Synchronous Programming

- Two simple ways of implementing reactive systems:
  - Event-driven approach
  - Sampling

```
<Initialize Memory>
Foreach input_event do
  <Compute Outputs>
  <Update Memory>
End
```

```
<Initialize Memory>
Foreach period do
  <Read Inputs>
  <Compute Outputs>
  <Update Memory>
End
```

Event-driven Programming

Sampling Programming
Synchronous Programming

- Program typically implements an automaton:
  - state: valuations of memory
  - transition: reaction, possibly involving many computations

- Synchronous paradigm: reactions are considered atomic, i.e., they take no time. (Computational steps execute like combinatorial circuits.)

- Synchronous broadcast: instantaneous communication, i.e., each automaton in the system considers the outputs of others as being part of its own inputs.

- Atomic reactions are called instants.
Overview

• **StateCharts:**
  - First, and probably most popular formal language for the design of reactive systems.
  - Focus on specification and design, not designed as a programming language.
  - Determinism is not ensured.
  - No standardized semantics.

• Programming languages for designing reactive systems:
  - **ESTEREL** [Berry]: imperative language.
  - **LUSTRE** [Caspi, Halbwachs], **SIGNAL** [Le Guernic, Beneviste]: data-flow languages.

• **ARGOS**: purely synchronoose variant of StateCharts.
LUSTRE

• Based on synchronous data-flow model:
  – Each variable takes a value at every cycle of the program.

• Programs are structured into nodes:
  – Node: subprogram defining its output parameters as functions of its input parameters.
  – Definition given by unordered set of equations.

• Variables are defined via equations, e.g. $X = E$ with variable $X$ and expression $E$.

• Expressions:
  – identifiers,
  – constants,
  – arithmetic, boolean and conditional operators,
  – 'previous' operator $pre$,
  – 'followed by' operator $\rightarrow$.
LUSTRE

• Specific Operators:
  - \((\text{pre}(E))_0 = \text{nil} \) (undefined)
  - \((\text{pre}(E))_n = E_{n-1}\)
  - \((E\rightarrow F)_0 = E_0\)
  - \((E\rightarrow F)_n = F_n\)
  - Example: \(x = 0\rightarrow(\text{pre}(x)+y)\)

\[s_n = 2*(x_n+y_n)\]
LUSTRE – Example Program

node Counter (init, incr: int; reset: bool)
  returns (count:int);
let
  count = init -> if reset then init
  else pre(count) + incr;
tel
ESTEREL - Principles

• Imperative language.

• Tailored for programming hardware or software synchronous controllers dominated by control-handling aspects.

• Most ESTEREL statements are conceptually instantaneous, i.e., are executed in the same reaction than other statements that sequentially precede or follow them in the program.
module Speedometer:
  input Second, Meter;
  output Speed: integer in
  loop
    var Distance:=0: integer in
    do
      every Meter do
        Distance:=Distance+1
      end every
      upto Second;
    emit Speed(Distance)
  end var
  end loop
end module

module SpeedSupervisor:
  input Second, Meter;
  output TooFast in
  signal Speed: integer in
  [ run Speedometer
    ||
    every Speed do
      if ?Speed > MaxSpeed
        then emit TooFast
      end if
    end every
  ]
end signal
end module
Compilation of Synchronous Languages

• Causality Analysis
  - Causality problem: the presence of a signal seems to depend on itself (problem of combinatorial loops in synchronous circuits).
  - Goal: have one (reactivity) and only one (determinism) consistent solution for each configuration of input signals.
  - Example situations:

    module P1:
    output O;
    present O
    else emit O
    end present
    end module

    module P2:
    output O;
    present O
    then emit O
    end present
    end module

    module P3:
    input I;
    output O;
    signal S in
    present I then emit S end
    ||
    present S then emit O end
    end signal
    end module

    inconsistent non-deterministic correct
Compilation of Synchronous Languages

module P4:
output O1, O2;
  present O1 then emit O1 end
 ||
  present O1 then
    present O2 else emit O2 end
  end present
end module

