sett fra et globalt synspunkt:

$$\mathcal{M} \triangleq \bigwedge_j (!r_j \textbf{ ismerge } 1!r_j, \ldots, n!r_j \wedge r_j = rt(r_j 0 \dashv !r_j))$$

I modellen for System 2 redeklareres hver felles variabel r_j slik at alle prosessene har en lokal kopi som de opererer på. Denne redeklarasjonen fjernes i modellen for System 3. Der har prosessene isteden en egen utgave av hver felles variabel, $i.r_j$. I tillegg vil de ha tilgang til r_j, men dette gjelder ikke i lokale tilstander, som forklart nedenfor. I $i.r_j$ huskes siste verdi prosessen har tilordnet eller lest. Ved tilordning må den globale variabelen r_j også endres.

De mytiske variablene deklareres på lignende vis som i modellen for System 2. Dermed vil vi få samme effekt for parallell sammensetning, med unntak av at de mytiske variablene bare har informasjon om tilordning til felles variable, mens de i System 2 også har informasjon om lesing og α-informasjon.

Modelleringen av setningene, som refererer til felles variable, tilsvarer det som skjer med den globale tilstanden. En lokal tilstand vil bare gi verdier til lokale variable. Derfor vil effekten på lokale betingelser ut fra modellen bare være på lokale variable og variable prefikset med i. Variablene r_j og $!r_j$, som er globale, kan altså ikke brukes i lokale betingelser. Der kan det bare refereres til $i.r_j$ og $i!r_j$, og derfor kan vi endre notasjonen slik at i-prefikset er fjernet lokalt.

For å få en lokal betingelse fra en global, må vi skjule alle variable som ikke gir mening i en lokal tilstand, og vi tar vekk i-prefikset for de felles variablene og !-sekvensene. Til dette bruker vi denne operatoren:

$$\mathcal{L}_i[P] \triangleq (\exists !r, r, 1!r, 1.r, \ldots, i-1!r, i-1.r, i+1!r, i+1.r, \ldots, n!r, n.r | P)^{i!r, i.r}_{!r,\ r}$$

For å få globale betingelser fra lokale trenger vi bare å prefikse variable med prosessidentifikasjon:

$$\mathcal{G}_i[P] \triangleq P^{!r,r}_{i!r,i.r}$$

Rett etter lesing og skriving av variabel r_j er lokal og global verdi lik. Dette kan vi ta med i $\mathcal{L}_i^j$ ved ikke å sette prosessidentifikasjon foran r_j og i $\mathcal{G}_i^j$ ved ikke å kvantorisere vekk

samme variabel. Inne i kritisk region er alle lokale og globale variable like, så vi trenger ikke å skjule r. Disse utgavene kaller vi $\mathcal{L}_i^*$ og $\mathcal{G}_i^*$.

I motsetning til i [Owe92] vil $\mathcal{G}_i$ navne om variable og ikke benytte kvantorer for å skjule dem. Dermed kan lokale betingelser uttrykkes globalt uten at vi mister informasjon.

Modellering av tilordning til felles variabel, $r_j := e$, gjøres som følger:

$$r_j := e;\ i.r_j := r_j;\ !r_j :=!r_j \vdash r_j\ i!r_j := i!r_j \vdash r_j$$

Både lokale og globale variable endres, og disse fire setningene ses på som én atomisk operasjon.

Ved lesing av felles variable, $v := e_j$, skjer det ingen endring av historievariable, men det er fortsatt ikke-deterministisk hva verdien av r_j er :

$$i.r_j := (\textbf{some}\ r_j \mid \mathcal{L}_i^j[I \wedge \mathcal{M}]);\ v := e_j$$

I tilordningen til $i.r_j$ vil **some**-konstruksjonen plukke ut en global verdi av r_j som oppfyller den globale invarianten, I og systeminvarianten, $\mathcal{M}$. Av denne grunn har vi brukt $\mathcal{L}_i^j$ som ikke skjuler r_j med en kvantor.

Kritisk region, **await** e **do** S **od**, modelleres som:

$$i.r := (\textbf{some}\ r \mid e \wedge \mathcal{L}_i^*[I \wedge \mathcal{M}]);\ S;\ !r :=!r \vdash r;\ i!r := i!r \vdash r$$

Ved inngang til kritisk region har vi en **some**-konstruksjon som gir verdier til variablene slik at $I \wedge \mathcal{M}$ er oppfylt. Ingen historievariable endres ved inngang til kritisk region, men ved utgang vil alle !-sekvensene bli oppdatert.

3.3 Regler

Ut fra modellen i forrige avsnitt kan vi lage følgende regler:

Tilordning:

T3:
$$\frac{I \wedge \mathcal{M} \wedge \mathcal{G}_i[P] \Rightarrow (I \wedge \mathcal{G}_i[Q])_{e,\ e,\ !r_j \vdash e, i!r_j \vdash e}^{r_j, i.r_j, !r_j,\ \ i!r_j}}{(I)\{P\}r_j := e\{Q\}}$$

Merk at vi bare har en premiss i denne regelen. Det er fordi all verifikasjon kan foregå på det globale nivået siden vi ikke bruker kvantor for å skjule mytiske variable i $\mathcal{G}_i$-operatoren, bare navner dem om.

Lesing:

L3:
$$(I)\{\forall r_j | \mathcal{L}_i^j[I \wedge \mathcal{M}] \Rightarrow P_{e_j}^v\} v := e_j\{P\}$$

r_j er variabelen som det refereres til i e_j. Det er kun nødvendig å vise den lokale delen her, siden I ikke kan referere til lokale variable, og lesing ikke avleires i mytiske variable.

Invarianten kan derfor ikke ødelegges av lesing, som den kan i System 2. Som vanlig brukes en $\forall$-kvantor i forbetingelsen for å gjenspeile ikke-determinismen vi har ved lesing.

Kritisk region:

$$\text{K3:} \quad \frac{\{e \wedge I \wedge \mathcal{M} \wedge \mathcal{G}_i^*[P]\} S \{(I \wedge \mathcal{G}_i^*[Q])^{!r,\ i!r}_{!r \vdash r, i!r \vdash r}\}}{(I)\{\forall r | P\}\textbf{await } e \textbf{ do } S \textbf{ od}\{Q\}}$$

Det er kun utgang fra kritisk region som avleires i mytiske variable, og dette gjenspeiles i regelen. Egentlig burde vi ha en global utgave av ventebetingelsen, $\mathcal{G}_i^*[e]$, i premissen, men siden $\mathcal{G}_i^*[e] = e$ forkorter vi dette til e. Forøvrig kan det være av interesse å merke seg at denne regelen ble svært lik den i System 2, med unntak av at her er det ingen endring av mytiske variable ved inngang til kritisk region.

Parallell sammensetning:

$$\text{P3:} \quad \frac{(I)\{P_i\} S_i \{Q_i\}}{\{\overline{I} \wedge r = r0 \bigwedge_i P_i{}^{!r}_{\varepsilon}\}\textbf{cobegin } S_1 \parallel \cdots \parallel S_n \textbf{ coend}\{\exists !r, 1!r, .., n!r | I \wedge \mathcal{M} \bigwedge_i \mathcal{G}_i[Q_i]\}}$$

for alle $i = 1, \ldots, n$. $\overline{I}$ er $I^{!r,1!r,\ldots,n!r}_{\varepsilon,\ \varepsilon,\ \ldots,\varepsilon}$.

Siden vi har tatt utgangspunkt i samme modell som i System 2 og bare fjernet en del mytisk informasjon, er det rimelig at denne regelen ligner svært på den i System 2. Som i System 1 og System 2 har vi implisitt deklarasjon av variable i S_i slik at P_i og Q_i ikke inneholder frie forekomster av lokale variable.

3.4 Eksempler

Begge eksemplene i dette avsnittet illustrerer hvordan den globale invarianten kan inneholde informasjon om hvordan de lokale !-sekvensene kan flettes sammen til en global. I [Owe92] finnes flere eksempler der det bare brukes !-sekvenser i tillegg til de felles variablene i spesifikasjonen.

I et av eksemplene der brukes imidlertid ?-sekvenser for å spesifisere en sekvens av verdier som er overført fra en prosess til en annen. Produsent-konsument-eksemplet her kan ses på som en forenklet utgave av dette eksemplet, der det vesentlige, nemlig overføringen av en sekvens av verdier som skal brukes av mottakerprosessen, er tatt med. Hvis vi sammenligner disse to eksemplene, ser vi at spesifikasjonen blir enklere om vi tillater ?-sekvenser også, men det er interessant å se at et så vanskelig eksempel kan vises med så få mytiske variable.

3.4.1 Et enkelt eksempel med kritisk region

I dette eksemplet må **await**-setningen i $P1$ utføres før den i $P2$. Resultatet av programmet vil være at x blir 1 og y blir 3.

$\{x = 0\}$
cobegin
$P1 :: \{!x = \varepsilon \wedge !y = \varepsilon\}$
await true do
$x := x + 1$
$y := 2$
od
$\{\#!x = 1 \wedge \#!y = 1\}$
$\|$
$P2 :: \{!x = \varepsilon \wedge !y = \varepsilon\}$
await $x = 1$ **do**
$x := 1$
$y := 3$
od
$\{\#!x = 1 \wedge \#!y = 1\}$
coend
$\{x = 1 \wedge y = 3\}$

Informasjon om hvilken rekkefølge tilordningene (**await**-setningene) kan utføres i ligger i den globale invarianten I:

$$!x \textbf{ head } \langle 1,1\rangle \wedge P1!x \textbf{ head } \langle 1\rangle \wedge P2!x \textbf{ head } \langle 1\rangle \wedge$$
$$!y \textbf{ head } \langle 2,3\rangle \wedge P1!y \textbf{ head } \langle 2\rangle \wedge P2!y \textbf{ head } \langle 3\rangle \wedge \#!x = \#!y$$

Denne invarianten inneholder fullstendig informasjon om hvordan historiesekvensene kan være til enhver tid. $\#!x = \#!y$ uttrykker at tilordning foregår til begge variable samtidig. Dette er tatt med fordi **await**-setningen modelleres slik at det foretas en tilordning til alle felles variable ved utgang av kritisk region, og all tilordning til felles variable i dette programmet foregår inne i kritisk region.

Ved **coend** vil alle de lokale sekvensene ha lengde 1, og de globale sekvensene vil dermed ha lengde 2. Vi får $x = rt(!x) = 1$ og $y = rt(!y) = 3$.

For å bevise at **await**-setningen i $P1$ vedlikeholder I bruker vi regelen K2 og følgende resonnement:

Anta $I \wedge P1!x = \varepsilon \wedge P1!y = \varepsilon$. Siden $P1!y = \varepsilon$ må vi også ha $!y = \varepsilon$. Fordi $!x$ og $!y$ er like lange er også $!x = \varepsilon$. Da vet vi at $x = 0$. Dette gir $P1!x \vdash (x + 1) = \langle 1\rangle \wedge P1!y \vdash 2 = \langle 2\rangle$. Det samme vil gjelde for de globale sekvensene. Dermed har vi:

$$I \wedge \mathcal{M} \wedge \mathcal{G}^*_{P1}[!x = \varepsilon \wedge !y = \varepsilon] \wedge e \Rightarrow (I^{!x,\ !y,\ P1!x,\ P1!y,\ P1.x,P1.y}_{!x\vdash x,!y\vdash y,P1!x\vdash x,P1!y\vdash y,x,\ \ y})^{y,x}_{2,x+1}$$

At invarianten vedlikeholdes over kritisk region i $P2$ viser vi slik:

Anta $x = 1 \wedge I \wedge P2!x = \varepsilon \wedge P2!y = \varepsilon$. Fra $x = 1$ og $\#!x = \#!y$ vet vi at $!x = \langle 1\rangle \wedge !y = \langle 2\rangle$. Dermed har vi $P2!x \vdash 1 = \langle 1\rangle \wedge P2!y \vdash 3 = \langle 3\rangle$ og $!x \vdash 1 = \langle 1,1\rangle \wedge !y \vdash 3 = \langle 2,3\rangle$. Tilsammen gir dette:

$$I \wedge \mathcal{M} \wedge \mathcal{G}^*_{P2}[!x = \varepsilon \wedge !y = \varepsilon] \wedge e \Rightarrow (I^{!x,\ !y,\ P2!x,\ P2!y,\ P2.x,P2.y}_{!x\vdash x,!y\vdash y,P2!x\vdash x,P2!y\vdash y,x,\ \ y})^{y,x}_{3,1}$$

Å bevise de lokale betingelsene gjøres som for sekvensielle programmer, og her er dette enkelt.

