Testbarhet av Distribuerade Realtidssystem

12 november 1998
Henrik Thane, Kristian Sandström
Institutionen för Datateknik
Mälardalens Högskola
{hte,ksm}@mdh.se

Abstract

I detta dokument så kommer vi att beskriva problematiken med att testa distribuerade realtidssystem. Vi kommer summariskt att måla upp problemen med att observera, styra och reproducera tester i distribuerade realtidssystem kontra vanliga sekventiella program. Då detta område är tämligen outforskat så kommer vi i möjligaste mån att beskriva relaterat arbete. Bland annat kommer vi att behandla monitorering av distribuerade system, samt beskriva hur man i programvara för parallella (concurrent) system gör för att reproducera beteendet hos systemet.

1. Inledning

Testbarhet kan definieras som benägenheten hos ett system att dölja fel. Dvs, ett mått på hur svårt det är att testa systemet. Ett mått kan t.ex., vara ett stoppkriterium som säger hur många gånger man minst måste testa ett system för att man med en viss konfidens kan säga att inga fel återstår.

Testbarheten hos ett system beror generellt av tre faktorer: Observerbarhet, reproducerbarhet och styrbarhet.

2. Observerbarhet

Med observerbarhet så menar man att ett system måste vara mätbart (observerbart) för att man överhuvud taget skall kunna säga något om dess beteende. Detta kan synes vara trivialt, men för program i distribuerade realtidssystem så är det ej trivialt. För att kunna observera ett system och ha någon tilltro till denna observation så måste observationen vara reproducerbar, dvs när omständigheterna är de samma (indata, internt tillstånd och omgivning) så skall resultatet av nya observationer vara det samma – från gång till gång. För att möjliggöra denna reproducerbarhet så måste systemet också vara styrbart. Om systemet ej är styrbart så kan vi ej påverka omständigheterna (indata, internt tillstånd, och omgivning). Det är just här som det blir svårt i distribuerade realtidssystem. Vi kommer nu att fördjupa oss i detta.

2.1. Test av sekventiella program

När man testar ett system (eller delsystem), så är det önskvärt att observera antingen indata, intermediära värden och variabler, utdata, eller allihop. När man observerar indata så ämnar man avgöra under vilka miljöbetingelser som testfallet körs. Detta är vanligtvis inget problem då testaren ofta har kontroll över indata. Om endast delar av systemet testas brukar man vanligtvis testa dessa isolerat från varandra, och resten av systemet, just för att testaren då kan kontrollera indata till den del som testas. Men när *tiden* för ankomst av indata är viktig, så kan indataobservation bli ett problem.

När man testar ett system så vill man undersöka om systemet för ett specifikt indata ger ett *korrekt* utdata – definierat av specifikationen. För att kunna avgöra om systemet beter sig som definierat så måste man alltså observera hur det reagerar på indata och i vissa fall *när* det gör det. Man måste alltså observera utdata. Detta är vanligtvis inget problem, men ofta är det också nödvändigt, och önskvärt, att observera intermediära värden och variabler, som inte syns i utdata, t.ex., för att underlätta detektivarbetet att finna feltillstånd (errors) i programmet, eller helt enkelt för att utröna *varför* programmet betedde sig som det gjorde. Vanligen gör man det med insättning av extra satser i programmet som skickar de interna datatillstånden till utdata (*intermediärt utdata*). En nackdel med detta är dock att man ofta måste kompilera om koden. En annan teknik för att öka observerbarheten, med avseende på de interna tillstånden, är att använda en interaktiv ”debugger”. Den tillåter testaren att styra exekveringen av programmet. Det görs bl.a. med hjälp av sk. *breakpoints* som gör avbrott i programmet vid förutbestämda punkter. Under dessa avbrott kan testaren granska (observera) innehållet i de interna variablerna.

2.2. Test av distribuerade system

Generering av intermediärt utdata och interaktiv debugging är ej lämpligt i distribuerade system. Om en process, t.ex., fördröjs p.g.a. att den måste exekvera en sats för att ge intermediärt utdata, och därför skickar ett meddelande till en annan process lite senare (än innan), så kan mottagaren undertiden ha fått ett meddelande från en tredje process, och därför erfar att meddelandenas ordning omvänds. Systemets beteende blir då annorlunda om meddelandenas ordningen ej är seriellt ekvivalent, dvs., meddelandenas ordning har betydelse.

Interaktiv debugging, som den tillämpas i centraliserade system, kan ej heller bli modifierad till att fungera i ett distribuerat system, p.g.a. kommunikationsfördröjningar, och frånvaro av en global synkroniserad klocka. Det är också väldigt svårt att stoppa exekveringen av alla processerna i systemet *samtidigt* eller ens i ett konsistent tillstånd.