Logically correct, but rejected by *Constructive Causality*:
no constructive explanation for solution.
Compilation of Synchronous Languages

• Sequential code generation
  - LUSTRE:
    • Generating single loop, after sorting the equations according to their dependences.
  - ESTEREL:
    • Compilation of control part into explicit automaton (ESTEREL -V2 and -V3 compilers).
      - efficient, but
      - possibly exponential expansion of code size.
    • Single loop code generation
      (ESTEREL-V4, and -V5 compilers).
Now: Esterel in more Depth

• Syntax and intuitive semantics

• Causality

• Documentation and Esterel-Distribution can be downloaded from
  www.cs.uni-sb.de/~kaestner/ES0203.html
Esterel: General Structure

module M:
    input names;
    output names;
    statement
end module

- Interface declaration specifies which objects a module exports or imports:
  - Data objects, which are declared abstractly in Esterel. Their actual value is given in the host language.
  - Signals and sensors. Host objects implementing them depend on the host language.

- Body is an executable statement.
module WATCH:

input UL, UR, LL, LR;
type Time;
constant Noon: Time;
constant WordLength = 16: integer;
function CompareTime (Time, Time): boolean;
procedure IncrementTime (Time) (integer);
type Beep;
output Beeper: Beep;
output CurrentTime := Noon : Time;
Signals and Sensors

- Interface signals or local signals, declared by the `signal` statement (see later).
- Instantaneously broadcast throughout the program.
- Pure signals: status is `present` or `absent`.
- In addition to their status, valued signals carry a value of arbitrary type.
- One predefined signal `tick`:
  - pure signal
  - represents activation clock of the reactive program
  - Status is `present` in each instant.

- Sensors have a value but no status;
  Example: `sensor temperature : integer;`
Statements

- A statement starts in some instant t, remains active for a while, and may terminate in some instant t >= t'.
  - t=t': statement is instantaneous
  - t'>t: statement takes time
Kernel Statements

- Selection of basic statements, most other statements can be programmed with:
  - nothing
  - pause
  - emit $S$
  - $p ; q$
  - $p \parallel q$
  - present $S$ then $p$ else $q$ end
  - suspend $p$ when $S$
  - loop $p$ end
  - trap $T$ in $p$ end
  - exit $T$
  - signal $S$ in $p$ end
Informal Semantics

• Basic pure control statements:
  - nothing: does nothing, ie terminates instantaneously when started.
  - pause: pauses when started and terminates in the next instant.

• Signal emission:
  - emit $S$: emits signal $S$ (ie sets its status to present) and terminates instantaneously.
  - emit $S(e)$: evaluates the data expression $e$, emits $S$ with that value and terminates instantaneously.
  - Valid for the current instant only; happens only once.
Informal Semantics

• Sequencing \((p ; q)\):
  - \(p\) is instantaneously started when \(p ; q\) is started and is executed up to completion or trap exit.
  - If \(p\) terminates, \(q\) is immediately started and the sequence behaves as \(q\) from then on.
  - If \(p\) exists via traps, the exits are immediately propagated and \(q\) is never started.

  - Example: emit S1; emit S2
Informal Semantics

• Parallel Statement ( \( p \parallel q \)):
  - Denotes explicit concurrency
  - Any signals emitted are instantaneously broadcast to all branches in each instant.
  - The sequencing operator ; binds tighter than \( \parallel \).
  - Upon start both branches \( p \) and \( q \) are instantaneously started.
  - It terminates in the precise instant where both branches are terminated (branch synchronization).

- Example: \( p; q\parallel r \) vs \([p; q]\parallel r \) vs \( p;[q\parallel r] \)
Informal Semantics

• present $S$ then $p$ else $q$ end
  - immediately starts $p$ if the signal $S$ is present, otherwise $q$ is immediately started.