3.4.2 Produsent- og konsument-prosess

I dette eksemplet produserer prosessen $P1$ data og sender de til konsument-prosessen $P2$. Vi ønsker å vise at alle data leses av konsument-prosessen. Data sendes i variabelen *data*, og den boolske variabelen *ack* brukes til synkronisering, slik at dataene blir lest nøyaktig én gang.

I dette beviset vil vi bruke følgende predikater:

$$\begin{aligned} HE(q,e) &\triangleq q = \varepsilon \vee rt(q) = e \\ E(q,e) &\triangleq q \neq \varepsilon \wedge rt(q) = e \end{aligned}$$

$HE(q,e)$ betyr at hvis det finnes minst et element i q, skal det siste elementet i sekvensen være e. $E(q,e)$ garanterer at det finnes minst et element i q, og det siste elementet i sekvensen skal være e. Dersom e er en boolsk variabel har vi følgende sammenheng mellom disse to predikatene:

$$E(q,e) \equiv \neg HE(q,\neg e)$$

Det dekorerte programmet blir som følger:

$\{\mathbf{true}\}$
$ack := \mathbf{false}$
cobegin
$P1 :: \{!data = !ack = \varepsilon\}$
    while true do $\{\#!data = \#!ack \wedge HE(!ack, \mathbf{true})\}$
        $\langle produserer\ element\ i\ data1 \rangle$;
        await not ack **do od**;
        $\{\#!data = \#!ack \wedge E(!ack, \mathbf{false})\}$
        $data := data1$;
        $\{\#!data = \#!ack + 1 \wedge E(!ack, \mathbf{false})\}$
        $ack := \mathbf{true}$
    od
$\|$
$P2 :: \{!data = !ack = \varepsilon\}$
    while true do $\{\#!data \cdot 2 = \#!ack \wedge HE(!ack, \mathbf{false})\}$
        await ack **do od**;
        $\{\#!data \cdot 2 = \#!ack + 1 \wedge E(!ack, \mathbf{true})\}$
        $data2 := data$;
        $\{data2 = rt(!data) \wedge \#!data \cdot 2 = \#!ack + 1 \wedge E(!ack, \mathbf{true})\}$
        $ack := \mathbf{false}$;
        $\{data2 = rt(!data) \wedge \#!data \cdot 2 = \#!ack \wedge E(!ack, \mathbf{false})\}$
        $\langle bruker\ elementet\ i\ data2 \rangle$;
    od
coend
$\{\mathbf{false}\}$

Vi definerer noen utsagn:

$$A \triangleq ALT(!ack) \wedge ALT1(P1!ack) \wedge ALT2(P2!ack)$$

der $ALT(q)$ betyr at element nr. $1, 4, 5, 8, 9, 12, 13 \ldots$ i sekvensen q er **false** og element nr. $2, 3, 6, 7, 10, 11, \ldots$ er **true**. $ALT1(q)$ betyr at odde elementer av q skal være **false** og like elementer **true**. $ALT2(q)$ betyr at odde elementer av q skal være **true** og like elementer **false**.

$$L \triangleq \#P1!data \geq \#P1!ack \geq \#P2!data \cdot 2 \geq \#P2!ack \geq \#P1!data - 2$$

sier hvilken rekkefølge de ulike operasjonene utføres i. Merk at vi må multiplisere lengden av sekvensen $P2!data$ med 2, fordi denne sekvensen bare er halvparten så lang som de andre sekvensene. Dette er fordi sekvensen $P2!data$ kun oppdateres én gang pr. runde i **while**-løkken i $P2$, mens de andre sekvensene oppdateres 2 ganger pr. runde. Alle !-sekvensene for en prosess oppdateres i en **await**-setning.

Den globale invarianten, I, er:

$$P2!data = P1!data/like[1{:}\#P2!data] \wedge HE(P1!data, data) \wedge A \wedge L \wedge$$
$$(E(P2!ack, \mathbf{true}) \Rightarrow \#P1!data = \#P1!ack \wedge E(P1!ack, \mathbf{true})) \wedge$$
$$(E(P1!ack, \mathbf{false}) \Rightarrow HE(P2!ack, \mathbf{false}))$$

der $q/like$ er sekvensen vi får ved å plukke ut elementene fra q hvor indeksen er et partall.

Den første delen av I uttrykker at hvert element som skrives av $P1$ blir overført til $P2$. Fordi tilordning avleires ved kritisk region vil $P1$ skrive hvert element to ganger. Resten av invarianten er nødvendig for å få med rekkefølgen prosessene utfører de ulike operasjonene i.

At den globale invarianten og løkkeinvariantene holder initielt er opplagt.

Vi må vise at den globale invarianten vedlikeholdes. Først viser vi at **await**-setningen i $P1$ gjør dette:

- Fra L har vi $\#P1!data \geq \#P2!data \cdot 2$. Sammen med $P2!data = P1!data/like[1{:}\#P2!data]$ gir dette:

  $$P2!data = P1!data \vdash data/like[1{:}\#P2!data]$$

- Fra den lokale betingelsen får vi $HE(P1!ack, \mathbf{true})$ og ventebetingelsen er $\neg ack$. Sammen med $ALT1(P1!ack)$ gir dette:

  $$ALT1(P1!ack \vdash ack)$$

- Fra A, L, $HE(P1!ack, \mathbf{true})$ og $\neg ack$ kan vi slutte at de to lokale sekvensene er like lange og at $!ack$ er tom eller slutter på $\langle \mathbf{false}, \mathbf{true}, \mathbf{true}, \mathbf{false} \rangle$. Dermed har vi

  $$ALT(!ack \vdash ack)$$

- Fra argumentet i forrige punkt vet vi at $\#P1!ack = \#P2!ack$, og fra den lokale forbetingelsen har vi $\#P1!data = \#P1!ack$. Tilsammen gir dette

  $$\#P2!ack \geq (\#P1!data + 1) - 2$$

- Vi har $HE(P1!ack, \mathbf{true})$ og $\neg ack$. Dermed må det være $P2$ som har satt ack til **false** eller begge de lokale sekvensene er tomme. Dermed har vi $HE(P2!ack, \mathbf{false})$, som igjen gir

$$(E(P2!ack), \mathbf{true}) \Rightarrow E(P1!ack \vdash ack, \mathbf{true})) \wedge$$
$$(E(P1!ack \vdash ack, \mathbf{false}) \Rightarrow HE(P2!ack, \mathbf{false}))$$

Tilsammen skulle disse punktene gi:

$$I \wedge \mathcal{M} \wedge \mathcal{G}^*_{P1}[\#!data = \#!ack \wedge HE(!ack, \mathbf{true})] \wedge e \Rightarrow I^{!data,\ P1!data,\ !ack,\ P1!ack}_{!data \vdash data, P1!data \vdash data, !ack \vdash ack, P1!ack \vdash ack}$$

slik at invarianten vedlikeholdes av **await**-setningen.

Neste del av beviset går ut på å vise at $data := data1$ i $P1$ vedlikeholder invarianten:

- Ved å bruke samme resonnement som for **await**-setningen får vi:

  $$P2!data = P1!data \vdash data1/like[1{:}\#P2!data]$$

- Anta A, L, $E(P1!ack, \mathbf{false}) \Rightarrow HE(P2!ack, \mathbf{false})$ og $E(P1!ack, \mathbf{false})$. Dette gir

  $$HE(P2!ack, \mathbf{false})$$

- I forrige punkt viste vi $HE(P2!ack, \mathbf{false})$, og vi kan anta $E(P1!ack, \mathbf{false})$. Da vet vi at sekvensen $!ack$ kun består av elementet **false** eller den slutter på $\langle \mathbf{false}, \mathbf{false} \rangle$. Fra A kan vi da slutte at $\#P1!ack = \#P2!ack + 1$. Fra den lokale betingelsen har vi at $\#P1!data = \#P1!ack$, og vi vet dermed at:

  $$\#P2!ack \geq (\#P1!data + 1) - 2$$

- Tidligere har vi vist $HE(P2!ack, \mathbf{false})$, og dermed har vi

  $$E(P2!ack, \mathbf{true}) \Rightarrow \#P1!data + 1 = \#P1!ack$$

Nå burde det være trivielt å vise:

$$I \wedge \mathcal{M} \wedge \mathcal{G}^{data}_{P1}[\#!data = \#!ack \wedge E(!ack, \mathbf{false})] \wedge e \Rightarrow I^{data\ !data,\ P1!data}_{data1\ !data \vdash data1, P1!data \vdash data1}$$

og vi er ferdig med den globale delen av beviset for denne tilordningssetningen.

At I vedlikeholdes over setningen $ack := \mathbf{true}$ viser vi slik:

- $ALT1(P1!ack)$ og $E(P1!ack, \mathbf{false})$ gir

  $$ALT1(P1!ack \vdash \mathbf{true})$$

- Anta A, L, $E(P1!ack, \mathbf{false}) \Rightarrow HE(P2!ack, \mathbf{false})$ og $E(P1!ack, \mathbf{false})$. Da har vi $HE(P2!ack, \mathbf{false})$, og $!ack$ består av bare elementet **false**, eller de to siste elementene i $!ack$ er **false**. Vi får dermed

  $$ALT(!ack \vdash \mathbf{true}).$$

- Fra den lokale forbetingelsen har vi $\#P1!data = \#P1!ack + 1$ og dermed

  $$\#P1!data \geq \#P1!ack + 1$$

Ved hjelp av disse punktene får vi

$$I \wedge \mathcal{M} \wedge \mathcal{G}^{ack}_{P1}[\#!data = \#!ack + 1 \wedge E(!ack, \mathbf{false})] \wedge e \Rightarrow I^{ack,\ !ack,\ \ P1!ack}_{\mathbf{true},!ack \vdash \mathbf{true},P1!ack \vdash \mathbf{true}}$$

og vi har dermed vist at invarianten vedlikeholdes.

Vi viser at **await**-setningen i $P2$ vedlikeholder invarianten slik:

- Anta A, L, $HE(P2!ack, \mathbf{false})$ og ack. Da må det være $P1$ som har tilordnet sist til ack, og denne prosessen har tilordnet **true**. Sekvensen $!ack$ må derfor slutte på $\langle \mathbf{false}, \mathbf{true} \rangle$, og $\#P1!ack = \#P2!ack + 2$. Sammen med $\#P2!data \cdot 2 = \#P2!ack$ fra den lokale betingelsen gir dette $(\#P2!data + 1) \cdot 2 = \#P1!ack$. Sammen med L gir dette $(\#P2!data + 1) \cdot 2 = \#P1!data$. Videre kan vi anta $HE(P1!data, data)$ og $P2!data = P1!data/like[1{:}\#P2!data]$, som sammen med informasjonen om sekvensenes lengder relativt til hverandre gir

  $$P2!data \vdash data = P1!data/like[1{:}\#P2!data + 1]$$

- Ved å se på de lokale sekvensenes lengder relativt til hverandre, slik vi gjorde i forrige punkt, innser vi fort at $\#P1!data = \#P1!ack$. I forrige punkt viste vi også at $P1$ sist tilordnet **true** til ack, og dermed har vi $E(P1!ack, \mathbf{true})$. Tilsammen får vi

  $$\#P1!data = \#P1!ack \wedge E(P1!ack, \mathbf{true}) \wedge$$
  $$(E(P1!ack, \mathbf{false}) \Rightarrow HE(P2!ack \vdash ack, \mathbf{false}))$$

- For å bevise $ALT(!ack \vdash ack)$, $ALT2(P2!ack \vdash ack)$, og $\#P1!ack \geq (\#P2!data + 1) \cdot 2$ bruker vi et symmetrisk argument som for å bevise det tilsvarende for **await** setningen i prosess $P1$.

Disse punktene gir

$$I \wedge \mathcal{M} \wedge \mathcal{G}^{*}_{P2}[\#!data \cdot 2 = \#!ack \wedge HE(!ack, \mathbf{false})] \wedge e \Rightarrow$$
$$I^{!data,\ \ P2!data,\ \ !ack,\ \ P2!ack}_{!data \vdash data, P2!data \vdash data, !ack \vdash ack, P2!ack \vdash ack}$$

og vi har vist at denne tilordningssetningen ikke ødelegger I.

For å bevise at I vedlikeholdes over $ack := \mathbf{false}$ i prosess $P2$ bruker vi et tilsvarende argument som for setningen $ack := \mathbf{true}$ i $P1$.

Dermed er vi ferdig med beviset for at den globale invarianten vedlikeholdes over hver setning som refererer til felles variable.

Å bevise at at de lokale betingelsene er sanne er nå rett frem, kanskje med unntak av betingelsen $data2 = rt(!data)$ i $P2$. Vi må vise at dette er sant etter tilordningssetningen $data2 := data$.