Ett distribuerat systems beteendet förändras alltså om det observeras. Fenomenet kallas för *probeffekten* [McDowell och Helmbold 1989]. Problemet med probeffekten är den interferens som uppstår mellan processer när deras relativa timing förändras. Detta kan leda till att vissa timing- och synkroniseringsfel inte uppenbaras, eller till att nya fel introduceras som inte skulle ha uppstått annars. Hur man än bestämmer sig för att observera ett distribuerat systemet så måste man hantera probeffekten.

2.3. Test av realtidssystem

Om vi nu även lägger till tidsaspekten, d.v.s. vi vill även observera om systemet uppfyller sina temporala krav, så graveras situationen ytterligare. Om vi, t.ex., vill verifiera den temporala korrektheten hos ett systems beteende så räcker det ej med att bara observera intressanta händelser (indata, utdata, eller kommunikation), och ordningen i vilka de sker, utan man måste även observera *när* i tiden de sker. Alltså måste mer information lagras, och presenteras, och därmed aggreveras probeffekten ytterligare.

Probeffekten kan även påverka beteendet hos en ensam sekventiell process. Detta kan ske då t.ex., ett programs resultat påverkas av *när* i tiden det exekveras, om då en fördröjning (p.g.a. probeffekten) får programmet att t.ex., sampla en extern fysisk process lite senare, än annars, så kan programmets resultat och beteende variera.

Generellt är det därför nödvändigt att undvika (eliminera) probeffekten.

2.4. Hantering av probeffekten

Beroende på vilken metod som används för att observera (monitorera) ett system kommer detsamma att påverkas i olika hög grad. Det är också av betydelse vilken typ av system som man monitoterar. Probeffekten kan vara svårare att undvika i parallella system/realtidssystem än i sekventiella system. Detta på grund av att tidsbeteendet och synkronisering har stor betydelse för parallella system.

Probeffekten kan attackeras på flera olika sätt:

Ignorera problemet: man hoppas att probeffekten ska ha ringa eller ingen inverkan.

Minimera effekterna: Man försöker implementera effektiv monitorering, på detta sätt förändras inte tidsbeteendet i så hög grad.

Dessa två angrepssätt kan knappast anses vara acceptabla för realtidssystem eller säkerhetskritiska system eftersom man inte alls vet om de delar av systemet man testat är korrekt utan proberna.

Undvika probeffekten: Man kan lämna kvar proberna i systemet, gömma dem bakom logisk tid eller använda sig av dedicerad hårdvara för monitorering.

Att lämna kvar proberna i systemet löser naturligtvis problemet eftersom man ej förändrar systemet. Det kräver dock att det finns tillräckligt med systemresurser för att låta alla prober i systemet sitta kvar. Det är en kostnad förenad med detta. Det kräver även att systemet designas från början med hänsyn tagen till proberna, i alla fall om proberna kräver mycket resurser av systemet.

Att gömma proberna bakom logisk tid innebär att man monitorerar systemet utan att påverka ordningen mellan händelser i systemet. Denna metod är användbar om systemet bara är beroende av att den relativa ordningen mellan händelser hålls upprättad. Ett problem är att händelsers tidsbeteende påverkas, det gör metoden mindre användbar för många (de flesta) realtidssystem.

Att använda dedicerad hårdvara innebär att man observerar systemet utifrån. Man kan använda dubbelportminnen och extern monitoreringsutrustning. Visst data kan vara svår att monitorera, t.ex. lokala variabler i CPU-register. I distribuerade system kan busstrafik monitoreras genom att en separat monitoreringsnod ”sniffar” av bussen. Även denna metod kräver att man tar hänsyn till testning i ett tidigt skede av designen. Metoden kan vara svår att tillämpa i system med befintlig hårdvara.

Speciellt gäller för tidstriggade statiskt schemalaggda system att man kan skapa temporala brandväggar för att undvika att prober förändrar tidsbeteendet i systemet. Finns det luckor i schemat så kan man låta prober exekvera i dessa luckor utan att det stör resten av systemet. Man kan även ta bort dessa prober från systemet efter testning utan att få någon probe-effekt. Att luckor i schemat ska uppstå på de platser man behöver är inte troligt, så även här krävs det att man designar med tanke på prober. När man tagit bort proberna så kan man dock använda deras tidsresurser för mjuka realtidsuppgifter. Dock krävs det att det underliggande run-time systemet arbetar efter vissa principer om temporala brandväggar ska vara möjliga (ett tasks starttid får ej vara beroende av variationer i andra tasks exekveringstid).

