Synchronous programming

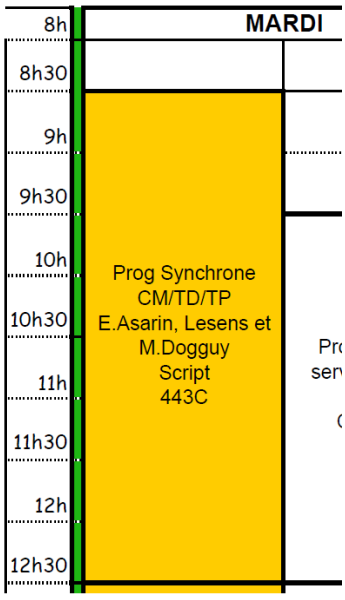
Critical Real Time Embedded Software

David Lesens

Wednesday, 06 October 2010

Synchronous programming

- Eugene Asarin
- Mehdi Dogguy
- David Lesens



Overview

- Critical real-time embedded software ☞
- Principles of the approach ☞
  - Introduction ☞
  - Formal semantics ☞
- SCADE ☞
- Model validation ☞

Where can we find software?

- Windows, Linux
- PowerPoint
- Latex
- Compilers
- Mathematical software (e.g. computation of π)
- Mobile phone
- Space
- Nuclear plant
- Airplane
- …

Software is everywhere…

What is embedded software?

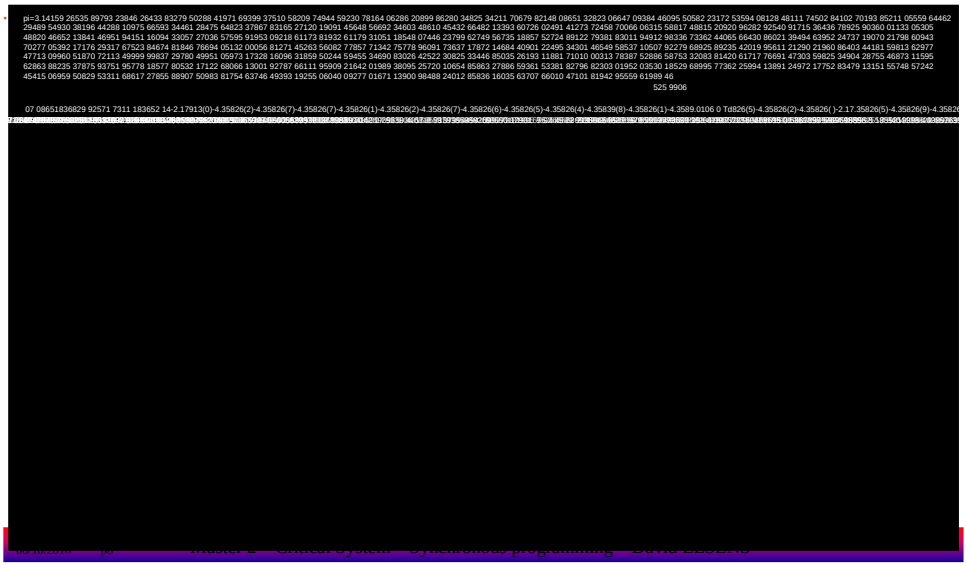
- Windows, Linux
- PowerPoint
- Latex
- Compilers
- Mathematical software (e.g. computation of π)

The software has its own objective
We can "buy" the software

- Mobile phone
- Space launcher
- Nuclear plant
- Airplane

The software is part of the system
We can only "buy" the system

Compute the first 10,000 digits of Pi



Real time?

- Transformational systems
  - Inputs available on execution start
  - Outputs delivered on execution end

} e.g. Mathematical computation

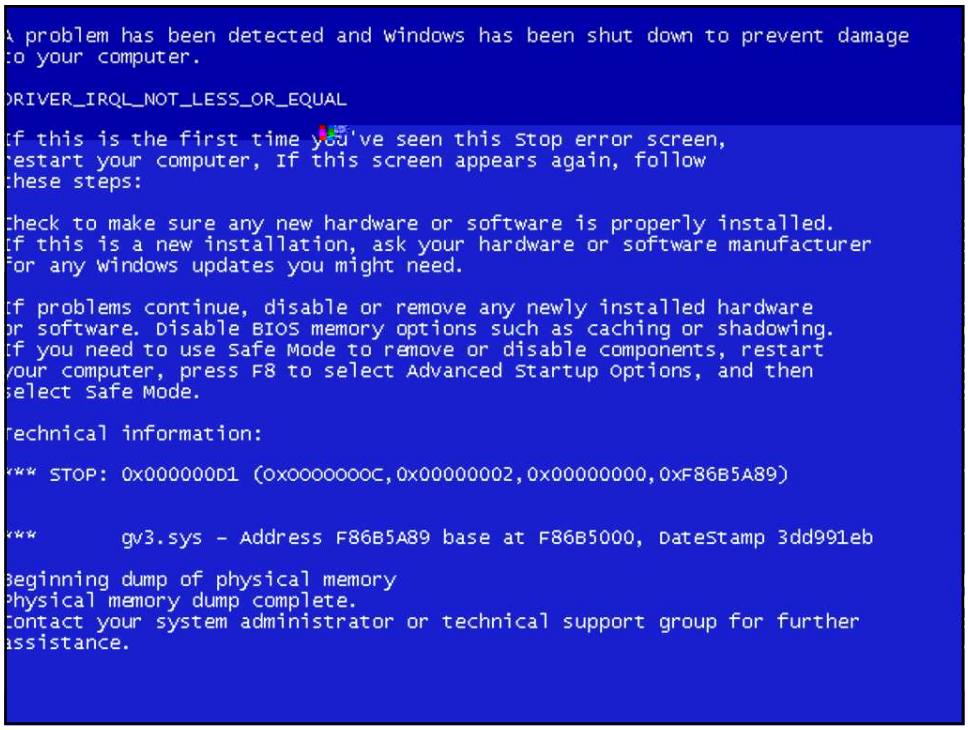
- Interactive systems
  - React to their environment
  - To their own speed

} e.g. Windows, Powerpoint

- Reactive systems
  - React to their environment
  - To a speed imposed by the environment

} e.g. Control / Command of a spacecraft

Critical? What does it mean?



Critical? What does it mean?

Intuitively, a critical system is a system which failure can have severe impacts

- Nuclear
- Aeronautic
- Automotive
- Railway
- Space
- …

Software criticality levels

Standards define precisely software criticality levels:

For instance:

- DO178B and DO178C for airborne systems
- ECSS for space systems
  - European Committee for Space Standardization

Software criticality categories ECSS-Q-80C

Software criticality category	Definition
A	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Catastrophic consequences
B	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Critical consequences
C	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Major consequences
D	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Minor or Negligible consequences

Software criticality categories ECSS-Q-80C

Software criticality category	Definition
A	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Catastrophic consequences
B	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Critical consequences
C	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Major consequences
D	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Minor or Negligible consequences

Software criticality categories ECSS-Q-80C

Software criticality category	Definition
A	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Catastrophic consequences
B	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Critical consequences
C	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Major consequences
D	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in: Minor or Negligible consequences

Software criticality categories ECSS-Q-80C

Software criticality category	Definition
A	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in Catastrophic consequences
B	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in Critical consequences
C	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in Major consequences
D	Software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in Minor or Negligible consequences

ECSS-Q-40B

Severity	Consequence
Catastrophic hazards	i) loss of life, life-threatening or permanently disabling injury or occupational illness, loss of an element of an interfacing manned flight system; ii) loss of launch site facilities or loss of system; iii) severe detrimental environmental effects.
Critical hazards	i) temporarily disabling but not life-threatening injury, or temporary occupational illness; ii) major damage to flight systems or loss or major damage to ground facilities; iii) major damage to public or private property; or iv) major detrimental environmental effects
Marginal hazards	minor injury, minor disability, minor occupational illness, or minor system or environmental damage
Negligible hazards	less than minor injury, disability, occupational illness, or less than minor system or environmental damage

ECSS-Q-40B

Severity	Consequence
Catastrophic hazards	i) loss of life, life-threatening or permanently disabling injury or occupational illness, loss of an element of an interfacing manned flight system; ii) loss of launch site facilities or loss of system; iii) severe detrimental environmental effects.
Critical hazards	i) temporarily disabling but not life-threatening injury, or temporary occupational illness; ii) major damage to flight systems or loss or major damage to ground facilities; iii) major damage to public or private property; or iv) major detrimental environmental effects
Marginal hazards	minor injury, minor disability, minor occupational illness, or minor system or environmental damage
Negligible hazards	less than minor injury, disability, occupational illness, or less than minor system or environmental damage

DO178B differs lightly from the ECSS

Severity	Consequence
Catastrophic	Failure conditions which would prevent continued safe flight and landing
Hazardous / Severe-Major	Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be: (1) a large reduction in safety margins or functional capabilities, (2) physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely, or (3) adverse effects on occupants including serious or potentially fatal injuries to a small number of those occupants
Major	Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries
Minor	Failure conditions which would not significantly reduce aircraft safety, and which would involve crew actions that are well within their capabilities. Minor failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight
No Effect	Failure conditions which do not affect the operational capability of the aircraft or increase crew workload

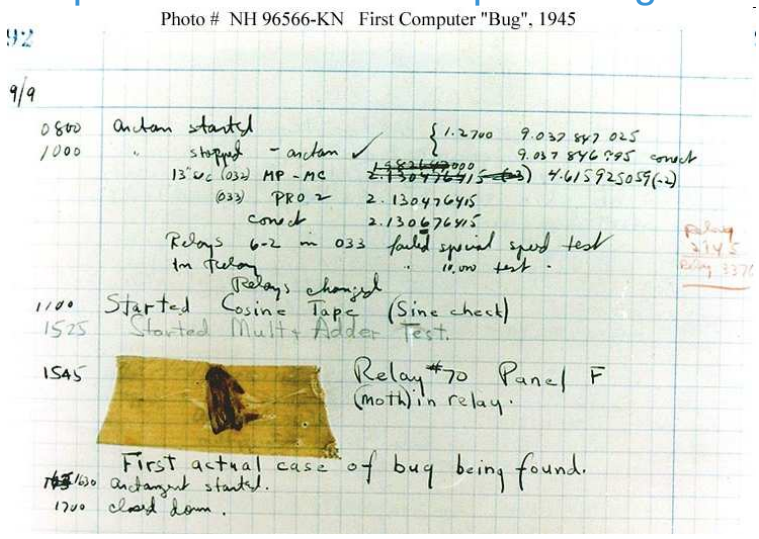
Wikipedia

Vocabulary

- Security
  - is the degree of protection against danger, loss, and criminals.
- Reliability
  - is the ability of a person or system to perform and maintain its functions in routine circumstances, as well as hostile or unexpected circumstances.
- Safety
  - is the state of being "safe" (from French sauf), the condition of being protected against […] consequences of failure, damage, error, accidents, harm or any other event which could be considered non-desirable. It can include protection of people or of possessions.

Example 1: The First “Computer Bug”

Photo # NH 96566-KN First Computer "Bug", 1945



Example 2: The Patriot Missile Failure

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people. A report of the General Accounting office, GAO/IMTEC-92-26, entitled *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia* reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors.

Failure in space

Trident **Sea launch**

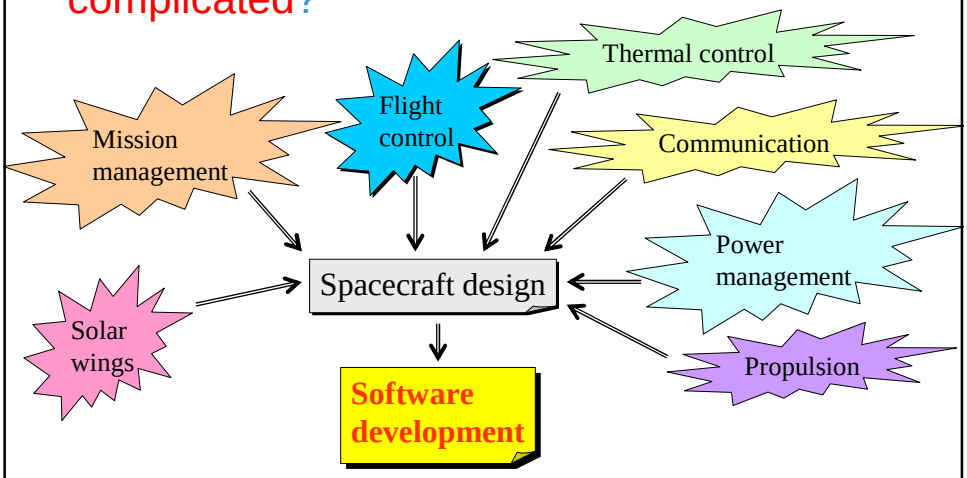
Overview

- Critical real-time embedded software ☞
- Principles of the approach ☞
  - Introduction ☞
  - Formal semantics ☞
- SCADE ☞
- Model validation ☞

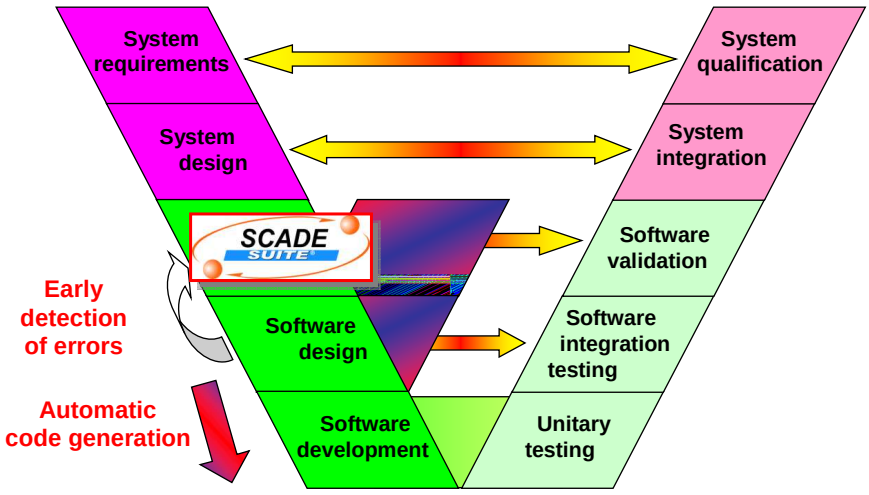
NASA's Climate Orbiter was lost September 23, 1999, due to a software bug

One engineering team used metric units while another used English units

Why is System (to Software) Engineering complicated?



Verification with model driven engineering



Formal Model Driven Engineering shall allow

- An early verification of the specification
  via a strong and intuitive semantic ensuring
  - Consistency
  - Completeness
  - Non ambiguity
- A behavioural validation within a simulation environment
-

Overview

- Critical real-time embedded software
- Principles of the approach
  - Introduction
  - Formal semantics
- SCADE
- Model validation

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

Wikipedia 

Formal semantics of programming languages

In theoretical computer science, formal semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages and models of computation

Statement groups

- In C, C++, Java

```
if ( light == red );
{
        Cancel_lift_off();
}
```

Legal statement
No warning

The call to Cancel_lift_off is always executed

- In Ada

```
if  light = red then;
        Cancel_lift_off;
end if;
```

Illegal statement
No compilation

Named notation

- In C, C++, Java

```
struct date {
  int day, month, year;
};
```

- In Ada

```
type Date is
  record
    Day, Month, Year : Integer;
  end record;
```

Named notation

- In C, C++, Java

```
struct date today = { 12, 1, 5 };
```

What does it mean?

