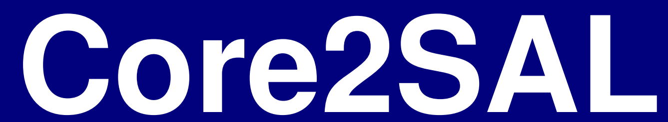
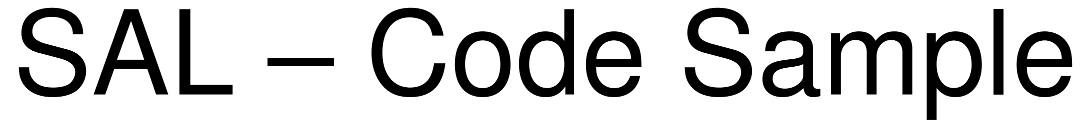


Design and Implementation review

Gilad Hemi

Ayelet Bar Niv



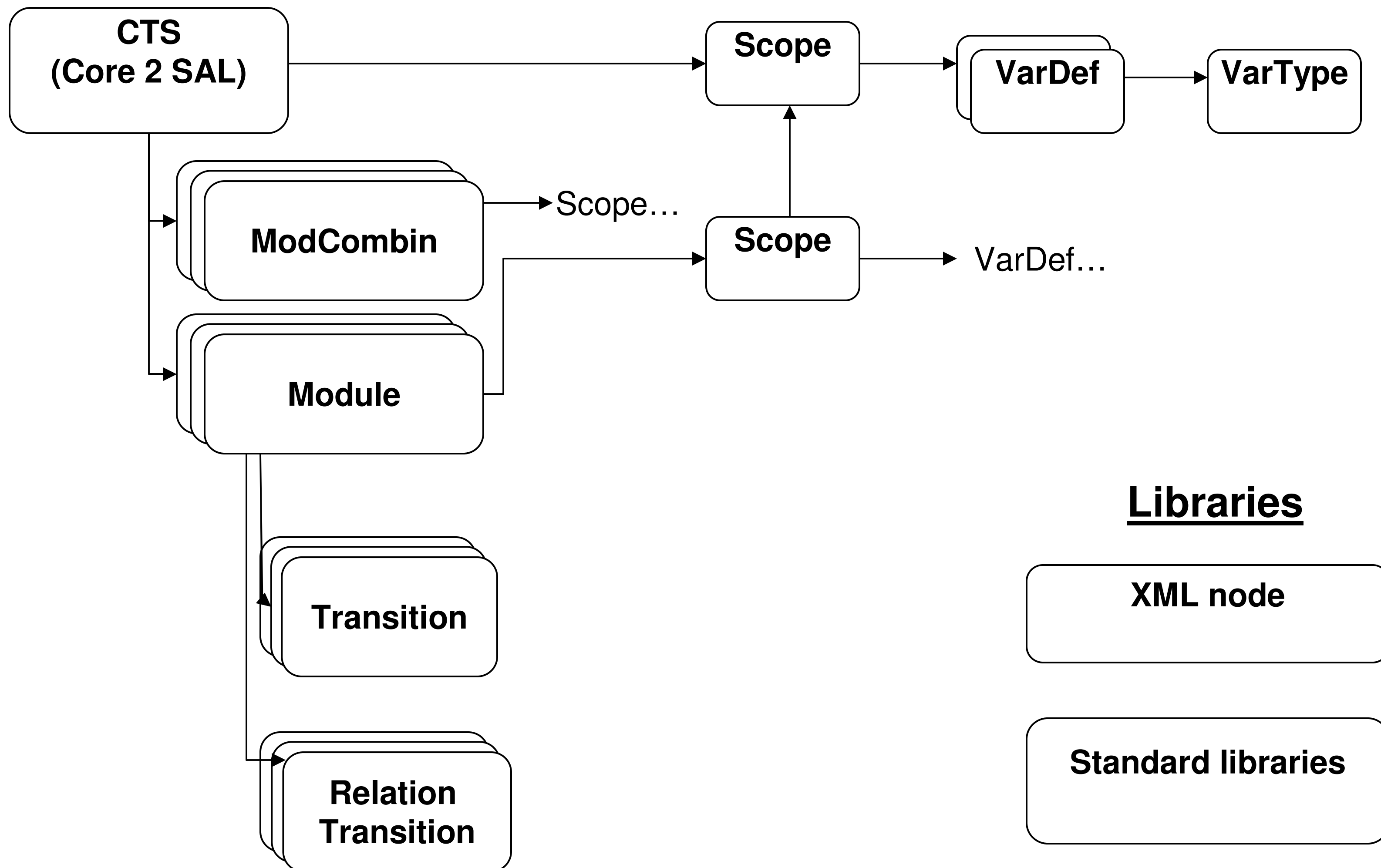
SAL – Code Sample

```
mutex : CONTEX =
BEGIN
PC: TYPE = {trying, critical,sleeping}
mutex [tval:boolean] : MODULE =
BEGIN
      INPUT          pc2: pc, x2: boolean
      OUTPUT         pc1: pc, x1: boolean

         INITIALIZATION
              pc1=sleeping;
              x1=tval
         TRANSITION
              pc1=sleeping  -->  pc1' = trying;
                     x1'=(x2=tval)
         [ ]
              pc1 =trying AND (pc2=sleeping OR x1=(x2/=tval))
              --> pc1'=critical
         [ ]
              pc1=critical --> pc1' = sleeping;
                             x1'=(x2=tval)
END

system: MODULE=
      LOCAL  x1, x2
      (mutex[FALSE] || (RENAME  pc2 TO pc1, x2 TO x1,
                            pc1 TO pc2,x1 TO x2
      mutex[TRUE]))
END
```