3. Reproducerbarhet

En väl etablerad teknik, är att man upprepar test av systemet efter att man rättat feltillstånd (errors), eller efter att man lagt till ny funktionalitet. Detta brukar kallas för *regressionstestning*. Meningen med regressionstestning är att (1) verifiera att feltillstånden verkligen har eliminerats, eller (2) att modifieringarna inte lett till nya oönskade effekter, eller feltillstånd. Vanligtvis återanvänder man samma testfall som innan, men testfallen kan också behövas uppdateras då den nya funktionaliteten kräver det.

En nödvändighet för att man överhuvudtaget skall kunna utföra regressionstest är att systemets beteende är reproducerbart. Ett system är reproducerbart (eller deterministiskt) om det för samma indata och interna tillstånd alltid levererar samma utdata. Om ett systems beteende ej är reproducerbart så kan man inte lita på att en repetition av testfallet aktiverar samma feltillstånd. Om systemet inte är reproducerbart, kan man inte heller dra slutsatsen att en feltillståndsrättning verkligen hade någon effekt.

Kostnadseffektiviteten hos regressionstestning kan också förbättras om man automatiserar processen. Automatisk jämförelse av resultat är dock mycket svårare (om inte omöjligt) då exekveringen inte är reproducerbar, d.v.s., om det finns flera möjliga svar. Reproducerbarhet är trivialt för sekventiella program, men i distribuerade system och realtidssystem kan så kallade ”raceconditions” uppstå.

Detta kan bero på:

- CPU last
- Trafiken på datornätet
- Icke determinism i kommunikations protokoll, t.ex., CSMA/CD – Ethernet
- Probers närvaro eller ej
- Närvaron av icke deterministiska satser t.ex., slumptal.

Källan till den ickedeterminism som belackar oss är bristen på a priori (i förväg) vetskap om hur systemet beter sig med avseende på timing, ordning och händelser. På en mer detaljerad nivå kommer vi nu att analysera källan till denna ickedeterminism. Vi kommer att ställa upp ett resonemang som visar hur (1) *synkronisering* mellan processer, (2) *preemption* (3) *avbrott*, och (4) *jitter* påverkar ett systems reproducerbarhet, samt vad graden av påverkan blir beroende på vilken typ av schemaläggnings- och synkroniseringsprincip systemet nyttjar. I resonemanget så antas systemet bli testat med hjälp av så kallad ”path-testing”, dvs att man testar *alla* möjliga exekveringsvägar, definierade av de selektioner som bygger upp kontrollflödet i programmet. Denna typ av test är generellt ”omöjlig” (antalet testfall går mot oändligheten) om det ej finns någon gräns för hur många gånger en loop får exekvera. Vi antar i detta resonemang att alla loopar körs maximalt en gång.

Två modeller

Vi kommer här att titta på modeller för identifiering av möjliga synkroniseringsscenarios och exekveringsvägar i realtidssystem. Vi har två approacher: En för statiskt schemalagda system och en för fix prioritets schemalagda system. Vi kommer i båda fallen att ta hänsyn till preemption, interrupter och jitter. I FPS fallet kommer vi bl.a. att titta närmare på relaterat arbete från test av parallella (concurrent) program (sektion 3.2).

I Fix prioritets schemalagda system, tas alla beslut om vilket task som skall köras under drift m.h.a. fasta prioriteter på tasken. All synkronisering och kommunikation sker med hjälpa av primitiver i programvaran.

I statiskt schemalagda system bestäms exekveringsordningen av tasken på förhand, dvs. innan systemet tagits i drift. Synkroniseringen mellan task definieras också av denna exekveringsordning. Hela schemat upprepas sedan med en periodtid lika med minsta gemensamma multipeln (LCM least common multiple) av de enskilda taskens periodtider.

3.1. Reproducerbarhet av statisk schema-lagda system

Vi inleder med ett enkelt statiskt schemalagt realtidssystem. Därefter lägger vi till preemption, avbrott, och jitter, för att analysera hur dessa påverkar systemets reproducerbarhet.

3.1.1 Modellen

Med den grundläggande modellen vi tillämpar kan man identifiera vilka vägar som måste exekveras för att garantera att alla möjliga vägar körts (under pathtesting med ett varv för varje loop). Hur detta görs rent praktiskt är utanför scopet i denna rapport – och är för övrigt ett forskningsområde

Emellertid definierar vi nedan ett mått på hur många dess vägar är. Ju fler vägar, desto sämre testbarhet.

För det enkla fallet, där ingen tidsdelning, avbrott eller jitter förekommer, beskriver följande uttryck (formel 1) det maximala antalet vägar i ett statiskt schemalagt system:

$$n_{tot} = \prod_{i=0}^{k} n_i^{\frac{LCM}{Ti}} \qquad \text{(formel 1)}$$

n_i = Antalet vägar i task *i*.

