Visualisering og animasjon av diskret-event simuleringsmodeller

Førsteamanuensis John I. Dalseng
Høgskolen i Finnmark
9500 Alta

Innledning

En diskret-event simuleringsmodell er et program som etterligner atferden til et virkelig system ved å følge mønsteret av hendelser i systemet. Systemet kan bestå av mange og sammensatte mønstre av hendelser, og modellen skal bevare rekkefølgen av gjensidig avhengige hendelser.

Vi vil beskrive en objekt-orientert simuleringsteknikk der simuleringsmodellen består av aktive og passive entiteter. De aktive entitetene, kalt prosesser, representerer systemkomponenter som utfører aktiviteter i systemet, og de passive entitetene representerer køer, ressurser og andre ikke aktive komponenter.

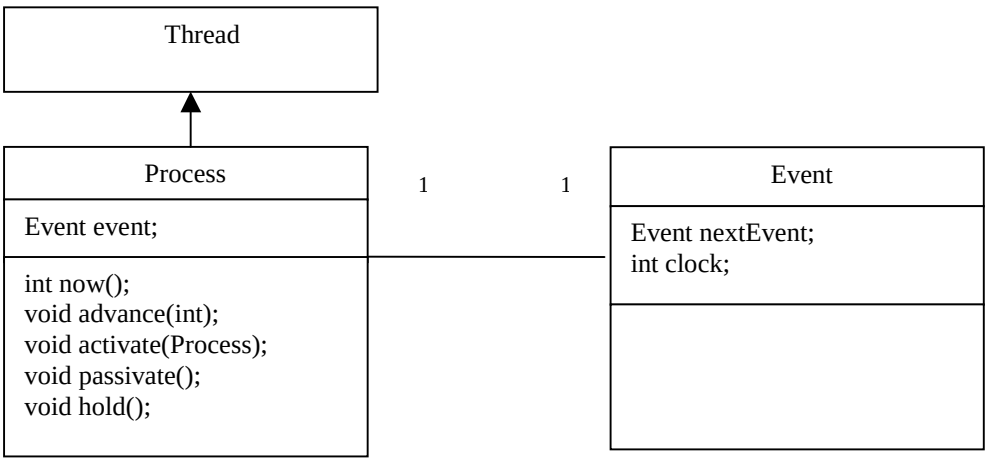
For å observere utførelsen av simuleringsmodellen kan visuell presentasjon av simuleringen være av stor nytte.
Simuleringen visualiseres ved hjelp av en grafisk modell, der entitetene og tilstandene i simuleringsmodellen er representert ved hjelp av grafiske objekter på skjerm.

Programsystemet består av to hovedmoduler: simulator og animator. Simulatoren definerer simuleringsmodellen og simuleringsalgoritmen, mens animatoren definerer den grafiske modellen og animasjonen. Simulatoren og animatoren kommuniserer via en CSP kanal [1].

Beskrivelsen av systemet er basert på Java™ ([8]). Modell-entitetene og de grafiske symbolene representeres ved hjelp av klasser. Klassene og interaksjonen mellom klasse-objektene er beskrevet ved bruk av UML-notasjon ([5]).

Simulatoren

Simuleringskonstruksjonene og simuleringsalgoritmen er definert i en package Simulation. Simulation definerer class Process, som representerer de aktive entitetene, prosessene, i modellen. Process er subklasse av class Thread i Java.



Class Process er assosiert med class Event. Hver hendelse i en prosess opptrer på et tidspunkt i simuleringen. Objekter av class Event angir tidspunktet for neste hendelse. Tidspunktet angis av variabelen clock, som også definerer lokal tid i prosessen. Simulert tid, systemtid, defineres som den laveste lokale tid i de aktive prosessene. Objekter av class Event er med i en liste, som angir skedulerte hendelser i simuleringsmodellen.

