

Proposal of extension of SpecC Language toward ease of coding PSM

April. 5, 2001

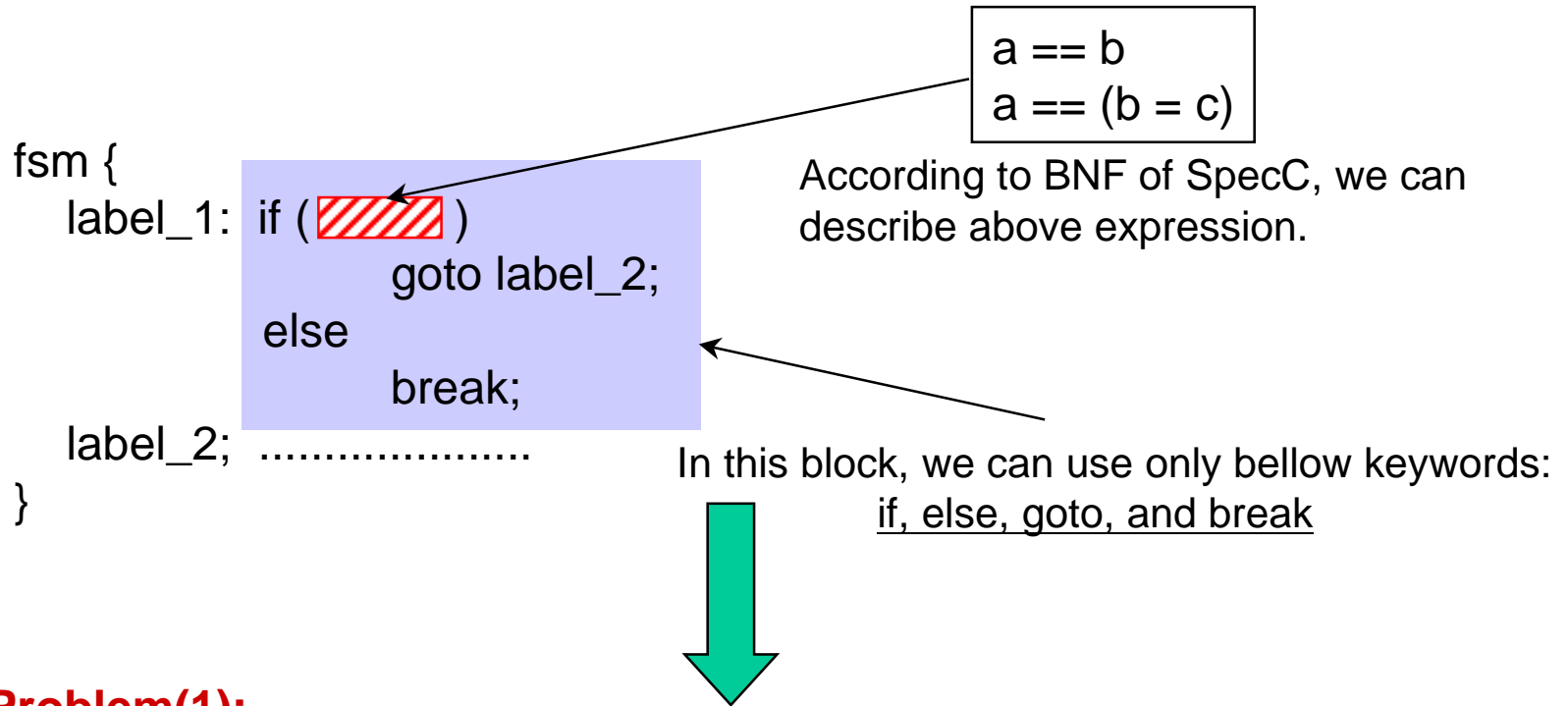
Tadaaki Tanimoto
Youichi Kobayashi

IP Technology Center
System LSI Business Division
Semiconductor & Integrated Circuits
Hitachi Ltd..

Agenda

1. Current FSM syntax
2. How to describe Mearly type FSM?
3. How to realize the hierarchy?
4. How to support for transition skipping hierarchy?
5. What happen if parallel FSM?

1. Current FSM syntax Problem(1)



Problem(1):

It's easy to describe Moore type FSM, but difficult to describe Mealy type FSM.

