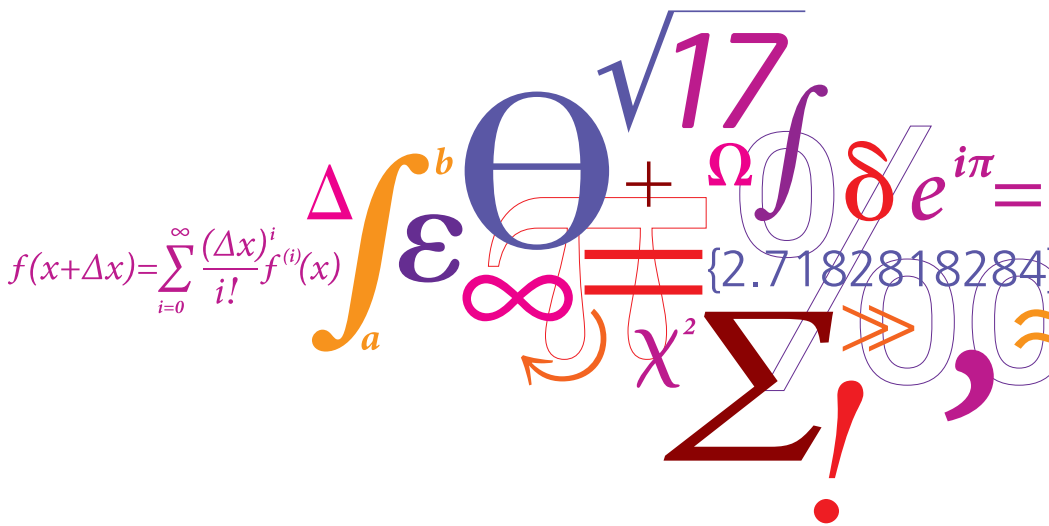
Construction of Simple Reactive programs

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

Asynchronous, reactive programs spend most of the wall-clock awaiting a request or a response from an external agent.

- The `async` computation expression (also called Asynchronous Workflows)
- Asynchronous computations can be used to make reactive, asynchronous programs with a very low resource demand
- A simple construction method, where a finite automaton specifies desirable event sequences
- The program skeleton for the dialogue program is systematically (almost automatically) derived from the automaton. Correctness

Goal: By the end of the day you have constructed a reactive program with a graphical user-interface with asynchronous operations.

Hopefully representable for a broad class of applications.

Some Terminology

- Asynchronous program: performs requests that are fulfilled at a later time. Meanwhile the program should work on other tasks.

  Examples: Network I/O, Web crawl

- Reactive program: spends most of the wall-clock time awaiting a request or a response from an external agent. A crucial problem for such a program is to minimize the resource demand while the program is waiting.

  Examples: GUI applications and web servers.

Focus here: Simple asynchronous, reactive programs where the system engage in a dialogue with a user.

The type: `Async('a)`

A value of type `Async<'a>` is an *asynchronous computation*.

- When started it becomes an executing task that either continues the current task or runs as an independent task.
- A terminating execution delivers a value of type `'a`.

Selected asynchronous operations:

webClient`.Async.DownloadString: Uri -> Async<string>` async.comp. to get WEB source determined by *webClient*. `Async.Sleep: int -> Async<unit>` `Async.Sleep` *n* `=` async.comp. sleeping *n* mS

`Async.Parallel: Seq<Async<'a>> -> Async<'a[]>`
`Async.Parallel [`c_0`; ...;`c_{n-1}`]`
= async.comp. of $c_0, \ldots, c_{n-1}$ put in parallel

`Async.RunSynchronously: Async<'a> -> 'a` Activates async.comp. Awaits completion.

Asynchronous computations by example

```
open System;;
open System.Net;;
let downLoadDTUcomp =
  async {
    let webCl = new WebClient()
    let! html = webCl.AsyncDownloadString(
                        Uri "http://www.dtu.dk")
    return html} ;;
 val downLoadDTUcomp : Async<string>
```

1. Create a `WebClient` object.
2. Initiate the download using `AsyncDownloadString`.
   Makes the task a wait item and will eventually terminate when the download has completed. It uses no thread while waiting.
3. At termination the rest of the computation is re-started with the identifier `html` bound to the result.
4. The expression `return html` returns the value bound to `html`, that is, the result of the download.