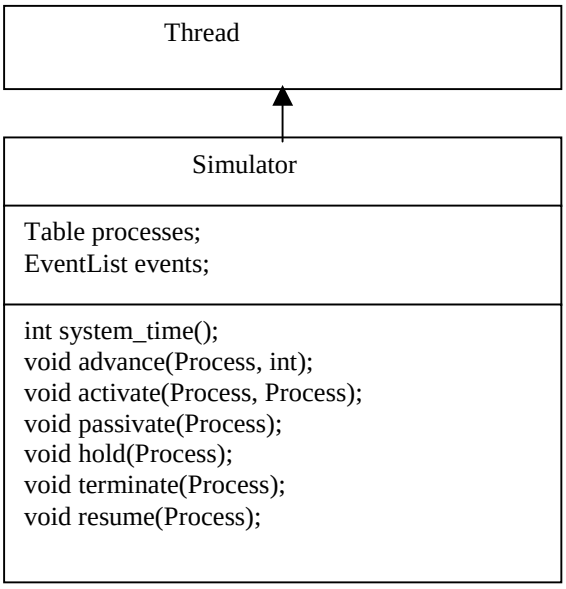
Et objekt av class Process kan være i en av fire tilstander:

1. active – prosessen er i gang med å utføre instruksjoner knyttet til hendelser og oppdatering av lokal tid.
2. passive – prosessen er blokkert og venter til den blir aktivisert av en annen prosess.
3. suspended – prosessen er blokkert og venter til system-tid når dens lokale tid.
4. terminated – prosessen er blokkert og kan ikke aktiviseres igjen.

class Process har følgende metoder:

- now: returnerer prosessens lokale tid (tidspunktet for neste hendelse).
- advance: øker lokal tid med et gitt antall tidsenheter, og fastsetter tidspunktet for neste hendelse.
- activate: prosessen aktiviserer en annen prosess nå, nåværende tidspunkt lokal tid.
- passivate: prosessen venter inntil den blir aktivisert av en annen prosess.
- hold: prosessen blokkeres midlertidig til system-tid når dens lokale tid.

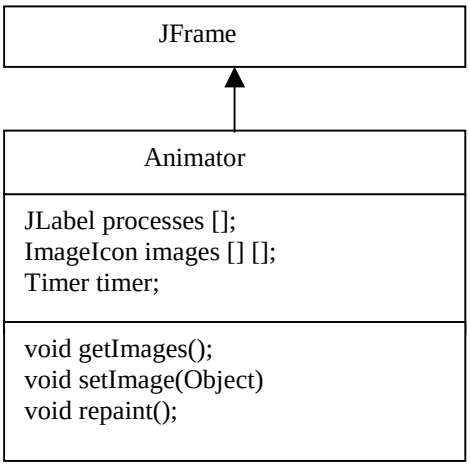
Vi har definert en simulator, som kontrollerer forløpet av system-tid og skedulerer utførelsen av hendelsene i de ulike prosessene.



Prosessene identifiseres ved hjelp av en tabell, Table processes, og de tilhørende neste-hendelse tidspunktene angis ved hjelp av en liste EventList events. Metoden system_time returnerer system-tid, og metodene advance, activate, passivate og hold implementerer de tilhørende operasjonene definert i class Process. Simuleringsalgoritmen implementeres av run-metoden. Vi skal ikke ta med representasjonen av køer, ressurser og andre ikke aktive komponenter her.

Animatoren

De grafiske komponentene og animasjonen av simuleringen på skjerm skal baseres på Java Swing grafiske komponenenter. De aktive og passive entitetene i simuleringsmodellen skal presenteres ved hjelp av grafiske symboler, og sekvenser av hendelser vises ved hjelp av serier av symboler. Animasjonsomgivelsene defineres ved hjelp av class Animator.