T_i = Task *i*:s periodtid

LCM = Minsta möjliga multipel av taskens periodtider

k = Antalet task i *LCM*

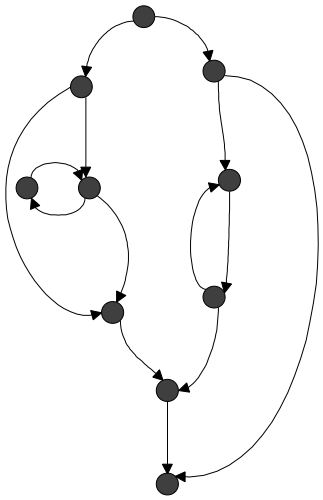
Formeln beräknar helt sonika, produkten av det maximala antalet exekveringsvägar för varje task som är del av en LCM cykel, även då inräknat flera instanser av samma task.

Exempel 1: Antag att vi har två task: *Task1* n = 3, T = 5; *Task2* n = 3, T = 8. Task 1 och 2 har då en *LCM* på 40, eftersom 5*8 är den minsta gemensamma multipeln.

Maximal antalet vägar, under en period av 40 tidsenheter, blir då:

$$n_{tot} = 3^{\frac{40}{5}} \times 3^{\frac{40}{8}} \approx 1{,}59 \times 10^{6}$$

Värdet är pessimistiskt, då troligen fler vägar i programmet är omöjliga på grund av semantiska restriktioner. Detta gör dock inget, ty vi kan ändå använda måttet för att göra jämförelser.

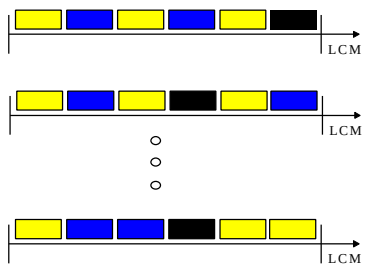


Figur 2-1 Kontrollflödet för ett program.

Figur 2-2 En kedja av task som alltid repeteras med en periodtid på LCM.

Ett mått för fix prioritetschemaläggning (FPS)

Givet att ett FPS system kör sina task periodiskt, och har en ekvivalent taskmodell så kan vi tillämpa det just beskrivna måttet för FPS system också. Skillnaden ligger i att för statiskt schemalagda system så finns det bara ett synkroniserings-scenario, medan det för FPS finns flera. Detta beror på att det under drift bestäms vem som får exekvera, betingat av synkronisering mellan task, nyttjande av delade resurser och kommunikation. Alla möjliga scheman för ett FPS system är lika många som det finns möjliga lösningar till ett statiskt schemalagt system. Slutsats: antalet möjliga exekveringsvägar för ett FPS system är *många* fler än för statiskt schemalagda system.



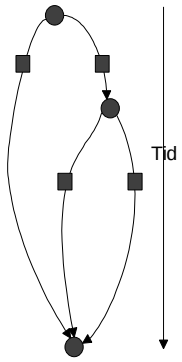
Figur 2-3 Olika scheman för FPS.

Men hur var det med reproducerbarheten?

En viktig sak att påpeka igen, är att det i statiskt schemalagda system bara finns *ett* synkroniseringsscenario som *bestäms* av det statisk schemat och *utförs* av realtidskärnan. I periodiska FPS system så bestäms alltså synkroniseringsordning under runtime, så om man vill erhålla reproducerbarhet i FPS system så måste man dels identifiera alla möjliga synkroniseringsordningar och exekveringsvägar, men också med *våld* styra upp de identifierade synkroniseringsordningarna för att kunna utföra path testing. Vi kommer att belysa detta mer utförligt i sektion 2.2.2

3.1.2 Modellen + Preemption

Om man tillåter preemption (task får avbryta varandra) blir situationen betydligt mer komplicerad, eftersom antalet möjliga exekveringsvägar ökar med varje möjlig preemptionpunkt i programmet, där tasket kan bli avbrutet och ett annat task kan börja exekvera (kvadraterna i figur 2-4). Preemption skiljer sig från interrupt (som vi avhandlar senare) genom att preemption uteslutande sker vid klock-tick, och inte som för interrupt, mellan godtycklig maskinkodsinstruktion.



Figur 2-4 Kontrollflöde för task med preemptionpunkter.

Om vi börjar med att anta att det, för var och en av de *k* exekverande tasken under en hel periodcykel, existerar en uppsättning punkter, *Qi,j* där preemption, kan ske, gäller följande formel: (Formel 2)

$$n_{sum(i)} = n_i + \sum_{\forall j \in S_i}((\sum_{\forall p \in Q_{i,j}} n_{A_p} \times n_j \times n_{B_p}) - n_i)$$

$\boldsymbol{n_{Ap}}$ = antal vägar i tasket före avbrottspunkten

$\boldsymbol{n_j}$ = antalet vägar i det avbrytande tasket *j*

$\boldsymbol{n_{Bp}}$ = antalet vägar i tasket efter avbrottspunkten

$\boldsymbol{S_i}$ = mängden av alla avbrytande task

$\boldsymbol{n_i}$ = antalet möjliga vägar genom task *i*.