For å gjøre det tydelig hvilke sekvenser det refereres til i denne delen av beviset, setter vi prosessidentifikasjon foran de lokale sekvensene. Dette tilsvarer å se på sekvensene globalt, men det samme kan gjenskapes fra et lokalt synspunkt.

Fra A, L, $E(P2!ack, \mathbf{true}) \Rightarrow \#P1!data = \#P1!ack \wedge E(P1!ack, \mathbf{true})$ og $E(P2!ack, \mathbf{true})$ får vi at sekvensen $!ack$ må slutte med $\langle \mathbf{true}, \mathbf{true} \rangle$, og dermed $\#P1!ack = \#P2!ack + 1$.

Vi får også $\#P1!data = \#P1!ack$ og fra den lokale forbetingelsen har vi $\#P2!data \cdot 2 = \#P2!ack + 1$. Tilsammen får vi $\#P1!data = \#P2!data \cdot 2$. Sammen med $P2!data = P1!data/like[1{:}\#P2!data]$ og $rt(P1!data) = data$ fra den globale invarianten gir dette

$$data = rt(P2!data)$$

Merk at dette er bevist uten å anta noe om verdien av *data* i den lokale forbetingelsen, som en anvendelse av regelen for lesing av felles variable krever.

Vi har bevist at hvert element blir overført fra P1 til $P2$ via variabelen *data*. Første konjunkt i I sammen med den lokale betingelsen foran klammene i P2 uttrykker dette. I et fullstendig program må det i tillegg vises at elementet blir brukt i $P2$. Dette beviset vil være en del av det lokale beviset inne i klammene der elementet brukes.

3.5 Sunnhet

Delene av modellen som ble gitt som motivasjon for reglene, burde overbevise om at systemet er sunt. Reglene i systemet er dessuten så like de i System 2 at dette systemet lett kan brukes for å vise sunnhet. Fremgangsmåten for et slikt bevis vil være den samme som for kompletthetsbeviset som følger.

3.6 Relativ kompletthet

En vanlig metode for å bevise relativ kompletthet av et bevissystem er å ta utgangspunkt i en modell som i [Coo78]. En annen metode er å ta utgangspunkt i et annet bevissystem og deretter avlede reglene innenfor det systemet det skal vises kompletthet av. For eksempel bevises kompletthet av et bevissystem med Hoares ADAP-regel på denne måten i [Old83].

Beviset i dette avsnittet er inspirert av en bevismetode fra kompleksitetsteori [GJ79]. Det tilsvarer en del av et bevis for at et desisjonsproblem er NP-komplett. I et slikt bevis må det vises at det finnes en transformasjon fra desisjonsproblemet vi tar utgangspunkt i til desisjonsproblemet vi ser på. Denne transformasjonen må gjøre hver ja-instans av problemet som var utgangspunkt, til en ja-instans av problemet vi ser på. Jeg har i ettertid sett kompletthetsbevis gjort på noe av den samme måten som i denne oppgaven.

Metoden med å avlede regler som nevnt over kan ses på som et spesialtilfelle av metoden med å konstruere bevisinstanser. I spesialtilfellet vil transformasjonen gi identiske betingelser i de to bevisene, og dermed kan man avlede direkte fra reglene uten å forklare i detalj hvordan bevisene må se ut i de to systemene. Dermed vil bevisene bli enklere i spesialtilfellet, men denne metoden er sannsynligvis ikke kraftig nok i alle tilfeller, som f.eks. i dette avsnittet.

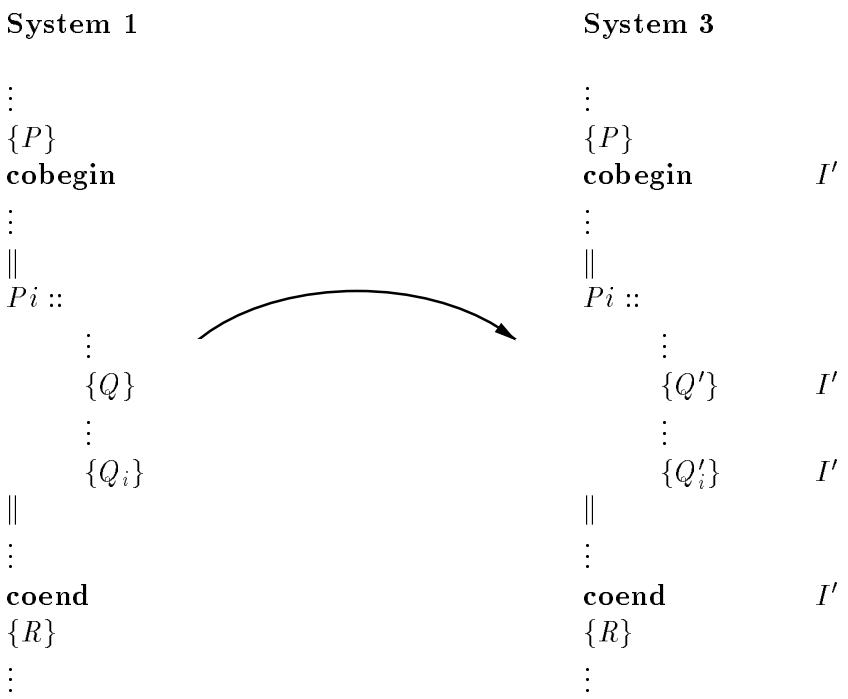
Teorem 1 *Bevissystemet med redusert mengde av mytiske variable* (System 3) *er relativt komplett.*

Bevis.

Innledning

Vi tar utgangspunkt i System 1, som er relativt komplett, og beviser at vi for ethvert bevis i dette systemet kan konstruere et bevis i System 3. Grunnen til at System 1, og ikke System 2, er valgt som utgangspunkt er at det er enklere å uttrykke betingelsene fra System 1. I System 2 har vi en global invariant hvor det kan ligge mye informasjon, og denne kan det være vanskelig å oversette til betingelser i System 3.

Vi lager en transformasjon fra et gitt bevis i System 1 til et konkret bevis i System 3. Siden vi ikke har de samme mytiske variable i de to systemene må betingelser med slike variable oversettes slik at de gir mening også i System 3. Dette gjelder betingelser inne i **cobegin**...**coend**-delen av programmet, og oversettelsesoperatoren er $'$. I tillegg finnes det en global invariant I', som skal være sann bestandig inne i hver enkelt prosess. Denne konstrueres også ut fra hvordan betingelsene i beviset i System 1 er. Skjematisk kan en transformasjon av et program dekorert med betingelser fremstilles slik:



De lokale betingelsene vil selvfølgelig inneholde mindre informasjon i beviset i System 3 enn i System 1. Imidlertid vil vi få den samme informasjonen for hele programmet, fordi vi har muligheten til å spesifisere mye i den globale invarianten. Legg merke til at P, betingelsen foran **cobegin**, og R, betingelsen etter **coend** gir samme informasjon i begge bevisene. Det skal dermed være mulig å bevise den samme spesifikasjonen av et program i System 3 som det ble gjort i System 1.

Vi starter med å gi definisjonen av operatoren som oversetter lokale betingelser:

$$Q' \triangleq \exists h_i | Q \bigwedge_j !r_j = h_i / !j$$

Q' er et uttrykk med det reduserte settet av mytiske variable, siden h_i skjules, og vi isteden får !-sekvenser. Disse representerer en del av den informasjonen vi hadde i h_i. Vi kan ikke

uttrykke alt om rekkefølge og hva som er lest i de lokale bevisene. Merk at $\bigwedge_j !r_j = h_i/!j$ delen av uttrykket bare inneholder sekvens-variable (h_i og !-sekvenser).

Oversettelse av de lokale betingelsene gjøres ikke utenfor **cobegin**...**coend**. Dersom vi likevel bruker oversettelsesoperatoren på en slik betingelse, vil det ikke ha noen effekt, fordi mytiske variable ikke omtales. For betingelsene i figuren over har vi dermed at $P' \equiv P$ og $R' \equiv R$.

Informasjon om hvilke flettinger som var riktige i det opprinnelige beviset legger vi i den globale invarianten:

$$I' \triangleq \exists h | \bigwedge_i (Q_i{}^{..r_j..}_{..final_j(h_i)..})^{h_i}_{local_i(h)} \bigwedge_j (!r_j \textbf{ head } h/!j \bigwedge_i i!r_j \textbf{ head } local_i(h)/!j)$$
$$\wedge \exists k | 0 \leq k \leq \#h \bigwedge_j (\bigwedge_i \#i!r_j = \#local_i(h[1{:}k])/!j)$$

der Q_i er bakbetingelsen til prosess i, j er indeks for variablene og k angir hvor stor del av h de nye mytiske variablene tilsvarer på et gitt sted i programmet. Den globale invarianten uttrykker at de globale sekvensene må kunne forlenges slik at de tilsvarer en h fra bakbetingelsen i beviset i System 1. Dette valget av global invariant gjør at det blir lett å avlede regelen for parallell sammensetning. Informasjon om hvilken rekkefølge de ulike prosessene kan ha skrevet til en bestemt variabel vil ligge her. I den siste delen angis det hvor lange sekvensene kan være i forhold til hverandre. Dette gir informasjon om rekkefølgen det er tilordnet til de ulike variablene, evt. om det er tilordnet til alle variablene samtidig (!!-element i h).

System 1 er relativt komplett. I lemma 3, 4, 5 og 6 viser vi at ethvert skritt der en av de fire reglene for parallelle programmer i beviset i System 1 resulterer i en dekorert programbit $\{P\}S\{Q\}$, kan vi, som en del av beviset vi konstruerer i System 3, vise *tilsvarende* skritt $(I')\{P'\}S\{Q'\}$. Anvendelse av andre regler i beviset vil ikke påvirke de mytiske variablene, så i disse tilfellene vil opplagt de lokale betingelser i de to systemene (P og P') gjennomgå samme forandringer. Dermed er også System 3 relativt komplett.

Delbevis

Først beviser vi et lemma om hvordan historiesekvensene er i det beviset vi tar utgangspunkt i (og dermed i det beviset vi konstruerer også). En sekvens som oppfyller en vilkårlig betingelse i en prosess, vil kunne forlenges slik at den er ekvivalent med en sekvens som oppfyller prosessens bakbetingelse. Merk at det ikke nødvendigvis må refereres eksplisitt til h_i i betingelsene. F.eks. vil alle historiesekvenser oppfylle betingelsen **true**.

Siden jeg tar utgangspunkt i et gitt bevis i System 1 og velger betingelser i beviset i System 3 utfra denne informasjonen, kan jeg bruke det jeg klarer å bevise om hvordan de mytiske variablene må være i det konkrete beviset jeg tok utgangspunkt i, i resten av kompletthetsbeviset.

I følgende lemma står begrepet "resten av programmet" for den delen av programmet som gjenstår å eksekvere. Begrepet vil ikke formaliseres her, men dette kan lett gjøres, f.eks. ved å se på sekvenser av programtilstander.

Lemma 2 *La Q være bakbetingelsen til en prosess og P en betingelse et tilfeldig sted i det lokale beviset for den samme prosessen. Da er*

$$\forall h_{i_P} \in \{h_i|P \wedge t_S\}|\exists h_{i_Q} \in \{h_i|Q^{..r_j..}_{..final_j(h_i)..}\}|h_{i_P} \textbf{ head } h_{i_Q}$$

der S betegner "resten av programmet", og t_S er termineringsvakt for S.

Bevis. Vi antar h_{i_P} **head** h_{i_Q}, og viser lemmaet ved induksjon på lengden av beviset av utsagnet $\{P\}S\{Q\}$.

Induksjonsbasis:

- Siden en sekvens er **head**-sekvens av seg selv har vi h_{i_Q} **head** h_{i_Q}. Vi har også $\{h_i|Q\} = \{h_i|Q^{..r_j..}_{..final_j(h_i)..}\}$ siden vi alltid har $r_j = final_j(h_i)$. Substitusjonen er tatt med for å unngå konflikt med andre referanser til felles variable tidligere i spesifikasjonen av prosessen.

Induksjonsskritt:

- En regel som ikke refererer til felles variable vil ikke endre h_i. Merk at Q ikke refererer til lokale variable fordi vi har en implisitt deklarasjon av disse variablene. Den siste setningen i prosessen blir dermed en skopterminator som gjør at de lokale variablene skjules med en kvantor.
- Tilordning: $\{P'\}r_j := e\{P\}$. Fra regelen får vi at P' er det samme som $P^{r_j,h_i}_{e,\ h_i \vdash (!j,e)}$. Spesifikasjonen av h_i i P kan ikke avhenge av r_j siden denne nettopp har blitt tilordnet en verdi, og h_i inneholder informasjon om det som har skjedd tidligere. Dermed vil ikke substitusjonen av e for r_j ha noe å si, og vi har $h_{i_{P'}} \vdash (!j, e) = h_{i_P}$. Sammen med h_{i_P} **head** h_{i_Q} fra induksjonshypotesen gir dette $h_{i_{P'}}$ **head** h_{i_Q}.
- For lesing og kritisk region får vi et tilsvarende argument som ved tilordning.
- Konsekvensregel: Anta $P' \Rightarrow P$ (premiss i regelanvendelsen). Dette gir $\{h_{i_{P'}}|P'\} \subseteq \{h_{i_P}|P\}$, og sammen med induksjonshypotesen gir dette at det finnes en h_{i_Q} slik at $h_{i_{P'}}$ **head** h_{i_Q}.