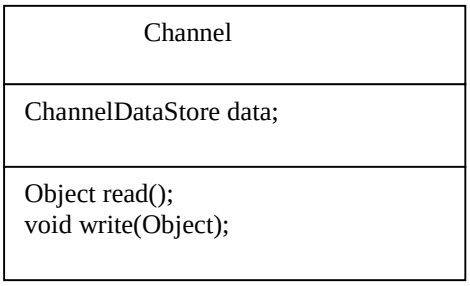


De grafiske symbolene for prosessene defineres ved hjelp av arrayet JLabel processes. Det grafiske elementet JLabel kan inneholde prosessens identifikator og lokale tid, og et bilde som viser tilstanden. Bildene som skal illustrere de ulike tilstandene en prosess kan være i representeres ved hjelp av arrayet ImageIcon images. Variabelen timer genererer hendelser som oppdaterer skjermbildet med jevne mellomrom.

Metoden setImage velger ut de bildene som skal presenteres på skjerm, og getImage laster inn bildene. Metoden repaint oppdaterer bildene på skjerm. Animatoren skal i tilleggg ha en run-metode (runnable), som implementerer animasjonsalgoritmen.

Kommunikasjon

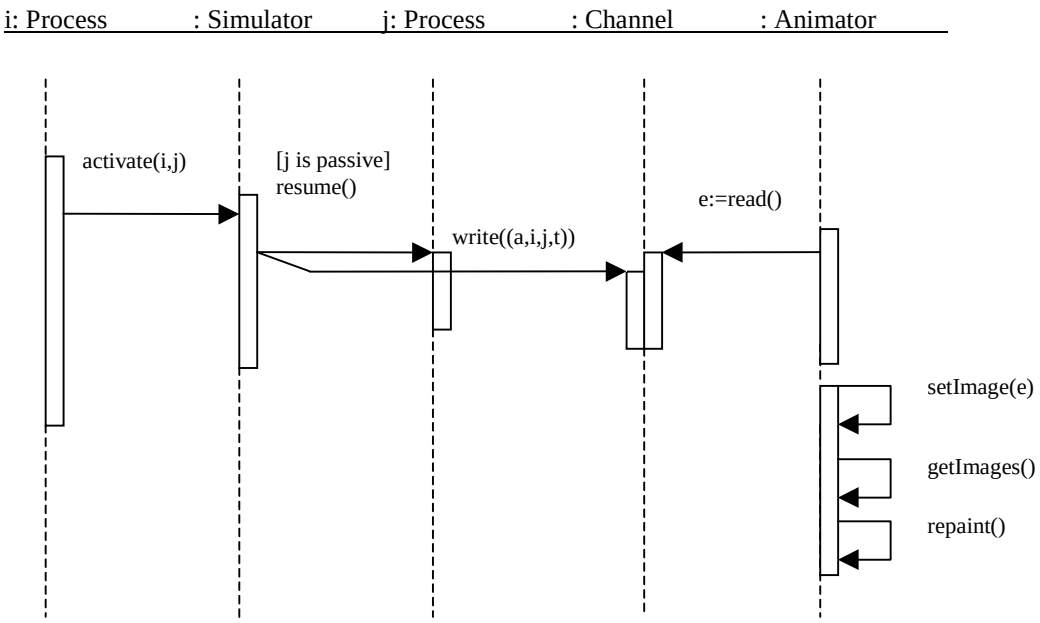
Simulatoren og animatoren kommuniserer via en CSP kanal. Kanalen er en synkronisert kommunikasjonsmekanisme. Simulatoren kan skrive på kanalen på et hvilket som helst tidspunkt, men må vente til animatoren har lest kanalen før den kan fortsette. Animatoren kan lese fra kanalen på et hvilket som helst tidspunkt, men må vente til simulatoren har skrevet på kanalen. Kommunikasjonsmekanismen representeres ved hjelp av class Channel. Det finnes for tiden to CSP pakker i Java: JCSP utviklet ved Universitetet i Kent, Canterbury, UK, ([4]) og CTJ utviklet ved Universitetet i Twente, Nederland ([5]). Vi har basert class Channel på JCSP.