Non Trivial Issues

- SAL transitions have no relations.
- SAL does not have partial synchronization.
- Unassigned next-state variables status.
- Analyzing Module arguments.

Design Overview (instances)



Simulating Relations – Simple cases

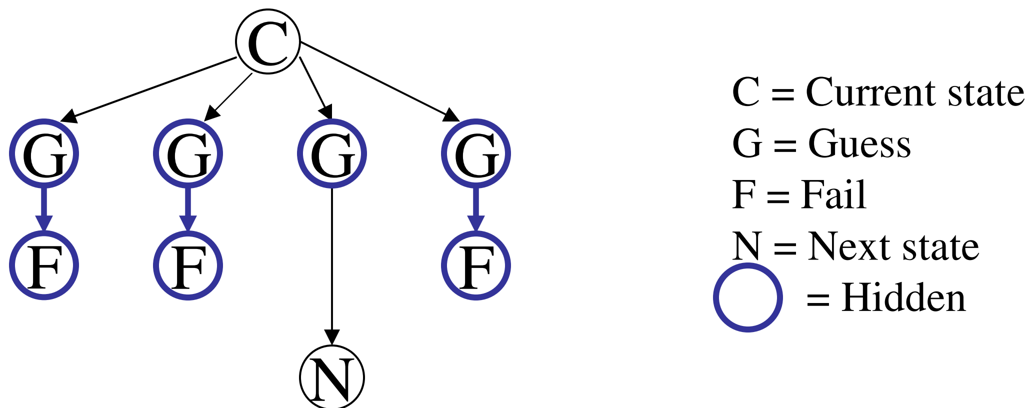
In simple cases we can convert relations to assignments.

1. Split the relation to terms.
2. If a term is an assignment (or a simple Boolean expression) to x’ and x’ has no assignment add it to the assignment section.
3. If the term does not have next-state variables add it to the enable section.

enable: x>0; relation: y’=8 /\ !z’ /\ x<5	X > 0 AND x < 5 --> y' = 8; z' = False

Simulating Relations

1. Assign non deterministic values to each next-state variable in the relation that does not have an assignment.
2. Check the assignment and raise the fail flag if necessary.
3. Use a PC to force the second transition.



Example

```
HOLD_PREVIOUS
MODULE SYSTEM()
{
VAR
        i : integer INITVAL 1;
        j : integer INITVAL 1;
        b : boolean INITVAL TRUE;
        b2 : boolean INITVAL FALSE;
TRANS RELAXABLE:
enable: !b;
relation:
                i=0 /\
                b‘ /\
                i'=j /\
                b2' = FALSE;

}
```

```
relation: CONTEXT =
BEGIN
CTS_PC_TYPE : TYPE = [0..1];
INTEGER : TYPE = [-5..5];
SYSTEM: MODULE =
BEGIN
LOCAL
  CTS_Force_PC , CTS_Fail: BOOLEAN, CTS_PC : CTS_PC_TYPE,
  b,b2 : BOOLEAN,  i: INTEGER,
  temp_j: INTEGER

INITIALIZATION
    CTS_Fail = FALSE; CTS_Force_PC = FALSE; CTS_PC = 0;
    b = TRUE;b2=FALSE; i=1;j=1
TRANSITION
 [
   RELAXABLE : !b AND i = 0 AND !CTS_FORCE -->
              b' = TRUE , b2' = FALSE ;
              temp_j' = j ;
              i' = {-5 , -4, -3 , -2 , -1 , 0 ,1 ,2 , 3 , 4 ,5} ;
              CTS_PC' = 0 ; CTS_Force_PC' = TRUE
 []
     RELAXABLE_relation : CTS_PC AND CTS_Force_PC = 0 -->
     CTS_Fail = IF i = temp_j THEN CTS_Fail ELSE TRUE ENDIF
 ]
END;
END
```