- In Ada

```
Today: Date := ( Day => 12, Month => 1, Year => 5 );
```

➔ Notation usable also for function call

Using distinct types

- In C++

```
int badcount, goodcount;
int b_limit, g_limit;
…
badcount++;
…
if ( badcount == b_limit ) {
…
goodcount++;
…
if ( goodcount == b_limit ) {
…
```

Using distinct types

- In Ada

```
type Goods is new Integer;
type Bads is new Integer;
badcount, b_limit : Goods
goodcount, g_limit: Bads
…
badcount := badcount+1;
…
if  badcount = b_limit then
…
goodcount := goodcount+1;
…
if goodcount = b_limit then
…
```

Strong typing is a good rule of critical software

Illegal
Bad typing

Formal languages

- Programming languages are more or less formal
  - …
  - Ada is more formal than Java
  - Java is more formal than C++
  - C++ is more formal than C
  - C is more formal than Matlab
  - …

The risk of errors is less important with a formal language

```
case State is
 when State1 =>
  Guard1 := X < 3;
  Guard2 := X > 3;
  if (EVENT1 and (Guard1 or Guard2)) then
    if (Guard1) then
      X := 5;
      State := State2;
    else
      if (Guard2) then
        X := 6;
      end if;
      State := State3;
    end if;
  end if;
 when State2 =>
  if (EVENT1) then
    X := 7;
    State := State1;
  end if;
 when State3 =>
  if (EVENT1 and EVENT1) then
    X := 8;
    State := State1;
  end if;
end case;
```

An other very simple example

Simple? Yes…

But what does this piece of code do?

Code (even Ada) is not an adequate way to communicate with system engineer

Overview

- Critical real-time embedded software
- Principles of the approach
  - Introduction
  - Formal semantics
- SCADE
- Model validation

Overview

- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity

Wikipedia

Need of deterministic algorithm

- In computer science, a deterministic algorithm is an algorithm which, in informal terms, behaves predictably
- Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states

Wikipedia

Determinism and ECSS

ECSS-Q-80C

- 6.2.3 Handling of critical software
- 6.2.3.2 The supplier shall define and apply measures to assure the dependability and safety of critical software. These measures can include:
  - …
  - prohibiting the use of language commands and features that are unpredictable;
  - use of **formal design language for formal proof**;

Synchronous languages

Semantics = synchronous hypothesis

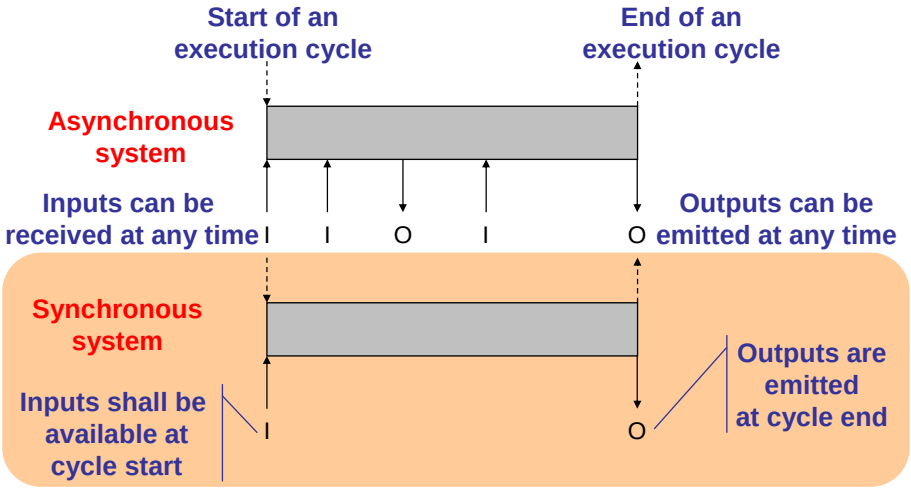
- Existence of a global clock
  - Software cyclically activated
  - Inputs read at the cycle beginning
  - Outputs delivered at cycle end
    (read / write forbidden during the cycle)
- The cycle execution duration shall theoretically be null

➔ No cycle overflow

- Mono-tasking

➔ Ensures the determinism

Asynchronous versus synchronous



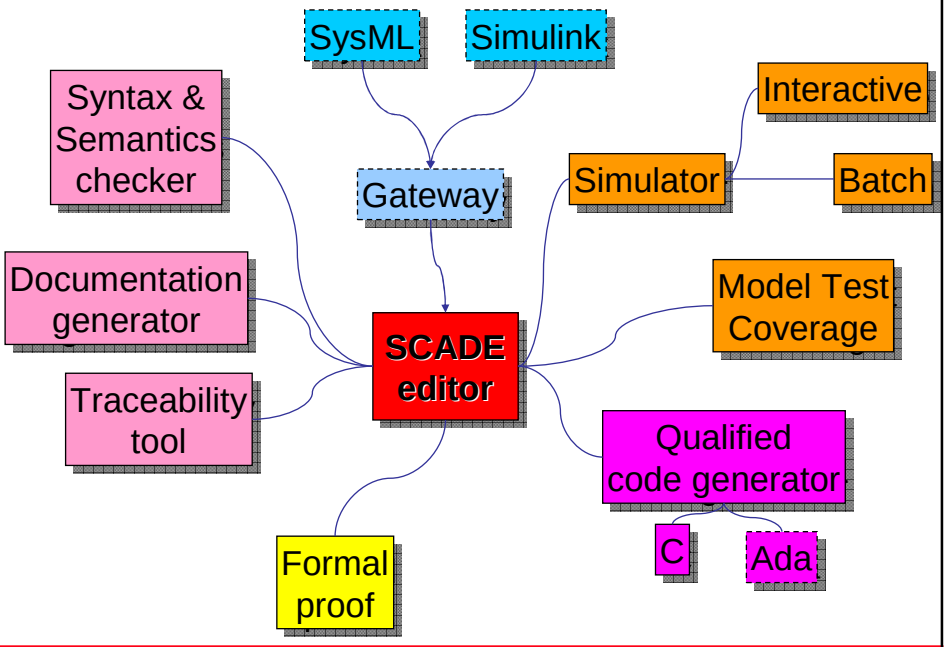
Overview

- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity

SCADE

"*Safety Critical Application Development Environment*"

- A textual language: Lustre
  - Formal language for reactive synchronous system
- A graphical language
  - Semantics equivalence SCADE ⇔ Lustre
  - Adapted to data flow and automata
- A software toolbox
  - Graphical editor, simulator, proof tool
  - Automatic documentation and certified code generation
- Synchronous approach



Time in Scade

- Global clock (known by all processes)
  - Time = discrete sequence of tick t_0, t_1, t_2, etc.
  - At each tick t_i a cycle is running
- Variable = flow which takes at each tick a unique value

  Example: integer variable x

	t_0	t_1	t_2	t_3	t_4	t_5
x	5	8	2	3	13	5

Operators

- An operator acts on flows of values (and not on values)

Example

- Operator « + »: int_n x int_n → int_n

	t_0	t_1	t_2	t_3	t_4	t_5
x	5	8	2	3	13	5
x + x	10	16	4	6	26	10

Temporal operators

- The "PRE" operator takes as input a data flow (i.e. a variable) and returns its value at the previous tick.

  At initial tick, its value is undefined.
- The "→" operator takes as input an initialisation value and a data flow of the same type. It returns an identical data flow, except for the initial value.

Example

	t_0	t_1	t_2	t_3	t_4	t_5
x	5	8	2	3	13	5
PRE x	null	5	8	2	3	13
9 -> x	9	8	2	3	13	5
9 -> PRE x	9	5	8	2	3	13

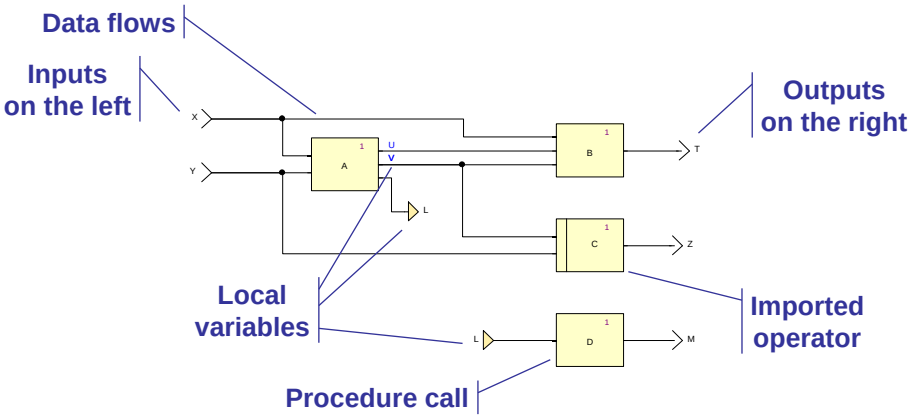
"Follow by" operator

FBY(x, n, init) = init → (PRE (PRE … x))

n times

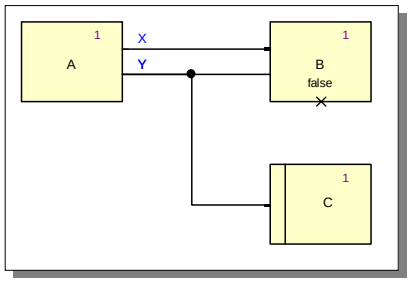
	t_0	t_1	t_2	t_3	t_4	t_5
x	5	8	2	3	13	5
9 -> PRE x	9	5	8	2	3	13
FBY(x,3,9)	9	9	9	5	8	2

SCADE at a glance: Data Flow

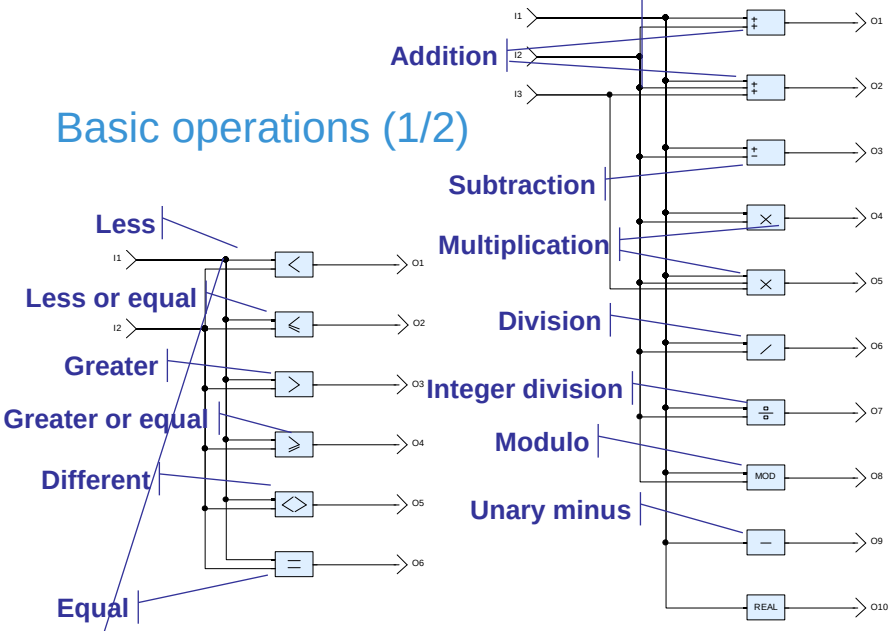


Textual versus graphical

```
( x, y ) = A();
B( x, y );
C( y )
```



Basic operations (1/2)



"Mutual exclusion" operator

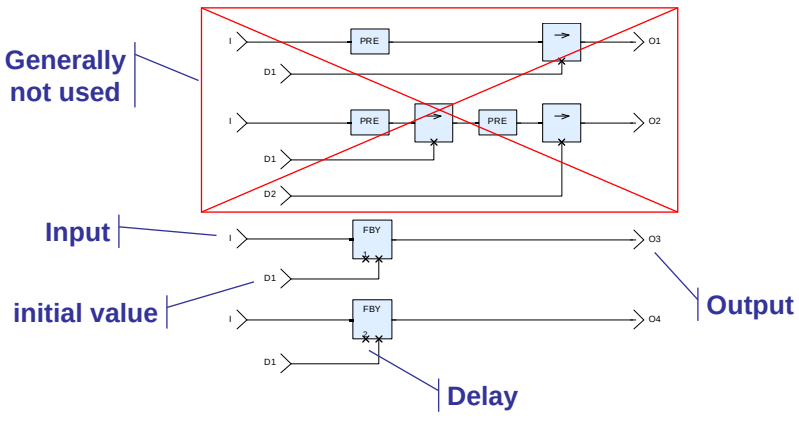
#: bool x bool x … x bool → bool

n times

Returns true if at most one of its inputs is true

e1	e2	e3	#(e1, e2, e3)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Delays

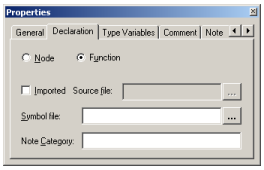


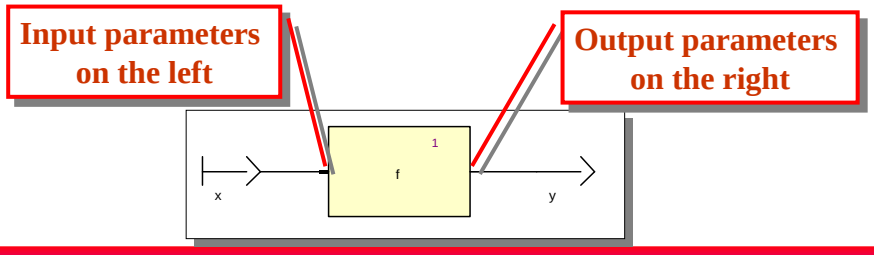
Node and function

y = f(x)

Function and nodes are represented by a rectangle

- A node has an internal state
- A function has **no** internal state





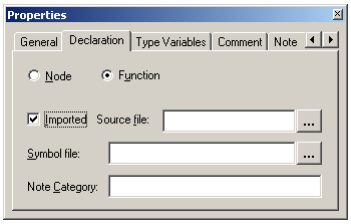
Imported function / node

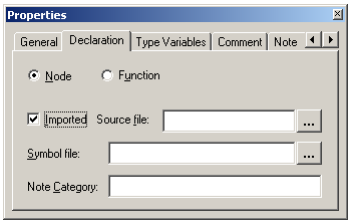
- Imported function

```
extern void C(
        bool Y );
```

- Imported node

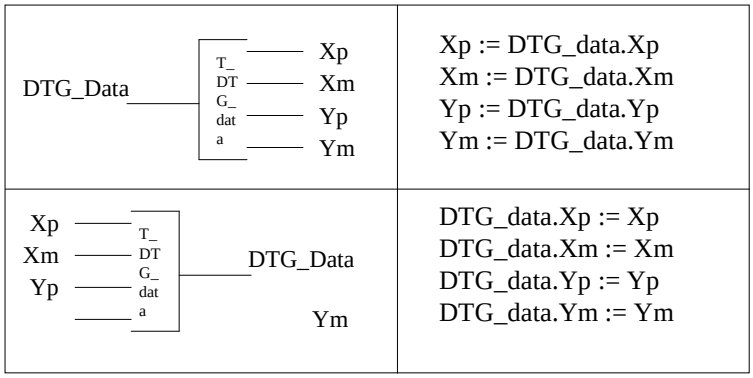
```
extern void C_reset(
        outC_C *outC );
extern void C(
        bool Y,
        outC_C *outC );
```