The expression: `async{ asyncExpr }`

The following constructs are (amongst others) allowed in *asyncExpr*:

- `let! v = e`
  `body`

Reads: perform the asynchronous operation e, and bind the result to v when it completes.

- `return e`

Reads: evaluate the expression e and return the result.

- `return! e`

Reads: execute e as an asynchronous computation and return the result.

- `do! e`

is equivalent to `let! _ = e`

Parallel downloads of web pages

```
let downloadComp url =
    let webCl = new WebClient()
    async {let! html = webCl.AsyncDownloadString(Uri url)
           return html};;
```

A computation for parallel downloads:

```
let downlArrayComp (urlArr: string[]) =
    Async.Parallel (Array.map downloadComp urlArr);;
val downlArrayComp : string [] -> Async<string []>
```

Activation of the computation:

```
let paralDTUandMScomp =
  downlArrayComp
    [|"http://www.dtu.dk"; "http://www.microsoft.com"|];;

Array.map (fun (s:string) -> s.Length)
          (Async.RunSynchronously paralDTUandMScomp);;
val it : int [] = [|45199; 1020|]
Real: 00:00:02.235,  CPU: 00:00:00.046
```

Uses limited CPU time.

Premature termination: Exception or Cancellation

A task executing an `async` computation reacts in case of an *exception* or a *cancellation* by calling the corresponding *continuation*.

- A continuation is a function for “the rest of the computation”
- A cancellation is requested (from outside the task) by setting a *cancellation token*. (A cancellation token is polled regularly)

A library function:

```
Async.StartWithContinuations:
   Async<'T>
   * ('T -> unit)                                  // normal term.
   * (exn -> unit)                                 // exception
   * (OperationCanceledException -> unit) // cancellation
   * CancellationToken                      -> unit
```

Activates an asynchronous computation with specified continuations and (optional) cancellation token.

Does not await completion of computation.

Simple examples

```
let okCon(s: string) = printf "Length = %d\n" (s.Length)
let exnCon _ = printf "Exception raised"
let canCon _ = printf "Operation cancelled"
```

Normal termination:

```
Async.StartWithContinuations
 (downloadComp "http://www.microsoft.com",
  okCon, exnCon, canCon);;
val it : unit = ()
Length = 1020
```

Simple examples cont'd

Termination due to an exception:

```
Async.StartWithContinuations
 (downloadComp "ppp", okCon, exnCon, canCon);;
val it : unit = ()
Exception raised
```

Termination due to a cancellation:

```
open System.Threading;;    // CancellationTokenSource
let ts = new CancellationTokenSource()
Async.StartWithContinuations
  (downloadComp "http://www.dtu.dk",
   okCon, exnCon, canCon, ts.Token);;
val it : unit = ()

ts.Cancel();;
Operation cancelled
```

Challenge

Consider a system engaging in a dialogue with a user comprising:

- Input events from a user.
- Status events from asynchronous computations.

How should the interactions be specified?

How can we design a dialogue program so that

- just the specified event sequences are accepted?

Correctness

Suggestion

- Interactions are specified by a finite automaton *A*
- The program design follows the structure of *A*

by use of an asynchronous event queue `ev` supporting the operations

- *ev*`.Post` *msg*: inserts the element *msg* in the event queue *ev*.
- *ev*`.Receive()`: Awaits the next element in the event queue *ev*.

Don Syme kindly provided the class `AsyncEventQueue`

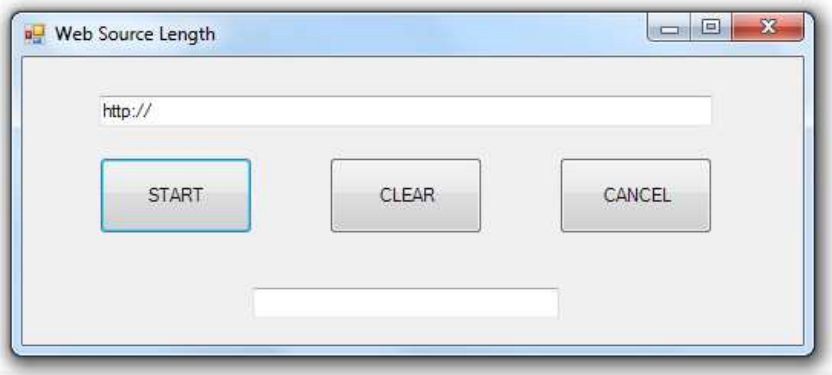
Reactive program: a simple example

Consider a reactive program to find lengths of HTML-sources of web-pages: The input events are:

`Start` *url* `:` Starts the download of the web-page.