$\boldsymbol{N_{sum(i)}}$ = summan av alla möjliga vägar genom task *i* p.g.a. preemption..

Exempel 2. Antag att vi har två task: task1 och task2, där enligt det statiska schemat task2 kan göra preemption på task1, för att den skall hinna med sin deadline. Antag vidare att:

- *task1* har *n* = 3 möjliga exekveringsvägar, och en periodtid på *T* = 5
- *task2* har *n* = 3 möjliga exekveringsvägar, och en periodtid på *T* = 8; task1 och task2 får då en *LCM* på 40 (precis som i förra exemplet).
- att task2 endast kan göra preemption på task1 en gång per båge i kontrollflödet för task1. Punkterna för preemption är illustrerade i figur 2-3 (de små fyrkanterna).

Summan av antalet vägar för task1 blir då:

$$\mathbf{n_{sum(1)} = 3 + (((1*3*1) + (1*3*2) + (1*3*1) + (1*3*1)) - 3) = 15}$$

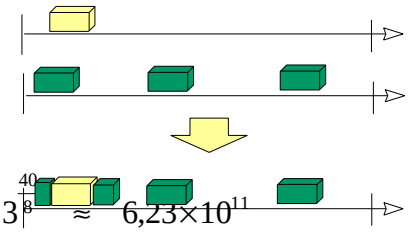
Och summan för task2 är oförändrad p.g.a. att den ej blir avbruten, dvs

$$\mathbf{n_{sum(2)} = 3 + (0) = 3}$$

Utifrån formel1 och formel2, skapas en ny formel (formel3) för det totala antalet exekveringsvägar i ett statiskt schemalagt system med preemption: (Formel 3)

$$n_{tot} = \prod_{i=0}^{k}(n_i + \sum_{\forall j \in S_i}(\sum_{\forall p \in Q_{i,j}} n_{A_p} \times n_j \times n_{B_p} - n_i))^{\frac{LCM}{Ti}}$$

Om vi nu fortsätter med exempel2 så kan vi nu räkna ut det nya antalet exekveringsvägar, när vi tillåter preemption:



$$n_{tot} = 15^{\frac{40}{5}} \times 3^{\frac{40}{8}} \approx 6{,}23 \times 10^{11}$$

Figur 2-5 Preemption.

Vilket är ungefär 10^5 gånger fler vägar än i exempel 1.

3.1.3 Modellen + interrupt

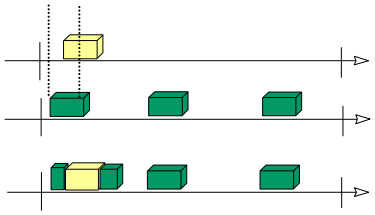
Om vi tillåter interrupt så blir det än värre, ty till skillnad från preemption som bara kan ske vid klock-tick, så kan interrupt slå in, i princip, mellan vilken maskinkodsinstruktion som helst.

I praktiskt hänseende kan ett sådant system betraktas som icke-deterministiskt, eftersom det är för komplext för att kunna testas fullständigt. Att interrupt ändå förekommer, kommer från krav på korta svarstider på för yttre signaler.

Således finns det en motsättning mellan svarstider och testbarhet för ett system.

3.1.4 Modellen + jitter

Jitter definieras som en variation av periodtider och svarstider för task. Om man tillåter preemption och är utsatt för jitter, så ökar antalet punkter där preemption kan ske, och därmed det totala antalet möjliga exekveringsvägar för systemet. Testbarheten kan därför betraktas som proportionellt mot magnituden på jittret. Vidare försvårar jitter förutsättningarna för att hålla de tidskrav som ställs på systemet, eftersom jitter påverkar systemets periodtid.



Figur2-6 Jitter + preemption.

3.1.5 Diskussion

Värt att nämna om denna modell är att den antar värsta fallet, dvs. vi antar att tasken kan ha vilka sidoffekter som helst på varandra. Den kräver också i fallet för preemption att man vet exakt när tasken kan göra preemption på varandra vilket i sin tur kräver att man har en fullständig exekveringstidsestimering. Om man ej antar att de kan ha godtyckliga sidoeffekter på varandra utan mer begränsat, så räcker det kanske med att undersöka var taskens preemption på varandra *egentligen* har någon effekt. I praktiken är detta troligen mer rimligt.