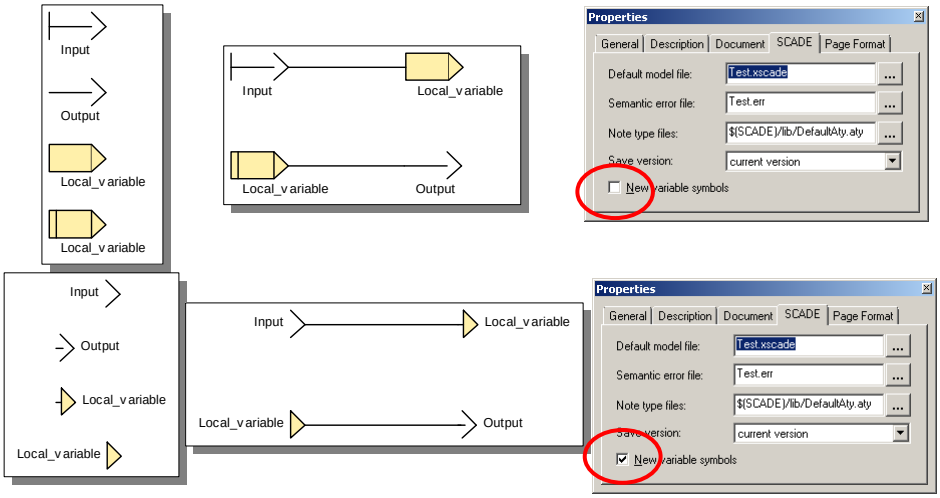


Context to be defined by the developer

Data structure

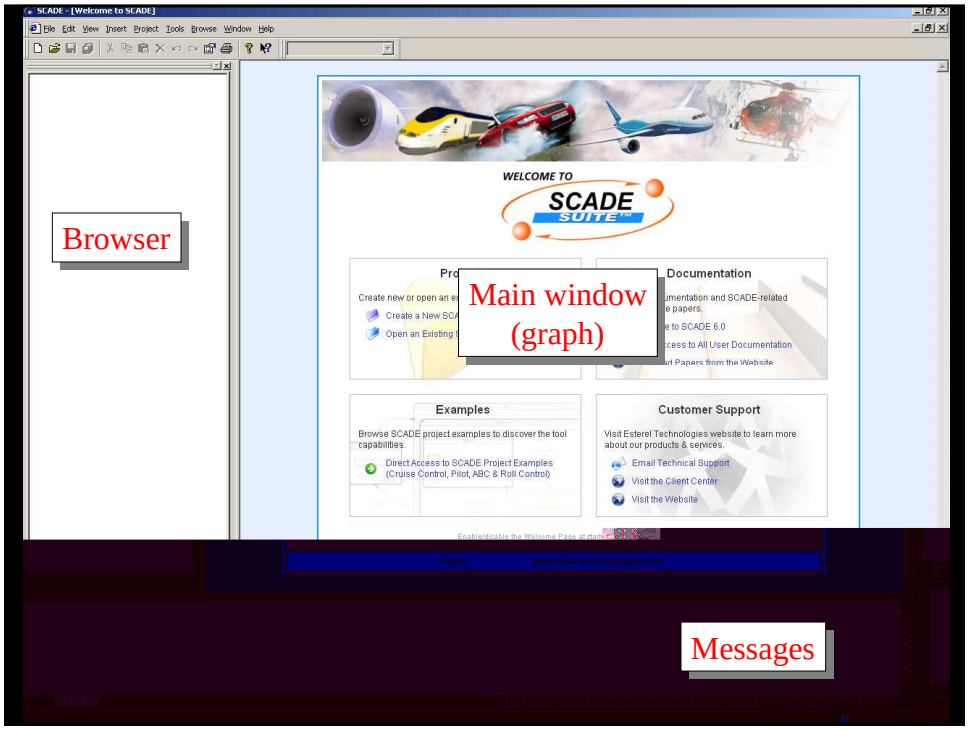


Variables representation

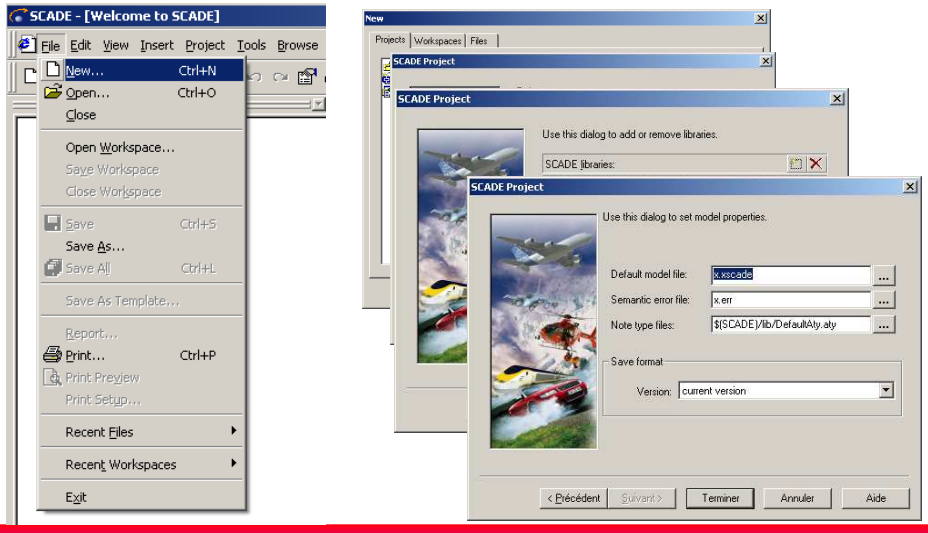


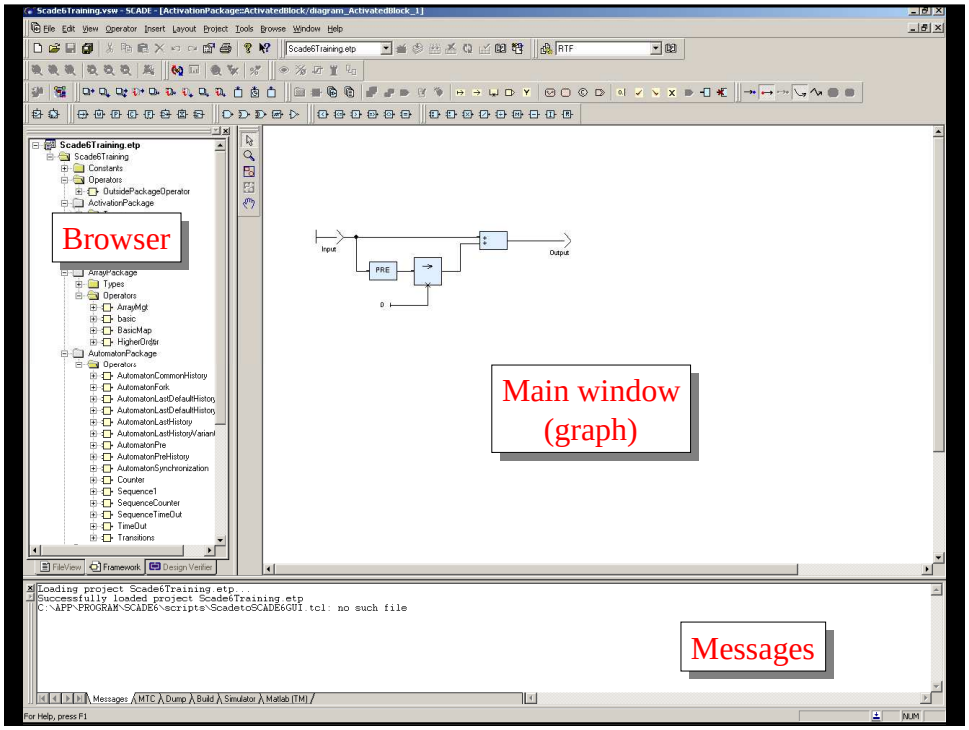
Overview

- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity



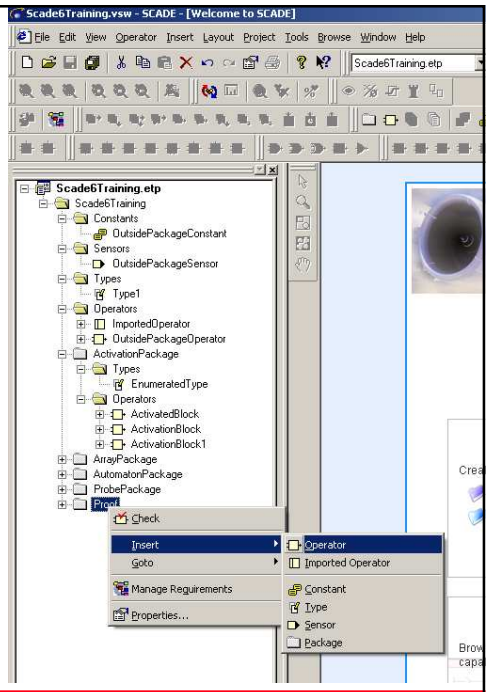
Creating a new project



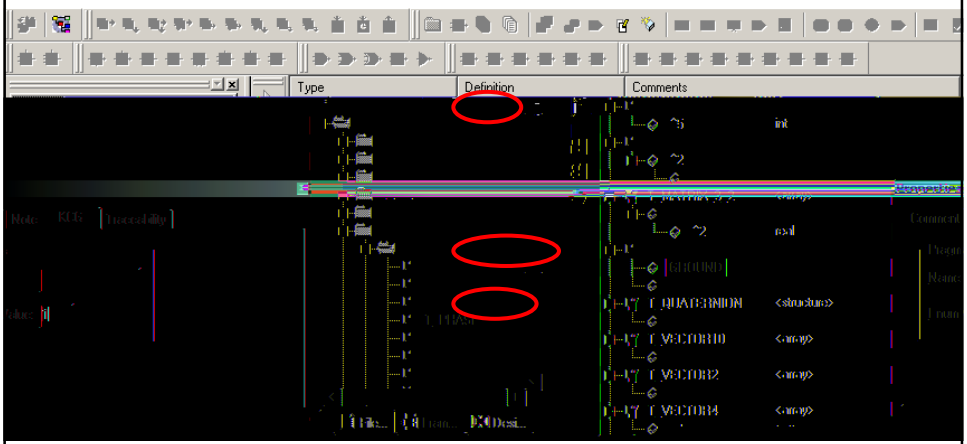


Packages

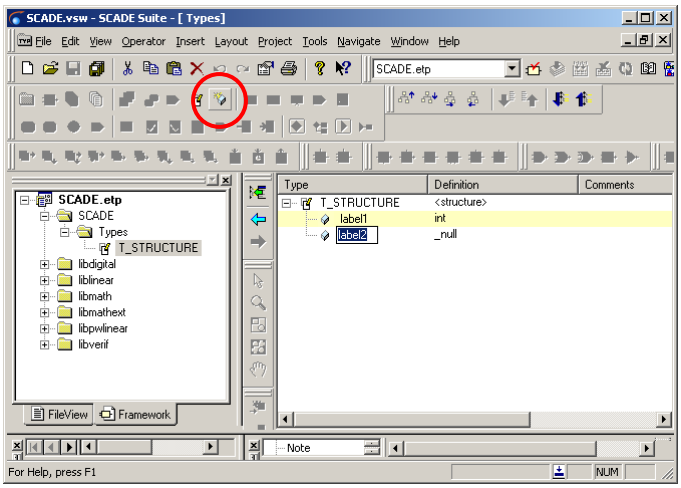
- Definitions of
  - Scade operators
  - Imported operators
  - Constants
  - Types
  - Sensors
  - Packages
- Inside or outside a package



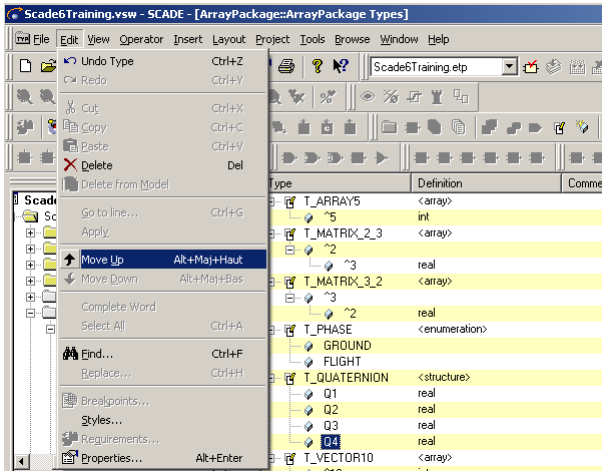
Management of types (1/3)



Management of types (2/3)



Management of types (3/3)

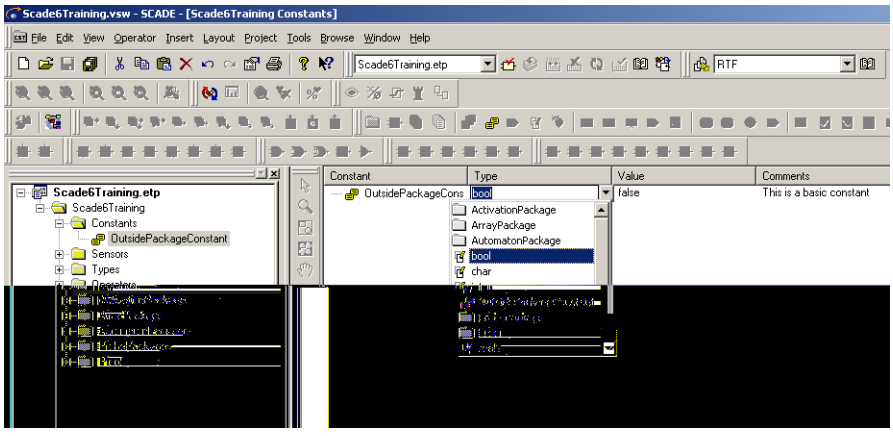


Integers and reals

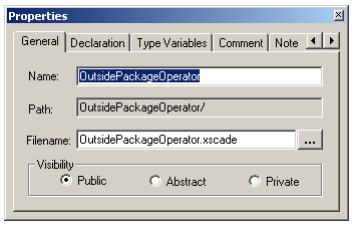
- Integers
  - Binary 0b01001
  - Octal 0563
  - Decimal 9637
  - Hexadecimal 0xAF6C
- Encoding
  - short, int, long
  - Float, double

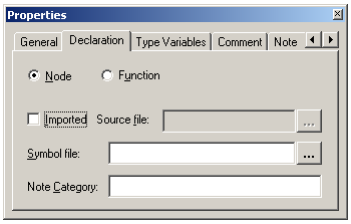
} Shall be defined by the user

Defining a constant



Changing an object properties

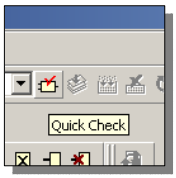




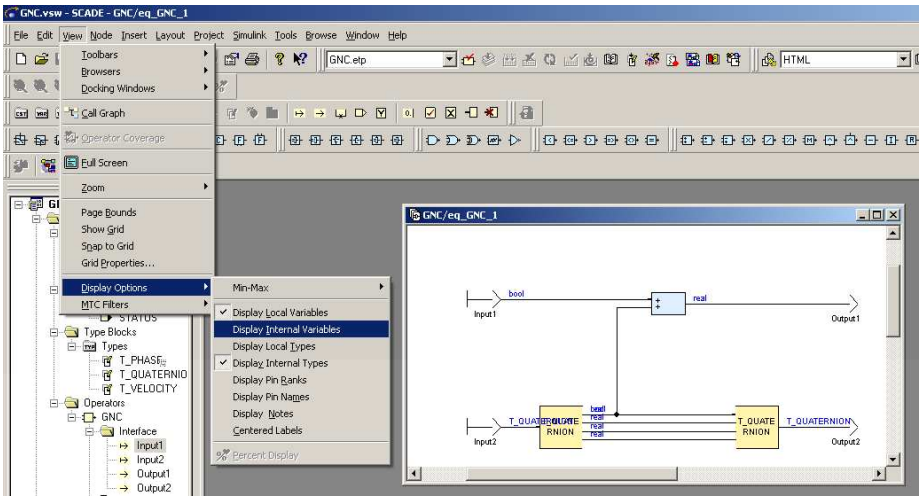
Keyword list

- Scade keywords
  - abstract, activate, and, assume, automaton, bool, case, char, clock, const, default, div, do, else, elsif, emit, end, enum, every, false, fby, final, flatten, fold, foldi, foldw, foldwi, function, guarantee, group, if, imported, initial, int, is, last, let, make, map, mapfold, mapi, mapw, mapwi, match, merge, mod, node, not, numeric, of, onreset, open, or, package, parameter, pre, private, probe, public, real, restart, resume, returns, reverse, sensor, sig, specialize, state, synchro, tel, then, times, transpose, true, type, unless, until, var, when, where, with, xor
- + Targeted programming language keywords