1. Current FSM syntax (cont'd) Problem(2)

Ideally, We want to describe FSM with gated clock by using **notify-wait** shown in bellow.

```
behavior A( in event GCLK ) {  
    void main void {  
        wait (GCLK);  
    }  
}
```

```
behavior FSM_with_GCLK )  
    in event CLK  
    in bool clock_enable) {  
        A IDLE(GCLK)  
        B ADDER(GCLK);  
        void main(void) {  
            if(clock_enable)  
                notify(GCLK);  
            fsm {  
                IDLE: {goto ADDER;}  
                ADDER: {goto IDLE;}  
            }  
        }  
    }
```

Problem(2):

FSM with event description does not work at all.

1. Current FSM syntax (cont'd) Problem(2)

Here, we show the example of defect of event description by **notify-wait**.

(1) Event description by notify-wait

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include "state.sh"
#define N 2
behavior A(in event CLK) {

    void main(void) {
        printf("State A\n");
        wait( CLK );
        printf("State A\n");
    }
};
behavior B(in event CLK) {

    void main(void) {
        printf("State B\n");
        wait( CLK );
        printf("State B\n");
    }
};
```

```
behavior Main {

    event CLK ;

    A IDLE( CLK ) ;
    B ADDR( CLK ) ;

    int main(int argc, char **argv) {
        int i ;

        for(i=0;i<=N;i++) {
            notify( CLK );
            fsm { IDLE: { goto ADDR; }
                ADDR: { goto IDLE; }
            }
            printf( "AAA\n" );
        }
    }
};
```

Result::

```
SpecC warning: dead lock in the program, aborted
State A
SpecC warning: dead lock in the program, aborted
State A
```

1. Current FSM syntax (cont'd) Problem(2)

Moreover, **notify-wait** does not work at all with **par**.

(2) Event description by notify-wait with par

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include "state.sh"
#define N 2
behavior A(in event CLK, out int O ) {

    void main(void) {
        printf("State A 1\n");
        wait( CLK );
        printf("State A 2\n");
        O = 1 ;
    }
};
behavior B(in event CLK, out int O) {

    void main(void) {
        printf("State B 1\n");
        wait( CLK );
        printf("State B 2\n");
        O = 0 ;
    }
};
```

```
behavior Main {


    event CLK ;
    int O ;

    A IDLE( CLK, O ) ;
    B ADDR( CLK, O ) ;

    int main(int argc, char **argv) {
        int i ;

        for(i=0;i<=N;i++) {
            notify( CLK );
            par { IDLE.main() ;
                ADDR.main() ;
            }
            printf( "O = %d\n", O );
        }
    }
};
```

Result::

SpecC warning: dead lock in the program, aborted
State A 1
State B 1
 ← "State A 2" and "State B 2"
can not be shown here.

Since we have this kind of problems, it's difficult to describe parallel execution of FSM.

1. Current FSM syntax (cont'd) Problem(2)

Bellow example shows the strange behavior of **par**.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include "state.sh"
#define N 2
behavior A(out int O) {

    void main(void) {
        if ( O == 0 ) {
            printf( "A O is 0%#n" );
        } else {
            printf( "A O is 1%#n" );
        }
        O = 0 ;
    }
};
behavior B(inout int O, event done) {

    void main(void) {
        if ( O == 0 ) {
            printf( "B O is 0%#n" );
        } else {
            printf( "B O is 1%#n" );
        }
        O = 1 ;
    }
};
```

```
behavior Main {

    int O ;
    event done ;

    A IDLE( O ) ;
    B ADDR( O, done ) ;

    int main(int argc, char **argv) {
        int i ;

        for(i=0;i<=N;i++) {
            O = 1 ;
            par { IDLE.main() ;
                ADDR.main() ;
            }
            printf( "O = %d%#n", O ) ;
        }
    }
};
```

Result::

