Synchronous Programming of Reactive Systems

Nicolas Halbwachs
Verimag/CNRS
Grenoble

Outline

- Introduction
- The Data-Flow Language Lustre
- The Imperative Language Esterel
- Compilation of Synchronous Languages
- Verification and Automatic Testing of Synchronous Programs
- Other Topics and Current Trends

Introduction

Reactive systems
- How are reactive systems commonly implemented?
- The synchronous model
- Synchronous languages

Reactive Systems

Permanent reaction to an environment that cannot wait
Embedded systems
- e.g., transportation, industrial control

Logical concurrency

ex. A digital watch:
- time keeper
- alarm
- stopwatch
- display manager
- button handler

Design these modules separately, compose them concurrently.

Specific features
- deterministic
- concurrent (logical ≠ physical)
- safety critical
Usual (asynchronous) languages for concurrency don’t work

Example: Every 60 seconds, emit a signal MINUTE
An attempt in ADA style:

```
task A : loop
  delay 60; B.MINUTE!
end
```

- Rendez-vous (symmetric communication) doesn’t work.
- Non-deterministic scheduling (asynchronous interleaving) doesn’t work.
- No broadcasting

How are reactive systems commonly implemented?

Simple implementation (event driven)
```
< Initialize Memory >
foreach input event do
  < Compute Outputs >
  < Update Memory >
end
```

Even simpler implementation (periodic sampling)
```
< Initialize Memory >
foreach period do
  < Read Inputs >
  < Compute Outputs >
  < Update Memory >
end
```

∼ interpreted automaton
- a loop iteration = a transition = a logical instant

Our example in Esterel:
```
every 60 SECOND do
  emit MINUTE
end
```

"Real-time" correctness condition
- max transition time < min environment delay

Synchronous programming
- high level, structured, modular
description of interpreted automata
- concurrency = synchronous product

Another point of view:
time, concurrency, and compositionality

```
\Delta(f(x))?
```

depends on implementation of \( f \), on the target machine, and generally on \( x \)

Abstraction: \( \Delta(f(x)) = \delta \)
Compositionality: \( f(x) = g(h(x)) \)
\[
\Delta_f = \Delta_g + \Delta_h \quad \delta = \delta + \delta
\]
Another point of view: time, concurrency, and compositionality

\[ \Delta(f(x)) \]
depends on implementation of \( f \), on the target machine, and generally on \( x \)

Abstraction: \( \Delta(f(x)) = \delta \)

Compositionality: \( f(x) = g(h(x)) \)

\[ \Delta_f = \Delta_g + \Delta_h \quad \delta = \delta + \delta \]

Two solutions:

\( \delta = 0 \) (synchrony), \( \delta = ? \) (asynchrony)

Concrete behavior

\[
\begin{align*}
\Delta_1 & \quad \Delta_2 \\
\delta_1 & \quad \delta_2 \\
\delta_3 & \\
\end{align*}
\]

Valid abstraction as long as \( \delta_i < \Delta_i \)

What’s new?

Classical in synchronous circuits

- synchronous communicating Mealy machines
- dynamic Boolean equations
- gate and latch networks

Classical in control engineering

- data-flow synchronous formalisms
  - differential or finite difference equations
  - block-diagrams, analog networks

Connexion with synchronous circuits (1/2)

\( (O_n, S_n) = F(L_n, S_{n-1}) \)

Data-flow languages (Lustre/Scade, Signal/Sildex)

In Lustre: \( (O, S) = F(I, \text{pre}(S)) \)

Connexion with synchronous circuits (2/2)

Parallel composition

\( (S1, L1, M1) = F1(E1, L2, \text{pre}(M1)) \)
\( (S2, L2, M2) = F2(E2, L1, \text{pre}(M2)) \)

Deterministic! (provided there is no combinational loop)
**Introduction**

The synchronous model

Connexion with synchronous automata (1/3)

Connexion with synchronous automata (2/3)

Connexion with synchronous automata (3/3)

---

**Imperative Languages (Esterel, Synccharts)**

In Esterel:

- every 60 sec do emit min end
- every 60 min do emit hour end

**Synchronous Languages**

- **Imperative**
  - StateCharts
  - Esterel
  - Argos, SyncChart

- **Declarative**
  - Lustre, Signal

---

**Industrial use**

- **Avionics, aerospace:**
  - Airbus, Honeywell, Eurocopter, Astrium (Lustre/Scade)
  - Dassault (Esterel)
  - Hispano-Suiza (Signal)

- **Nuclear plants:**
  - Rolls-Royce CN, EDF (Lustre)

- **CAD:**
  - Cadence, Synopsys, TI (Esterel)

- **Telecom:**
  - TI (Esterel)

- **Many more...**

---

**The data-flow approach**

- The combinational part
- Temporal operators
- Examples
- Clocks
- Assertions
- The story of Lustre/Scade

---
The data-flow approach (1/3)

Classical in control theory (equations, data-flow networks) and circuits (equations, gate networks)

$$A = \frac{X+Y}{2};$$

Synchronous interpretation: time = $\mathbb{N}$

$$\forall n \in \mathbb{N}, A_n = \frac{X_n + Y_n}{2}$$

N. Halbwachs (Verimag/CNRS) Synchronous Programming 30 / 198

The data-flow approach (2/3)

Lustre (textual) and Scade (graphical)

N. Halbwachs (Verimag/CNRS) Synchronous Programming 31 / 198

The data-flow approach (3/3)

Other solution

$$\text{node Average (X, Y: int) returns (A: int);}$$

$$\text{let}$$

$$\text{A = S/2;}$$

$$\text{S = X+Y;}$$

$$\text{tel}$$

N. Halbwachs (Verimag/CNRS) Synchronous Programming 32 / 198

Equations and flows

System of equations

- One definition for each output/local variable
- Meaningless order
- Substitution principle (referential transparency)

Flows

- Each variable, or constant, or expression, represents an infinite sequence of values

$$X = x_0, x_1, \ldots, x_n, \ldots$$

- $x_n$ is the value of $X$ at the $n$-th cycle of the program

N. Halbwachs (Verimag/CNRS) Synchronous Programming 33 / 198

The combinational part of the language

Base types: bool, int, real

Constants

- $2 = 2, 2, 2, \ldots$
- $true = true, true, true, \ldots$

Pointwise operators

- standard arithmetic and logic operators

$$X+Y = x_0 + y_0, x_1 + y_1, \ldots$$

A Boolean example

$$\text{node Nand (X, Y: bool) returns (nand: bool);}$$

$$\text{let}$$

$$\text{nand = not (X and Y);}$$

$$\text{tel}$$

Execution:

<table>
<thead>
<tr>
<th>X</th>
<th>true true false true true ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>false true false false true ...</td>
</tr>
<tr>
<td>nand</td>
<td>true false true true false ...</td>
</tr>
</tbody>
</table>
Pointwise operators: the conditional operator

node Max (A, B: int) returns (max: int):
let
  max = if A >= B then A else B;
tel

Execution:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>10</th>
<th>8</th>
<th>25</th>
<th>12</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>
max | 5 | 10| 8  | 25| 17 | . . .|

Temporal operators

- "pre" (previous) operator
  One step delay:
  \[
  X \begin{bmatrix} x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ . \ . \ . \\ pre(X) \end{bmatrix}
  \]
  \[\text{pre}(X) \begin{bmatrix} nil \ x_0 \ x_1 \ x_2 \ x_3 \ . \ . \ . \end{bmatrix}\]

- "->" (followed-by) operator
  Initialization:
  \[
  X \begin{bmatrix} x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ . \ . \ . \\ Y \begin{bmatrix} y_0 \ y_1 \ y_2 \ y_3 \ y_4 \ . \ . \ . \end{bmatrix} \end{bmatrix}
  \]
  \[X -> Y \begin{bmatrix} x_0 \ y_1 \ y_2 \ y_3 \ y_4 \ . \ . \ . \end{bmatrix}\]

Formal semantics

- pointwise operators
  \[(\text{op}(X, Y, \ldots, Z))_i = \text{op}(X_i, Y_i, \ldots, Z_i)\]

- temporal operators
  \[(\text{pre}(X))_i = \begin{cases} \text{nil} & \text{if } i = 0 \\ X_{i-1} & \text{otherwise} \end{cases}\]

  \[(X -> Y)_i = \begin{cases} X_0 & \text{if } i = 0 \\ Y_i & \text{otherwise} \end{cases}\]

Simple examples

Rising edge of a Boolean flow

node Edge (B: bool) returns (edge: bool):
let
  edge = false -> B and not pre(B);
tel

Min and Max values of a sequence

node MinMax (X: int) returns (min, max: int):
let
  \begin{align*}
  \text{min} &= X -> \text{if } X < \text{pre}(\text{min}) \text{ then } X \\
  \text{else } \text{pre}(\text{min}); \\
  \text{max} &= X -> \text{if } X > \text{pre}(\text{max}) \text{ then } X \\
  \text{else } \text{pre}(\text{max}); \\
  \end{align*}
tel

Recursive definitions

Min and Max values of a sequence (cont.)

Other writing (definitions of pairs):

node MinMax (X: int) returns (min, max: int):
let
  \begin{align*}
  (\text{min}, \text{max}) &= (X,X) -> \text{if } X < \text{pre}(\text{min}) \text{ then } (X, \text{pre}(\text{max})) \\
  \text{else if } X > \text{pre}(\text{max}) \text{ then } (\text{pre}(\text{min}),X) \\
  \text{else } \text{pre}(\text{min},\text{max}); \\
  \end{align*}
tel

Recursive definitions
Correct recursive definitions

Example

alt = false -> not pre(alt);

The sequence of values can be computed step-by-step:

- \( alt_0 = false_0 = false \)
- \( alt_1 = (not\ pre(alt))_1 = not\ alt_0 = true \)
- ...

Correct recursive definitions

Example

alt = false -> not pre(alt);

The sequence of values can be computed step-by-step:

- \( alt_0 = false_0 = false \)
- \( alt_1 = (not\ pre(alt))_1 = not\ alt_0 = true \)
- ...

Incorrect recursive definitions

Example

\[ X = \frac{1}{2-X}; \]

- Unique solution \( X=1 \) but cannot be obtained constructively.
- Any dependence loop (not cut by a "pre") is rejected by the compiler ("causality error"), even:

\[ X = if\ C\ then\ Y\ else\ Z; \]
\[ Y = if\ C\ then\ W\ else\ X; \]

Double initialization

Define \( P \equiv F, F, T, F, F, T, F, F, T, F, F, \ldots \)

- \( P = false \rightarrow pre(Q); \)
- \( Q = false \rightarrow not\ pre(Q); \)

Warning: \( X \rightarrow Y \rightarrow Z = X \rightarrow Z \)

Double initialization

Define \( P \equiv F, F, T, F, F, T, F, F, T, F, F, \ldots \)

- \( P = false \rightarrow pre(Q); \)
- \( Q = false \rightarrow not\ pre(Q); \)

Warning: \( X \rightarrow Y \rightarrow Z = X \rightarrow Z \)

Double initialization

Define \( P \equiv F, F, T, F, F, T, F, F, T, F, F, \ldots \)

- \( P = false \rightarrow pre(Q); \)
- \( Q = false \rightarrow not\ pre(Q); \)

Warning: \( X \rightarrow Y \rightarrow Z = X \rightarrow Z \)

Define \( fib \equiv 1, 1, 2, 3, 5, 8, 13, \ldots \)

- \( fib = 1 \rightarrow pre(f); \)
- \( f = 1 \rightarrow (fib + pre(fib)); \)

or (substitution)

- \( fib = 1 \rightarrow pre(1 \rightarrow (fib + pre(fib))); \)

Double initialization

Define \( P \equiv F, F, T, F, F, T, F, F, T, F, F, \ldots \)

- \( P = false \rightarrow pre(Q); \)
- \( Q = false \rightarrow not\ pre(Q); \)

Warning: \( X \rightarrow Y \rightarrow Z = X \rightarrow Z \)

Define \( fib \equiv 1, 1, 2, 3, 5, 8, 13, \ldots \)

- \( fib = 1 \rightarrow pre(f); \)
- \( f = 1 \rightarrow (fib + pre(fib)); \)

or (substitution)

- \( fib = 1 \rightarrow pre(1 \rightarrow (fib + pre(fib))); \)

Counter

Write a node

\[ \text{node Counter (B: bool) returns (count: int);} \]

where \( \text{count} \) is the number of \textit{true} occurrences of \( B \)
Examples

Counter

Write a node

\[ \text{node Counter (B: bool) returns (count: int);} \]

where \( \text{count} \) is the number of \text{true} occurrences of \( \text{B} \)

\[ \text{node Counter (B: bool) returns (count: int);} \]

\[ \text{let} \]

\[ \text{count} = 0 \rightarrow \text{if B then pre(count) + 1 else pre(count);} \]

\[ \text{tel} \]

misses possible initial occurrence of \( \text{B} \)

N. Halbwachs (Verimag/CNRS) Synchronous Programming 48 / 198

Counter (cont.)