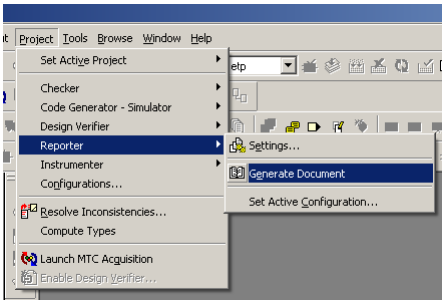
Quick check

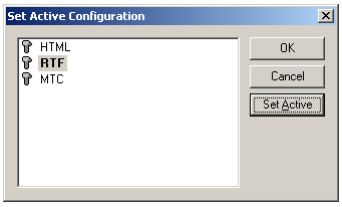


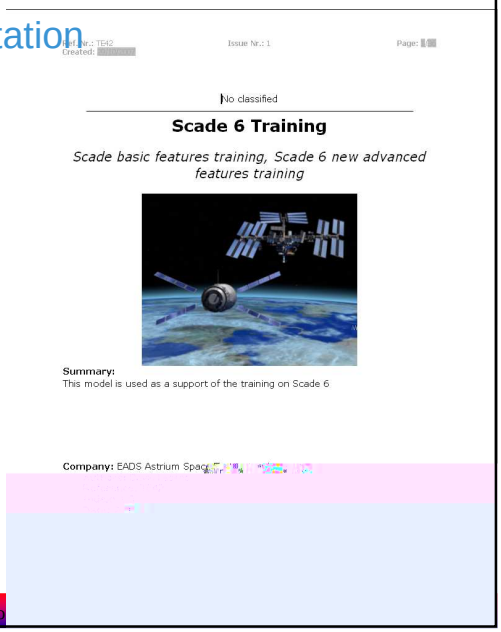
Display types / variable names / …



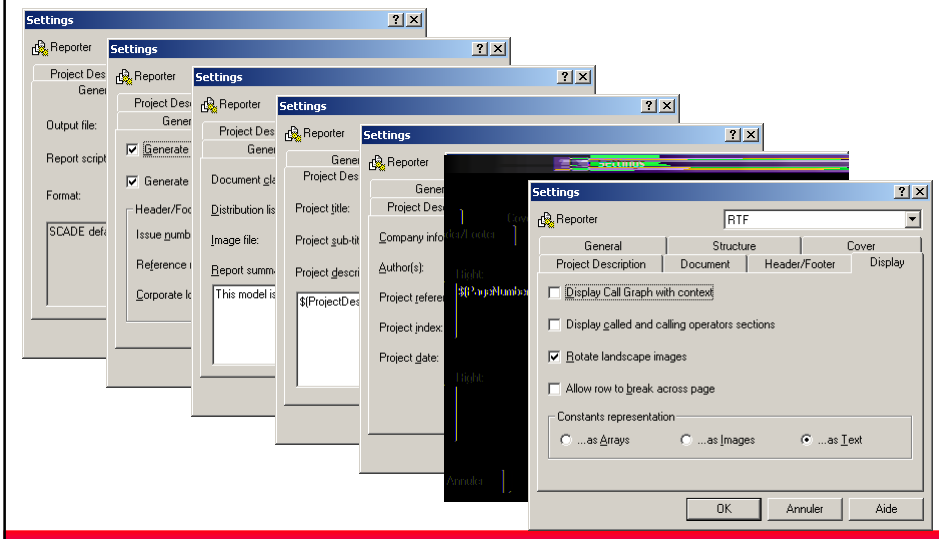
Generation of documentation



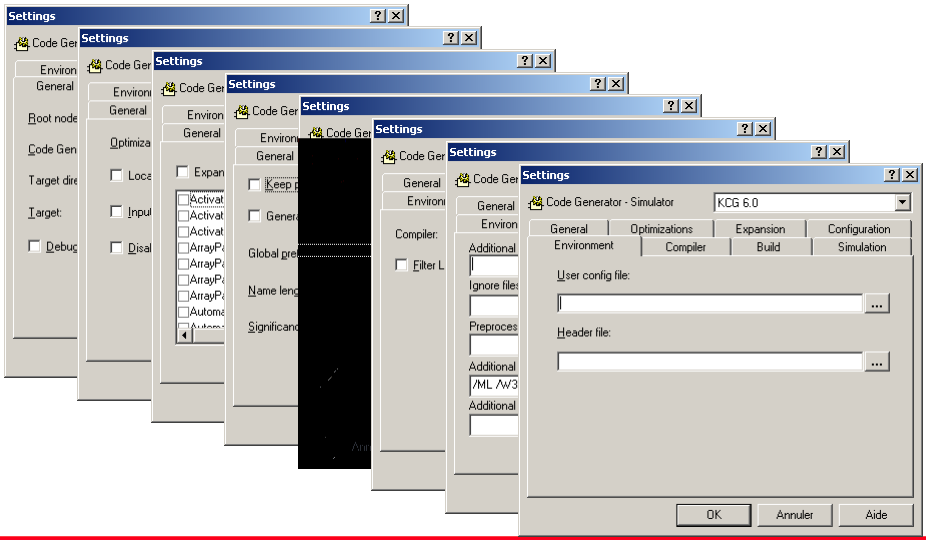




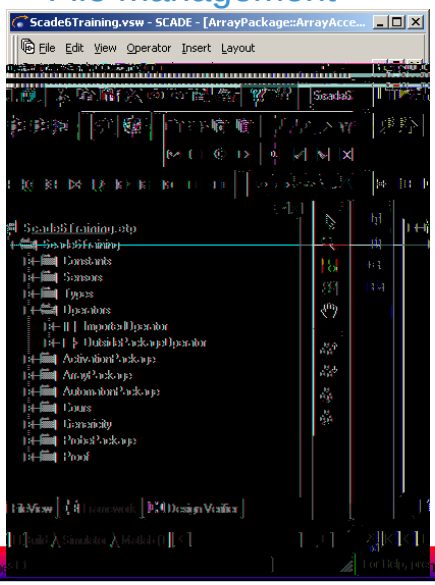
Report customization



Code generation customization



File management



Scade6Training.xscade
ImportedOperator.xscade
OutsidePackageOperator.xscade
ActivationPackage.xscade
ArrayPackage.xscade
AutomatonPackage.xscade
Cours.xscade
Genericity.xscade
ProbePackage.xscade
Proof.xscade

Documentation

- Welcome to SCADE 6.o
- Getting Started with SCADE
- Scade Language Tutorial
- Scade Language Primer
- SCADE Libraries Manual
- SCADE UML Metamodel Card
- SCADE Gateway for Rhapsody Guidelines
- Simulink™ Gateway Guidelines

age Reference Manual
Manual
nical Manual
Simulink™ Modeling Guidelines
RTOS Wrapper Guidelines
About Requirements Management Gateway documentation, check from RMGC interface at Help > Documentation or Coupling Notes
Scade Langu
SCADE User
SCADE Tech

Overview

- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity

“IF” operator

x = if b then y else z

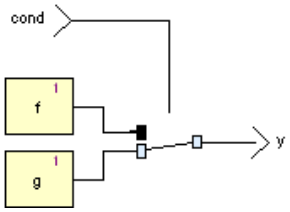
If “b” is true, “x” takes the value “y”,
else, “x” takes the value “z”

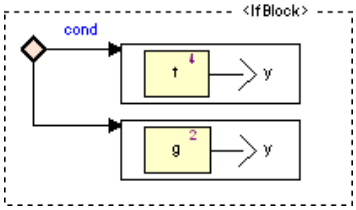
Note:

Does not mean

If “b” is true, execute “y”,
else, execute “z”

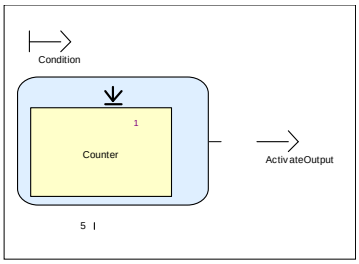
If versus IfBlock






```
void IfBlockWithNodes(bool cond) {
  int y;
  if (cond) { y = f (); }
  else { y = g (); }
}
```

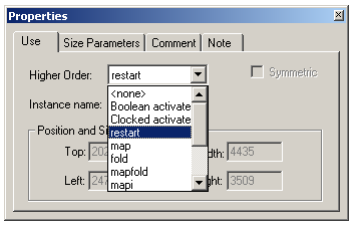
Activation conditions



- Activation condition

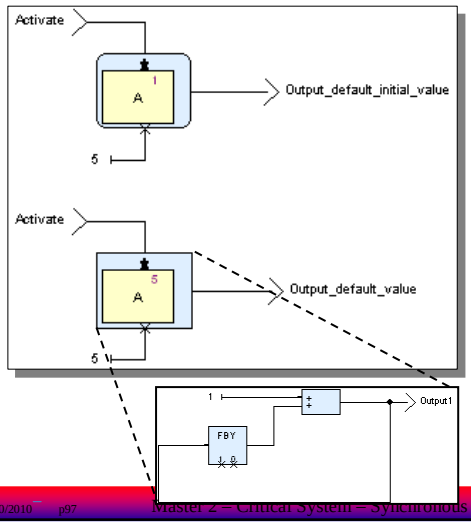
  - Condition true = Block activated
  - Condition false = Previous outputs used (was “condact” in Scade 5)

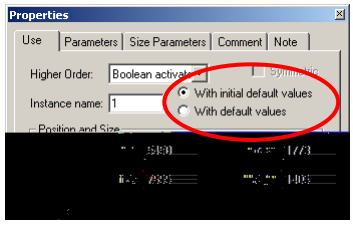
    or Default values
  - Init values before first use

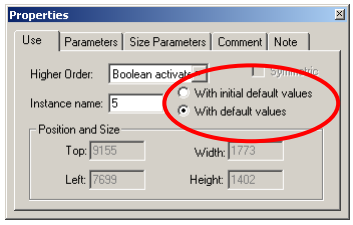


- Restart condition
  - Condition true = Internal memory reset

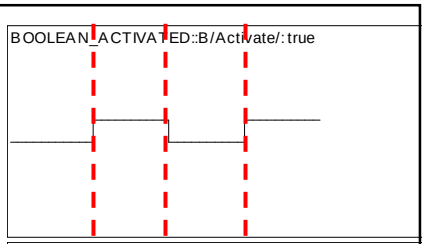
Activation: Example (2/3)

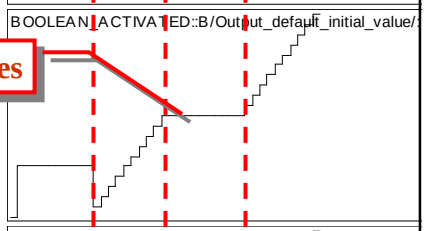


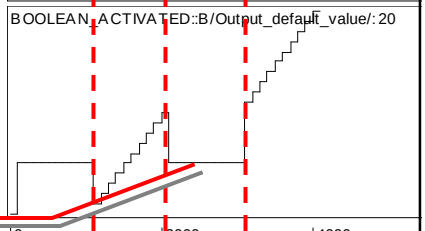


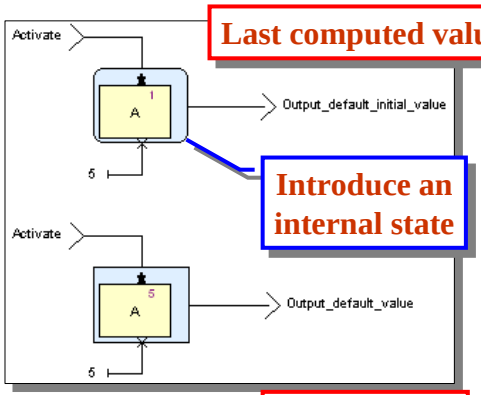



```
if (Activate) {
    Output_default_initial_value = A();
    Output_default_value = A();
} else {
    if (init) Output_default_initial_value = 5; }
    Output_default_value = 5;
init = false;
```







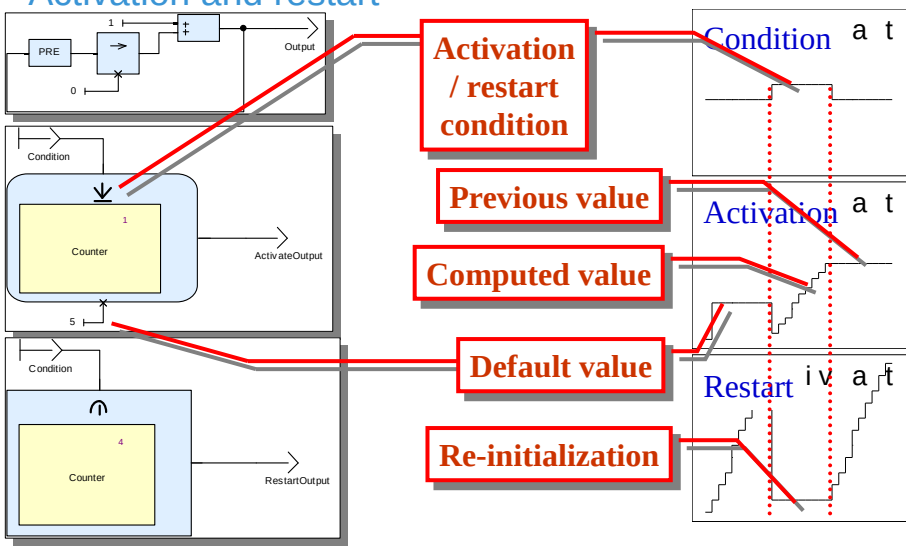


Last computed values

Introduce an internal state

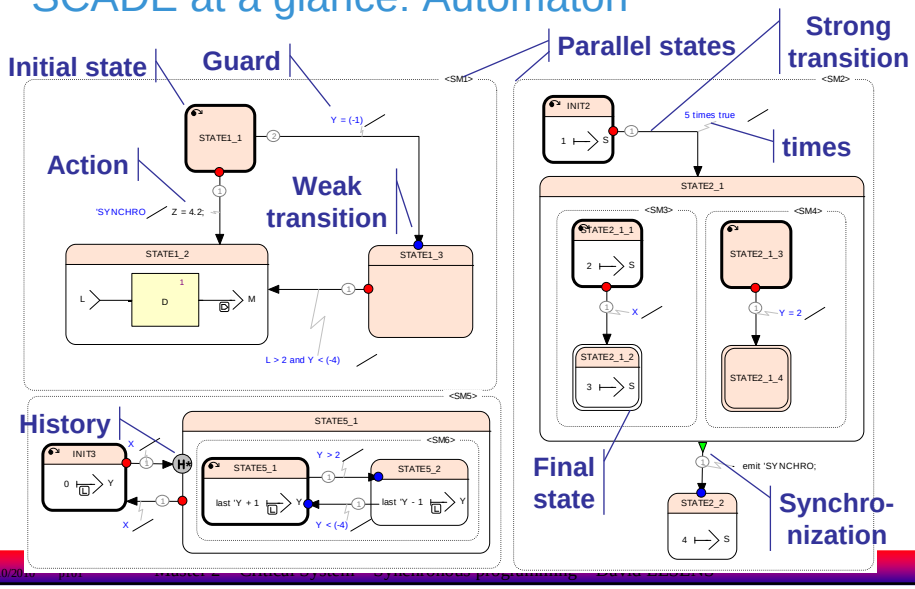
Default value

Activation and restart



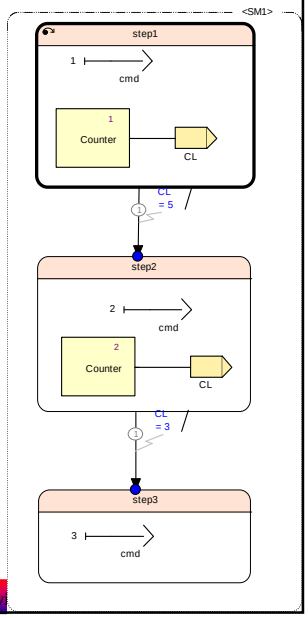
Overview

SCADE at a glance: Automaton



Data flow and automata

- A node is composed of
  - Equations (data-flow)
  - Automata (event driven)
- An automaton is composed of
  - States
  - Transitions
- A state is composed of
  - Equations
  - Automata