`Clear:` Clears the text boxes.

`Cancel:` Cancels a progressing download.



The status events are

`Web` *html*: The result of a download

`Cancelled`: The downloading was cancelled

`Error`: An exception occurred

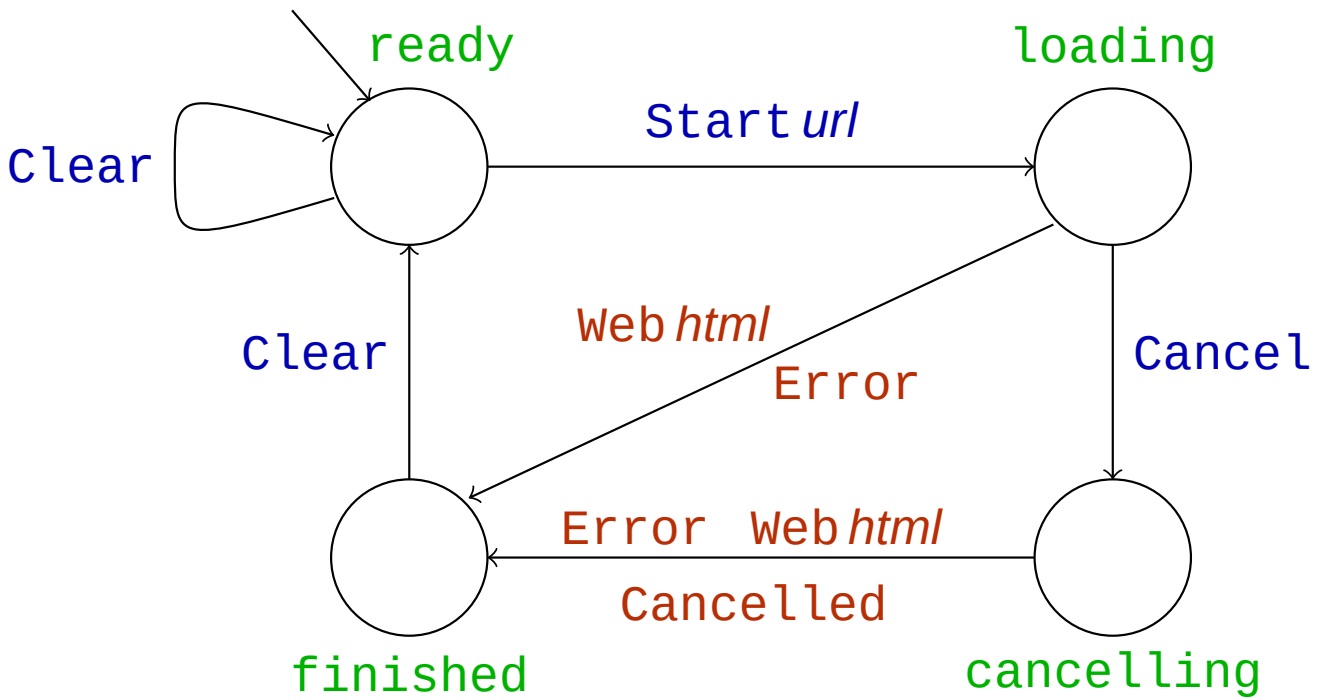
Output from the program appears in text boxes.

We abstract from output in the specification of the interaction.