Dermed skulle induksjonsskrittet i beviset være gjennomført for reglene for alle programkonstruksjoner, unntatt dersom prosessen ikke terminerer, f.eks. fordi den aborterer eller går i evig løkke. Siden vi har et bevis for $\{P\}S\{Q\}$ kan dette forsterkes med en termineringsvakt, $\{P \wedge t_S\}S\{Q\}$. Betingelsen $P \wedge t_S$ i en gren av programmet som leder til ikke-terminering vil være ekvivalent med betingelsen **false**, og dermed er lemmaet trivielt sant. □

Lemma 3 *La*

$$\{Q^{r_j,h_i}_{e,\ h_i \vdash (!j,e)}\}r_j := e\{Q\}$$

være en anvendelse av aksiomskjemaet T1 *på en setning med tilordning til en felles variabel i beviset i* System 1. *Da gjelder tilsvarende Hoare-setning i* System 3:

T1′: $$(I')\{(Q^{r_j,h_i}_{e,\ h_i \vdash (!j,e)})'\}r_j := e\{Q'\}$$

Bevis. Ved å anvende T3, får vi T1′ dersom følgende premiss kan vises:

$$\vdash I' \wedge \mathcal{M} \wedge \mathcal{G}_i[(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})'] \Rightarrow (I' \wedge \mathcal{G}_i[Q'])^{r_j,i.r_j,!r_j,\quad i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e}$$

Dette gjøres i to deler. Først vises den lokale betingelsen, $(\mathcal{G}_i[Q'])^{r_j,i.r_j,!r_j,\quad i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e}$. Her er det unødvendig å anta I' og $\mathcal{M}$, så vi viser:

$$\mathcal{G}_i[(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})'] \vdash (\mathcal{G}_i[Q'])^{r_j,i.r_j,!r_j,\quad i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e}$$

Ved å ekspandere $\mathcal{G}_i$ og de lokale $'$-uttrykkene får vi:

$$\exists h_i|(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})^r_{i.r}\textstyle\bigwedge_{j'} i!r_{j'} = h_i/!j' \vdash (\exists h_i|Q^r_{i.r}\bigwedge_{j'} i!r_{j'} = h_i/!j')^{r_j,i.r_j,!r_j,\quad i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e}$$

Her er j i $\bigwedge_j$-delen av uttrykket byttet ut med j' for å unngå sammenblanding med indeksen til den felles variabel det tilordnes til.

Siden vi ikke har noen frie forekomster av h_i kan kvantoren på venstre side fjernes, og substitusjonene på høyre side utføres:

$$(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})^r_{i.r}\textstyle\bigwedge_{j'} i!r_{j'} = h_i/!j' \vdash \exists h_i|(Q^r_{i.r})^{i.r_j}_e\bigwedge_{j'\neq j} i!r_{j'} = h_i/!j' \wedge i!r_j\vdash e = h_i/!j$$

Vi velger $h_i\vdash(!j,e)$ for h_i på høyre side:

$$(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})^r_{i.r}\textstyle\bigwedge_{j'} i!r_{j'} = h_i/!j'$$
$$\vdash ((Q^r_{i.r})^{i.r_j}_e)^{h_i}_{h_i\vdash(!j,e)}\bigwedge_{j'\neq j} i!r_{j'} = h_i\vdash(!j,e)/!j' \wedge i!r_j\vdash e = h_i\vdash(!j,e)/!j$$

som er sant, fordi e ikke inneholder referanser til h_i eller r, slik at substitusjonene i Q gir samme resultat på begge sider, og fordi uttrykket $\bigwedge_{j'} i!r_{j'} = h_i/!j'$ er det samme som $\bigwedge_{j'\neq j} i!r_{j'} = h_i\vdash(!j,e)/!j' \wedge i!r_j\vdash e = h_i\vdash(!j,e)/!j$.

Dermed er vi ferdig med å vise den lokale betingelsen.

I neste del av beviset må vi vise at den globale invarianten vedlikeholdes:

$$\vdash I' \wedge \mathcal{M} \wedge \mathcal{G}_i[(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})'] \Rightarrow I'^{r_j,i.r_j,!r_j,\quad i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e} \tag{3.1}$$

Hvis vi velger samme h i $I'^{r_j,i.r_j,!r_j,\quad i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e}$ som i I', følger det meste av konklusjonen trivielt, unntatt følgende tre utsagn:

$$i!r_j\vdash e \textbf{ head } local_i(h)/!j \tag{3.2}$$

$$!r_j\vdash e \textbf{ head } h/!j \tag{3.3}$$

$$\exists k|\ldots\wedge \#i!r_j\vdash e = \#local_i(h[1{:}k])/!j \wedge \ldots \tag{3.4}$$

(3.2) viser vi ved først å vise

$$\mathcal{G}_i[(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})'] \vdash (\exists h_i|Q^{r_j}_e \wedge i!r_j\vdash e = h_i/!j)^r_{i.r} \tag{3.5}$$

som blir følgende uttrykk hvis venstre side skrives helt ut, og substitusjonen på høyre side utføres:

$$\exists h_i|(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})^r_{i.r}\bigwedge_{j'} i!r_{j'} = h_i/!j' \vdash \exists h_i|(Q^{r_j}_e)^r_{i.r} \wedge i!r_j \vdash e = h_i/!j$$

Ved å velge $h_i \vdash (!j, e)$ for h_i på høyre side blir dette:

$$(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})^r_{i.r}\bigwedge_{j'} i!r_{j'} = h_i/!j' \vdash ((Q^{r_j}_e)^r_{i.r})^{h_i}_{h_i\vdash(!j,e)} \wedge i!r_j \vdash e = h_i \vdash (!j, e)/!j$$

som opplagt må være sant.

Vi kan ikke bruke r_j i spesifikasjonen av h_i i System 1, fordi denne variabelen nettopp har fått tilordnet en ny verdi, mens h_i er avhengig av gamle verdier. Dermed har vi:

$$\vdash \{h_i|Q\} = \{h_i|Q^{r_j}_e\} \tag{3.6}$$

der Q er betingelsen etter denne tilordningssetningen i System 1.

Fra (3.1) kan vi anta $\mathcal{G}_i[(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})']$, og fra (3.5) får vi da:

$$(\exists h_i|Q^{r_j}_e \wedge i!r_j \vdash e = h_i/!j)^r_{i.r}$$

som ved å bruke (3.6) blir:

$$(\exists h_i|h_i \in \{h_i|Q\} \wedge i!r_j \vdash e = h_i/!j)^r_{i.r}$$

La Q_i være denne prosessens bakbetingelse, fortsatt i System 1. Siden det er bevist at denne betingelsen holder, kan vi forsterke den med t_S. Lemma 2 gir da at alle h_i som er slik at $Q \wedge t_S$ er sann, må være **head**-sekvens av en h_i som gjør Q_i sann. Dette må også gjelde dersom vi ser på en projeksjon av sekvensene, og uttrykket blir dermed:

$$(\exists h_i|h_i \in \{h_i|Q_{i\ldots final_{j'}(h_i)\ldots}^{\ldots r_{j'}\ldots}\} \wedge i!r_j \vdash e \textbf{ head } h_i/!j)^r_{i.r}$$

og siden substitusjonene av r ikke vil ha noen effekt og $h_i = local_i(h)$, har vi også:

$$\exists h|h \in \{h|(Q_{i\ldots final_{j'}(h_i)\ldots}^{\ldots r_{j'}\ldots})^{h_i}_{local_i(h)}\} \wedge i!r_j \vdash e \textbf{ head } local_i(h)/!j$$

og dermed har vi vist (3.2).

Fra $\mathcal{G}_i[(Q^{r_j,h_i}_{e,\ h_i\vdash(!j,e)})']$ har vi $\bigwedge_{j'} i!r_{j'} = h_i/!j'$ der h_i oppfyller forbetingelsen til setningen vi beviser. Fra I' har vi $0 \leq k \leq \#h \bigwedge_{j'}(\bigwedge_i \#i!r_{j'} = \#local_i(h[1{:}k])/!j')$, og fra $\mathcal{M}$ kan vi vite hvordan de lokale og globale sekvensene skal stemme overens med hverandre. Vi velger $k+\delta$ for k i $I'^{r_j,i.r_j,!r_j,\ \ i!r_j}_{e,\ e,\ \ !r_j\vdash e,i!r_j\vdash e}$. δ angir hvor mange elementer fremover i h elementet $(i!j, e)$ er. Elementene mellom k og $k+\delta$ vil være ?-elementer. Fra alle disse antagelsene og (3.2) får vi at det finnes en h slik at bakbetingelsen til alle prosessene i System 1, og dermed I', holder og:

$$h[k+\delta] = (i!j, e)$$

og fra denne, I' og $\mathcal{M}$ følger (3.3) og (3.4). Vi er ferdig med denne delen og dermed hele beviset for dette lemmaet. □

Lemma 4 *La*

$$\{\forall r_j | Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)}\} v := e_j \{Q\}$$

være en anvendelse av aksiomskjemaet L1 *på en setning med lesing av en felles variabel i beviset i* System 1. *Da gjelder tilsvarende Hoare-setning i* System 3:

L1′:
$$(I')\{(\forall r_j | Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)})'\} v := e_j \{Q'\}$$

Bevis. Ved å anvende L3, med Q' som bakbetingelse og I' som global invariant, får vi:

$$\vdash (I')\{\forall r_j | \mathcal{L}^j_i [I' \wedge \mathcal{M}] \Rightarrow (Q')^v_{e_j}\} v := e_j \{Q'\}$$

Hvis vi bruker CQL på denne, får vi L1′ ved å vise:

$$\vdash (\forall r_j | Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)})' \Rightarrow (\forall r_j | \mathcal{L}^j_i [I' \wedge \mathcal{M}] \Rightarrow (Q')^v_{e_j})$$

som ved å fjerne den unødvendige antagelsen, $\mathcal{L}^j_i [I' \wedge \mathcal{M}]$, og skrive $'$-delene helt ut blir:

$$\exists h_i | (\forall r_j | Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)}) \bigwedge_{j'} !r_{j'} = h_i / !j' \vdash \forall r_j | (\exists h_i | Q \bigwedge_{j'} !r_{j'} = h_i / !j')^v_{e_j}$$

Siden vi ikke har frie forekomster av h_i eller r_j kan $\exists$-kvantoren på venstre side og $\forall$-kvantoren på høyre side i uttrykket fjernes. Videre utføres substitusjonen på høyre side:

$$(\forall r_j | Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)}) \bigwedge_{j'} !r_{j'} = h_i / !j' \vdash \exists h_i | Q^v_{e_j} \bigwedge_{j'} !r_{j'} = h_i / !j'$$

Videre velger vi $h_i \vdash (?j, r_j)$ for h_i på høyre side, mens vi velger r_j som fri forekomst av r_j på venstre side:

$$Q^{v,\ h_i}_{e_j, h_i \vdash (?j, r_j)} \quad , \quad \bigwedge_{j'} !r_{j'} = h_i / !j' \vdash (Q^v_{e_j})^{h_i}_{h_i \vdash (?j, r_j)} \bigwedge_{j'} !r_{j'} = h_i \vdash (?j, r_j) / !j'$$

og dette uttrykket er trivielt siden $h_i / !j' = h_i \vdash (?j, r_j) / !j'$ og e_j ikke inneholder referanser til h_i. □

Lemma 5 *For enhver anvendelse av regelen* K1 *på en* **await***-setning i beviset i* System 1:

$$\frac{\{P\} S \{Q^{h_i}_{h_i \vdash (!!r)}\}}{\{\forall r | e \Rightarrow P^{h_i}_{h_i \vdash (??r)}\} \textbf{await } e \textbf{ do } S \textbf{ od} \{Q\}}$$

kan vi avlede følgende for den samme setningen i System 3:

K1′:
$$\frac{\{P'\} S \{(Q^{h_i}_{h_i \vdash (!!r)})'\}}{(I')\{(\forall r | e \Rightarrow P^{h_i}_{h_i \vdash (??r)})'\} \textbf{await } e \textbf{ do } S \textbf{ od} \{Q'\}}$$

Bevis. Vi viser først at premissen i anvendelsen av K1 gir premissen i K1′. Vi bruker aksiomskjemaet CONS og får følgende uttrykk, der betingelsene bare inneholder mytiske variable:

$$\{\bigwedge_j !r_j = h_i / !j\} S \{\bigwedge_j !r_j = h_i / !j\}$$

Ved å bruke reglene CJ og CQL på dette uttrykket og premissen i K1′ får vi:

$$\{P \bigwedge_j !r_j = h_i / !j\} S \{Q^{h_i}_{h_i \vdash (!!r)} \bigwedge_j !r_j = h_i / !j\}$$

og ved å bruke EXST og CQR får vi:

$$\{\exists h_i | P \bigwedge_j !r_j = h_i/!j\} S \{\exists h_i | Q^{h_i}_{h_i \vdash (!!r)} \bigwedge_j !r_j = h_i/!j\} \tag{3.7}$$

som er det samme som premissen i K1′. Inne i S vil det ikke skje noe med de mytiske variablene, så effekten på merkete og umerkete betingelser vil være den samme.