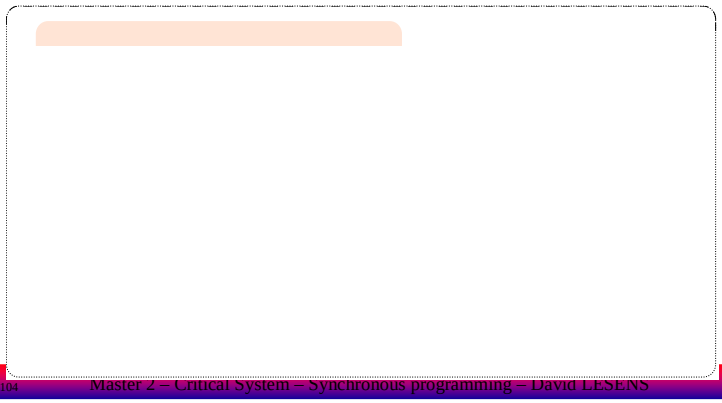
Principles of Automata

- Semantics equivalence
  - There exists a data-flow model semantically equivalent to any automaton
- Automaton scheduling
  - At most one transition fired *per cycle*
  - Exactly one active state per *cycle*

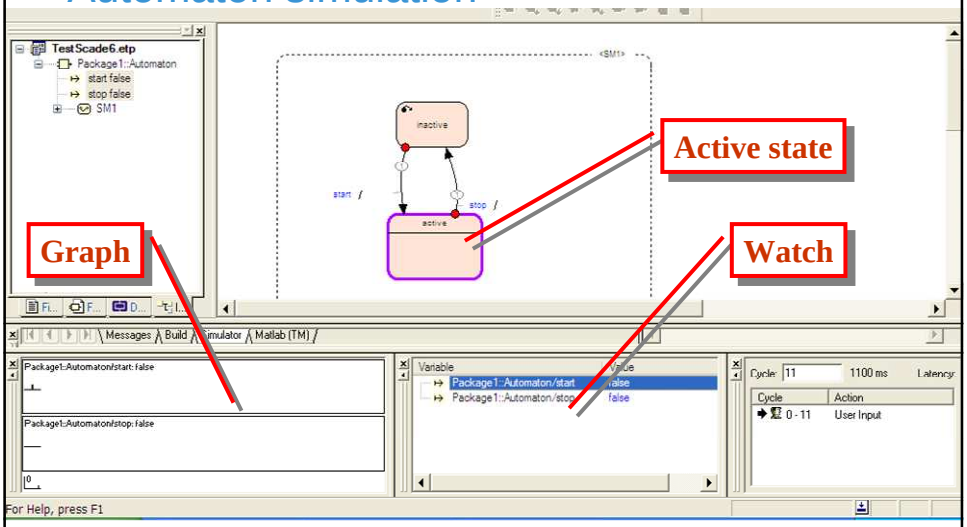
    (except then parallel states are defined)

States

- A state can be
  - An initial state / a final state
  - Hidden / nested

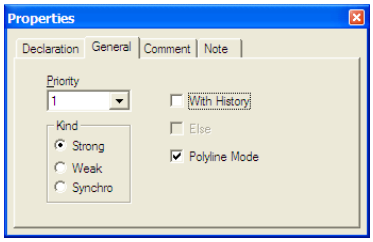


Automaton simulation

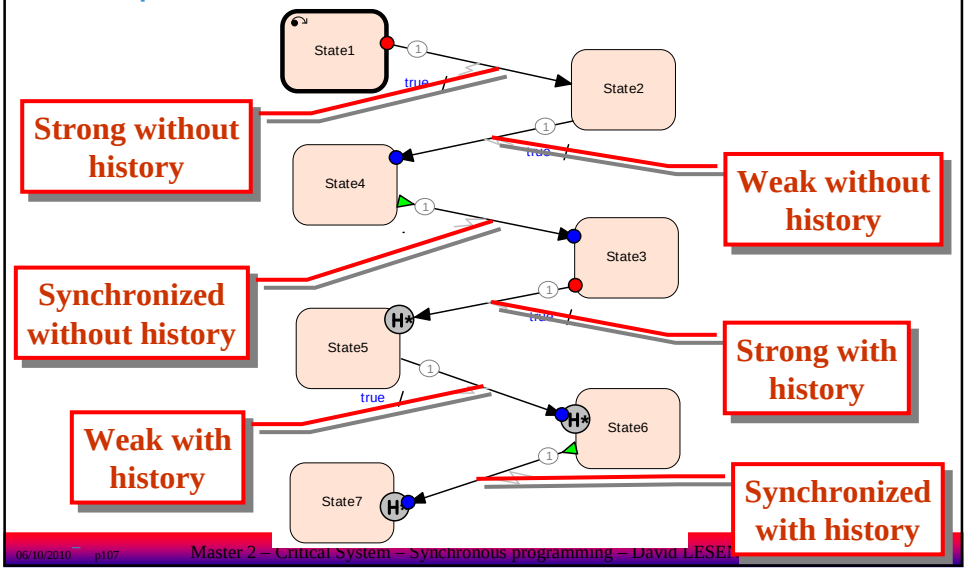


Transitions

- A transition can
  - have a **weak** pre-emption
  - have a **strong** pre-emption
  - be **synchronized**
- It can have
  - A guard
  - An action
- It has a priority
- It can be with or without a history



Graphical transitions

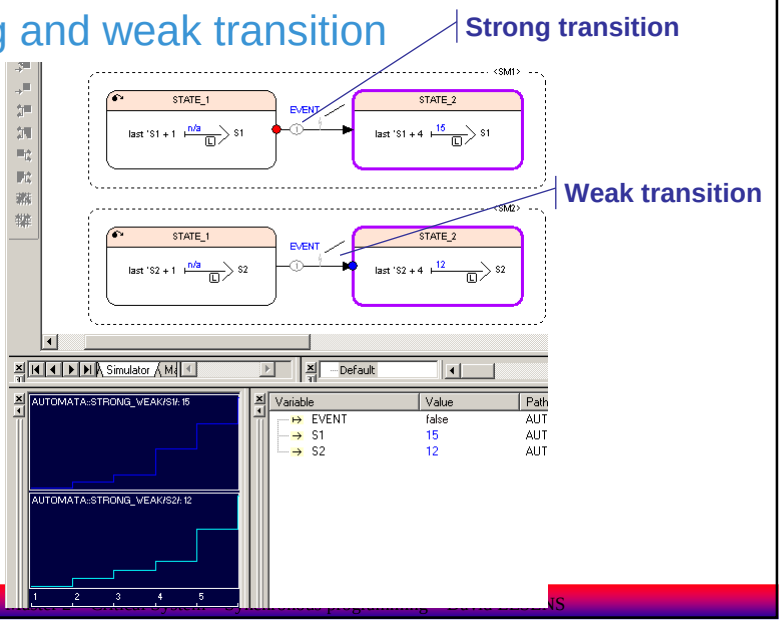


Strong and weak transitions

- Strong transition
  - The transition is triggered before the state execution
  - ➔ The guard can not depend on the current value of a data
- Weak transition

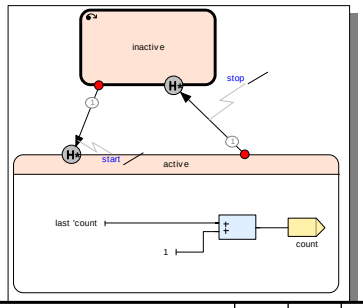
  (or "*weak delayed*")
  - The state is executed before the transition triggering
  - ➔ The guard can depend on the current value of a data

Strong and weak transition

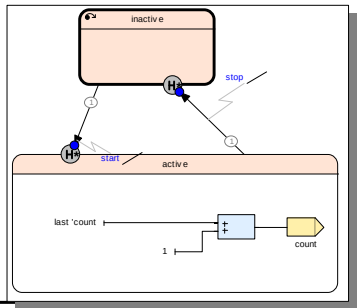


Example (1/2)

Strong transition



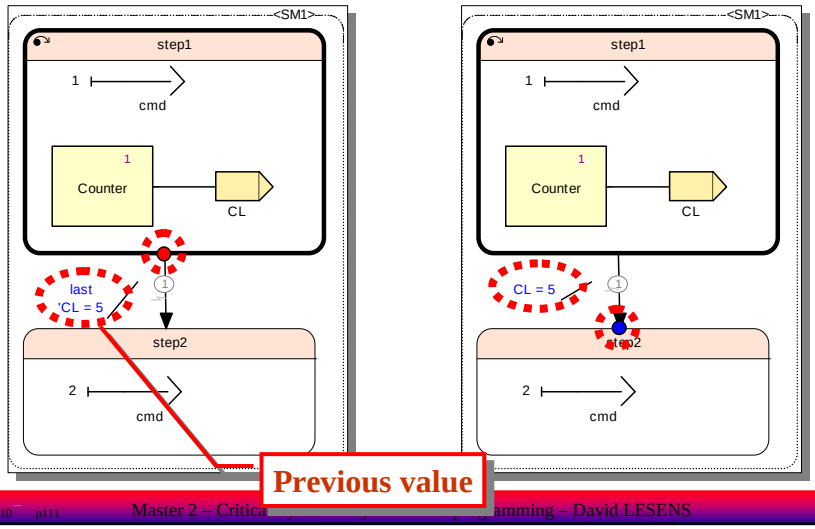
Weak transition



start				T							
stop								T			
count strong	0	0	0	1	2	3	4	4	4	4	4
count weak	0	0	0	0	1	2	3	4	4	4	4

Example (2/2)

The behaviours of the two following models are equivalent

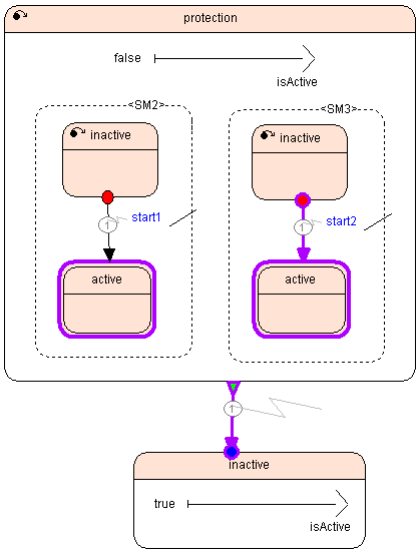


Synchronized transition

- A synchronized transition
  - Has no guard
  - Is triggered as soon as all nested automata reach a final state

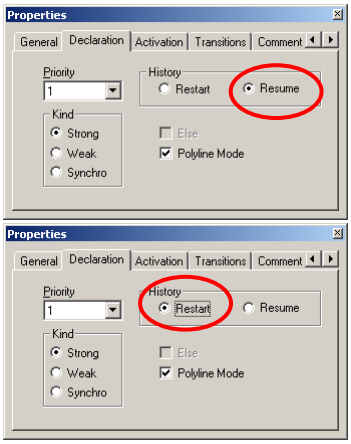
Example

- **Start1 received**
  - Still in protection state
- **Start2 received**
  - Final states reached
- **Transition inactive triggered**

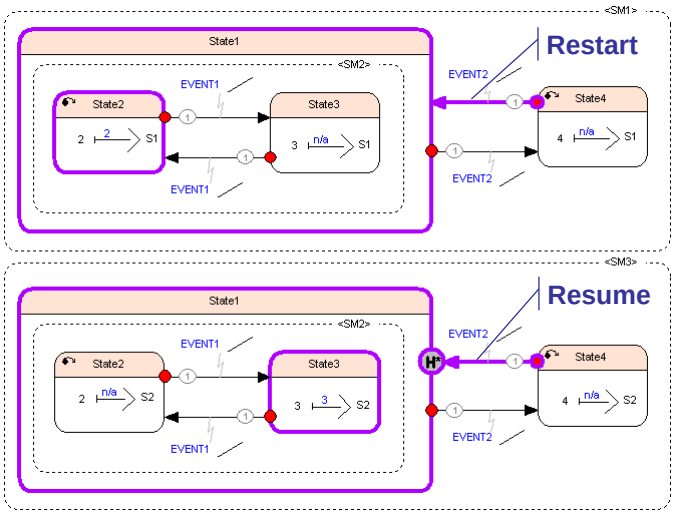


Transition history

- Transition without history
  - The state resumes its execution
  - The memories are reset
- Transition with history
  - The state resumes its execution
  - The memories are not reset
- Two types of memories
  - PRE : local to the state
  - LAST : common to the node