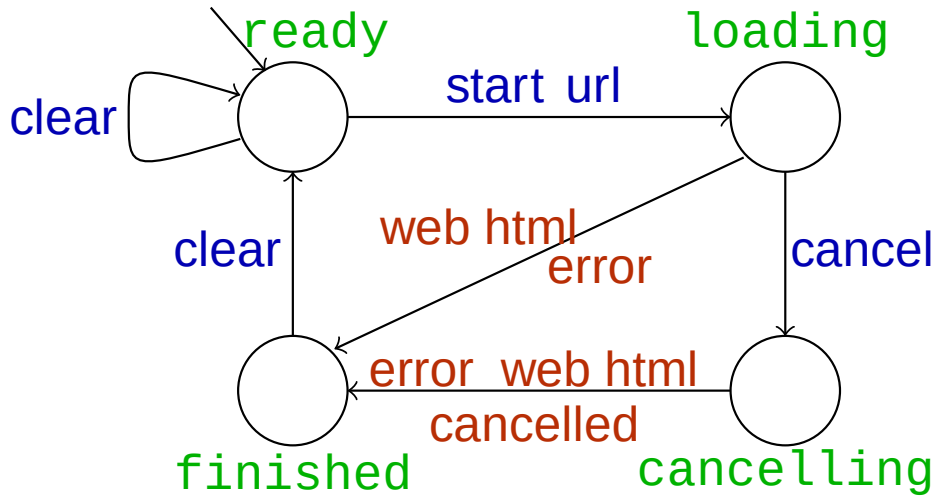
Modelling the dialogue

A finite automaton describes the desirable event sequences:

- States: `finished`, `cancelling`, `ready`, `loading`
- Input events: `Start` *url*, `Clear`, `Cancel`
- Status events: `Web` *html*, `Cancelled` and `Error`

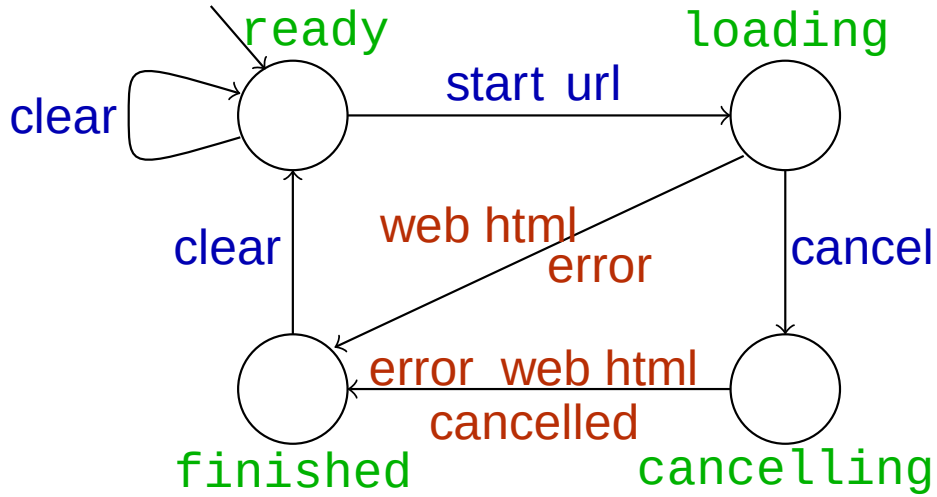


Skeleton of dialogue program: ready




```
let ready() =
    async { .... // actionReady: actions for incoming events
            disable [cancelButton]
            let! msg = ev.Receive()
            match msg with
            | Start url -> return! loading(url)
            | Clear     -> return! ready()
            | _         -> failwith("ready: unexpected message")}
```

Skeleton of dialogue program: loading




```
and loading(url)  =
    async { .... // actionLoading: actions for incoming events
            disable [startButton; clearButton]
            let! msg = ev.Receive()
            match msg with
            | Web html -> let ans = "Length = " +  ....
                                    return! finished(ans)
            | Error    -> return! finished("Error")
            | Cancel   -> ts.Cancel()
                          return! cancelling()
            | _        -> failwith("loading: unexpected message")}
```

Actions for incoming event: loading(url)