Better solution:

\[ \text{node Counter (B: bool) returns (count: int);} \]

\[ \text{let} \]

\[ \text{count} = \text{if B then (1 \rightarrow \text{pre(count)} + 1) else (0 \rightarrow \text{pre(count)});} \]

\[ \text{tel} \]

N. Halbwachs (Verimag/CNRS) Synchronous Programming 49 / 198

Exercise: Counter with reset

Write a node

\[ \text{node RCounter (B, reset: bool) returns (count: int);} \]

which does the same, but is reset to 0 when "reset" is true

N. Halbwachs (Verimag/CNRS) Synchronous Programming 50 / 198

Exercise: Counter with reset (cont.)

N. Halbwachs (Verimag/CNRS) Synchronous Programming 51 / 198

Another example: an integrator

\[ f(t) \rightarrow \text{Integ} \]

\[ F(t) = \int_0^t f(\tau) d\tau \]

N. Halbwachs (Verimag/CNRS) Synchronous Programming 52 / 198
Another example: an integrator

\[ F(t) = \int_0^t f(\tau) d\tau \]

Program structure

Any defined node can be re-used, as a basic operator.

Called as a function:

```
nb_seconds =
  RCounter( second,
            second and (false -> pre(nb_seconds = 59) ));
```

Other example: Switches

Write a node TwoStates receiving 3 Boolean inputs init, set, and reset, and behaving like a switch or a flip-flop:

Its boolean output state

- is initially equal to init,
- is set to true when set is true,
- is reset to false when reset is true,
- keeps its previous value otherwise.

Switches (cont.)

First solution

```
node TwoStates(init, set, reset: bool)
returns (state: bool);
let
  state = init -> if set then true
           else if reset then false
           else pre(state);
```

```
Problem: what happens with the call
TwoStates(false, change, change)?
```
Switches (cont.)

Can you fix this problem?

Mouse click (cont.)

click
wait
< d

double = wait and click

click
wait = d
expire
single

single = expire and not click

Mouse click: complete program

node Mouse (d: int; click: bool)
returns (single, double: bool);
var wait, expire: bool; waiting_delay: int;
let single = expire and not click;
double = wait and click;
expire = false -> pre(waiting_delay)=d-1;
wait = false ->
pre(TwoStates(click, click, click or expire));
waiting_delay = Counter(click or expire, wait);
tell

A complete example: Heating control

HControl

goal

heat

goal

HControl

Goal

heat

T

DEMO
Examples

A complete example: Heating control

node HControl (T, goal : real) returns (heat: bool);
let
heat = true -> if (T > goal + 1.) then false
else if (T < goal -1.) then true
else pre(heat);
tel

goal
heat
T

Heating control: simulating the environment

node Env (heat: bool; ExtT: real) returns (T: real);
var Tprime : real;
let
Tprime = if heat then alpha * (h-T) else beta * (ExtT -T);
T = Integ(Tprime, ExtT);
tel

T
Env
heat
Text
HControl
goal
T

Heating control: Simulation program

node Simul (goal, ExtT: real)
returns (heat: bool; T: real);
let
heat = HControl(T, goal);
T = Env(heat, ExtT);
tel

DEMO

Clocks

Clocks

Sampling: the "when" operator

Goal: define a flow "slower" that inputs

X 4 1 -3 0 2 7 8 3 12
C F T F F T T F T F

When C is false, X when C does not exist

One may only operate on flows with the same clock:

X + (X when C) is forbidden

Clocks (cont.)

Projection: the "current" operator

Goal: Coming back to a "faster" clock

X 4 1 -3 0 2 7 8 3 12
C F T F F T T F T F

When C is false, X when C does not exist

One may only operate on flows with the same clock:

X + (X when C) is forbidden

Clocks

Sampling: the "when" operator

Goal: define a flow "slower" that inputs

X 4 1 -3 0 2 7 8 3 12
C F T F F T T F T F

When C is false, X when C does not exist

One may only operate on flows with the same clock:

X + (X when C) is forbidden

Clocks: typical use
Clocks

Clocks: Initialization problem

Solution 1: clock initially true

\[
\begin{array}{c|cccccccc}
X & 4 & 1 & -3 & 0 & 2 & 7 & 8 & 3 \\
C & F & F & F & F & T & T & T & T \\
Ci = (true \rightarrow C) & T & T & F & F & T & T & F & T \\
{\text{current}}(X \text{ when } Ci) & 4 & 1 & 1 & 2 & 7 & 7 & 3 & 3
\end{array}
\]

Solution 2: Forcing an initial value

\[E = \text{if } C \text{ then } \text{current}(X \text{ when } C)\]
\[\text{else } (\text{default} \rightarrow \text{pre}(E));\]
or (to save a memory)
\[X1 = (\text{if } C \text{ then } X \text{ else default}) \rightarrow X;\]
\[E = \text{current}(X1 \text{ when } C);\]

Nodes and clocks

A node works at the rate of its actual input parameters (data-flow behavior)

\[
\begin{array}{c|cccccccc}
C & F & F & F & F & T & T & T & T \\
\text{true when } C & T & T & T & T & T & T & T & T \\
\text{Counter(true when } C) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

sampling of inputs \neq sampling of outputs

\[
\begin{array}{c|cccccccc}
\text{Counter(true)} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{Counter(true) when } C & 2 & 4 & 7 & 8 & 1 & 2 & 3 & 4 \\
\end{array}
\]

Assertions

assert \langle \text{bool exp} \rangle;
allows the programmer to write general assumptions (about the program environment).

Examples:

assert not (R and S);
assert true \rightarrow (U - \text{pre}(U)) <= 5;

Assertions (cont.)

These assertions are taken into account

- by the simulator
- by the compiler (under some options)
- by verification and testing tools

The story of Lustre/Scade
The story of Lustre/Scade

- 1984: first design (Caspi, Halbwachs - IMAG)
- 1987: first academic compiler (Plaice - IMAG)
- 1989: first industrial environment, SAGA (Verilog, Schneider-Electric)
- 1995: SAGA becomes SCADE (Verilog, Schneider-Electric, Airbus)
- 2001: SCADE is bought by Esterel-Technologies see www.esterel-technologies.com/v2/
- 2006: SCADE V6 (automata, arrays, . . .)

