

Fladuno

Funktionel reaktiv programmering på indlejrede enheder

Martin Dybdal Troels Henriksen Jesper Reenberg

Datalogisk Institut, Københavns Universitet

16. juni 2009



Behovet for hændelser

Fundamentet for et reaktivt system er noget at reagere på.

Essensen af et reaktivt system

Hændelse → System → Respons

Begrebet “hændelse” dækker over en ekstern begivenhed, en ændring i miljøet, eller et signal sendt til systemet.

Det er pålagt at vi konstruerer en abstraktion der gør det praktisk at arbejde med hændelser.



Strategi

Der er to overordnede implementeringsstrategier:

Asynkron hændelser, hvor vi bruger hardwarefaciliteter (interrupts) til at acceptere hændelser, selv hvis vi er midt i en beregning (for eksempel, i at beregne en respons til en tidligere modtaget hændelse).

Synkron hændelser, hvor vi eksplicit checker om der er sket en ændring i systemets input der svarer til en hændelse, f.eks. hver eneste gang systemet ikke har andet at lave.

Synkron hændelser kan betyde at vi overser hændelser eller spilder ressourcer. Hardware-interrupts gør brug af elektroniske kredsløb der direkte er designet til at løse dette problem.



Asynkroner hændelser

Vi baserer hændelser på hardware interrupts, specifikt *Pin Change Interrupts*.

Dette betyder at hændelser er totalt knyttet til ændring af inputværdier på Arduino'ens I/O-pins.

Det er ikke muligt at have hændelser tilknyttet mere obskure interrupts, f.eks. timer-interrupts, som vi derfor håndterer specielt.



Asynkroner hændelser

Vi baserer hændelser på hardware interrupts, specifikt *Pin Change Interrupts*.

Dette betyder at hændelser er totalt knyttet til ændring af inputværdier på Arduino'ens I/O-pins.

Det er ikke muligt at have hændelser tilknyttet mere obskure interrupts, f.eks. timer-interrupts, som vi derfor håndterer specielt.

Vi kræver ikke en 1-til-1 mapping mellem interrupts og hændelser, idet visse hændelser kan svare til en værdi på blot en af adskillige I/O pins.



Asynkroner hændelser versus atomiske delgrafer

Problem

Asynkroner hændelser kan ankomme på vilkårlige og uforudsigelige tidspunkter, men vi garanterer også at atomisk delgrafer er atomiske.



Asynkrone hændelser versus atomiske delgrafer

Problem

Asynkrone hændelser kan ankomme på vilkårlige og uforudsigelige tidspunkter, men vi garanterer også at atomisk delgrafer er atomiske.

Løsningen er en hændelseskø:

- Når vi modtager en hændelse bliver den lagt på en hændelseskø.
- Hver gang vi er blevet færdige med en atomisk delgraf behandler vi det ældste element på hændelseskøen.

En følge er at vi ikke kan garantere en responstid på vores behandling af hændelser. Fladuno giver ingen real-time garantier.



Designovervejelser for hændelseskøen

- Indsættelse skal være (deterministisk) hurtigt, da interrupt-handlers skal terminere så hurtigt som muligt.
- Køen må ikke gro ubegrænset, idet vi kan ende med for lidt fri hukommelse til at evaluere de atomiske delgrafer.
- Det er nemmere at håndtere manglende plads til en ny hændelse i køen, end at håndtere at vi løber tør for stakplads under evalueringen af en atomisk delgraf.

Vi må have en statisk øvre grænse for køens størrelse.

Vi risikerer at gå glip af hændelser hvis køen er fyldt, men dette vil sjældent ske.

Under normal brug vil opfyldning af køen kun ske i sjældne øjeblikke, men

kørsel kan fortsætte. At korrumpere stakken under evalueringen af en atomisk delgraf vil sandsynligvis være en katastrofal fejl.



Implementering af hændelseskøen

Designparametre

- Hurtig indsættelse
- Statisk (maksimal-)størrelse

Den valgte løsning er en cirkulær liste implementeret over et C array med statisk størrelse. Element-indsættelse er garanteret til at terminere inden for et konstant antal instruktioner (hård realtime).



Målsætning

Hændelser har *typer* og *payloads*.

En hændelsesdefinition

- Liste af I/O pins
- Prædikاتفunktion der kaldes når en interrupt opstår på de angivne pins
- Funktion der kaldes til at give hændelsens payload

Payloads allokeres dynamisk:

- Nondeterministisk køretid (men sandsynligvis hurtig nok på Arduino)
- Der bruges ikke mere plads end der rent faktisk er af hændelses-payloads i hændelseskøen



Eksempel på definition

```
instance Event PushButtonPressEvent () where
  setupEvent e@(PushButtonPressEvent (d@(PushButton pin))) =
    do addDevice d
       pv <- statevar d "press_predicate"
       let v = H.Var (mkName pv)
           addCImport pv [$ty|() -> Bool|] [$cexp|$id:pv|]
           addCFundef [$cedec|int $id:pv () {
                                   return (digitalRead($int:pin) == HIGH);
                                   }|]
       return $ mkEvent e Nothing (Just v)
interruptPins (PushButtonPressEvent (PushButton pin)) = [DPin pin]
```



Eksempel på definition

```
instance Event PushButtonPressEvent () where
  setupEvent e@(PushButtonPressEvent (d@(PushButton pin))) =
    do addDevice d
       pv <- statevar d "press_predicate"
       let v = H.Var (mkName pv)
           addCImport pv [$ty|() -> Bool|] [$cexp|$id:pv|]
           addCFundef [$cedec|int $id:pv () {
                       return (digitalRead($int:pin) == HIGH);
                       }|]
       return $ mkEvent e Nothing (Just v)
interruptPins (PushButtonPressEvent (PushButton pin)) = [DPin pin]
```

Vi vedkender os at programmør-interface måske er lidt uoptimalt.



Eksempel på brug

```
onEvent (PushButtonPressEvent $ PushButton 2)  
  >>> toggle (diode 13)
```



Idle-waiters

Potentielt problem

I mange reaktive systemer har man behov for konstant *polling*. Brugere af timer-hændelser kan fylde hændelseskøen, hvilket kan medføre tab af videre (interessante!) hændelser.



Idle-waiters

Potentielt problem

I mange reaktive systemer har man behov for konstant *polling*. Bruger af timer-hændelser kan fylde hændelseskøen, hvilket kan medføre tab af videre (interessante!) hændelser.

- Det er sjældent vigtigt med eksakt timing for denne polling.
- *Idle-waiters* er atomiske delgrafer der evalueres (round-robin) når hændelseskøen er tom.
- Der er naturligvis ingen garanti på at hændelseskøen nogensinde er tom