• suspend $p$ when $s$
  - $s$ is a signal expression (see later)
  - When the suspend statement starts, $p$ is immediately started.
  - $s$ has no effect in the initial instant in which the statement becomes active.
  - If the signal expression $s$ is true, $p$ remains in its current state and the suspend statement pauses for the instant.
  - If $s$ is false, $p$ is executed for the instant. If $p$ pauses, terminates or exits a trap, so does suspend.
Informal Semantics

• loop $p$ end loop:
  - $p$ is instantaneously restarted anew upon termination.
  - $p$ must never be able to terminate instantaneously when started. Note: the condition check is static!
  - If $p$ exists some enclosing trap, the loop is exited.

  trap $T$
  
  loop $p$ end loop || present $S$ then exit $T$
  end trap

  - Example: loop emit $S$ end loop (not allowed)
Static Termination Check

loop
    present I then
        present J else
            p
        end present
    else
        q
    end present;
end loop

Program is rejected!
Informal Semantics

• trap $T$ in $p$ end
  - A trap defines a lexically scoped exit for $p$.
  - $p$ is immediately started when the trap statement starts.
  - If $p$ terminates so does the trap statement.
  - exit $T$ (occurring inside of $p$) causes $T$ to terminate immediately.
  - When traps are nested, the outer one takes priority.

trap $U$ in
  trap $T$ in
    $p$
  end trap;
q
end trap;
r

• $p$ exits $T$: $q$ is immediately started
• $p$ exits $U$: $r$ is immediately started
• $p$ exits $T$ and $U$ simultaneously: $U$ takes priority.
Informal Semantics

• Local signal declaration: signal $S$ in $p$ end signal
  - Signal $S$ is local to $p$.
  - Scoping is lexical, ie any redeclaration of a signal hides the outer declaration.
  - A local signal placed within a loop can be executed several times in the same instant. Then each execution declares a new copy / incarnation of the signal.
  - Example:

    ```
    signal Alarm,
      Distance : integer,
    in
      p
    end signal
    ```
Further Statements

- Third basic control statement:
  halt: pauses forever and never terminates.
- sustain S / sustain S(e): continuous emission of signal
- Assignment (instantaneous):
  \( X := e \) where \( X \) is a variable and \( e \) is a data expression
- Procedure call (instantaneous): call \( P(X,Y)(e_1,e_2) \)
- repeat \( e \) times
  \( p \)
  end repeat

- Local variable declaration:

```plaintext
var X : double,
    Count := ?Distance : integer
in
    \( p \)
end var
```
Further Statements

• if Data Test: if $e$ then $p$ else $q$ end if
  – $e$ is a data expression: The conditions are evaluated in sequence.

• await $d$
  – $d$ is a delay expression
  – The delay is started when await is started. The statement pauses until the delay elapses and terminates in that instant.

• abort $p$ when $d$
  – $p$ is run until termination or until the delay $d$ elapses.
  – If $p$ terminates before the delay elapses, so does the abort statement. Otherwise, $p$ is preempted when the delay elapses.
  – strong abort vs. weak abort:
    • strong abort: If the delay elapses before termination of $p$, $p$ is preempted and not executed.
    • weak abort: If the delay elapses before termination, $p$ is preempted and executed for a last time.
# Valued Signals vs. Variables

<table>
<thead>
<tr>
<th>The value can be changed only if the status is present. Unlike the status, the value is permanent: if it is unchanged in an instant, its value is that of the previous instant.</th>
<th>The value of a variable is written by an instantaneous assignment statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A valued signal has exactly one status and exactly one value at a time. Both the status and the value are broadcast.</td>
<td>Unlike a signal a variable can take several successive values in an instant. The order in which the values are taken is the internal control flow of the program, the so-called constructive order.</td>
</tr>
<tr>
<td>A signal is shared throughout its scope (whole program for interface signal and the scope of its declaration for a local signal)</td>
<td>A variable is local to a thread in case the thread writes it. If the thread forks on `</td>
</tr>
</tbody>
</table>
Valued Signals vs Variables

- **Forbidden:**
  - $x := 0$
  - $x := x + 1$ || $x := 1$

- **Forbidden:**
  - $\text{emit } S(?S+1)$

- **Allowed:**
  - $\text{emit } S(1)$ || $\text{emit } S(2)$

- **Allowed:**
  - $x := x + 1$