Videre viser vi K1′ ved å bruke følgende instans av regelen K3:

$$\frac{\{e \wedge I' \wedge \mathcal{M} \wedge \mathcal{G}_i^*[e \Rightarrow P']\} S \{(I' \wedge \mathcal{G}_i^*[Q'])^{!r, \; i!r}_{!r \vdash r, i!r \vdash r}\}}{(I')\{\forall r | e \Rightarrow P'\}\textbf{await } e \textbf{ do } S \textbf{ od}\{Q'\}} \tag{3.8}$$

For å få konklusjonen i K1′ bruker vi CQL og viser:

$$\vdash (\forall r | e \Rightarrow P^{h_i}_{h_i \vdash (??r)})' \Rightarrow \forall r | e \Rightarrow P'$$

Å ekspandere ′-delene gir følgende uttrykk:

$$\exists h_i | (\forall r | e \Rightarrow P^{h_i}_{h_i \vdash (??r)}) \bigwedge_j !r_j = h_i/!j \vdash \forall r | e \Rightarrow \exists h_i | P \bigwedge_j !r_j = h_i/!j$$

Siden vi ikke har frie forekomster av r eller h_i, blir dette:

$$(\forall r | e \Rightarrow P^{h_i}_{h_i \vdash (??r)}) \bigwedge_j !r_j = h_i/!j \vdash e \Rightarrow \exists h_i | P \bigwedge_j !r_j = h_i/!j$$

Ved å velge r_j på venstre side og $h_i \vdash (??r)$ på høyre side, får vi:

$$e \Rightarrow P^{h_i}_{h_i \vdash (??r)} \quad , \quad \bigwedge_j !r_j = h_i/!j \quad , \quad e \vdash P^{h_i}_{h_i \vdash (??r)} \bigwedge_j !r_j = h_i \vdash (??r)/!j$$

som burde være opplagt.

Vi antar premissen i K1′ og skal vise premissen i (3.8). I' refererer kun til mytiske variable, og ved å bruke aksiomskjemaet CONS får vi:

$$\{I'\} S \{I'\}$$

Vi anvender reglene CJ og CQL på dette uttrykket og (3.7), og får:

$$\{\mathcal{G}_i^*[P'] \wedge I'\} S \{\mathcal{G}_i^*[(Q^{h_i}_{h_i \vdash (!!r)})'] \wedge I'\} \tag{3.9}$$

Ved å bruke konsekvensreglene, må følgende to uttrykk vises for at vi skal ha premissen i (3.8):

$$e \wedge I' \wedge \mathcal{M} \wedge \mathcal{G}_i^*[e \Rightarrow P'] \Rightarrow \mathcal{G}_i^*[P'] \wedge I' \tag{3.10}$$

$$\mathcal{G}_i^*[(Q^{h_i}_{h_i \vdash (!!r)})'] \wedge I' \Rightarrow (I' \wedge \mathcal{G}_i^*[Q'])^{!r, \; i!r}_{!r \vdash r, i!r \vdash r}) \tag{3.11}$$

(3.10) skulle være rimelig opplagt, siden $\mathcal{G}_i^*$-operatoren ikke navner om felles variable og e ikke inneholder mytiske variable. Dermed er $\mathcal{G}_i^*[e \Rightarrow P']$ ekvivalent med $e \Rightarrow \mathcal{G}_i^*[P']$.

For (3.11) viser vi den lokale delen først. Da er det tilstrekkelig å anta $\mathcal{G}_i^*[(Q^{h_i}_{h_i \vdash (!!r)})']$, og vi må vise:

$$\vdash \mathcal{G}_i^*[(Q^{h_i}_{h_i \vdash (!!r)})'] \Rightarrow (\mathcal{G}_i^*[Q'])^{!r,\ i!r}_{!r \vdash r, i!r \vdash r}$$

Vi ekspanderer $\mathcal{G}_i^*$ og $'$-uttrykk og utfører substitusjonene på høyre side og får:

$$\exists h_i | Q^{h_i}_{h_i \vdash (!!r)} \bigwedge_j i!r_j = h_i/!j \vdash \exists h_i | Q \bigwedge_j i!r_j \vdash r_j = h_i/!j$$

Ved å velge $h_i \vdash (!!r)$ når kvantoren på høyre side i uttrykket fjernes, får vi:

$$Q^{h_i}_{h_i \vdash (!!r)} \bigwedge_j i!r_j = h_i/!j \vdash Q^{h_i}_{h_i \vdash (!!r)} \bigwedge_j i!r_j \vdash r_j = h_i \vdash (!!r)/!j$$

som skulle være opplagt.

Da gjenstår det å vise I'-delen:

$$\vdash \mathcal{G}_i^*[(Q^{h_i}_{h_i \vdash (!!r)})'] \wedge I' \Rightarrow I'^{!r,\ i!r}_{!r \vdash r, i!r \vdash r}$$

Det meste av $I'^{!r,\ i!r}_{!r \vdash r, i!r \vdash r}$ følger trivielt fra I', unntatt følgende tre utsagn:

$$\bigwedge_j i!r_j \vdash r_j \ \textbf{head}\ local_i(h)/!j \tag{3.12}$$

$$\bigwedge_j !r_j \vdash r_j \ \textbf{head}\ h/!j \tag{3.13}$$

$$\exists k | \ldots \bigwedge_j \#i!r_j \vdash r_j = \#local_i(h[1{:}k])/!j \ldots \tag{3.14}$$

Disse bevises på samme måte som de tilsvarende utsagnene i lemma 3:

(3.12) vises ved å anta $(\mathcal{G}_i^*[Q'])^{i!r}_{i!r \vdash r}$, dvs.:

$$\vdash \exists h_i | Q \bigwedge_j i!r_j \vdash r_j = h_i/!j$$

Q er bakbetingelsen til **await**-setningen i System 1. Denne kan vi forsterke med en termineringsvakt. La Q_i være bakbetingelsen til hele prosessen i System 1. Da har vi fra lemma 2 at for enhver h_i som gjør at Q er sann, finnes det en forlengelse av denne som gjør Q_i sann. Dette må gjelde selv om vi bare ser på en projeksjon av sekvensene, og dermed har vi:

$$\exists h_i | h_i \in \{h_i | Q_{i \ldots final_j(h_i) \ldots}^{\ldots r_j \ldots}\} \bigwedge_j i!r_j \vdash r_j \ \textbf{head}\ h_i/!j$$

og siden $h_i = local_i(h)$, har vi:

$$\exists h | h_i \in \{h_i | (Q_{i \ldots final_j(h_i) \ldots}^{\ldots r_j \ldots})^{h_i}_{local_i(h)}\} \bigwedge_j i!r_j \vdash r_j \ \textbf{head}\ local_i(h)/!j$$

og fordi denne h oppfyller I', har vi vist (3.12).

Resten av beviset ligner på det som gjøres for tilordningssetningen. Vi antar $\mathcal{M}$ for å få riktige flettinger. Videre antar vi $0 \leq k \leq \#h \bigwedge_j \#!r_j = \#h[1{:}k]/!j$ og resten av I'. Da finnes det h og k slik at vi i konklusjonen kan velge $k + \delta$ for k, der elementene mellom k og $k + \delta$ bare er ?-elementer, og vi har $h[k + \delta] = (!!r)$. Sammen med (3.12) gir dette (3.13) og (3.14). □

Lemma 6 *For enhver anvendelse av regelen* P1 *på en* **cobegin**...**coend**-*blokk i beviset i* System 1:

$$\frac{\{P_i\}S_i\{Q_i\}}{\{r=r0\bigwedge_i P_{i\,\varepsilon}^{h_i}\}\textbf{cobegin } S_1 \parallel \cdots \parallel S_n \textbf{ coend}\{\exists h|\bigwedge_i \overline{Q_i}\bigwedge_j r_j = final_j(h)\}}$$

kan vi avlede følgende i System 3:

$$\text{P1}': \quad \frac{(I')\{P_i'\}S_i\{Q_i'\} \qquad i=1,\ldots,n}{\{r=r0\bigwedge_i P_{i\,\varepsilon}^{h_i}\}\textbf{cobegin } S_1 \parallel \cdots \parallel S_n \textbf{ coend}\{\exists h|\bigwedge_i \overline{Q_i}\bigwedge_j r_j = final_j(h)\}}$$

$\overline{Q_i}$ *er* $(Q_{i\,..final_j(h_i)..}^{..r_j..})_{local_i(h)}^{h_i}$.

Bevis. Anta premissen i P1′. Ved å anvende P3 får vi:

$$\{I'^{!r,1!r,\ldots,n!r}_{\varepsilon,\,\varepsilon,\,\ldots,\varepsilon} \wedge r=r0\bigwedge_i P_{i\,\varepsilon}'^{!r}\}\textbf{cobegin } S_1 \parallel \cdots \parallel S_n \textbf{ coend}\{\exists !r, 1!r, .., n!r|I' \wedge \mathcal{M}\bigwedge_i \mathcal{G}_i[Q_i']\}$$

Vi får konklusjonen i P1′ ved å anvende konsekvensregler og vise følgende:

$$r=r0\bigwedge_i P_{i\,\varepsilon}^{h_i} \Rightarrow I'^{!r,1!r,\ldots,n!r}_{\varepsilon,\,\varepsilon,\,\ldots,\varepsilon} \wedge r=r0\bigwedge_i (P_i')_{\varepsilon}^{!r} \tag{3.15}$$

$$\exists !r, 1!r, .., n!r|I' \wedge \mathcal{M}\bigwedge_i \mathcal{G}_i[Q_i'] \Rightarrow \exists h|\bigwedge_i \overline{Q_i}\bigwedge_j r_j = final_j(h) \tag{3.16}$$

Først viser vi $(P_i')_{\varepsilon}^{!r}$-delen av (3.15). Det er nok å anta $P_{i\,\varepsilon}^{h_i}$. Skrevet helt ut blir dette:

$$P_{i\varepsilon}^{h_i} \vdash \exists h_i|P_i\bigwedge_j \varepsilon = h_i/!j$$

og ved å velge ε for h_i får vi følgende trivielt sanne uttrykk:

$$P_{i\varepsilon}^{h_i} \vdash P_{i\varepsilon}^{h_i}\bigwedge_j \varepsilon = \varepsilon/!j$$

For å vise $I'^{!r,1!r,\ldots,n!r}_{\varepsilon,\,\varepsilon,\,\ldots,\varepsilon}$, dvs.:

$$\begin{aligned}\vdash \exists h|\bigwedge_i (Q_{i\,..final_j(h_i)..}^{..r_j..})_{local_i(h)}^{h_i}\bigwedge_j (\varepsilon \textbf{ head } h/!j\bigwedge_i \varepsilon \textbf{ head } local_i(h)/!j)\\ \wedge\exists k|0\le k\le \#h\bigwedge_j \#\varepsilon = \#h[1:k]/!j\end{aligned}$$

antar vi at forbetingelsen i beviset i System 1 var forsterket med en termineringsvakt. Dermed vet vi at det eksisterer en h slik at bakbetingelsen $\bigwedge_i (Q_{i\,..final_j(h_i)..}^{..r_j..})_{local_i(h)}^{h_i}$ og dermed hele uttrykket er sant.

Dermed er beviset av (3.15) ferdig.