3.2. Väganalys (Path analysis)

I sektion 4.1.1 beskrevs att i FPS system så bestäms synkroniseringsordningen mellan task under runtime. För att erhålla reproducerbarhet i FPS system så måste man dels identifiera alla möjliga synkroniseringsordningar och exekveringsvägar, men också med *våld* styra upp de identifierade synkroniseringsordningarna. I denna sektion beskriver vi en metod som erbjuder en möjlighet att identifiera alla möjliga synkroniseringsordningar och exekveringsvägar för parallella program och som har potential att tillämpas i FPS system.

Exekveringsväganalys är en beprövad metod som ofta används för test av sekventiella program, här presenteras en metod som har utökats till att även täcka parallella system. Det som tillförts är mekanismer för att skildra hur parallella program samverkar, så att man kan observera och styra systemets beteende.

Ett sekventiellt programs beteendet beskrivs av dess indata samt en specifik exekveringsväg. Det vill säga, exekveras samma kod med samma indata fås samma resultat. Det förutsätter dock att varje programsats är deterministisk. Ett exempel på en icke-determinism i program kan vara slumptalssatser. För parallella system räcker det inte med att samma indata och kod exekveras, eftersom olika exekveringsordning mellan interagerande parallella program (task) kan påverka slutresultatet. Beteendet hos ett system med parallella task, bestäms av taskens indata, den kod de exekverar (kontrollflöde) samt de synkroniseringar som sker mellan tasken (synkroniseringsflöde). Kan man styra dessa tre parametrar så är systemets beteende reproducerbart. Vi kommer under detta avsnitt att beskriva en metod benämnd *väganalys* som angriper dessa tre attribut (indata, kontrollföde, synkronisering).

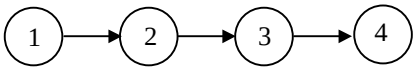
3.2.1 Kontrollflöde

Den syntaktiska strukturen av ett tasks kod kan beskrivas med en kontrollflödesgraf, se exempel 3 (nedan). En sådan graf ger de olika vägar som exekveringen av ett program kan ta. Ett specifikt exekveringsscenario för ett task bestäms då av en sådan väg genom kontrollflödesgrafen. Exekveringen av ett parallellt system inbegriper således ett antal parallella vägar. Om en väg genom en flödesgraf för något task, i, benäms p_i och systemet består av tasken t_1 till t_n, så kan ett exekveringsscenario för alla task i systemet beskrivas av n-tupeln $(p_1,p_2,\ldots,p_n)$. En sådan tupel är giltig om det finns åtminstone en indatamängd som för varje task t_i traverserar vägen p_i.

Exempel 3: Tre task, T1, T2 och T3 samt deras flödesgrafer.

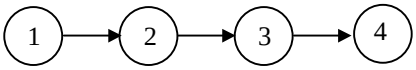
T1()

```
int y;
{                          1
  y = receive(R);          2
  out(y)                   3
}                          4
```



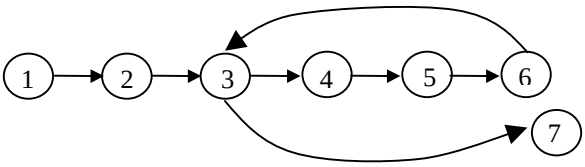
T2()

```
int z;
{                          1
  z = receive(R);          2
  out(z)                   3
}                          4
```



T3()

```
int x;
{                                1
   read(x);                      2
   for(int i=1;i≤2;i++){  3
      send(R,x);                 4
      x = x + 1;                 5
   }                             6
}                                7
```



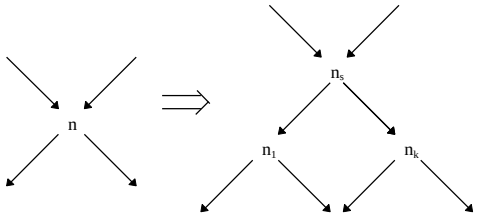
Exemplet visar koden och kontrolflödesgraferna för tre task, T1, T2 och T3. Task T3 synkroniserar med ett task innan data skickas, send anropet skickar alltså inget data innan något task gör ett anrop till receivesatsen med motvarande parameter (R). T3 har genom send anropet ingen kontroll över ordning med vilken den sänder data till T1 och T2. Ordningen beror på vilket task som anropar receivesatsen först. Exekveringsvägen (endast en möjlig) för T3 är (1,2,3,4,5,6,3,4,5,6,3,7).

För parallella system kan två exekveringar med samma indata och samma exekveringsvägar ge skilda resultat. Detta beror på interaktion mellan task, där olika ordningsföljd mellan task kan medföra skilda resultat. Därför måste man även ta hänsyn till hur task synkroniserar med varandra för att kunna ge en komplett bild av ett parallellt system.