- The SAL does not have a HOLD_PREVIOUS flag.
- It behaves as HOLD_PREVIOUS = true.
- If HOLD_PREVIOUS = false we have to add a non deterministic assignment for each variable X that does not have an assignment.

  e.g. *B' = {FALSE,TRUE}* for a boolean variable B.
- Integer range is restricted so we can add a non deterministic assignment

  I' = { min_int, … ,max_int }

HOLD_PREVIOUS : Arrays

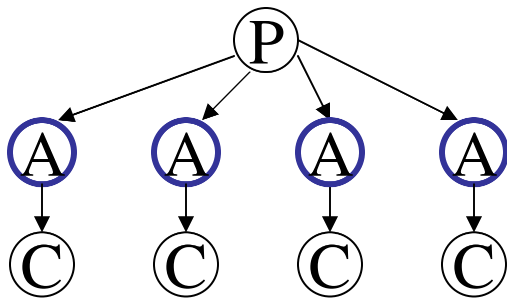
- Sometimes it’s hard to determine the to which indices we need to add a non deterministic assignment.

For instance :

```
TRANSITION a:
 enable :
        bVal
    assign :
        arr'[i] = arr[i]+1;
        arr’[i-7] = arr[i];
```

HOLD_PREVIOUS : Arrays (cont')

- In each module we insert a new transition with assignments to all the indices of every array in the scope



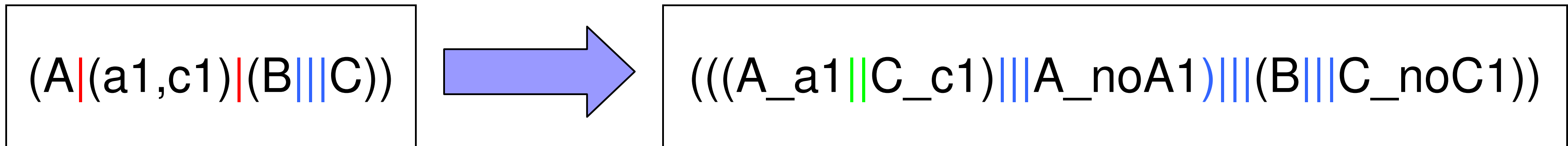
P = Previous state
A = Array assignment
C = Current state
= Hidden