```
A O is 1
B O is 0
O = 1
A O is 1
B O is 0
O = 1
A O is 1
B O is 0
O = 1
```

If concurrent execution is done,
both A and B will be "1"

Non-preemptive concurrent execution.

par_hw -> We focus on this semantics
par_sw -> preemptive, non-preemptive

There's argument for definition of par_sw.
But par_hw is highest priority for us.

We do need parallel execution !!
It's essential for describing HW !!

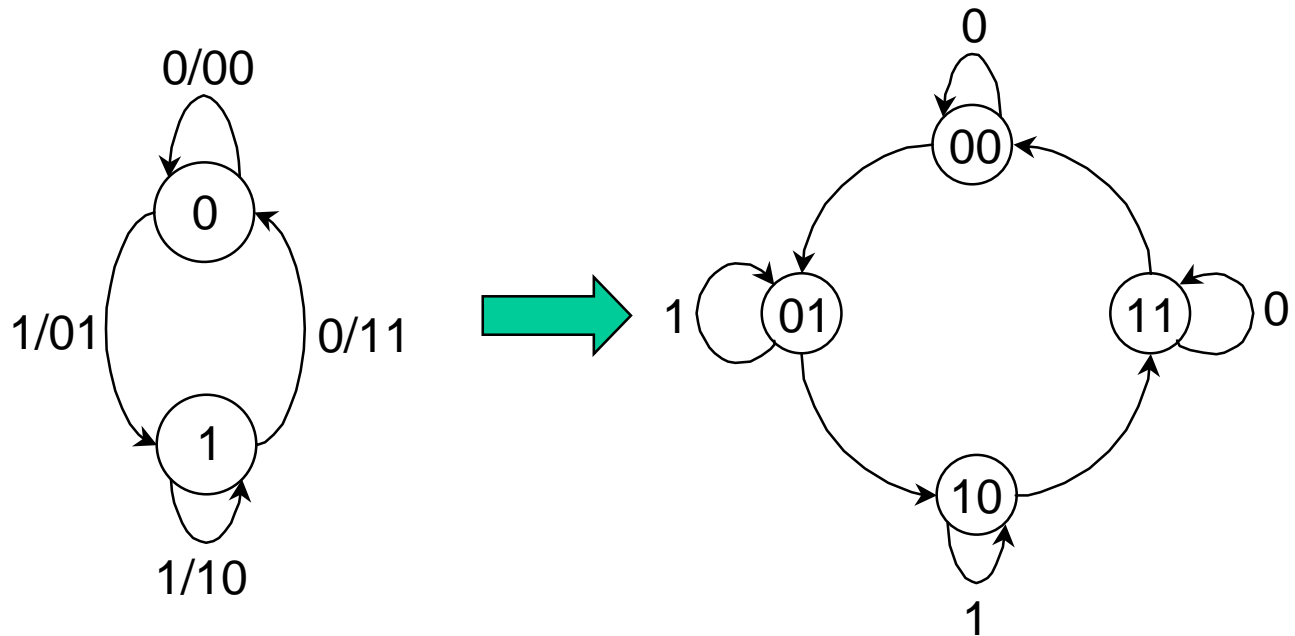
1. Current FSM syntax(cont'd)

Most important issues are “par in HW semantics” and “fixing the bugs around notify-wait”.

Since we want to focus on how to extend SpecC language for describing PSM. We assume above two big problem will be fixed sooner.

2. How to describe Mearly type FSM

Idea 1: Transform from Mearly to Moore.



Time consuming !!

2. How to describe Mearly type FSM (cont'd)

Idea 2: Spec C is extension of C. So we can write FSM in C like bellow.