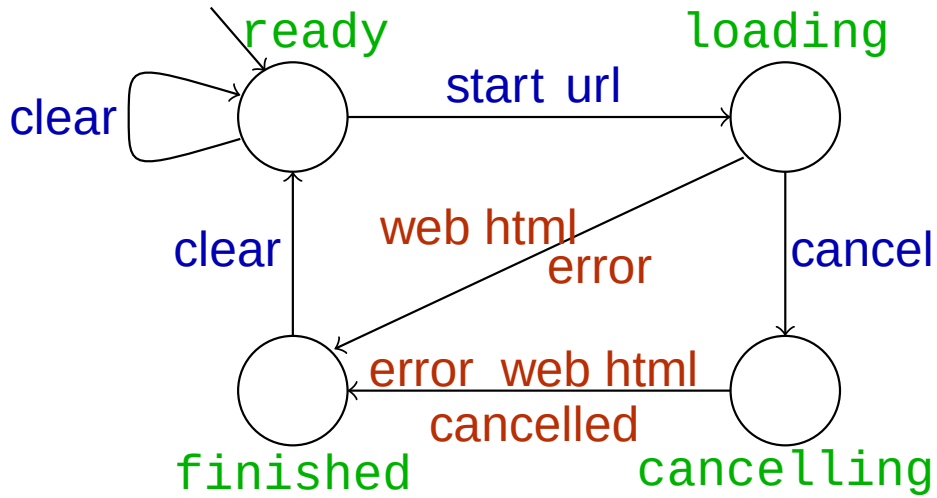
- the text box for the answer is set
- an asynchronous download with continuations is started

```
ansBox.Text <- "Downloading"
use ts = new CancellationTokenSource()

Async.StartWithContinuations
(async { let webCl = new WebClient()
let! html = webCl.AsyncDownloadString(Uri url)
return html },
 (fun html -> ev.Post (Web html)), // normal termination
 (fun _ -> ev.Post Error),         // error (e.g. wrong url)
 (fun _ -> ev.Post Cancelled),     // cancellation
ts.Token)
```

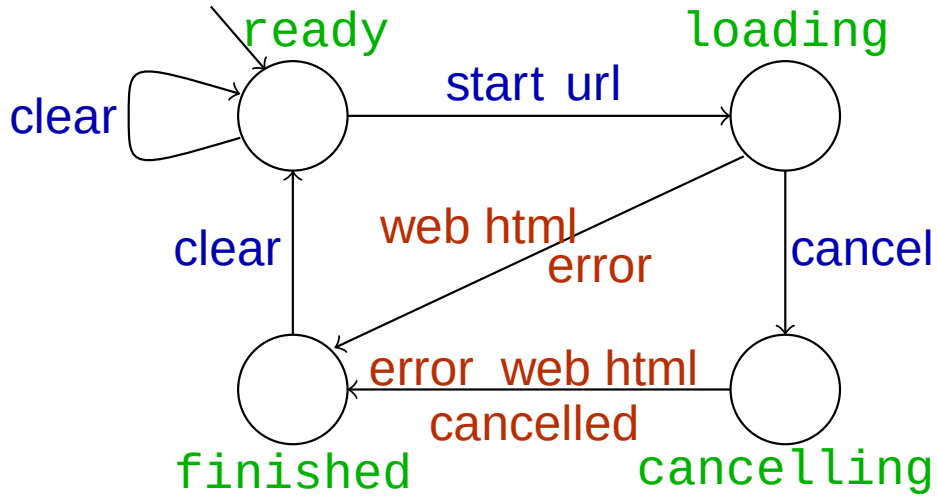
Incoming actions for the other states are simpler.

Skeleton of dialogue program: cancelling




```
and cancelling() =
    async { .... // actionCancelling: actions for incoming events
           disable [startButton; clearButton; cancelButton]
           let! msg = ev.Receive()
           match msg with
           | Cancelled | Error
           | Web _ -> return! finished("Cancelled")
           | _     ->  failwith("cancelling: unexpected message")}
```

Skeleton of dialogue program: finished




```
and finished(s) =
    async { .... // actionFinished: actions for incoming events
            disable [startButton; cancelButton]
            let! msg = ev.Receive()
            match msg with
            | Clear -> return! ready()
            | _     -> failwith("finished: unexpected message")}
```

Summary

Asynchronous, reactive programs spend most of the wall-clock awaiting a request or a response from an external agent

- The `async` computation expression was introduced
- It was shown how asynchronous computations can be used to make reactive, asynchronous programs with a very low resource demand
- A simple construction method was introduced, where a finite automaton specifies desirable event sequences
- The program skeleton for the dialogue program is systematically (almost automatically) derived from the automaton.
- An bachelor project used the same recipe as described here for the dialogue with the user in an app for an Android phone.

The combination of asynchronous and functional constructs is powerful