The Imperative Language

Esterel

N. Halbwachs (Verimag/CNRS) Synchronous Programming

Introductory example

An Esterel process communicates with its environment by means of (pure or valued) signals.
Its behavior is a sequence of reactions, each of which triggered by input signals
Ex.: a behavior of the speedometer

- In a reaction, a signal is either present or absent (instantaneous broadcast)
module Speedometer:
input second, meter;
output speed: integer in
loop
var Distance := 0 : integer in
abort
every meter do  
Distance := Distance + 1
end every
when second ;
emit speed(Distance);
end var
end loop
end module

N. Halbwachs (Verimag/CNRS) Synchronous Programming 81 / 198

The language
10 Signals, Events, and occurrences
Kernel language
Some derived statements

Examples
The story of Esterel
SyncCharts

Signals, Events, and occurrences

Event = set of present signals
A reaction consists of adding signals to an input event.
tick is a special signal which belongs to all events
If S is a valued signal, ?S refers to the value carried by its last occurrence

Occurrence:
In statements like await S, abort ... when S, the considered occurrence of S is in the strict future.
but you can write also await immediate S, ...
An occurrence may also consist of a number of signal occurrences:
every 60 SEC do emit MIN end
### The language

- Signals, Events, and occurrences
- Kernel language
- Some derived statements

### Kernel language

- `nothing, halt`
- `v := exp`
- `stat ; stat`
- `if exp then stat else stat`
- `loop stat end`
- `trap id in stat`
- `exit id`
- `var var_decls in stat`
- `signal signal_decls in stat`
- `[ stat | stat ]`
- `emit S, emit S(exp)`
- `present S then stat1 else stat2`
- `abort stat when S`

### Some derived statements

- `await S (= abort halt when S)`
- `weak abort stat when S (= trap T in [ abort stat1; exit T when S; stat2 | loop abort Stat ; halt when Occ ; end loop ] end )`
- `pause (= await tick)`

### Example 1: Parallel composition and instantaneous broadcast

```plaintext
input Second;
output Minute, Hour;

[ every 60 Second do emit Minute end |
  every 60 Minute do emit Hour end ]
```
Exercise: ABRO
emit O as soon as both A and B have occurred. Restart on any occurrence of R.

Example 2: Mouse Click
- receives pure signals click and hsec.
- emits double whenever two clicks happen within d hsec, and single whenever a click is not followed by a second click within that delay.

module mouse:
constant d: integer;
input click, hsec; output single, double;

loop
await click;
abort
await click; emit double
when d hsec

timeout emit single end
end

slightly wrong when simultaneous "click" and "d hsec"
### Structure nesting and priorities

Correct solution:

```plaintext
module mouse:
constant d: integer;
input click, hsec; output single, double;
loop
  await click;
  abort
  await d hsec; emit single
  when click
  timeout emit double end
end loop
end
```

N. Halbwachs (Verimag/CNRS) Synchronous Programming

### Exercise: Reflex game

- The player puts a coin in the machine
- After a random delay, the machine switches on the go lamp
  The player should press the stop button as soon as possible
- Then the machine displays the time (in ms.) elapsed
  between go and stop. The go lamp is switched on, and a new game can start
- **Exception cases:**
  - the player presses the stop before go (cheating! ring the bell and end the game)
  - the player does not press stop within limit time ms. after the go lamp is on (abandon, end the game)
  - Initially, only the game_over in on, the machine displays 0.

### Reflex game: declarations

```plaintext
module REFLEX_GAME :
constant limit_time : integer;
function RANDOM() : integer;
input MS, COIN, STOP;
output DISPLAY(integer), GO_ON, GO_OFF, GameOver_ON, GameOver_OFF, RingBell;
```

### Reflex game: initializations

**Overall initializations:**
- emit DISPLAY(0);
- emit GO_OFF;
- emit GameOver_ON;

**Game initializations**
- emit DISPLAY(0);
- emit GO_OFF;
- emit GameOver_OFF;

### Reflex game: the game (1) [Exercise]

% Phase 1:
% wait for a random delay, and switch on the GO lamp
% if STOP is pushed while waiting,
% ring bell and end the game
Reflex game: the game (2) [Exercise]

% Phase 2: count the time until STOP
% if limit time is reached, end the game

Reflex game: end of a game

emit GO_OFF;
emit Game_Over.ON;

The story of Esterel

developed since early 80th at CMA/ENSMP and Inria [G. Berry & co.]
now equipped with a graphical syntax: SyncCharts [Ch. Andre, 1996]
commercialized by Esterel-Technologies until 2009
several sources, many users
see www.esterel.org

SyncCharts: a graphical language based on Esterel

Automata are very useful for describing control
The best way for describing an automaton is by drawing it
(not always easy to specify in Esterel)

Example: ABRO as an automaton
emit O as soon as both A and B have occurred.
Restart on any occurrence of R.

Graphical, automata-based language: Needs

Structure (cf. Statecharts: parallel and hierarchical composition)
Avoid text (texts and drawings don’t merge so well)
Small number of graphical primitives
(Of course!) Precise semantics
SyncCharts

**ABRO in SyncCharts**

**Advantages**
- Better structure
- Each signal appears once
- Waiting also for C doesn't complexify too much (linear increase)

**SyncCharts: Basic constructs (1)**

Automata with (input events)/(output signals) on the transitions, possibly final states, possible priorities on transitions.

**SyncCharts: Basic constructs (2)**

Hierarchical composition

**SyncCharts: Basic constructs (3)**

Parallel composition, local signal

**SyncCharts: Basic constructs (4)**

Inhibition and inhibiting transitions

**SyncCharts: Basic constructs (5)**

Transitions on termination

**ABRO simulation**

Initial
Advantages of SyncCharts over Esterel

- Graphical (people sometimes like it ?!)
- Easy automata description
- Basically, same difference as between automata and regular expressions
- Nesting of interrupts/preemption/exceptions more readable than with textual nesting

Compilation of Synchronous Languages
### Static semantics

- Causality analysis

- Sequential code generation

### Causality analysis

#### Causality errors

Some programs don’t make sense

\(\sim\) combinational loops in circuits

- **In Lustre**
  \[x = \text{not } x;\]

- **In circuits**
  \[\text{module P1: output } x;\]
  \[\text{present } x \text{ else emit } x\]
  \[\text{end present}\]

  ![Causality errors](image1)

- **In Esterel**
  \[\text{module P1: output } x;\]
  \[\text{present } x \text{ else emit } x\]
  \[\text{end present}\]

  ![Causality errors](image2)

  _no behavior_

#### Causality errors (2/3)

but some loops do make sense!