```
while(1) {  
    switch (state)  
        case :  
            .....  
        case :  
            .....  
}
```

Advantage

This coding style is suitable to describe Mearly type FSM.

Disadvantage

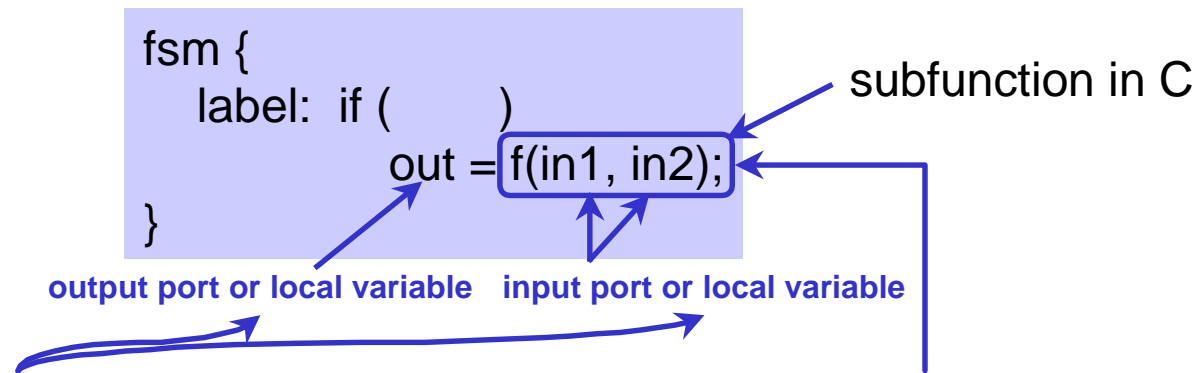
This coding style can not represent FSM explicitly.



It's a big problem !!

2. How to describe Meary type FSM (cont'd)

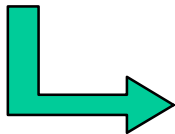
Idea 3: Support the syntax for Meary type FSM.



We need to check whether these are described in ports which described in port declaration of behavior including this FSM.

If this function include 'event' type argument, it is difficult to guarantee the synchronization between transition and output of FSM.

According to BNF of SpecC, we can use 'event' type argument in C subfunction. It's strange. We should modify BNF to prohibit to use 'event' type argument in C subfunction.

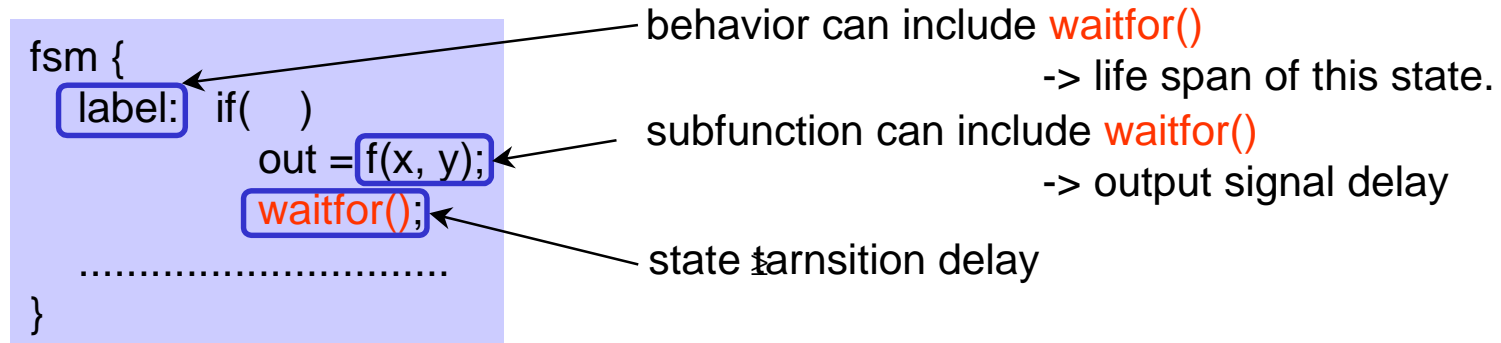


i.e, we should be

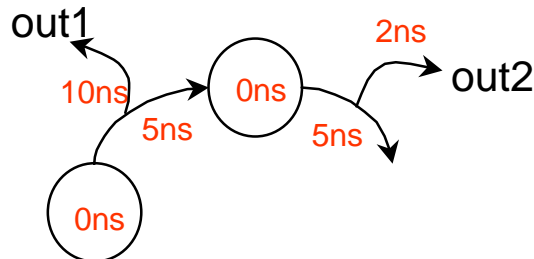
event \in behavior	event \notin C subfunction
channel	
Interface	

2. How to describe Mearly type FSM (cont'd)

Here, we show that **waitfor()** can cause a problem in Mearly type FSM.



Example:



In this example, out2 should be asserted earlier than out1. --> **Contradiction !!**

Constraints:

$$(\text{state transition delay}) + (\text{life span of state}) \geq (\text{output signal delay})$$

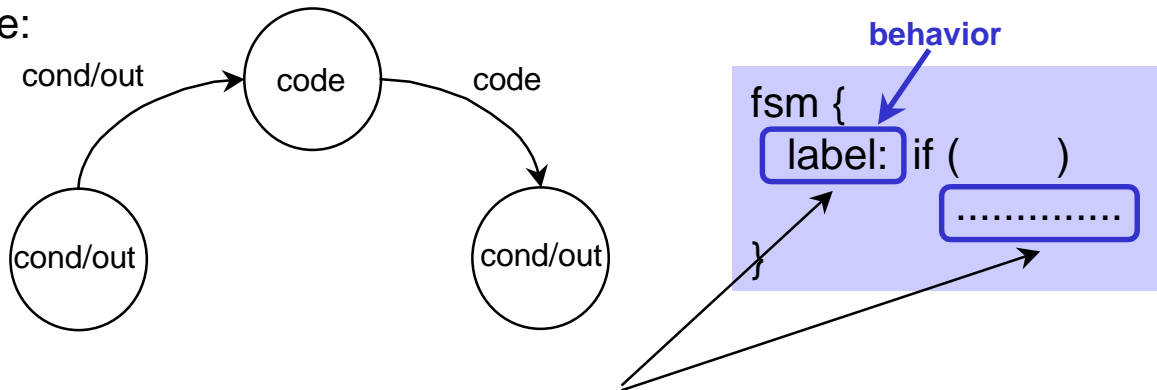


Compiler should check this constraints!!

2. How to describe Mealy type FSM (cont'd)

We can not construct below strange. Mealy type FSM and Moore type FSM are mixed.

Example:



Only one can have output function



Compiler should check this constraints !!

3. How to realize the hierarchy ?

Support to call main method including FSM.