Transition with history

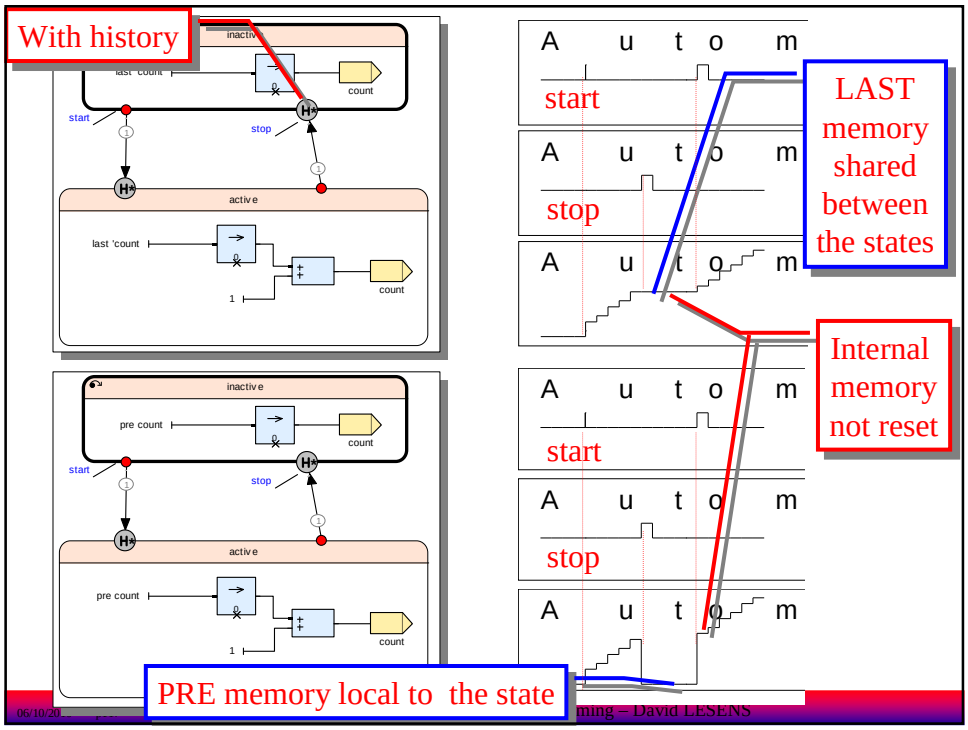


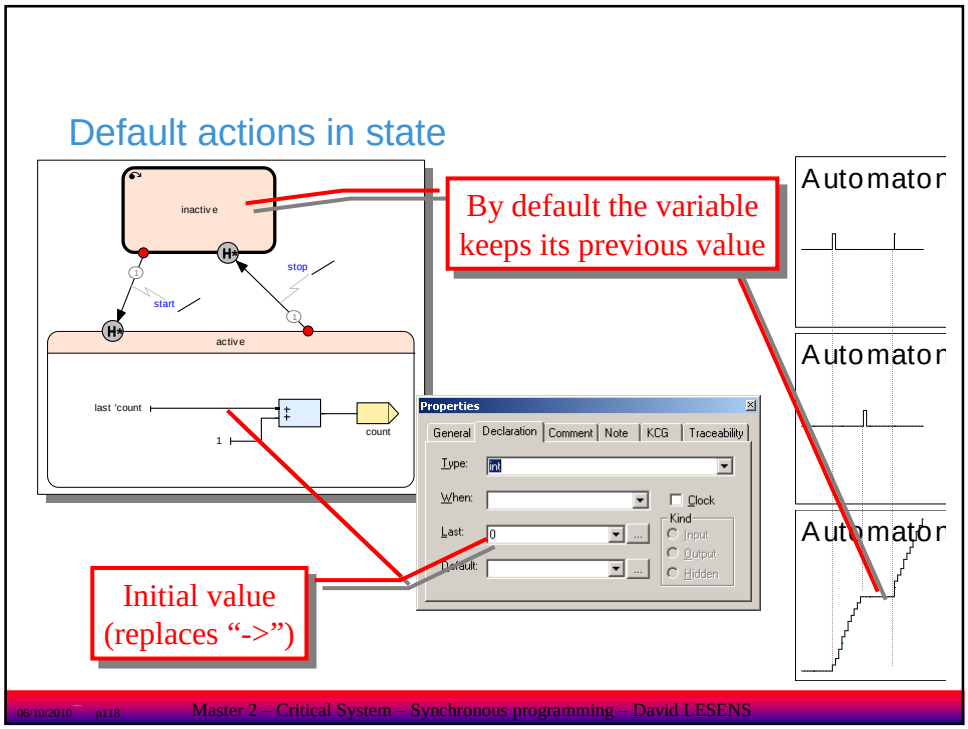
Shared memory

- Data flow point of view
  - Access to the last value of a flow in its scope
    - "pre *expression* "
- Mode automata point of view
  - Access to values computed in other states
    - "last 'x"

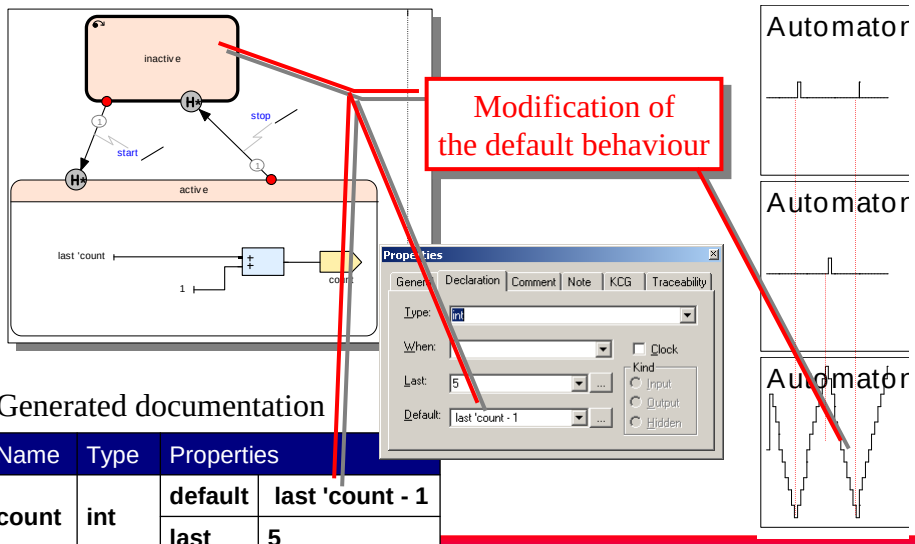
  ("x" is a named flow, not an expression

  ➔ utilization of ')


With history
inactive
last 'count
count
start
stop
active
last 'count
1
count
inactive
pre count
count
start
stop
active
pre count
1
count
A u t o m
start
A u t o m
stop
A u t o m
A u t o m
start
A u t o m
stop
A u t o m
LAST memory shared between the states
Internal memory not reset
PRE memory local to the state
06/10/20
ming – David LESENS


Default actions in state
inactive
stop
start
active
last 'count
1
count
By default the variable keeps its previous value
Automator
Automator
Automator
Properties
General
Declaration
Comment
Note
KCG
Traceability
Type:
int
When:
Clock
Last:
0
Default:
Kind
Input
Output
Hidden
Initial value (replaces "->")
06/10/2010
p118
Master 2 – Critical System – Synchronous programming – David LESENS

Modifying the default action

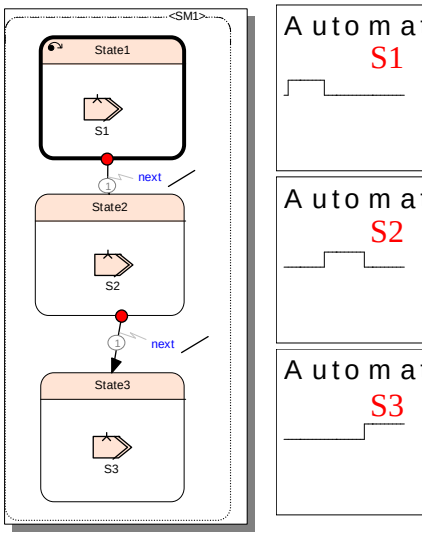


Generated documentation

Name	Type	Properties	
count	int	default	last 'count - 1
		last	5

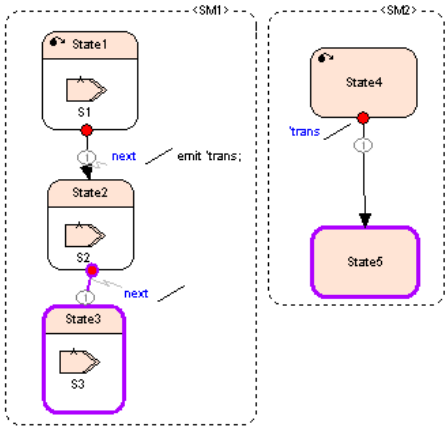
Signals

- A signal can be
  - Present ➔ true
  - Absent ➔ false
- A signal can not be
  - An input / output
- ≠ Boolean value
  - A Boolean value keeps its previous value then non updated in a state



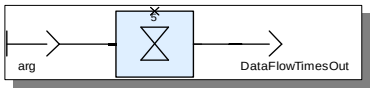
Composition and communication

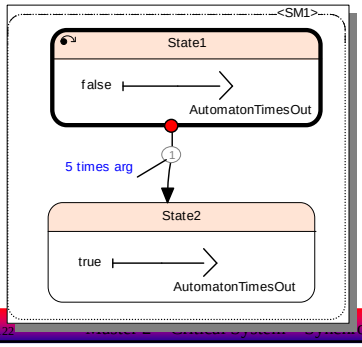
- A signal can be
  - Emitted in a state
  - Emitted on a transition
- A transition can be triggered by a signal

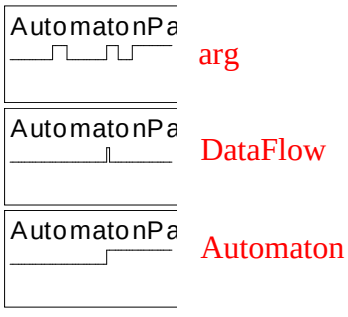


Factor

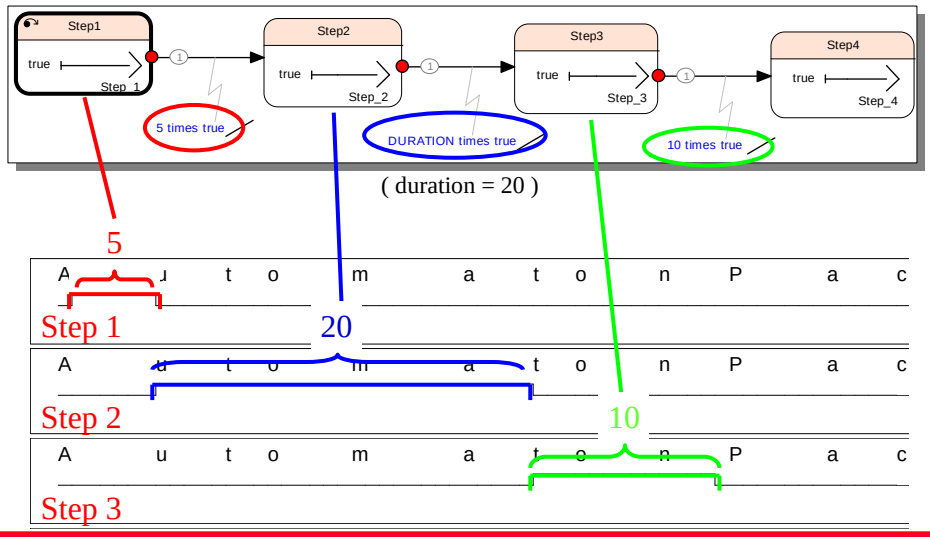
- A factor specifies on many time a condition must be true
  - In a data flow view
  - In a guard (automaton)







Time-out with factor

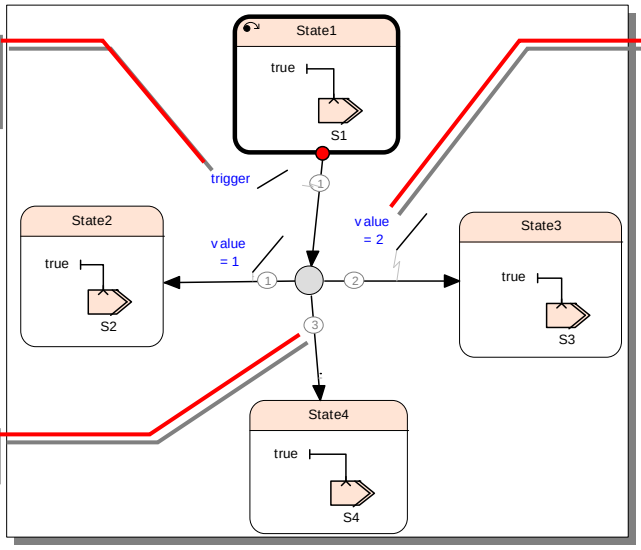


Fork

Common guard

Specific guard

Priority

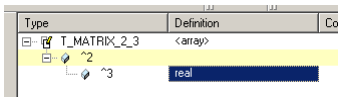


Overview

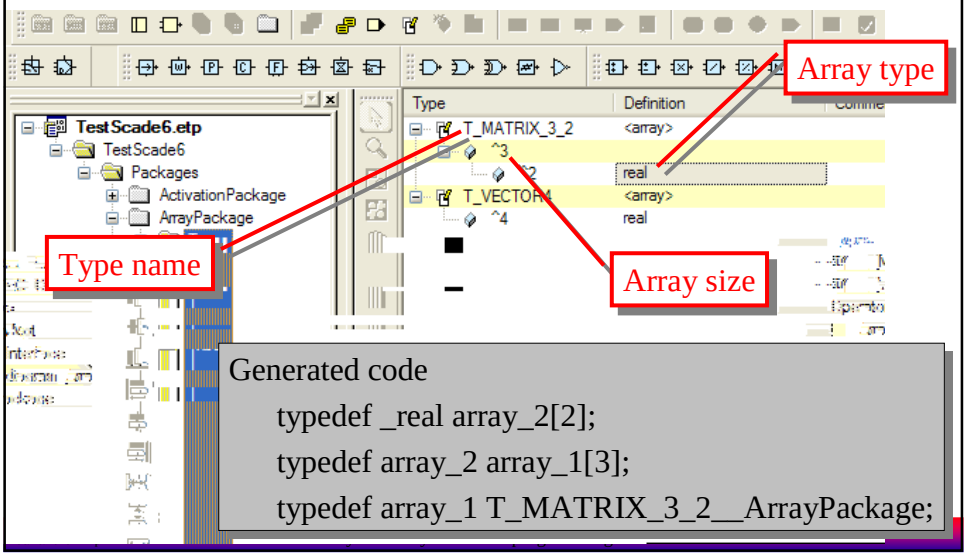
- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity

Arrays definition

- Restrictions
  - Static size
  - First element = index 0
- Definitions
  - type VECTOR = real ^ 4 ;
  - type MATRIX_2_3 = real ^ 3 ^ 2 ;
    - 2 lines, 3 columns
    - typedef real LINE_3[3];
    - typedef LINE_3 MATRIX_2_3 [2];



Editing array types



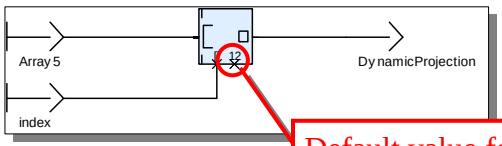
Generated code

```
typedef _real array_2[2];
typedef array_2 array_1[3];
typedef array_1 T_MATRIX_3_2__ArrayPackage;
```

Array access

Array5=[2,4,6,8,10], Index=3

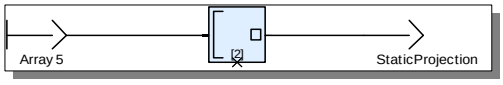
- Dynamic



Index = 3 ➔ Output = 8
Index = 10 ➔ Output = 12

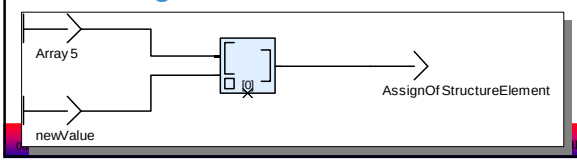
Default value for index out of range

- Static



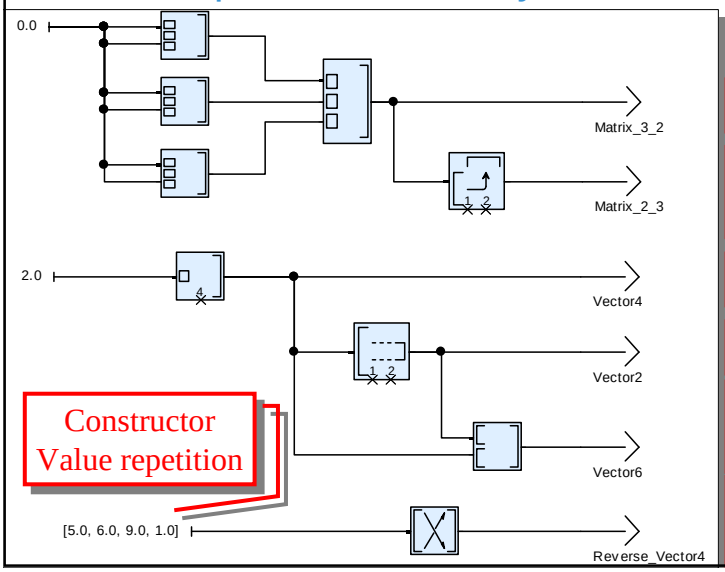
Output = 6

- Assignment



newValue = 3 ➔
Output = [3,4,6,8,10]