- **In Lustre**
  \[x = \text{if } c \text{ then } y \text{ else } z;\]
  \[y = \text{if } c \text{ then } z \text{ else } x;\]

- **In circuits**
  ![Causality errors](image3)

- **In Esterel**
  \[\text{module P1: input } c, z;\]
  \[\text{output } x, y;\]
  \[\text{present } c;\]
  \[\text{present } y \text{ then emit } x\]
  \[\text{end present}\]
  \[\text{present } z \text{ then emit } y\]
  \[\text{end present}\]
  \[\text{else;}\]
  \[\text{present } z \text{ then emit } x\]
  \[\text{end present}\]
  \[\text{present } x \text{ then emit } y\]
  \[\text{end present}\]

  ![Causality errors](image4)

  _only one behavior in classical logic_

  _no behavior in constructive logic!_

#### Causality analysis

#### Solutions:

- forbid loops (Lustre)
- forbid instantaneous reaction to absence
  
  (SL [Boussinot], ReactiveML [Mandel])

In pure Esterel, or in Boolean Lustre, the problem boils down to determining whether a system of Boolean equations has one and only one solution.

- one and only one behavior in _classical_ logic
- one and only one behavior in _constructive_ logic (Esterel V5)

#### Causality analysis (3/3)

- Source of all the problems with the semantics of Statecharts and Sequential Function Charts (Grafcet)

- Where does the problem come from?
  
  The result of a reaction is a fixpoint of a function which is not necessarily increasing (because of the negation, or the reaction to absence)
Causality analysis (3/4)
Computing in constructive logic:

- Use Scott's Boolean domain with

\[ 0 = \bot = \top 0 = 0 \]

\[ 1 = \bot = \top 1 = 1 \]

- or use dual rail encoding:

\[ x_0, x_1 \text{ with } \begin{cases} x_0 = 1 & \text{iff } \text{x surely 0} \\ x_1 = 1 & \text{iff } \text{x surely 1} \end{cases} \]

Causality analysis (4/4)

Example:

\[ x = x ; \]

\[ y = x \text{ and not } y ; \]

Example: McMillan/DeSimone’s bus arbiter

\( n \) units connected to a bus. At each clock tick, some of them can ask for bus access for this tick. At most one may be granted, and the allocation should be fair.

Basic idea: travelling token:

\[ \begin{align*}
\text{r}_0 & = \text{m}_0 + \text{t}_2 \cdot \text{r}_2 \\
\text{r}_1 & = \text{m}_1 + \text{t}_0 \cdot \text{r}_0 \\
\text{r}_2 & = \text{m}_2 + \text{t}_1 \cdot \text{r}_1 \\
\text{r}_3 & = \text{m}_3 + \text{t}_2 \cdot \text{r}_2
\end{align*} \]

Example: McMillan/DeSimone’s bus arbiter

Fairness: change the master at each tick

\[ \begin{align*}
\text{t}_0 & = \text{m}_0 + \text{t}_2 \cdot \text{r}_2 \\
\text{t}_1 & = \text{m}_1 + \text{t}_0 \cdot \text{r}_0 \\
\text{t}_2 & = \text{m}_2 + \text{t}_1 \cdot \text{r}_1
\end{align*} \]

but...

- if \( \text{m}_0 = 1 \), \( \text{t}_0 = 1 \):

\[ \begin{align*}
\text{t}_1 & = \text{m}_1 + \text{r}_0 \\
\text{t}_2 & = \text{m}_2 + (\text{m}_1 + \text{r}_0) \cdot \text{r}_1
\end{align*} \]

- if \( \text{m}_1 = 1 \), \( \text{t}_1 = 1 \):

\[ \begin{align*}
\text{t}_2 & = \text{m}_2 + \text{r}_1 \\
\text{t}_0 & = \text{m}_0 + (\text{m}_2 + \text{r}_1) \cdot \text{r}_0
\end{align*} \]

- if \( \text{m}_2 = 1 \), \( \text{t}_2 = 1 \):

\[ \begin{align*}
\text{t}_0 & = \text{m}_0 + \text{r}_2 \\
\text{t}_1 & = \text{m}_1 + (\text{m}_0 + \text{r}_2) \cdot \text{r}_0
\end{align*} \]

Combinational loop!

\[ \text{m}_0 \uparrow \text{m}_1 \uparrow \text{m}_2 \]

remains to show that \( \text{m}_0 \lor \text{m}_1 \lor \text{m}_2 \) always true
Static semantics

Sequential code generation
- Single loop for Lustre
- Explicit Automata
- From Esterel to Lustre
- Reincarnation in Esterel

Compilation: sequential code generation

Single loop (implicit automaton)
- initializations
  forever do
    get inputs
    compute outputs
    update memory
  end

obvious for data-flow programs:
- sort the variables according to their dependences
  (an order, from the causality criterion in Lustre)
- choose a suitable set of memories

Single loop (2/3)

Sorting variable computation, and choice of memories
- $x > y$ iff $x$ appears outside of any "pre" in the definition of $y$
  ($x$ must be computed after $y$)
- $>$ is a partial order, if there is no causality error
- $x > y$ iff $\text{pre}(y)$ appears in the definition of $x$ ($x$ should be computed before $y$, i.e., from the previous value of $y$)
- Consider the graph of the relation "$>$".
- Remove $>$ edges to cut all loops (if any). The resulting graph is a partial order.
- Whenever an edge $x \leadsto y$ is removed, a buffer $py$ must be introduced.

Single loop (2/3)

Sorting variable computation, and choice of memories
- $x > y$ iff $x$ appears outside of any "pre" in the definition of $y$
  ($x$ must be computed after $y$)
- $>$ is a partial order, if there is no causality error
- $x > y$ iff $\text{pre}(y)$ appears in the definition of $x$ ($x$ should be computed before $y$, i.e., from the previous value of $y$)
- Consider the graph of the relation "$>$".
- Remove $>$ edges to cut all loops (if any). The resulting graph is a partial order.
- Whenever an edge $x \leadsto y$ is removed, a buffer $py$ must be introduced.