Som sagt ovenfor vet vi at det finnes en h slik at $\bigwedge_i (Q_{i\,..final_j(h_i)..}^{..r_j..})_{local_i(h)}^{h_i}$ er sann. Fra $\bigwedge_i \mathcal{G}_i[Q_i']$ og $h_i = local_i(h)$ får vi da:

$$\bigwedge_j (\bigwedge_i i!r_j = local_i(h)/!j)$$

og ved I' og $\mathcal{M}$ blir dette:

$$\bigwedge_j (!r_j = h/!j\bigwedge_i i!r_j = local_i(h)/!j)$$

og dermed

$$\bigwedge_j r_j = final_j(h)$$

og beviset av (3.16) er også ferdig. □

3.7 Diskusjon

Kompletthetsbeviset og produsent-konsument-eksemplet antyder at det kan bli mye som skal spesifiseres i den globale invarianten, for å kunne verifisere et program. Vi kunne derfor som nevnt på side 28, ønske å bruke ?-sekvenser også.

Ved å ta utgangspunkt i en modell slik vi har gjort for System 3 vil det være lett å lage et bevissystem med ?-sekvenser som er sunt og komplett. Det forutsettes selvfølgelig at modellen er riktig og at reglene utledes riktig. Et slikt bevissystem kan ses på som en utvidelse av System 3, og System 2 kan ses på som en utvidelse av dette igjen. I dette nye systemet vil vi kunne bevise minst like mye som i System 3. Dermed vil et slikt system være komplett, fordi System 3 er komplett. Samtidig vil vi ikke kunne vise mer enn det vi kan i System 2, som er sunt, og dermed har vi også sunnhet.

I en del tilfeller vil et bevis i System 3 ikke bli så hierarkisk oppbygget som ønskelig, siden en endring av en prosess vil kunne medføre at den globale invarianten må endres. Dermed risikerer vi å måtte gjøre store deler av beviset om igjen. Dersom vi har historiesekvenser som i System 1 og ikke har global invariant, vil en endring av en prosess medføre at denne prosessens historiesekvens endres. Beviset for parallell sammensetning må gjøres om igjen, og dette inkluderer å se på flettinger av alle historiesekvenser. Det er vanskelig å si hva som blir mest arbeidskrevende, men begge metoder vil nok kreve en del arbeid.

Ved å fjerne en del av den mytiske informasjonen kan vi få enklere regler. Vi får ihvertfall mindre mytisk informasjon å holde rede på. Det er teoretisk interessant å se hvor lite mytisk informasjon som er nødvendig for å ha et komplett bevissystem.

I de tilfelle hvor ingen felles variable endres i kritisk region, brukes det i et eksempel i [Owe92] en regel der !-sekvensene ikke endres. En slik regel vil gjøre spesifikasjonen enklere, og det kunne derfor være ønskelig med en slik regel i System 3. Uten ?-sekvenser kan det virke som om et bevissystem med en slik regel ikke vil være komplett, noe produsent-konsument-eksemplet illustrerer. Det synes som om vi trenger enten full oppdatering av de mytiske variablene, som i denne oppgaven, eller ?-sekvenser, som i eksemplet i [Owe92]. Ser vi på modellen virker dette rimelig. Å fjerne oppdateringen av !-sekvensene ved utgang av kritisk region vil medføre at det ikke tas vare på noe historieinformasjon i modellen av en slik programsetning.

I kompletthetsbeviset forsterket vi betingelser med termineringsvakt ved de anledninger det var nødvendig for å gjennomføre beviset. Derfor er vi ikke sikret at spesifikasjonen, dvs. lokale betingelser og den globale invarianten, av ikke-terminerende programmer vil kunne uttrykke så mye fornuftig. Det eneste kompletthetsbeviset garanterer er at vi kan bevise at bakbetingelsen er **false**.

Mange ganger ønsker vi å vise mer enn dette selv om programmet ikke alltid terminerer. F.eks. viser vi i produsent-konsument-eksemplet at data overføres riktig, og til dette bruker vi den globale invarianten. Ved å inkludere i den globale invarianten alle muligheter for hvordan de mytiske variablene kan komme til å se ut, ikke bare hvordan de vil se ut etter **coend**, kan vi vise sterkere spesifikasjoner for programmer som ikke alltid terminerer.

Kapittel 4

Vranglås

I dette kapitlet skal vi se om bevissystemet kan brukes til å bevise vranglås/fravær av vranglås i programmer. Vi bruker definisjonen av total vranglås fra side 7:

> Alle prosesser som ikke har terminert er i venting.

Dette innebærer at all aktivitet har opphørt. I denne oppgaven ser jeg ikke på partiell vranglås, selv om resonnementene kan utvides til å gjelde dette også.

De eneste stedene vi har noen form for venting er ved **await**-setninger. Det er altså i forbindelse med disse at det er fare for at vi får vranglås. Fravær av vranglås innebærer at i en tilstand der alle prosessene er foran en **await**-setning, må minst en prosess ha sann ventebetingelse.

4.1 Modellen for kritisk region

For å løse problemet med å bevise vranglåsfrihet, tar vi utgangspunkt i modellen for kritisk region. Modellen ser opprinnelig slik ut, som gitt i [Owe92] (modell 1):

$$|_j r_j := \textbf{some};\ \textbf{if}\ e\ \textbf{then}\ h_i := h_i \vdash (??r);\ S;\ h_i := h_i \vdash (!!r)\ \textbf{else abort fi}$$

Den skrives også slik (modell 2):

$$r := (\textbf{some}\ r|e);\ h_i := h_i \vdash (??r);\ S;\ h_i := h_i \vdash (!!r)$$

I begge modeller utføres hele programbiten som en atomisk operasjon. Regelen som hører til ser slik ut:

$$\frac{\{e \wedge I \wedge P \wedge R \wedge R_i\} S \{(H_i \Rightarrow Q \wedge I)^{h,\ h_i}_{h \vdash (i!!r),\ h_i \vdash (!!r)}\}}{(I)\{\forall r | P^{h_i}_{h_i \vdash (??r)}\}\textbf{await}\ e\ \textbf{do}\ S\ \textbf{od}\{Q\}}$$

Modell 1

I den første modellen er det **abort**-setningen som er ment å modellere vranglås. Vi får vranglås dersom programkontrollen kommer inn i **else**-grenen. Å bevise fravær av vranglås vil da tilsvare å bevise betingelsen **false** på begynnelsen av **else**-grenen. Vi må altså bevise at e *aldri* kan være **false** i det **if**-testen evalueres. Dette er imidlertid et for sterkt krav, siden det er tilstrekkelig å vise at én av prosessenes ventebetingelser er sann for å unngå vranglås.

Problemet med denne modellen i sammenheng med vranglås (eller synkronisering) er at $|_j r_j :=$ **some** kan utføres før e er sann, uten at dette influerer Hoare-analysen. Siden e ikke omtales i **some**-konstruksjonen, men først blir evaluert i **if**-setningen, er det vanskelig å modifisere forutsetningene slik at modellen vil være egnet til også å kunne omfatte venting.

Modell 2

I [Owe92] defineres det at **some**-konstruksjonen aborterer hvis e er **false**. Dette gir samme problemer som for modell 1. I dette tilfellet evalueres e i **some**-konstruksjonen, slik at her er det mulig å modifisere måten denne konstruksjonen fungerer på. Vi lar programkontrollen stoppe foran **some**-setningen og vente på sann ventebetingelse istedenfor å abortere. Dette gjøres på en slik måte at andre prosesser slipper til og kan endre felles variable og dermed verdien av e.

Modifikasjon av modell 2

I første omgang endrer vi modell 2 slik at venting uttrykkes eksplisitt i historiene. Dette gjør vi for å forenkle resonnementet for hvordan vi finner et uttrykk for vranglåsfrihet. Vi legger setningen $h_i := h_i \vdash await$ først. Dette tilsvarer å melde seg på i en ventekø hvor prosessen blir liggende til e er sann:

$$h_i := h_i \vdash await;\ r := (\textbf{some}\ r|e);\ h_i := h_i \vdash (??r);\ S;\ h_i := h_i \vdash (!!r)$$

await-elementene er med i både lokale og globale historiesekvenser, fordi vi ser på dette som to atomiske operasjoner:

1. At prosessen melder seg på i ventekø, $h_i := h_i \vdash await$
2. Resten av uttrykket. Dette var også en atomisk operasjon i den opprinnelige modellen, under forutsetning av at e var sann før utførelsen.

Det må bevises at den globale invarianten, I, holder over disse to operasjonene, og vi får følgende regel:

$$\frac{\{e \wedge I \wedge P \wedge R \wedge R_i\} S \{(H_i \Rightarrow Q \wedge I)^{h,\ h_i}_{h \vdash (i!!r),\ h_i \vdash (!!r)}\}\ ,\ I \wedge G^j_i[P^{h_i}_{h_i \vdash await \vdash (??r)}] \Rightarrow I^h_{h \vdash await}}{(I)\{\forall r | P^{h_i}_{h_i \vdash await \vdash (??r)}\}\textbf{await}\ e\ \textbf{do}\ S\ \textbf{od}\{Q\}}$$

Vi skal nå se hvordan vi kan uttrykke fravær av vranglås. Hvis vi får vranglås, vet vi at alle de lokale historiesekvensene har *await* som siste element og ingen av ventebetingelsene er sanne. For å vise fravær av vranglås er det derfor tilstrekkelig å vise at én ventebetingelse, e_i, er sann. Indeksen i angir hvilken prosess uttrykket er fra, og dette uttrykket globaliseres ikke. Vi kan også anta den lokale forbetingelsen til **await**-setningen, fordi det vil ikke bli vranglås hvis vi ikke kan nå denne tilstanden. Her brukes b_i som forkortelse for $G_i[P]$, den lokale betingelsen som hører sammen med e_i, sett med globale øyne. Videre kan vi anta $rt(local_i(h)) = await$, siden vi vet at det siste elementet før en vranglås vil være en *await*. Vi kan som vanlig anta den globale invarianten.

I første omgang ser vi bare på programmer der hver prosess har én **await**-setning. Å bevise fravær av vranglås i et slikt program tilsvarer å vise at den globale invarianten impliserer følgende uttrykk:

$$(\bigwedge_i rt(local_i(h)) = await) \Rightarrow (\bigwedge_i b_i \Rightarrow \bigvee_i e_i)$$

Vi kommer tilbake til hvordan programmer med flere **await**-setninger i en prosess skal bevises.

Forenkling av uttrykket for vranglåsfrihet

I uttrykket for vranglåsfrihet som vi har satt opp, er antagelsen $\bigwedge_i rt(local_i(h)) = await$ unødvendig, fordi det alltid vil være mulig å ha den samme informasjonen i de lokale betingelsene. Dermed vil et bevis for fravær av vranglås være å bevise at den globale invarianten impliserer:

$$\bigwedge_i b_i \Rightarrow \bigvee_i e_i$$

Siden vi nå kan ha all informasjon i de lokale betingelsene, trenger vi strengt tatt ikke *await*-elementet, og vi sløyfer derfor all spesifikasjon av det.

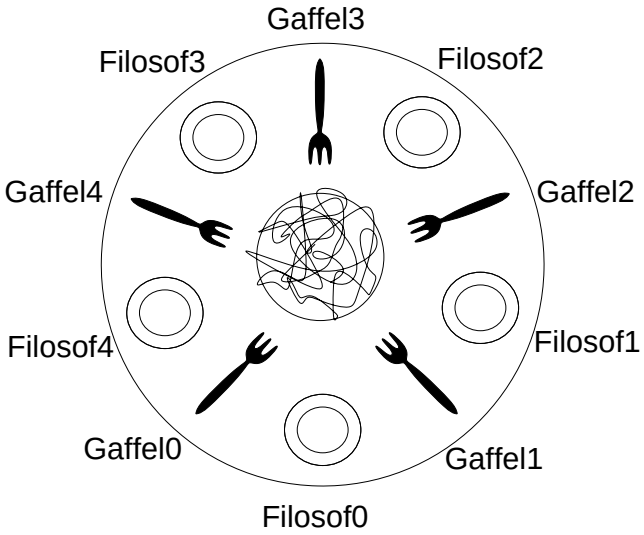
Hvis vi ser på modellen, finner vi ut at dette virker rimelig. Dersom det oppstår en vranglås, stopper programkontrollen foran setningen $r := (\mathbf{some}\, r|e)$. Å bevise at vi ikke får vranglås blir derfor ekvivalent med å bevise at det er mulig å oppfylle en e, slik at programkontrollen fortsetter over denne setningen.

Det blir altså ingen avleiring av *await*-elementer i historiesekvensene. Regelen for kritisk region blir den samme som i det opprinnelige systemet, siden ingen utvidelse av dette systemet er nødvendig. Spesifikasjonen av et program kan bli enklere dersom vi fortsatt har *await*-elementet, men regelen blir mer komplisert.

4.2 Eksempel: Dining Philosophers

Som et eksempel på hvordan vi bruker denne metoden for å bevise fravær av vranglås, kan vi bruke Dijkstras klassiske eksempel, Dining Philosophers [Dij72].