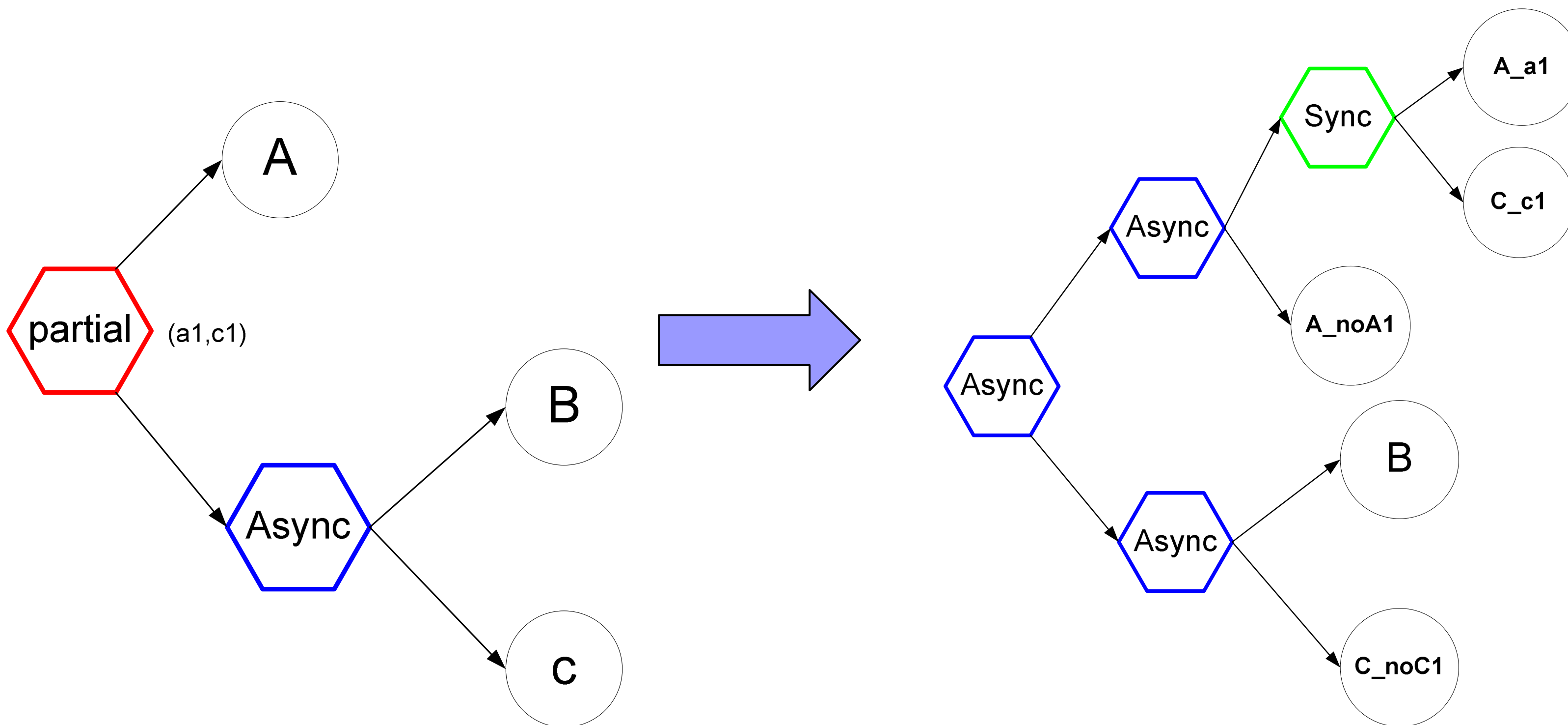
Example

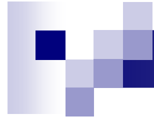
```
MODULE SYSTEM()
{
VAR
        a : ARRAY[2] OF boolean;
        b : boolean INITVAL TRUE;

TRANS No_Hold:
  enable: !b;
  assign: b':= TRUE;

TRANS No_Hold2:
  enable: !b;
  assign: a[1]' := TRUE;

}
```

```
holdprev: CONTEXT =
BEGIN
CTS_ARRAY_PC_TYPE : TYPE = [0..1];

SYSTEM: MODULE =
BEGIN
LOCAL
  CTS_Force_ARRAY : BOOLEAN,
  CTS_ARRAY_PC : CTS_ARRAY_PC_TYPE,
  a : ARRAY 2 OF BOOLEAN,
  b : BOOLEAN

INITIALIZATION
   CTS_Force_ARRAY = FALSE;
   b = TRUE;
TRANSITION
 [
   No_Hold : !b AND CTS_Force_ARRAY AND CTS_ARRAY_PC = 0 -->
                 b' = TRUE
 []
  No_Hold2: !b AND CTS_Force_ARRAY AND CTS_ARRAY_PC = 0 -->
                 a'[1] = TRUE
 []
      arrayAssign : !CTS_Force_ARRAY -->
                 CTS_Force_ARRAY' = TRUE ; CTS_ARRAY_PC' = 0 ;
                 a'[0] = { TRUE, FALSE };
                 a'[1] = { TRUE, FALSE };
]
END;
END
```

Partial Synchronization – Original Design







Implementation issues

- Renaming
- Module Variables
- Parameters
- Complex Combinations

Partial Synchronization - Renaming

In order to solve renaming issues, we resolve the renaming prior to handling partial synchronization, simply by cloning the modules and changing the name of the transitions in the new module to a unique name.

Partial Synchronization - Module Variables

While handling partial synchronization, we split some of the modules.

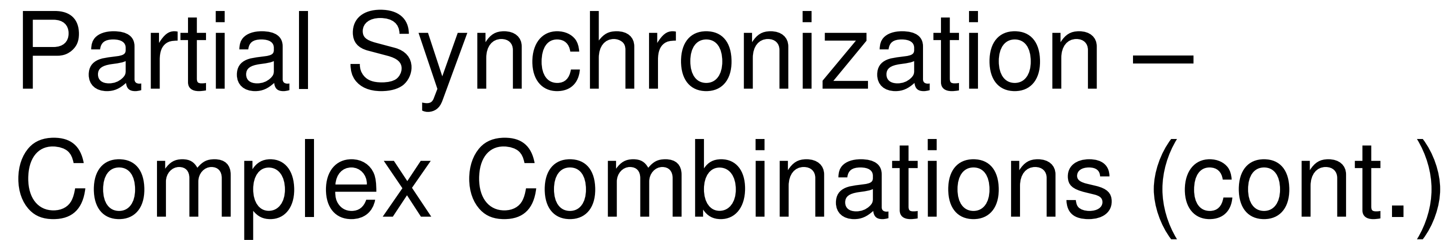
All local variables of the split module becomes local in the parent scope, so they can be shared by all the child modules that were created.