Når en hendelse utføres i simuleringmodellen, vil simulatoren sende en melding til animatoren om hendelsen via kanalen. Meldingen tas vare på i variabelen ChannelDataStore data. Meldingen inneholder data om prosessene som er involvert i hendelsen, hendelse tidspunkt og hvilken type hendelse som er utført. Simulatoren må vente til animatoren har lest meldingen. Når animatoren har lest meldingen kan simulatoren fortsette. Animatoren vil oppdatere den grafiske modellen, og presentere modellen på skjerm før neste melding leses.

Interaksjon mellom simulator og animator

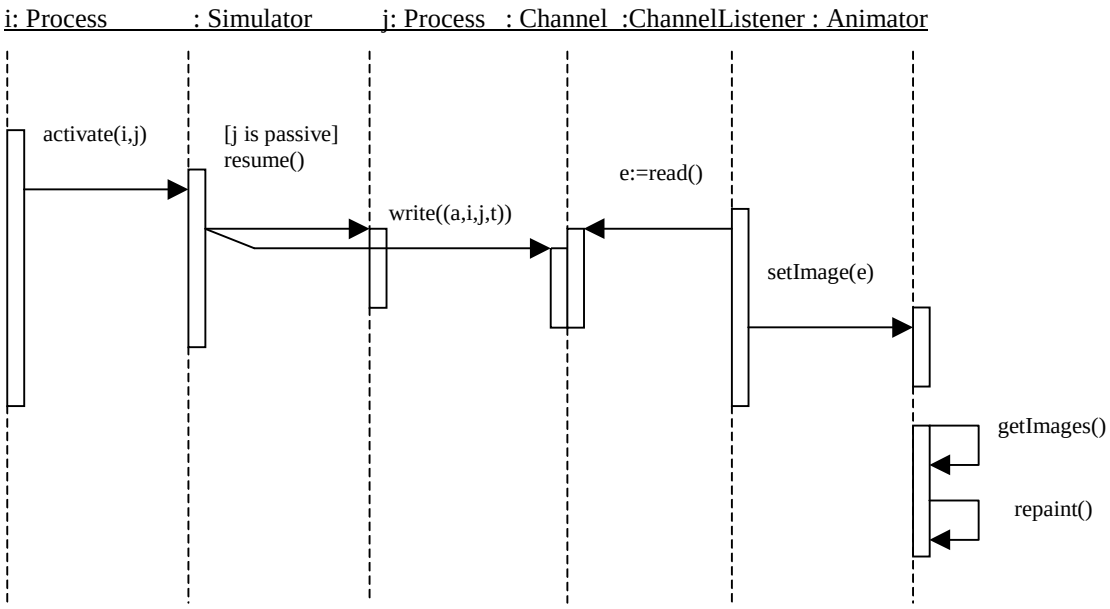
Vi skal se på interaksjonen mellom klassene for å utføre animasjonen av en hendelse i simuleringsmodellen. Som eksempel skal vi bruke hendelsen ”prosess i aktiviserer prosess j på tidspunkt t”. Interaksjonen beskrives ved hjelp av et UML sekvens diagram.



Prosess i starter interaksjonen ved å sende en melding activate(i,j) til simulatoren på tidspunkt t=i.now(). Simulatoren vil prøve å aktivisere j ved å sende en melding resume() til j når system-tid=t. Samtidig vil simulatoren sende en melding write((a,i,j,t)) til animatoren via kanalen. Animatoren leser kanalen ved å sende read() til kanalen. Read returnerer meldingen fra simulatoren i en variabel e. Simulatoren må vente til animatoren har lest kanalen før den kan fortsette. Animatoren vil på grunnlag av meldingen oppdatere bildene som skal fremstille prosessene på skjerm ved hjelp av setImage() og getImages(), og tegne den grafiske modellen ved hjelp av repaint(). Deretter kan den lese neste melding på kanalen. Denne teknikken gir

full synkronisering av simulatoren og animatoren på hendelse nivå. Hendelsene fremstilles på skjermen i den rekkefølgen de utføres i simulatoren.