Single loop (3/3)

Sorting variable computation, and choice of memories
- Example
  $x = 0$ -> if edge then
  $\quad$ (pre($x$) + $y$)
  $\quad$ read($c$, $y$);
  $\quad$ else
  $\quad$ pre($x$);
  $\quad$ edge = $c$ -> (c and not pre($c$));
  $\quad$ { edge := $c$; $x := 0$ }
  $\quad$ else
  $\quad$ edge := $c$ and not _pc ;
  $\quad$ if edge then $x := x+y$
  $\quad$ . _init := false ; _pc := $c$;
  $\quad$ end

Single loop (1/3)

Example
- $x = 0$ -> if edge then
  $\quad$ _init := true ;
  $\quad$ foreach step do
  $\quad$ read($c$, $y$);
  $\quad$ else
  $\quad$ pre($x$);
  $\quad$ edge = $c$ -> (c and not pre($c$));
  $\quad$ { edge := $c$ ; $x := 0$ }
  $\quad$ else
  $\quad$ edge := $c$ and not _pc ;
  $\quad$ if edge then $x := x+y$
  $\quad$ . _init := false ; _pc := $c$;
  $\quad$ end for
Sequential code generation
- Single loop for Lustre
- Explicit Automata
- From Esterel to Lustre
- Reincarnation in Esterel

Explicit control automaton (1/5)
- first way of compiling Esterel
- also applied to data-flow languages
- basis for verification methods
  (notion of control automaton)

Principle: specialize the code executed at each step (e.g., at initial step)

Explicit control automaton (2/5)
An example in Lustre
\[
\begin{align*}
x = 0 & \rightarrow \text{if edge then } \text{pre}(x)+1 \\
& \text{else pre}(x) \\
\text{edge} & = c \rightarrow \\
& \text{c and not pre}(c);
\end{align*}
\]

Explicit control automaton (2/5)
An example in Lustre
\[
\begin{align*}
x = 0 & \rightarrow \text{if edge then } \text{pre}(x)+1 \\
& \text{else pre}(x) \\
\text{edge} & = c \rightarrow \\
& \text{c and not pre}(c);
\end{align*}
\]

Explicit control automaton (2/5)
An example in Lustre
\[
\begin{align*}
x = 0 & \rightarrow \text{if edge then } \text{pre}(x)+1 \\
& \text{else pre}(x) \\
\text{edge} & = c \rightarrow \\
& \text{c and not pre}(c);
\end{align*}
\]

Explicit control automaton (2/5)
An example in Lustre
\[
\begin{align*}
x = 0 & \rightarrow \text{if edge then } \text{pre}(x)+1 \\
& \text{else pre}(x) \\
\text{edge} & = c \rightarrow \\
& \text{c and not pre}(c);
\end{align*}
\]

Explicit control automaton (3/5)
An example in Esterel
\[
\begin{align*}
every \ R \ do \\
& \text{[ await A \ || \ await B ]}; \\
& \text{emit } O; \\
& \text{halt}
\end{align*}
\]
Explicit control automaton (3/5)

An example in Esterel

every R do
    [ await A || await B ];
    emit O ;
end every

Explicit control automaton (4/5)

An example in Esterel

abort halt\(^1\) when R;
loop
    abort
    [ abort halt\(^2\) when A
      || abort halt\(^3\) when B
    ];
    emit O; halt\(^4\)
    when R
end

In kernel language:

1. \text{abort halt}^1 \text{ when R;}
2. \text{loop}
3. \text{abort}
4. \text{[ abort halt}^2 \text{ when A
}\text{ || abort halt}^3 \text{ when B
]};
5. \text{emit O; halt}^4 \text{ when R
}\text{end}

Abstract Control Automaton

\text{DEMO: Boolean mouse}
Explicit control automaton (5/5)

- Very efficient code
- Possible exponential growth of the code (less than for asynchronous languages)

Esterel is now compiled also into an implicit automaton (single loop)

- Much more tricky!
- Consequence of silicon compiling
- No explosion of the code size

From Esterel to Lustre (1/7)

Principles of the translation of (pure) Esterel into implicit automata (or Esterel → Lustre, or Esterel → circuit)

Each Esterel statement is translated into a node with the same interface:

```plaintext
node N
  (start, run, S,... : bool)
returns (term, halt, S',...)
```

Examples:

```plaintext
emit S': term = start
halt = false;
S' = start;
```

From Esterel to Lustre (2/7)

Examples:

```plaintext
await S:
  term = run and w and S;
  halt = run and w and not S;
  w = false → pre(start or halt);
```

From Esterel to Lustre (3/7)

Examples:

```plaintext
stat1 || stat2:
  (term1, halt1, S'1) = Stat1(start, run,...);
  (term2, halt2, S'2) = Stat2(start, run,...);
  halt = halt1 or halt2;
  term = (term1 or term2) and not halt;
  S' = S'1 or S'2;
```

From Esterel to Lustre (4/7)

Examples:

```plaintext
stat1 || stat2:
  (term1, halt1, S'1) = Stat1(start, run,...);
  (term2, halt2, S'2) = Stat2(start, run,...);
  halt = halt1 or halt2;
  term = (term1 or term2) and not halt;
  S' = S'1 or S'2;
```
Sequential code generation

From Esterel to Lustre

\[
\text{stat1; stat2:}
\]
\[
(\text{term1, halt1, } S'1) = \text{Stat1}(\text{start, run},...);
\]
\[
(\text{term2, halt2, } S'2) = \text{Stat2}(\text{term1, run and not halt1},...);
\]
\[
\text{halt} = \text{halt1 or halt2};
\]
\[
\text{term} = \text{term2};
\]
\[
S' = S'1 \text{ or } S'2;
\]

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop stat:}
\]
\[
(\text{term1, halt1, } S',...) = \text{Stat(start or term1, run,...)};
\]
\[
\text{term} = \text{false};
\]
\[
\text{halt} = \text{halt1};
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop}
\]
\[
\text{signal S in}
\]
\[
[\text{ await T ; emit S)}
\]
\[
|\text{ present S then emit O)}
\]
\[
]\end
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop}
\]
\[
\text{signal S in}
\]
\[
[\text{ await T ; emit S)}
\]
\[
|\text{ present S then emit O)}
\]
\[
]\end
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop}
\]
\[
\text{signal S in}
\]
\[
[\text{ await T ; emit S)}
\]
\[
|\text{ present S then emit O)}
\]
\[
]\end
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop}
\]
\[
\text{signal S in}
\]
\[
[\text{ await T ; emit S)}
\]
\[
|\text{ present S then emit O)}
\]
\[
]\end
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop}
\]
\[
\text{signal S in}
\]
\[
[\text{ await T ; emit S)}
\]
\[
|\text{ present S then emit O)}
\]
\[
]\end
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)

\[
\text{loop}
\]
\[
\text{signal S in}
\]
\[
[\text{ await T ; emit S)}
\]
\[
|\text{ present S then emit O)}
\]
\[
]\end
\]

Sequential code generation

Reincarnation in Esterel

Sequential code generation

Reincarnation in Esterel

A last compilation problem:

“Reincarnation” in Esterel (1/4)
A last compilation problem: “Reincarnation” in Esterel (1/4)

loop
  signal S in
  | await T; emit S
  || present S then emit O
  | end
end

“Reincarnation” in Esterel (2/4)

Solution 1: Code replication

loop
  signal S1 in
  | await T; emit S1
  || present S1 then emit O
  | end
end
  signal S2 in
  | await T; emit S2
  || present S2 then emit O
  | end

“Reincarnation” in Esterel (3/4)

Solution 2: Distinguish between
- the “surface” of the body, i.e., a piece of code corresponding to its first reaction,
- and its “depth”, i.e., a piece of code corresponding to other reactions.

surf( await T ) = pause
depth( await T ) = await immediate T
surf( present S then emit O ) = present S then emit O
depth( present S then emit O ) = nothing

“Reincarnation” in Esterel (4/4)

loop
  signal S1 in
  | pause
  || present S1 then emit O
  | end
end
  signal S2 in
  | await immediate T; emit S2
  || nothing
  | end

Verification and Automatic Testing
of Synchronous Programs

Validation of Reactive Systems (1/2)

- crucial goal
- critical properties (safety) “something (bad) never happens”
- the environment must be taken into account
Validation of Reactive Systems (2/2)

Two complementary techniques
- Formal verification
- Testing
Both require the formal description of
- the expected behavior of the system (properties)
- the assumed behavior of the environment (assumptions)

Expressing Properties using Synchronous Observers

A program observing relevant variables, and computing a Boolean output, true as long as the property is satisfied (safety).

Example – “any occurrence of “danger” is followed by an “alarm” before the next “deadline”.

Synchronous Observers (1/3)

An observer in Lustre

node Omega (danger, alarm, deadline: bool)
returns (ok: bool);
var alarm_since_danger: bool
let
  ok = deadline => alarm_since_danger;
  alarm_since_danger =
    if alarm then true
    else if danger then false
    else (true => pre(alarm_since_danger));
tel

Synchronous Observers (2/3)

Advantages
- Transform path properties into state properties
- Same language to write programs and properties
- Observers are executable
  - they can be tested
  - they can be executed on-line, concurrently with the program, during its actual execution (self-test, redundancy, monitoring, “runtime verification”)

Synchronous Observers (3/3)

“Verification Program”

Check: “if ok=false then realistic=false before”

Formal Verification

Synchronous programs as Interpreted Automata
- Split the program into a finite control part and a data part.
- In Lustre, the control part may be obtained by a restriction to Boolean (or finite state) variables (partial evaluation).
- Result: an explicit or implicit interpreted automaton
  Already used in compilation (Esterel and Lustre)
Verification by Model-Checking

Exhaustive exploration of the control part of the automaton (Boolean abstraction): build all the states that can be reached from the initial state with relevant always true, checking that ok is always true.

- exact for purely Boolean programs
- conservative for general programs

**Model-Checking: Example**

The Switch example in Lustre:

```plaintext
node TwoStates1
  (init, set, reset: bool)
returns (state: bool);
let
  state = init ->
  if set and not pre(state)
  then true
  else if reset then false
  else pre(state);
end
node TwoStates2
  (init, set, reset: bool)
returns (state: bool);
let
  state = init ->
  if set and not pre(state)
  then true
  else if reset then false
  else pre(state);
end

Show that they behave the same if set and reset are never true together
```

**Model-Checking: Example**

```plaintext
node Verif(init, set, reset: bool) returns (ok: bool);
var s1, s2: bool;
let
  s1 = TwoStates1(init, set, reset);
  s2 = TwoStates2(init, set, reset);
  ok = (s1=s2);
tel
```

**Model-Checking: Example**

```plaintext
node Verif(init, set, reset: bool) returns (ok: bool);
var s1, s2: bool;
let
  s1 = TwoStates1(init, set, reset);
  s2 = TwoStates2(init, set, reset);
  ok = (s1=s2);
tel
```

LESAR: a verification tool.

- several engines: enumerative search, symbolic (BDD-based) forward or backward search

**LESAR: BAD STATES**
LESAR: a verification tool.

- several engines: enumerative search, symbolic (BDD-based) forward or backward search

Forward

BAD STATES

DEMO

N. Halbwachs (Verimag/CNRS)  Synchronous Programming  175 / 198

Backward

BAD STATES

DEMO

N. Halbwachs (Verimag/CNRS)  Synchronous Programming  175 / 198

Unfeasible transitions

Statically unfeasible transitions

x \leq 1 \land x > 2

Dynamically unfeasible transitions

x \leq 0

x := x + 1

x \geq 2

N. Halbwachs (Verimag/CNRS)  Synchronous Programming  176 / 198

Numerical Properties

Approaches based on the control automaton only provide exact results when applied to pure control programs (Pure Esterel, Boolean Lustre, ...)

Interpreted Automata (with data): approximate, conservative results

Some unfeasible transitions are considered
Example: A subway speed regulation system

Each train counts

- the number \( \#b \) of beacons encountered along the track
- the number \( \#s \) of seconds received from a central clock

Ideally, a train should meet a beacon each second.

In fact,

- When \( \#b \) becomes \( \geq \#s + 10 \), the train is considered early, until \( \#b \leq \#s \) (hysteresis)
- When \( \#s \) becomes \( \geq \#b + 10 \), the train is considered late, until \( \#s \leq \#b \)


A subway speed regulation system (3/5)

Code of state: “On-time”

\[
\begin{align*}
\text{if BEACON then DIFF} &:= \text{DIFF} + 1 \\
\text{else if SECOND then DIFF} &:= \text{DIFF} - 1; \\
\text{if DIFF} &< -10 \text{ then} \\
\text{if DIFF} &> 10 \text{ then goto “Early_and_Late”} \\
\text{else goto “Late”} & \\
\text{else if DIFF} &> 10 \text{ then goto “Early”} \\
\text{else goto “On_time”} &
\end{align*}
\]

A subway speed regulation system (4/5)

Removing statically unfeasible transitions

- Use assumptions (boring and error-prone)
- Satisfiability of linear constraints can be easily checked

Result on the example

A subway speed regulation system (5/5)

Removing dynamically unfeasible transitions

- Use abstract interpretation to build, in each state of the automaton, an upper approximation (system of linear constraints) of the set of variable valuations when the control is at this state.

Result on the example

The complete subway example

- When a train is early, it puts on brakes; continuously braking makes the train stop before encountering 10 beacons.
- When a train is late, it warns the central clock, which stops sending the SECOND signal until no train is late.