Partial Synchronization – Parameters

- The actual parameters for the child modules are the same as the actual parameters for the original module.

Partial Synchronization – Complex Combinations

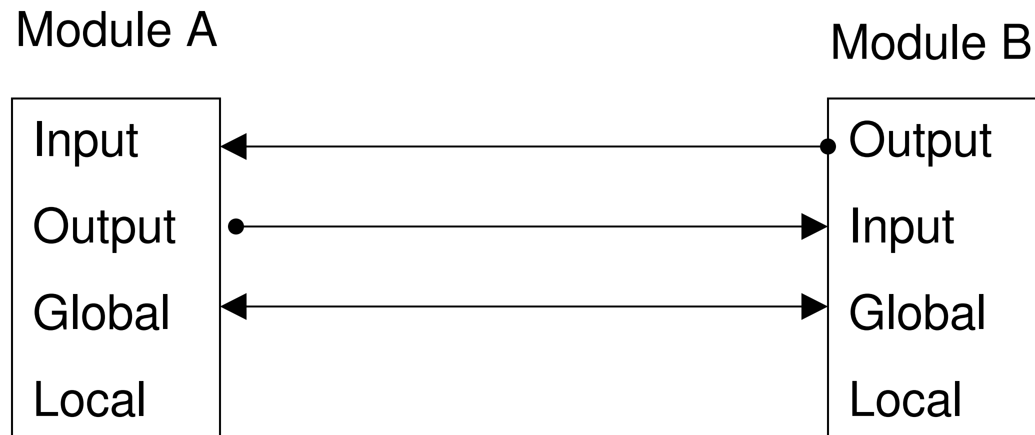
- In cases of complex combination our solution doesn't work.
- Complex combinations can be partial synchronization inside another partial synchronization, or synchronized combination inside partial synchronized combination.



Partial Synchronization – Complex Combinations (cont.)

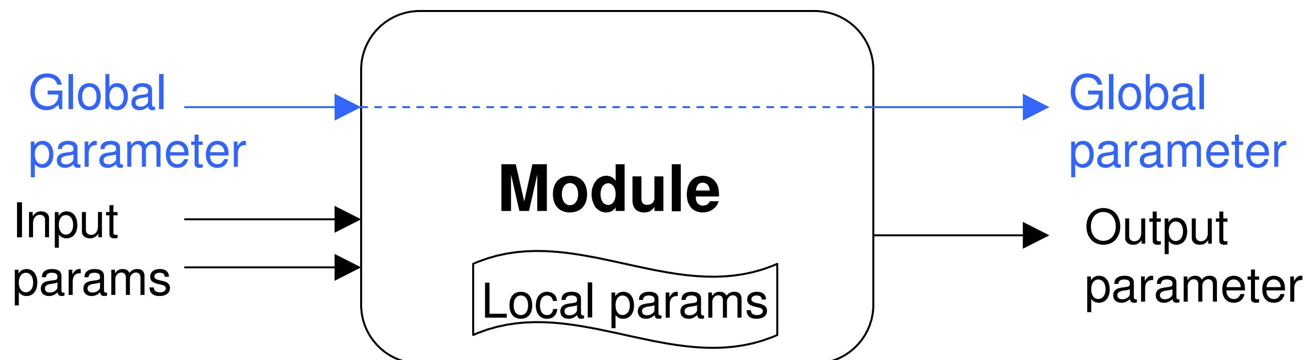
- in cases of complex combinations we will use flattening before handling the partial synchronization.

Variables is SAL



Variables in SAL

- SAL variables are similar to input/output ports. A global variable symbolize a connection between an input port and an output port.



Variables in SAL - Algorithm

- L = { all variables declared locally}
- O_1 = { all variables with assignments } \ L
- I_1 = { all variables in expressions } \ L
- $G_1 = O_1 \cap I_1$
- $O = O_1 \setminus G_1$
- $I = I_1 \setminus G_1$