En annen tilnærming går ut på å introdusere en class ChannelListener, som leser kanalen og frigjør denne oppgaven fra animatoren.



Animatoren utfører bare animasjonen, og kan frigjøres fra simulatoren. Når kanalleseren mottar en melding fra simulatoren, vil den sende en melding setImage() til animatoren for å oppdatere den grafiske modellen. Når modellen er oppdatert kan kanalleseren fortsette og leser neste melding fra kanalen. Animatoren henter bildene som skal fremstilles på skjermen ved hjelp av getImages(). Simulatoren vil fortsette straks kanalleseren har lest meldingen, mens animatoren fremstiller den grafiske modellen på skjerm med jevne mellomrom og uavhengig av simulatoren.

Forskjellen mellom de to tilnærmingene er at den første svarer til en hendelse-drevet animasjon, mens den siste svarer til en tids-steg animasjon. Hendelse-drevet animasjon vil si at animatoren fremstiller forløpet av simuleringen hendelse for hendelse. Tids-steg animasjon vil si at animasjonen viser øyeblikksbilder av tilstanden i simuleringsmodellen med jevne mellomrom. Tids-steg animasjon vil følge tilstanden i modellen i ”grove” trekk; alle tilstander vil ikke bli vist. Den relative hastigheten mellom simulatoren og animatoren vil ha innvirkning på hvilke tilstander en tids-steg drevet animasjon viser.

Konklusjon

Vi har gitt en kort beskrivelse av en teknikk for å vise forløpet av en simulering ved hjelp av en grafisk modell på skjerm. Simuleringsmodellen består av objekter som fremstilles av tilhørende grafiske objekt på skjermen. Forløpet av en simulering vises ved å angi de enkelte objektene og objekt tilstandene ved hjelp av grafiske figurer.

Simulatoren er basert på en objekt-orientert simuleringsmetode, der class Process utgjør hoved byggesteinen i modellen. Prosessene kan i prinsippet utføres i parallell, men en skedulerer kontrollerer fremdriften av simulert tid og hendelse rekkefølgen.

Animatoren kan fremstille forløpet av simuleringen enten hendelse for hendelse eller i faste tids-steg. Begge metodene har fordeler og ulemper. Fordelen med hendelse-drevet animasjon er at brukeren kan følge simuleringen i detalj hendelse for hendelse. Ulempen er at simuleringen kan bestå av så mange hendelser per tidsenhet at det er praktisk umulig å observere simuleringen over et lengre tidsintervall. Metoden kan brukes til å observere kritiske sekvenser av hendelser over avgrensede tidsintervaller. Fordelen med tids-steg animasjon er at brukeren selv kan definere frekvensen av øyeblikks bilder som skal fremstilles. På denne måten kan forløpet av simuleringen observeres over et lengre tidsintervall. Ulempen er at animasjonen bare gir et grovt inntrykk av forløpet av simuleringen. Metoden kan brukes til for eksempel å observere om tilstanden i en modell stabiliseres, om tilstanden fluktuerer, eller lignende.

Referanser

1. Communicating Sequential Processes, C.A.R. Hoare. Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).
2. Java Threads in the Light of occam/CSP, P.H.Welch, in Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG-21, pp. 259-284, IOS Press (Amsterdam), ISBN 90 5199 391 9, April 1998.
3. Java Communicating Sequential Processes - Paul Austin, University of Kent at Canterbury, UK, Early Access Release 0.5 released on 7 July 1998.
4. Java Communicating Sequential Processes (JCSP) Library, Peter Welch & al, University of Kent at Canterbury, UK
5. Communicating Threads for Java, G.H. Hilderink & al, University of Twente, Nederlands
6. A Distributed Real-Time Java System Based on CSP, The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000, Newport Beach, California, pp.400-407, March 15-17, 2000.
7. The Unified Modeling Language - Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, ISBN 0-201-57168-4, 1999.
8. The Java™ Language Specification, Java Team, James Gosling, Bill Joy and Guy Steele, Java™ Series, ISBN 0-201-63451-1, 1996