3.2.2 Synkronisering

Ett tasks synkroniseringar med andra task kan representeras av en synkroniseringsgraf. Grafen är en modell av hur task interagerar med varandra och ger en deterministisk beskrivning av synkroniseringen, det gör att synkroniseringsgrafen inte behöver avspeglas i kontrollflödesgrafen, se exempel 4 (nedan). Det kommer av att en enda programsats kan synkronisera med flera andra task och där synkroniseringsordningen kan variera. Ta som exempel en meddelandekö som läses av ett task och där flera andra task kan skicka meddelanden till kön. Om två task skickar meddelanden i omvänd ordning till denna kö så är synkroniseringen inte densamma. En synkroniseringsgraf kan konstrueras från en kontrollflödesgraf enligt följande [Yang92]:

1) Ta bort alla noder, förutom start och slutnod, som inte härrör från synkroniseringssatser.
2) För varje nod *n* som motsvarar en ickedeterministisk synkroniseringsstats som kan synkronisera med *k* task; ersätt noden *k* med en nod n_s som representerar ett ickedeterministiska val. Låt n_s få *k* lövnoder ($n_1..n_k$) som motsvarar synkronisering med $task_1$ till och med $task_k$. Se figur 2-7 nedan.

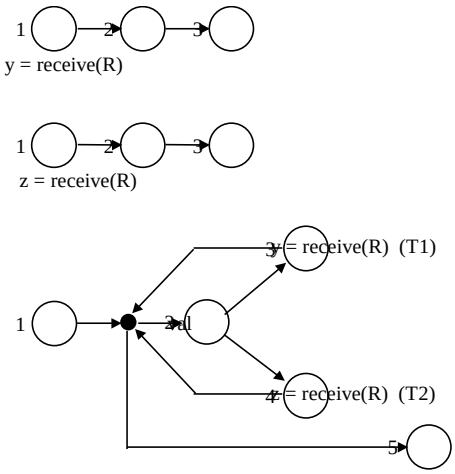


Figur 2-7. Transformering av icke-deterministisk synkronisering.

Om en väg genom en synkroniseringsgraf för något task, *i*, benäms s_i och systemet består av tasken t_1 till t_n, så kan ett synkroniseringsscenario för alla task i systemet beskrivas av *n*-tupeln (s_1,s_2,…,s_n).

Beteendet hos ett parallellt system bestående av *n* stycken task kan nu, för ett givet scenario, bestämmas av en indatamängd ∝ (fnurkel), en *n*-tupel (p_1,p_2,…,p_n) som representerar kontrollflödesvägar samt en *n*-tupel (s_1,s_2,…,s_n) som representerar synkroniseringen i systemet.

Exempel 4. Synkroniseringsgrafer för T1, T2 och T3.



I exempel 4, består synk-roniseringsgraferna för T1 och T2, förutom start- och slutnod, bara av synkroniseringssatsen receive. Task T3 har två synk-roniseringsnoder på grund av att tasket kan synkronisera både med T1 och T2. I detta fall har synkroniseringssatsen expanderats enligt punkt 2 ovan. Den svart punkten är införd för att öka läsbarheten. För T3 skulle vektorn (1,2,3,2,4,5) beskriva ett möjligt synkroniseringsscenario där T3 först synkroniserar med T1 och sedan med T2.

3.2.3 Testning med hjälp av Väganalys

Två olika tillvägagångssätt kan användas vid test:

- **Ickedeterministisk exekvering**. Här används bara indatadelen av Väganalysen. Synkroniseringsdelen ses här som utdata från testet. Om man utför samma test två gånger och identiska vägar i synkroniseringsgraferna har traverserats så har samma beteende testats. Detta tillvägagångssätt medför att testningen är enkel att utföra, man exekverar bara systemet med ett givet indata. I allmänhet är det dock ej säkert att testen är reproducerbara (dvs. att man råkade köra samma synroniseringsväg).
- **Styrd exekvering**. Detta innebär att man kan styra exekveringen så att en given tupel av synkroniseringsvägar kan upprepas varje gång testet genomförs. Här kan tupeln av synkroniseringsvägar ses som en extra indatamängd. Det här tillvägagångssättet kräver att synkroniseringen mellan tasken kan styras fullständigt.

Teststrategier

En teststrategi för Väganalys kan sammanfattas som [Yang92]:

1) Välj en mängd exekveringsvägar för varje task
2) Välj en mängd tupler ($P_1..P_n$) sammansatta av taskens exekveringsvägar ($p_1..p_k$)
3) Konstruera tupler ($S_1..S_n$) med synkroniseringsvägar ($s_1..s_k$) från (2)
4) Generera indatamängder för varje tupel S_i längs P_i
5) Exekvera systemet med indatadelen (ickedeterministisk exekvering)
6) Exekvera systemet med styrd exekvering

Exempel 5. Vi nyttjar data från exempel 3 och 4:

1) Exekveringsvägar för tasken. Det finns bara en möjlig väg för varje task.