Some operators on arrays



Constructor
Value repetition

- Data to Vector
- Transpose of an Array
- Scalar to Vector
- Slice of a vector
- Concatenation of Arrays
- Reverse of a Vector

Overview

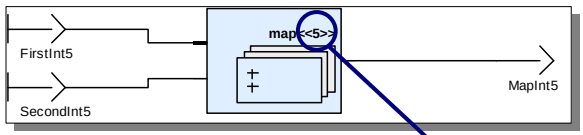
- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity

Iterations

- Equivalent to “for” in C

Map / Mapi / Mapw / Mapiw
Fold / Foldi / Foldi / Foldiw

Map

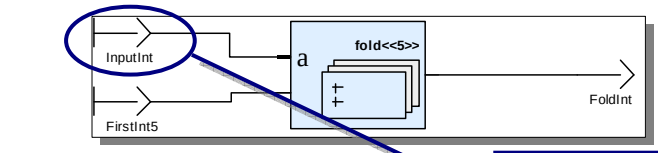


Size of the input vector

```
for (i = 0; i < 5; i++) {
        MapInt5[i] = FirstInt5[i] + SecondInt5[i];
}
```

MAP: Apply the operator successively on each element of the input vector(s) element[i] .element’[i]

Fold



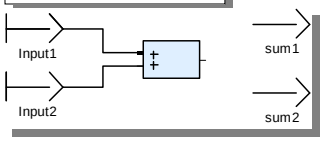
First element of the iteration

```
FoldInt = InputInt;
for (i = 0; i < 5; i++) {
        FoldInt = FoldInt + FirstInt5[i];
}
```

FOLD: Apply recursively the operator on input vector element[i] .element[i+1]

Mapfold

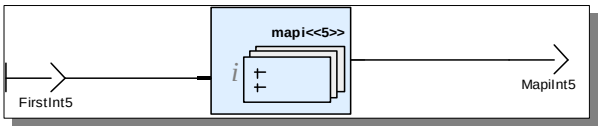
Operator add_2




```
MapFold1Int = InputInt;
for (i = 0; i < 5; i++) {
        add_2_ArrayPackage(MapFold1Int, FirstInt5[i],
                           &MapFold1Int, &MapFold2Int[i]);
}
```

Nodes used with a mapfold iterator should duplicate their output
We obtain both results at the same time

Mapi = Map with iterator as input

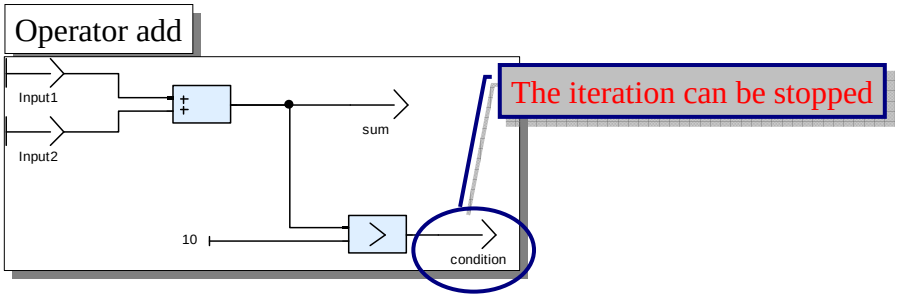



```
for (i = 0; i < 5; i++) {
        MapiInt5[i] = i + FirstInt5[i];
}
```

The index of the iteration
is the first argument of the node

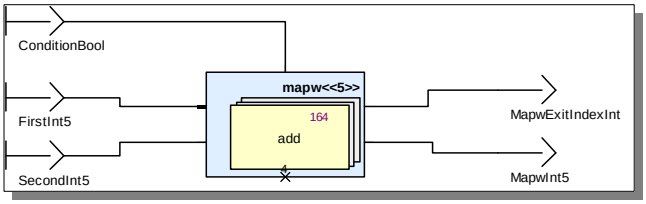
Mapw / Foldw = Partial operators

- Capability to stop an iteration on a Boolean condition computed by the operator



As soon as the condition is false, the iteration is topped

Mapw = Map partial operator

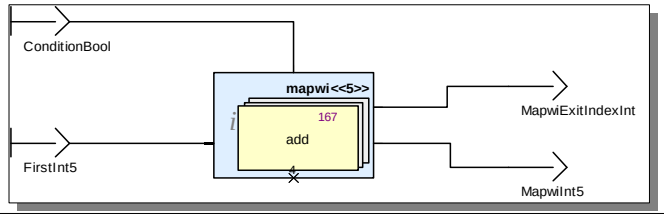



```
MapwExitIndexInt = 0;
for (i = 0; i < 5; i++) {
 if (ConditionBool) {
   add(FirstInt5[i], SecondInt5[i], &ConditionBool, &MapwInt5[i]);
   MapwExitIndexInt = i + 1;
 } else { MapwInt5[i] = 4; } }
```

The iteration can be stopped

It is recommended to not use this operator (WCET)

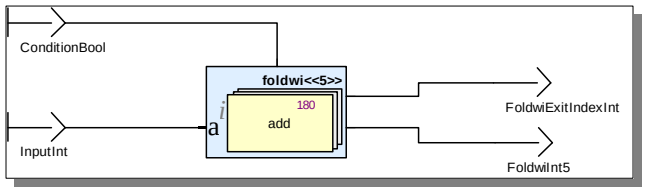
Mapwi = Mapi + Mapw




```
MapwiExitIndexInt = 0;
for (i = 0; i < 5; i++) {
 if (ConditionBool) {
   add(i, FirstInt5[i], & ConditionBool, &MapwiInt5[i]);
   MapwiExitIndexInt = i + 1;
 } else { outC->MapwiInt5[i] = 4; } }
```

The iterator is the first argument

Foldwi = Foldi + Foldw




```
FoldwiInt5 = InputInt; tmp = ConditionBool;
for (i = 0; i < 5; i++) {
  if (ConditionBool) { break; }
  add(i, FoldwiInt5, & ConditionBool, &tmp);
  FoldwiInt5 = tmp;
}
FoldwiExitIndexInt = i;
```

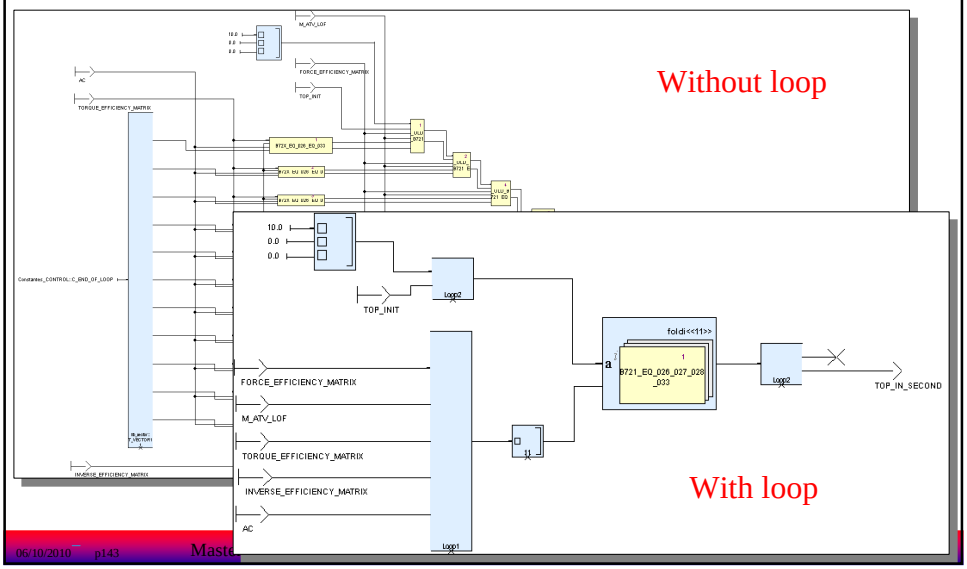
The iteration can be stopped

The input flow is the iterator

Iteration summary

- Map = Successive application
- Fold = Recursive application
- Mapfold = Map + Fold
- Mapi = Map with iterator as input
- Foldi = Fold with iterator as input
- Mapw = Map partial operator
- Mapwi = Mapi + Mapw
- Foldw = Fold partial operator
- Foldwi = Foldi + Foldw

Example 1

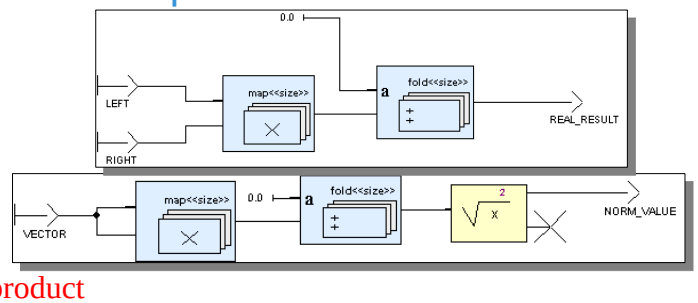


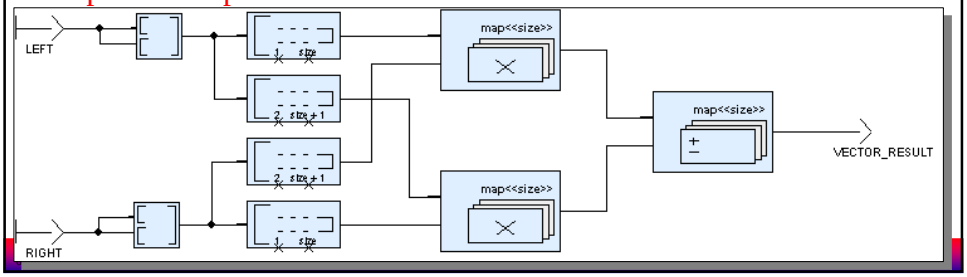
Example 2: cross product

Compute scalar product

Compute vector norm

Compute cross product





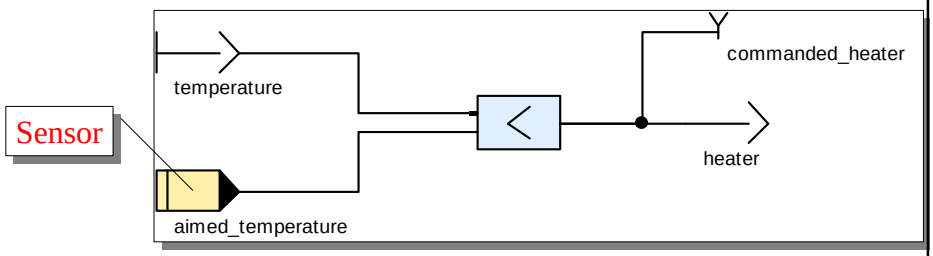
Overview

- Synchronous model
- Introduction to the Scade language
- Editing a Scade model
- Activation conditions
- Automata
- Arrays
- Iterations
- Global flows: Sensors and probes
- Genericity

Sensors

- Sensor: Global system input

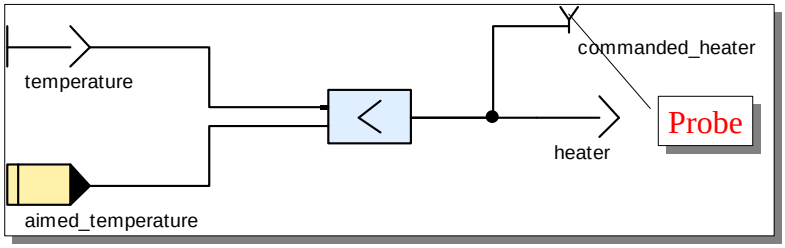
```
Input   temperature
Output heater
```



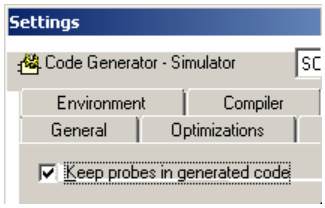

```
extern _int aimed_temperature__ProbePackage;
```

Probes

- Probe: Global system output

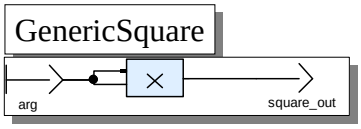



```
typedef struct {/* context */
_bool heater; /* outputs */
_bool commanded_heater; /* probes */
} C_controller__ProbePackage;
```

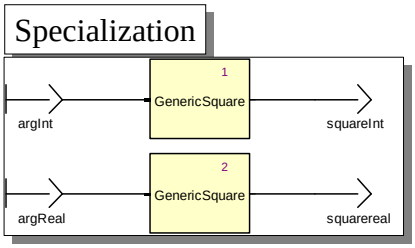


Overview

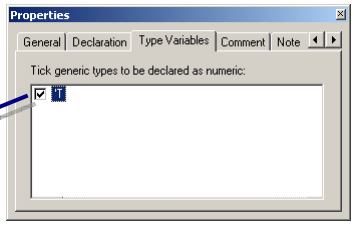
Generic operator definition



	Name	Type
Input	arg	'T
Output	square_out	'T



Name	Generic Type
'T	numeric



Definition of a generic numeric type

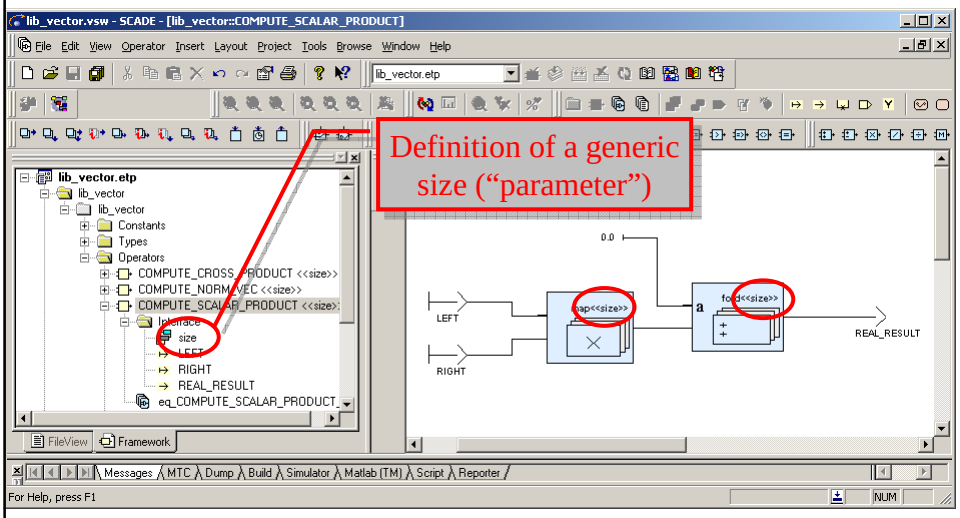
Generic operator instantiation

```
int GenericSquare_int ( int arg ) {
        int square_out;
        square_out = arg * arg;
        return square_out;
}
```

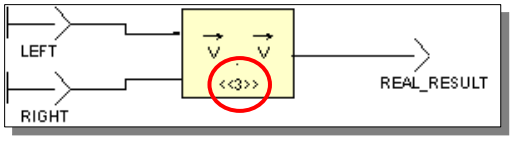
```
real GenericSquare_real ( real arg ) {
        real square_out;
        square_out = arg * arg;
        return square_out;
}
```

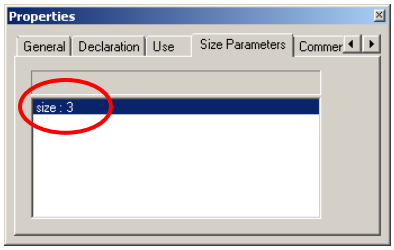
```
void Specialization(    int argInt; real argReal;
                        int squareInt; real squarereal; ) {
*squareReal = GenericSquare_real ( argReal );
*squareInt =  GenericSquare_int ( argInt );
}
```

Definition of parameters



Parameter instantiation






```
REAL_RESULT = 0.0;
for (i = 0; i < 3; i++) {
   REAL_RESULT = REAL_RESULT + (*LEFT)[i] * (*RIGHT)[i];
}
return REAL_RESULT;
```

Overview

- Critical real-time embedded software
- Principles of the approach
  - Introduction
  - Formal semantics
- SCADE
- Model validation

Semantics verification (1/2)

Semantics of a SCADE model

- Syntax
- Typing verification
  - Types compatibility
    - Example: Integer ≠ real
- Non uninitialized variables
- Temporal causality
- …

Temporal causality

SCADE is an equational language

- The evaluation order depends only on data flows

x = y;
y = z;

→ "y = z" evaluated first
"x = y" evaluated secondly

x = y;
y = z;
z = x;

→ Impossible computation of the evaluation order
"x = y = z = x = …"

Causality problem

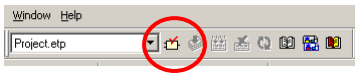
Semantics verification (2/2)

A SCADE model with a correct semantics is:

- Complete
- Consistent
- Implementable

➔ The good properties of a specification

➔ “Semantics check” to be systematically performed



But does the software behave as expected?