Fem filosofer sitter rundt et stort bord med en bolle spaghetti midt på bordet, og de veksler mellom å tenke og spise. Det er en tallerken til hver, og mellom hver tallerken ligger en gaffel, se figur.



For å kunne spise må en filosof ha gaflene på begge sider av seg. Disse gaflene deles med naboene, så de kan være opptatt. Et eksempel er $filosof_1$, som bruker $gaffel_1$ og $gaffel_2$. $gaffel_1$ deler han med $filosof_0$ og $gaffel_2$ med $filosof_2$.

4.2.1 Med vranglås

Vi kan tenke oss at en filosof først tar opp venstre gaffel og så høyre når han ønsker å spise. Når han er ferdig med å spise legger han begge gaflene ned igjen, slik at naboene får spist også.

La hver filosof representere en prosess og hver gaffel en ressurs som skal deles. Vi lar en gaffel være en boolsk variabel som er **true** når den er ledig og **false** når den er opptatt. Dette gir følgende system:

$gaffel_0 := gaffel_1 := gaffel_2 := gaffel_3 := gaffel_4 :=$ **true**;
cobegin $filosof_0 \parallel \cdots \parallel filosof_4$ **coend**

En filosof-prosess blir da:

$filosof_i$ :: **loop**
await $gaffel_i$ **do** $gaffel_i :=$ **false od**
await $gaffel_{i+_5 1}$ **do** $gaffel_{i+_5 1} :=$ **false od**
$\langle spis \rangle$;
$gaffel_i :=$ **true**
$gaffel_{i+_5 1} :=$ **true**
$\langle tenk \rangle$;
repeat

$+_5$ betyr pluss modulo 5.

Denne måten å dele gaflene på innebærer en risiko for at vi kan få vranglås. Sett at alle filosofene bestemmer seg for å spise samtidig og plukker opp sin venstre gaffel. Da vil alle sitte med en gaffel og vente på at den andre skal bli ledig. Den gaffelen er imidlertid plukket opp av filosofen til høyre, som igjen venter på en gaffel. Alle prosessene venter på at en annen prosess skal frigi en ressurs, og vi har fått en vranglås. Et bevis for dette blir gitt i avsnitt 4.3.2

4.2.2 Uten vranglås

En måte å sikre at vi unngår vranglås [Dij72] er å la den filosofen som skal spise, vente på at begge gaflene er tilgjengelige og så plukke opp begge gaflene i en operasjon.

En filosof-prosess som bruker denne strategien, ser slik ut:

$filosof_i$ :: **loop**
$\{h_i \in SEKV_i\}$
await $gaffel_i \wedge gaffel_{i+_5 1}$ **do** $gaffel_i :=$ **false**; $gaffel_{i+_5 1} :=$ **false od**
$\langle spis \rangle$;
$gaffel_i :=$ **true**
$gaffel_{i+_5 1} :=$ **true**
$\langle tenk \rangle$;
repeat

$SEKV_i$ betyr mengden av sekvenser som oppfyller følgende regulære uttrykk

$$\langle (??\ldots, \mathbf{true}, \mathbf{true}, \ldots), (!!\ldots, \mathbf{false}, \mathbf{false}, \ldots), (!gaffel_i, \mathbf{true}), (!gaffel_{i+_5 1}, \mathbf{true}) \rangle *$$

der $\langle e \rangle *$ betyr gjentakelse av e null eller flere ganger.

Elementet $(??\ldots, \mathbf{true}, \mathbf{true}, \ldots)$ betyr at vi leser **true** fra variablene $gaffel_i$ og $gaffel_{i+_5 1}$, og $(!!\ldots, \mathbf{false}, \mathbf{false}, \ldots)$ står for tilordning av **false** til de samme variablene.

Den lokale betingelsen foran **await**-setningene i hver prosess er $h_i = SEKV_i$ og tilhørende ventebetingelse er $gaffel_i \wedge gaffel_{i+_5 1}$. Dette gir følgende uttrykk for fravær av vranglås:

$$\bigwedge_i G_i[h_i \in SEKV_i] \Rightarrow \bigvee_i gaffel_i \wedge gaffel_{i+_5 1}$$

Vi lar den globale invarianten være **true**. Å bevise at den lokale invarianten i filosof-prosessene vedlikeholdes og holder initielt, er rett frem. Til slutt må vi vise uttrykket for fravær av vranglås. Vi ser på hvordan de lokale historiesekvensene kan flettes sammen. Alle prosessene har sist tilordnet **true** til de gaffel-variablene de bruker, og dermed er alle ventebetingelsene sanne.

4.3 Prosesser med flere await-setninger

Når vi har flere **await**-setninger får vi også flere ventebetingelser. Det er ikke sterkt nok å bevise at en av disse er sann, siden det kan være ved en annen **await**-setning vi venter.

På den annen side blir det for sterkt å kreve at alle ventebetingelser skal være sanne hver gang vi kommer til en **await**-setning.

Det er nødvendig å skille mellom **await**-setningene for å vite hvilken ventebetingelse som må være sann. En måte å gjøre dette på er å se på de lokale betingelsene ved hvert **await**-sted. Vi bruker de globale utgavene av betingelsene.

Vi må sjekke alle tilstander der prosessene venter. Dette gir et eksponensielt antall tilstander å undersøke. Antall kombinasjoner er:

$$k_1 \cdot k_2 \cdot \cdots \cdot k_n$$

der k_1 er antall **await**-setninger i prosess 1, k_2 i prosess 2, osv, og det er n prosesser. Dersom det er like mange **await**-setninger i hver prosess kan dette uttrykkes som k^n der k er antall **await**-setninger og n er antall prosesser.

En tilstand hvor alle prosesser venter gir opphav til en konjunkt i uttrykket som vi må vise at den globale invarianten impliserer. Det blir dermed k^n konjunkter i uttrykket. En slik konjunkt ser ut som:

$$\bigwedge_i b_{ij_i} \Rightarrow \bigvee_i e_{ij_i}$$

Indeksen i står som vanlig for prosess, og j_i står for hvilken **await**-setning det er i prosessen. Denne indeksen bestemmes av hvilken av tilstandene vi undersøker i denne konjunkten. e_{ij_i} er ventebetingelsen i $j_i$te **await**-setning i prosess i, og b_{ij_i} er forbetingelsen til samme **await**-setning globalisert.

Hvis vi tar med de lokale historiesekvensene i de lokale betingelsene, og uttrykker i den globale invarianten hva de lokale sekvensene skal være **head**-sekvenser av, slik vi gjorde i forrige eksempel, blir bevisene lette å gjennomføre. I praksis blir det store problemet å sette opp uttrykket for fravær av vranglås, siden dette uttrykket blir så stort.

4.3.1 Eksempel: Alternativ måte å programmere Dining Philosophers

Vi bruker nok en gang Dining Philosophers som eksempel. Denne løsningen er foreslått av Hoare [Hoa72]. Den har to **await**-setninger pr. prosess og er uten vranglås.

Filosof-prosessene 0–3 tar opp først venstre og så høyre gaffel, mens prosess 4 gjør det i motsatt rekkefølge. Prosess 0–3 ser da slik ut:

filosof$_i$ :: **loop**
  $\{h_i \in SEKV_i\}$
  await *gaffel*$_i$ **do** *gaffel*$_i$:= **false od**
  await *gaffel*$_{i+1}$ **do** *gaffel*$_{i+1}$:= **false od**
  $\langle spis \rangle$;
  gaffel$_i$:= **true**
  gaffel$_{i+1}$:= **true**
  $\langle tenk \rangle$;
repeat

der $SEKV_i$ betyr mengden av sekvenser som oppfyller det regulære uttrykket

$$\langle(??\ldots,\mathbf{true},\ldots),(!!\ldots,\mathbf{false},\ldots),(??\ldots,\mathbf{true},\ldots),(!!\ldots,\mathbf{false},\ldots),$$
$$(!gaffel_i,\mathbf{true}),(!gaffel_{i+1},\mathbf{true})\rangle*$$

I de to første elementene vil **true** og **false** tilsvare verdien av $gaffel_i$ og i de to neste variabelen $gaffel_{i+1}$. Som vanlig brukes $*$ for gjentakelse. Vi bruker også $SEKV_i'$ som betyr mengden av sekvenser som oppfyller

$$\langle(??\ldots,\mathbf{true},\ldots),(!!\ldots,\mathbf{false},\ldots),(??\ldots,\mathbf{true},\ldots),(!!\ldots,\mathbf{false},\ldots),$$
$$(!gaffel_i,\mathbf{true}),(!gaffel_{i+1},\mathbf{true})\rangle*(??\ldots,\mathbf{true},\ldots),(!!\ldots,\mathbf{false},\ldots)$$

En sekvens fra denne mengden vil være lik en sekvens fra mengden $SEKV_i$ med elementene $(??\ldots,\mathbf{true},\ldots),(!!\ldots,\mathbf{false},\ldots)$ lagt til. Verdiene **true** og **false** tilhører variabelen $gaffel_i$.

Prosess 4 blir slik:

$filosof_4$ :: **loop**
$\{h_4 \in SEKV_4\}$
await $gaffel_0$ **do** $gaffel_0 :=$ **false od**
await $gaffel_4$ **do** $gaffel_4 :=$ **false od**
$\langle spis\rangle$;
$gaffel_0 :=$ **true**
$gaffel_4 :=$ **true**
$\langle tenk\rangle$;
repeat

$SEKV_4$ betyr mengden av sekvenser som oppfyller

$$\langle(??\mathbf{true},\ldots),(!!\mathbf{false},\ldots),(??\ldots,\mathbf{true}),(!!\ldots,\mathbf{false}),(!gaffel_0,\mathbf{true}),(!gaffel_4,\mathbf{true})\rangle*$$

Her vil **true** og **false** i de to første elementene tilsvare $gaffel_0$ og i de to neste variabelen $gaffel_4$. $SEKV_4'$ betyr (som for de andre prosessene) mengden av sekvenser som oppfyller

$$\langle(??\mathbf{true},\ldots),(!!\mathbf{false},\ldots),(??\ldots,\mathbf{true}),(!!\ldots,\mathbf{false}),(!gaffel_0,\mathbf{true}),(!gaffel_4,\mathbf{true})\rangle*$$
$$(??\mathbf{true},\ldots),(!!\mathbf{false},\ldots)$$

Vi lar den globale invarianten være **true**. Å bevise lokale betingelser for disse prosessene er trivielt. Til slutt må vi sette opp et uttrykk for vranglåsfrihet og bevise at den globale invarianten impliserer dette.

Siden det er to **await**-setninger i hver prosess, og det er fem prosesser, får vi $2^5 = 32$ mulige tilstander der alle prosessene venter. Dermed får vi en konjunksjon med 32 ledd, som vi må vise.

Som et eksempel på hvordan konjunktene vil se ut, kan vi ta det leddet som tilsvarer at $filosof_0$ er ved andre **await**-setning og resten av prosessene ved første:

$$G_0[h_0 \in SEKV'_0] \wedge G_1[h_1 \in SEKV_1] \wedge G_2[h_2 \in SEKV_2] \wedge G_3[h_3 \in SEKV_3] \wedge$$
$$G_4[h_4 \in SEKV_4] \Rightarrow gaffel_1 \vee gaffel_1 \vee gaffel_2 \vee gaffel_3 \vee gaffel_0$$

Ved å se på informasjonen i premissen i dette uttrykket, ser vi at alle *gaffel*-variablene unntatt $gaffel_0$ må være sanne og dermed er uttrykket sant.

Vi kaller dette leddet for $KOMB$[2,1,1,1,1]. Uttrykket for vranglåsfrihet med de 32 konjunktene kan nå uttrykkes som:

$$\forall p_1, p_2, p_3, p_4, p_5 | KOMB[p_1, p_2, p_3, p_4, p_5]$$

der hver p_i tar verdiene 1 eller 2.

Vi ser på hvordan vi beviser et ledd til. Konjunkten $KOMB$[2,2,1,2,2] kan skrives helt ut som:

$$G_0[h_0 \in SEKV'_0] \wedge G_1[h_1 \in SEKV'_1] \wedge G_2[h_2 \in SEKV_2] \wedge G_3[h_3 \in SEKV'_3] \wedge$$
$$G_4[h_4 \in SEKV'_4] \Rightarrow gaffel_1 \vee gaffel_2 \vee gaffel_2 \vee gaffel_4 \vee gaffel_4$$

Dette tilsvarer at $filosof_2$ er ved sin første **await**-setning, mens resten er ved sin andre. Premissen i uttrykket kan ikke være sann, fordi h_0 og h_4 kan ikke begge lese **true** fra $gaffel_0$ og skrive **false**. Prosessene vil aldri komme i denne tilstanden. Resten av leddene vises på samme måte.