   T1: p1=(1,2,3,4)

   T2: p2=(1,2,3,4)

   T3: p3=(1,2,3,4,5,6,3,4,5,6,3,7)

2) Tupler

   P = [p1,p2,p3]

3) T3 har två möjliga synkroniseringsvägar.

   T1: s1=(1,2,3)

   T2: s2=(1,2,3)

   T3: s3=(1,2,3,2,4,5) s3'=(1,2,4,2,3,5)

   S1 = [s1,s2,s3]

   S2 = [s1,s2,s3']

4) T1 och T2 saknar indata förutom det data som de tar emot från T3.

   S1,P: α1=[(ε)(ε)(x=7)]

   S2,P: α2=α1

5) Utför testet med α1 (α2 är identisk). För detta test kan variabel y i T1 få värdet 7 eller 8 beroende på synkroniseringsordning. Dito gäller för variabel z i T2.

6) Utför test:

   (a) med α1 och styrd exekvering enligt S1,P

   (b) med α2 och styrd exekvering enligt S2,P

För test (a*)* får variabel *y* i *T1* värdet 7 och variabel *z* i *T2* får värdet 8. För test (b) får variabel *y* i *T1* värdet 8 och variabel *z* i *T2* får värdet 7.

3.3. Diskussion

Vi har i detta avsnitt analyserat och beskrivit olika sätt för att modellera reproducerbarhet i parallella system och i distribuerade realtidssystem i synnerhet. Vi fann att det i statiskt schemalagda system bara finns *ett* synkroniseringsscenario som *bestäms* av det statiska schemat och *utförs* av realtidskärnan. I periodiska FPS system så bestäms dock synkroniseringsordningen under runtime. Om man vill erhålla reproducerbarhet i FPS system så måste man, dels identifiera alla möjliga synkroniseringsordningar, och exekveringsvägar, men också med *våld* styra upp de identifierade synkroniseringsordningarna för att kunna utföra path testing. Vi beskrev även en metod från test av parallella system kallad Väganalys. Den kunde i viss mån användas till FPS system för att identifiera möjliga exekekveringsvägar och möjliga synkroniseringsscenarion. För att i slutänden erhålla ett reproducerbart FPS system så måste man alltså styra exekveringen och synkroniseringen mellan tasken. Hur detta kan göras är inte helt uppenbart, men troligtvis så måste realtidskärnan understödja styrd exekvering. Författarna ponerar att det skulle kunna lösas med något liknande ett statiskt schema där man kan bestämma exakt vilket task som får köra och när, eller en realtidskärna som tillhandahåller varianter på primitiver för synkronisering och kommunikation där specifika synkroniseringsscenarios framtvingas. Frågan om hur detta skall göras för att undvika probeffekten med avseende på timing verkar svårt (minst sagt). Probeffekten med avseende på synkroniseringsordning har dock eliminerats med temporala brandväggar eller styrd exekvering.

4. Konklusion

Gällande test och testbarhet av distribuerade realtidssystem så kan vi med säkerhet dra slutsatsen att det finns mycket att göra inom detta område.

De arbeten som finns är i stort sett: En problembeskrivning om testning av DRTS [Schutz94], några arbeten om monitorering av DRTS [Dodd92, Tsai96], samt lite relaterat arbete från test av parallella system utan tidskrav [Tai91, Yang92].

För att summera: Vi har i denna rapport beskrivit problematiken med att testa distribuerade realtidssystem. Vi har målat upp problemen med att observera, styra och reproducera tester i distribuerade realtidssystem, men även givit modeller med vars hjälp man kan analysera systemen för att bland annat kunna möjliggöra reproducerbarhet.

Referenser

1. [Dodd92] P. S. Dodd, et. al. *Monitoring and debugging distributed real-time programs.* Software-practice and experience. Vol. 22(1), pp. 863-877, October 1997.
2. [McDowell89] C.E. McDowell and D.P.Hembold. Debugging concurrent programs. ACM Computing Serveys, 21(4):593-622, December 1989.
3. [Schütz94] W. Schütz. Fundamental Issues in Testing Distributed Real-Time Systems. Real-Time Systems, 7, 129-157, 1994.
4. [Tai91] K.C Tai, et. al. *Debugging concurrent Ada programs by deterministic execution*. IEEE transactions on software engineering. Vol. 17(1), pp. 45-63, January 1991.
5. [Tsai96] J.P. Tsai, et. al. *A system for visualizing and debugging distributed real-time systems with monitoring support*. Journal of Software engineering and Knowledge engineering. Vol. 6(3), pp. 355-400, 1996.
6. [Yang92] R-D Yang and C-G Chung. Path analysis testing of concurrent programs. Information and software technology. Vol 34 No 1 January 1992.