For two trains, we get \(|\#b_1 - \#b_2| \leq 29\), showing that two trains initially separated by more than 29 beacons will never collide.
Automatic Testing

Validation of reactive systems

Synchronous observers

Verification

N. Halbwachs (Verimag/CNRS) Synchronous Programming 186 / 198

Testing

- Strong industrial demand
- To be used when verification fails
  - Too complex programs (numbers)
  - Black-box programs (partly described in low-level languages, distributed implementations)

N. Halbwachs (Verimag/CNRS) Synchronous Programming 187 / 198

Automatic Testing

- Use the assumption observer to generate realistic test sequences
- Use the property observer as an "oracle" to analyze the results of the test

N. Halbwachs (Verimag/CNRS) Synchronous Programming 188 / 198

Automatic Testing: Example (1/3)

The controller \(C\) should maintain the temperature \(u\) between 17\(\degree\) and 20\(\degree\) by turning a heater on and off.

When the heater is running, the temperature \(u\) increases with \(0.2 \leq \frac{du}{dt} \leq 0.5\); otherwise it decreases with \(-0.3 \leq \frac{du}{dt} \leq -0.1\).

Initially, the heater is turned off, and 18\(\degree \leq u \leq 19\)\(\degree\).

N. Halbwachs (Verimag/CNRS) Synchronous Programming 189 / 198

Automatic Testing: Example (2/3)

node Assumption(on, off: bool; u: real) returns (realistic: bool);

let realistic = (u \geq 18 and u \leq 19) -> if heating then (dudt \geq 0.2 and dudt \leq 0.5)
else (dudt \geq -0.3 and dudt \leq -0.1);

\(dudt = (u – \text{pre}(u))\);

heating = false -> if pre(on) then true
else if pre(off) then false
else pre(heating);
tel

N. Halbwachs (Verimag/CNRS) Synchronous Programming 190 / 198

Automatic Testing: Example (3/3)

node Property(on, off: bool; u: real) returns (ok: bool);

let ok = (u \geq 17 and u \leq 20);
tel

N. Halbwachs (Verimag/CNRS) Synchronous Programming 191 / 198

Testing tool: Lurette

Given
- the observers “Assumption” and “Property”
- the executable code of the program under test

generate and run arbitrarily long test sequences, satisfying the “Assumption”, while checking the “Property”

N. Halbwachs (Verimag/CNRS) Synchronous Programming 192 / 198

Generation of input sequences (1/3)

Realistic test sequences

\((i_1, i_2, \ldots, i_n)\) involving outputs

\((o_1, \text{rea}_1), (o_2, \text{rea}_2), \ldots, (o_n, \text{rea}_n)\) with \(\text{rea}_i = \text{true}, i = 1..n.\)
**Generation of input sequences (2/3)**

First attempt: Observer as an acceptor
- randomly choose input \(i\)
- get the correspond. output \(o\)
- submit \((i, o)\) to the observer
- in case of refusal choose again.

Two problems:
- the observer needs the program output \(o\)
- the probability to randomly choose a realistic input can be very small (e.g., \(X = Y\))

**Generation of input sequences (3/3)**

Second idea: Observer as a generator

Global state: \(S = \langle S_P, S_A \rangle\)

\[\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)\]

\[\langle S_P, S_A \rangle\]

choice of \(i\)
(s.t. \(f(S_A, i) = \text{true}\))

\(S_P\) running \(P\)

\(i/o\)

\(S'_A\) running \(A\)

\(S_P, S'_A\)

**Generation of input sequences (3/3)**

Second idea: Observer as a generator

Global state: \(S = \langle S_P, S_A \rangle\)

\[\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)\]

\[\langle S_P, S_A \rangle\]

choice of \(i\)
(s.t. \(f(S_A, i) = \text{true}\))

\(S_P\) running \(P\)

\(i/o\)

\(S'_A\) running \(A\)

\(S_P, S'_A\)

**Generation of input sequences (3/3)**

Second idea: Observer as a generator

Global state: \(S = \langle S_P, S_A \rangle\)

\[\text{rea} = f(S_A, i), \quad S'_A = g(S_A, i, o)\]

\[\langle S_P, S_A \rangle\]

choice of \(i\)
(s.t. \(f(S_A, i) = \text{true}\))

\(S_P\) running \(P\)

\(i/o\)

\(S'_A\) running \(A\)

\(S_P, S'_A\)

**Example (1/2)**

First step

\[\text{realistic} = (u > 18 \text{ and } u < 19) \rightarrow\]

- if heating then (\(\text{dudt} > 0.2\) and \(\text{dudt} \leq 0.5\))
- else (\(\text{dudt} = 0.3\) and \(\text{dudt} = 0.1\))

\[\text{dudt} = (u - \text{pre}(u));\]

\[\text{heating} = \text{false} \rightarrow\] if \(\text{pre(on)}\) then \(\text{true}\)

else if \(\text{pre(off)}\) then \(\text{false}\)

else \(\text{pre(heating)}\);
Automatic Testing
Example (1/2)

First step
realistic = (u≥18 and u≤19) ->
if heating then (dudt≥0.2 and dudt≤0.5) else (dudt≥–0.3 and dudt≤–0.1);
dudt = (u – pre(u));
heating = false -> if pre(on) then true else if pre(off) then false else pre(heating);

Select, e.g., u=18, run a program step, get the output, e.g., on=off=false

Example (2/2)

Second step:
You know pre(u)=18, pre(on)=pre(off)=pre(heating)=false
realistic = (u≥18 and u≤19) ->
if heating then (dudt≥0.2 and dudt≤0.5) else (dudt≥–0.3 and dudt≤–0.1);
dudt = (u – pre(u)); i.e., u–18
heating = false -> if pre(on) then true else if pre(off) then false else pre(heating); i.e., false

You get –0.3 ≤ u–18 ≤ –0.1, i.e., 17.7 ≤ u ≤ 17.9.
Select, e.g., u=17.7, run a program step, and so on...

Conclusion

Both for formal verification and for automatic testing, the user has to provide the same information:

- an observer specifying properties of interest (safety)
- an observer specifying known assumptions about the environment

Automatic tools are then available to make the job

Presently, we also have a specific language to express input scenarios (non deterministic, constrained)
A short bibliography

1 General principles, basic papers and books

References


2 Synchronous languages

References


3 Compilation

References


4 Verification and Validation

References


5 Synchrony/Asynchrony, Code distribution

References


6 Reactive programming, Higher order languages

References