4.3.2 Eksempel: Bevis av Dining Philosophers med vranglås

Vi skal se at vi ikke kan bevise vranglåsfrihet for den første utgaven av Dining Philosophers. En filosof-prosess ser ut som:

$filosof_i$ :: **loop**
$\{h_i \in SEK_i\}$
await $gaffel_i$ **do** $gaffel_i$:= **false od**
await $gaffel_{i+_5 1}$ **do** $gaffel_{i+_5 1}$:= **false od**
$\langle spis \rangle$;
$gaffel_i$:= **true**
$gaffel_{i+_5 1}$:= **true**
$\langle tenk \rangle$;
repeat

der SEK_i betyr mengden av sekvenser som oppfyller det regulære uttrykket

$$\langle (??\ldots, \mathbf{true}, \ldots), (!!\ldots, \mathbf{false}, \ldots), (??\ldots, \mathbf{true}, \ldots), (!!\ldots, \mathbf{false}, \ldots),$$
$$(!gaffel_i, \mathbf{true}), (!gaffel_{i+_5 1}, \mathbf{true}) \rangle *$$

der **true** og **false** i de to første elementene tilsvarer verdiene for $gaffel_i$ og i de to neste for $gaffel_{i+_5 1}$. Videre betyr SEK'_i mengden av sekvenser som oppfyller

$$\langle (??\ldots, \mathbf{true}, \ldots), (!!\ldots, \mathbf{false}, \ldots), (??\ldots, \mathbf{true}, \ldots), (!!\ldots, \mathbf{false}, \ldots),$$
$$(!\mathit{gaffel}_i, \mathbf{true}), (!\mathit{gaffel}_{i+_5 1}, \mathbf{true})\rangle * (??\ldots, \mathbf{true}, \ldots), (!!\ldots, \mathbf{false}, \ldots)$$

La $KOM[j_1, j_2, j_3, j_4, j_5]$ være predikatet som sier at prosessene er ved **await**-setning j_1, j_2,..., dvs.:

$$\bigwedge_i b_{j_i} \Rightarrow \bigvee_i e_{j_i}$$

Uttrykket for vranglåsfrihet blir

$$\forall p_1, p_2, p_3, p_4, p_5 | KOM[p_1, p_2, p_3, p_4, p_5]$$

De fleste av KOM-leddene vil kunne bevises, men $KOM[2,2,2,2,2]$ vil gi problemer. Dette leddet tilsvarer at alle prosessene venter foran sin andre **await**-setning. Hvis vi ser på programteksten skjønner vi fort at det må bli vranglås her fordi alle *gaffel*-variablene er **false**, og alle prosessene venter på at en av disse variablene skal bli **true**. Formelt ser dette leddet ut som

$$\bigwedge_i G_i[h_i \in SEK_i'] \Rightarrow \bigvee_i \mathit{gaffel}_i$$

Ved å se på hvordan historiesekvensene i premissen kan flettes sammen, ser vi at premissen impliserer at alle *gaffel*-variablene er **false**. Dermed kan ikke konklusjonen i uttrykket være sann. Videre er det ikke noe i premissen som vil gjøre denne **false**. Vi vet dermed at her er det en mulighet for at vi kan få vranglås, og vi har bevist det vi sa mer uformelt tidligere i avsnittet.

4.4 Oppsummering

Ved å legge inn et *await*-element i historiesekvensene vises det eksplisitt hvordan venting kan modelleres. Dette kan gi enklere betingelser, og ikke minst bidrar det til å klargjøre hvordan vi kan komme frem til et uttrykk for vranglåsfrihet. Imidlertid blir regelen for kritisk region mer komplisert, og det blir mer mytisk informasjon å holde rede på, så derfor tar vi ikke med *await*-elementet likevel.

Beviset for et program der prosessene har flere **await**-setninger blir ganske komplisert hvis vi skal gjøre det helt detaljert, men prinsippet er enkelt.

Enklere systemer har ofte for sterke krav slik at det finnes vranglåsfrie programmer som ikke oppfyller disse kravene. Problemet er så komplekst at det nok ikke finnes noen enklere måte å utføre slike bevis.

Det som er gjort i dette kapitlet ligner på måten dette er behandlet i noen bevissystemer for CSP [AFdR80, LG81] men er gjort uavhengig av disse arbeidene.

Kapittel 5

Konklusjon

Utgangspunktet for denne oppgaven var at jeg skulle vise relativ kompletthet av et bevissystem uten α-sekvenser[1], som foreslått i [Owe92]. I oppgaven har jeg kommet frem til et sterkere resultat, relativ kompletthet av et bevissystem der sekvenser av leste verdier og α-sekvenser er fjernet. Dette bevissystemet har jeg motivert utfra en skisse av en modell med tilordningssekvenser som eneste mytiske variable. I tillegg til beviset for relativ kompletthet har jeg gitt et uformelt argument for at det er sunt.

Jeg har også brukt bevissystemet på et par eksempler. Det viser seg å være tungt å benytte til bevisføring i praksis, ihvertfall for den klassen av programmer der prosessene synkroniseres som i produsent-konsument-eksemplet.

Imidlertid blir reglene enklere enn i System 2. Det er også teoretisk interessant å se hvor lite mytisk informasjon vi kan klare oss med. Dette kan hjelpe på vår forståelse av hvordan prosesser samhandler og hvordan dette kan bevises.

Det er også interessant å se om det er en fordel med en global invariant eller ikke. Fordelen er, som vi har sett i denne oppgaven, at vi får svært stor uttrykkskraft. Ulempen er at det må vises at den globale invarianten vedlikeholdes i hele programmet, og bevissystemet blir noe mer komplisert. Dersom man er av den oppfatning at en global invariant er en uting kan man i System 2 sette den til **true**, og kompleksiteten vil bli som i System 1, som er uten global invariant.

En som skal lære å bevise parallelle programmer, vil nok synes at et bevissystem som ligner på System 1, vil være enklest å forstå og anvende. Etterhvert som man får litt mer erfaring med beviser kan det være en fordel å ha større spillerom i utformingen av spesifikasjonen og beviset, med flere valgmuligheter som vil føre frem. Da vil System 2 være mere egnet, kanskje med den modifikasjonen at historiesekvensene er oppsplittet (men uten at noe er fjernet).

Som en oppsummering kan vi si at det (foreløpig) ikke finnes et "beste" bevissystem. Vi trenger forskjellige formalismer/bevissystemer avhengig av hva de skal brukes til og hvem som skal bruke dem.

Videre i oppgaven har jeg også sett på hvordan fravær av vranglås kan bevises i System 2. Dette kan bli svært komplisert i praksis fordi vi kan risikere å måtte sjekke et eksponensielt

[1] Definisjonen av α-sekvenser finnes på side 14.

antall globale tilstander. Jeg tror at mye av grunnen til dette er den iboende kompleksiteten i problemet med vranglåssjekking, selv om noe forenkling bør være mulig i forhold til det jeg har gjort.

Det som er gjort i kapittel 4 om vranglås har mye til felles med [AFdR80] og [LG81]. Fordelene med det som er gjort her og i [AFdR80] er at vi kan benytte en global invariant for å beskrive mulige globale tilstander, mens det har vi ikke muligheten til i [LG81].

Siden det er begrenset tid til rådighet for å gjøre en hovedoppgave, er det naturlig å runde av her. Det er imidlertid noen problemstillinger det kunne være interessant å se mer på.

Jeg tror det kan være av interesse å se om det finnes et relativt komplett bevissystem der den mytiske informasjonen er ytterligere redusert i forhold til det jeg har sett på. F.eks. kan det tenkes at man kan klare seg med tidligere verdier av variable og ikke behøve rekkefølgen av tilordninger. Dette vil ligne på det som er gjort i [Jon83] og [Stø91]. Hvis det er umulig å redusere den mytiske informasjonen, er det interessant å se om dette kan vises formelt.

I [Owe92] er det nevnt at hans system kan utvides til å håndtere nestet parallellitet etter mønster av [Mel86a]. Det kan være interessant å se om et system med redusert mengde av mytiske variable også kan utvides til å håndtere nestet parallellitet.

Videre kunne jeg tenke meg å se på hvor mye som kreves for å kunne resonnere om total korrekthet og tid.

Det er også av interesse å se om resonnementer om fravær av vranglås kan forenkles. Her kunne jeg tenke meg å se om det er beskrevet en egnet metode i litteraturen, evt. om en ny metode kan utvikles.

Hittil har jeg nevnt noen retninger for videre arbeid med vanlige felles variable. Det kan også være interessant å se på andre typer variable, f.eks. monitor-variable, i sammenheng med historiesekvenser.

Jeg håper denne oppgaven er et bidrag til å øke forståelsen av parallelle prosesser og resonnementer om disse. Ihvertfall har den bidratt til øke min egen forståelse av emnet.

Referanser

[AFdR80] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey — part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[Apt83] K.R. Apt. Formal justification of a proof system for Communicating Sequential Programs. *Journal of the ACM*, 30(1):197–216, 1983.

[Ash75] E.A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, 1975.

[Bri73] P. Brinch Hansen. Concurrent programming concepts. *Computing Surveys*, 5(4):223–245, 1973.

[Coo78] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[Dæh87] J.H. Dæhlin. Semantikk om invarians og progresjon. Hovedoppgave, Institutt for informatikk, Universitetet i Oslo, 1987.

[Dah92] O.-J. Dahl. *Verifiable Programming*. Prentice-Hall, 1992.

[Dah93] O.-J. Dahl. Parallell programmering. Kompendium, Institutt for informatikk, Universitetet i Oslo, 1993.

[Dij72] E.W. Dijkstra. Hierarchical ordering of sequential processes. In C.A.R. Hoare and R.H. Perrot, editors, *Operating System Techniques*, pages 72–93. Academic Press, 1972.

[Dij76] E.W. Dijkstra. *A Dicipline of Programming*. Prentice-Hall, 1976.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics*, number 19. American Mathematical Society, 1967.

[FS81] L. Flon and N. Suzuki. The total correctness of parallel programs. *SIAM J. Computing*, 10(2):227–246, 1981.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GMK89] S. Gjessing and E. Munthe-Kaas. Trace based verification of parallel programs with shared variables. In *22nd Hawaii Int. Conf. on System Sciences*, 1989.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa72] C.A.R. Hoare. Towards a theory of parallel programming. In Hoare and Perott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa81] C.A.R. Hoare. A calculus of total correctness for Communicating Sequential Processes. *Science of Computer Programming*, 1:49–72, 1981.

[Jon83] C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Proc. IFIP 9th World Computer Congress*, pages 321–332. Elsevier Science Publishers B.V., North-Holland, 1983.

[Lam80] L. Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[LG81] G.M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15(3):281–302, 1981.

[MC81] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4), 1981.

[Mel86a] S. Meldal. An axiomatic semantics for nested concurrency. *BIT*, 26(2):164–174, 1986.

[Mel86b] S. Meldal. Axiomatic semantics of access type tasks in Ada. Research Report 100, Institute of Informatics, University of Oslo, 1986.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, 1984.

[Ngu85] V. Nguyen. The incompleteness of Misra and Chandy's proof system. *Information Processing Letters*, 21:93–96, 1985.

[Old83] E.-R. Olderog. On the notion of expressiveness and the Rule of Adaptation (Note). *Theoretical Computer Science*, 24(3):337–347, 1983.

[Owe92] O. Owe. Axiomatic treatment of processes with shared variables revisited. *Formal Aspects of Computing*, 4:323–340, 1992.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Department of Computer Science, Cornell University, 1975. (Also in Outstanding Dissertations in the Computer Sciences. Garland Publishing, 1980).

[Sou84] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31:13–29, 1984.

[Sou86] N. Soundararajan. Total correctness of CSP programs. *Acta Informatica*, 23:193–215, 1986.

[Sou91] N. Soundararajan. Completeness issues in trace based systems. Research Report 153, Institute of Informatics, University of Oslo, 1991.

[Sti88] C. Stirling. A generalization of Owicki-Gries Hoare logic for a concurrent language. *Journal of Theoretical Computer Science*, 58:347–359, 1988.

[Stø91] K. Stølen. An attempt to reason about shared-state concurrency in the style of VDM. In S. Prehn and W.J. Toetenel, editors, *VDM '91. Formal Software Developments. 4th International Symposium of VDM Europe*, number 551 in Lecture Notes in Computer Science, pages 324–342. Springer-Verlag, 1991.

[WGS92] J. Widom, D. Gries, and F.B. Schneider. Trace-based network proof systems: Expressiveness and completeness. *ACM Transactions on Programming Languages and Systems*, 14(3):396–416, 1992.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.