What is testing?

Compare the observed behaviour
with the expected behaviour

- Several levels of test
  - Unitary / integration / validation / system qualification
  - Host / target
  - Real equipment / simulator
  - “White” box / “Black” box

At code or model level

Objectives of unitary tests

- Robustness
  - Absence of “runtime error”
- Functional validity
  - Comparison with the expected results
- Contractual objectives
  - **Coverage**
    - Intuitively satisfactory
    - Measurable
    - But not a proof of exhaustiveness

Unitary tests: Coverage

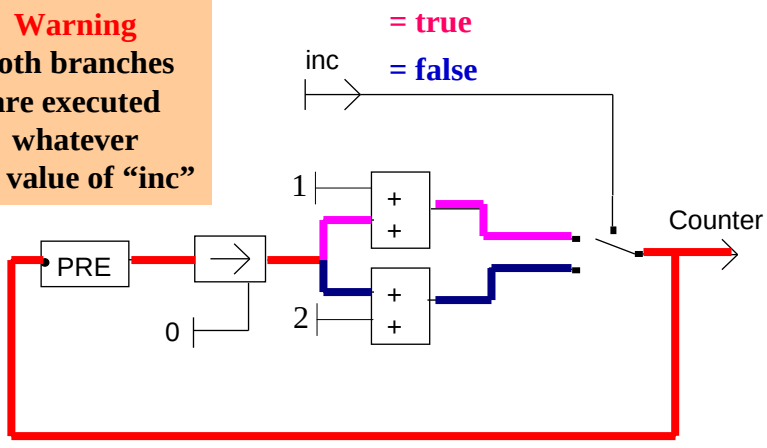
```
Procedure f(x : in real; y: in real; z : out real)
        if (x > 1.0) or (x < -1.0) then
                z := y/x;
        else
                z := y;
        if z < 2.0 then
                z = 2.0;
```

Coverage

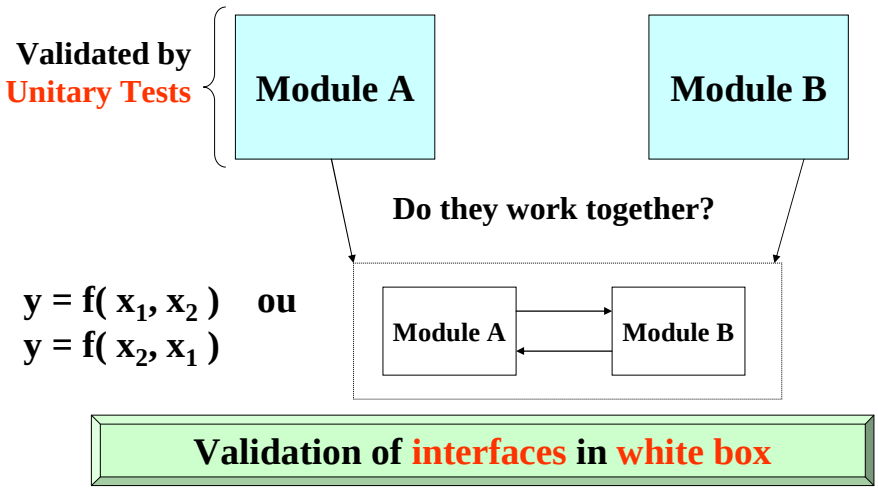
- **branch** — (x=2.0, y=6.0), (x=-1.0,y=1.0)
- **decision** — + (x=-2, y=3.0)
- **path** — + (x=2.0, y=1.0), (x=0.5,y=2.0)

Coverage of a SCADE model

Warning
Both branches are executed whatever the value of "inc"



Integration test



Limit of the white box approach

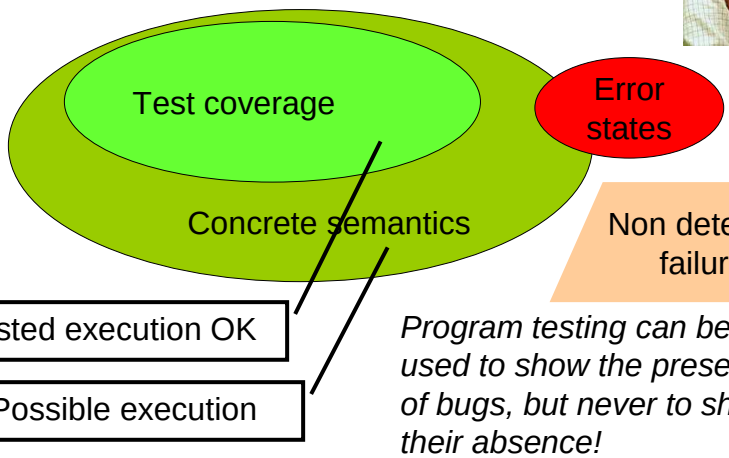
- The presence of a spy may modify the real time behaviour
- What happens if the debugger / simulator has … a bug?

Validation

- Black box tests
  - Control of the inputs
  - Observations of the outputs
- On host or on target
  - Tests on target are more expensive

Non intrusive

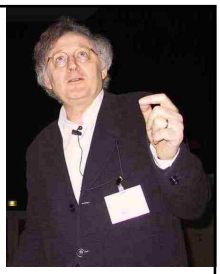
Software testing

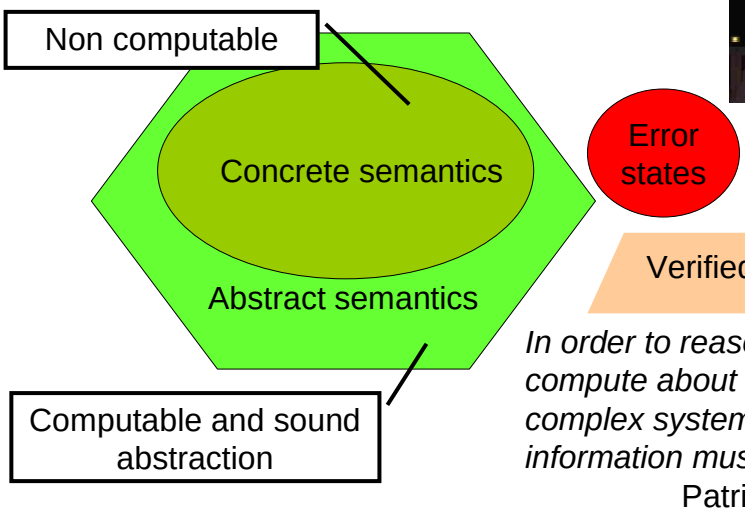


Program testing can be used to show the presence of bugs, but never to show their absence!

Edgser W. Dijkstra

Principle of the proof

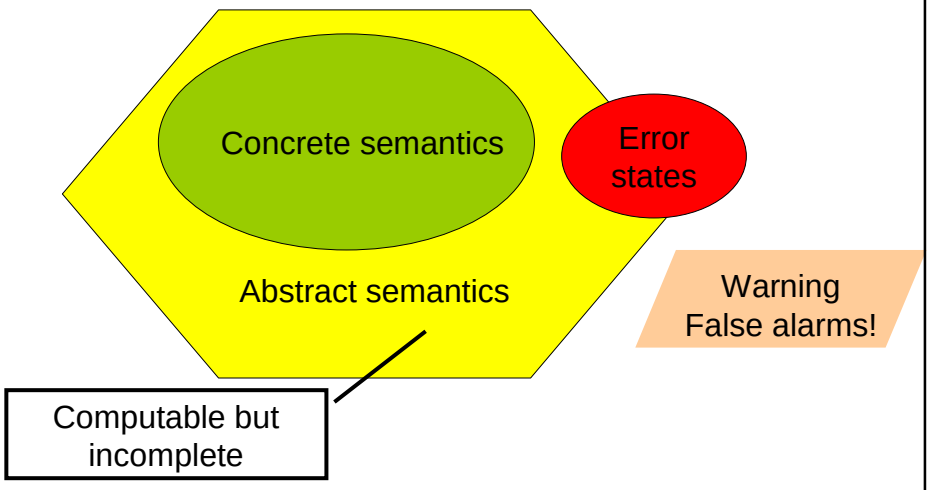




In order to reason or compute about a complex system, some information must be lost

Patrick Cousot

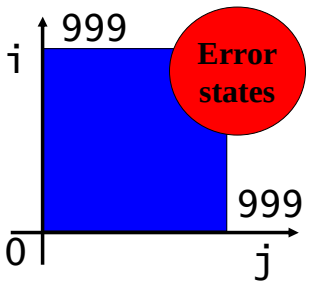
Proof limitation



Example (1)

```
int a[1000];
for (i = 0; i < 1000; i++) {
  for (j = 0; j < 1000-i; j++) {
    // 0 <= i <= 999
    // 0 <= j <= 999
    a[i+j] = 0;
  }
}
```

Warning

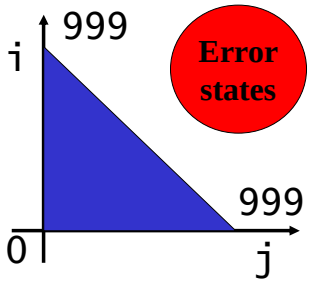


Non conclusive

Example (2)

```
int a[1000];
for (i = 0; i < 1000; i++) {
  for (j = 0; j < 1000-i; j++) {
    // 0 <= i and 0 <= j
    // i+j <= 999
    a[i+j] = 0;
  }
}
```

Safe



Conclusive

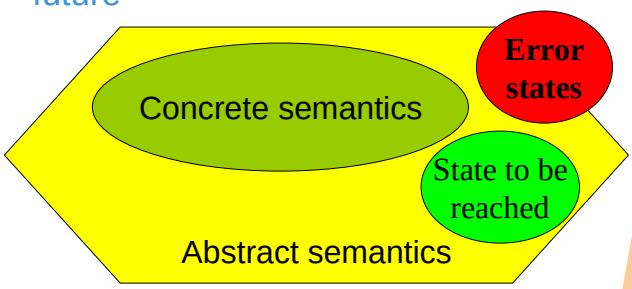
Safety et liveness properties

- **Safety**

"Bad" things never happen

- **Liveness**

Some thing "good" will eventually happen in the future



The proof tool of SCADE can not prove liveness properties

Interest of the liveness properties

- "Liveness" property / "timed" property
  - Example: if an error is detected, the software shall raise an alarm toward the user
    - Liveness: the alarm will mandatorily be raised (one day or another)

But when?

➔ Not *acceptable for a critical real time piece of software*

❖ Timed property: the alarm will mandatorily be raised 1 second after the failure occurence

➔ **Safety property**

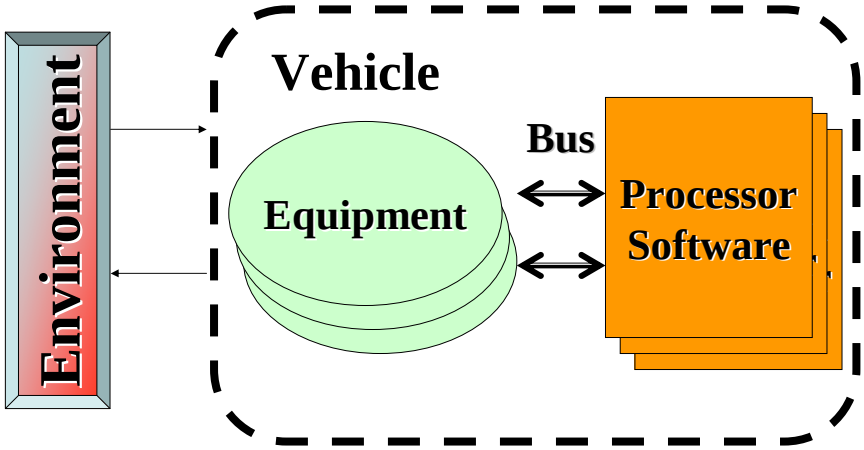
Formal proof

- "Mathematical" exhaustive demonstration that a piece of software/code satisfied a property

➔ Rarely the case!

A piece software generally satisfies a property only in a correct environment

The software is part of a complex system



Formal proof principles

- Software under validation
- Properties to be satisfied
- Software environment

($\Box$ correct environment) $\wedge$ software $\Rightarrow$ properties

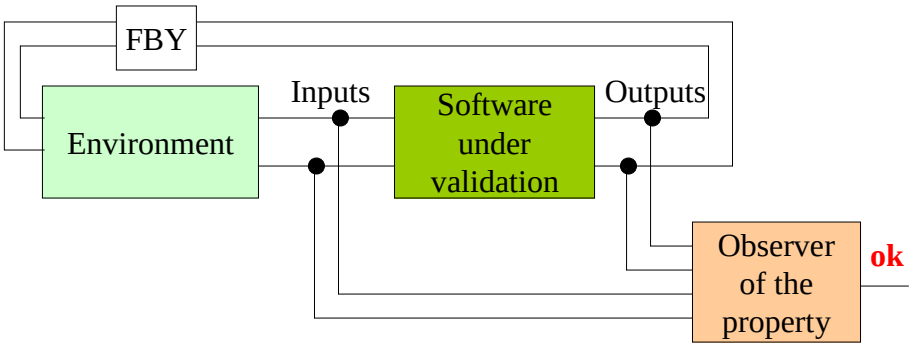
- Environment in open or close loop

Expression of properties

Notion of observer

- An observer is a software observing the software under validation and returning “true” as long as the property is satisfied
  - Observation of the software inputs
  - Observation of the software outputs
- Idem for the environment properties

Observers in SCADE



➔ Use for testing (oracle)
➔ Use by SCADE proof tool

Non deterministic environment (1/2)

The software environment is generally not fully deterministic

- Human action
- Failure
- …

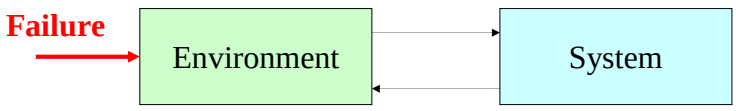
➔ Non deterministic environment

But SCADE is a deterministic language!

Non deterministic environment (2/2)

The non determinism is modelled by an additional input

Example: Failure occurrence



Assertion

An assertion allows to restrict an environment “too much” non deterministic

Example:

- Input “gf” models a gyroscope failure
- Input “tf” models a thruster failureune panne d’une tuyère
  - ➔ To develop a “one fault tolerant” system

Hypothesis: assert #(gf, tf)

The End