```
fsm {
  label: if ( )
  .....
}
```

We can use behavior including main method including FSM.

PSM has two type of semantics:

TOC (Termination on Completion)

TOI (Transition on Interrupt)

FSM has two type:

Mealy type

Moore type

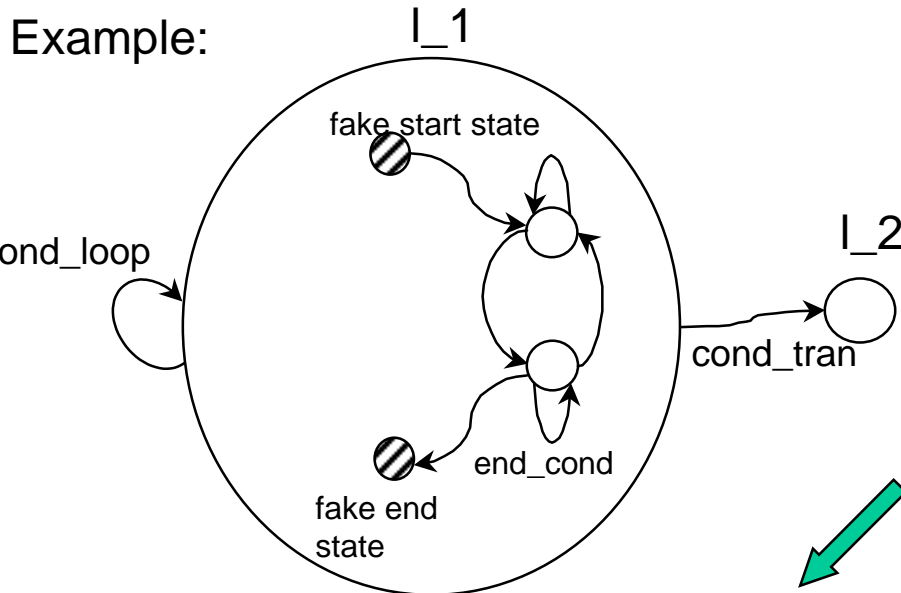
We should consider 16 classification.

↓ i.e.

Termination semantics		FSM type	
hie_level	hie_level + 1	hie_level	hie_level + 1
2	2	2	2

→ 2 x 2 x 2 x 2 = 16 (classification)

3. How to realize the hierarchy ? (cont'd)



TOC

```
fsm {
  l_1: if(end_cond & cond_tran)
        goto l_2;
        .....
}
```

TOI

Behavior including main method including FSM

```
fsm {
  l_1: if(cond_tran)
        goto l_2;
        .....
}
```

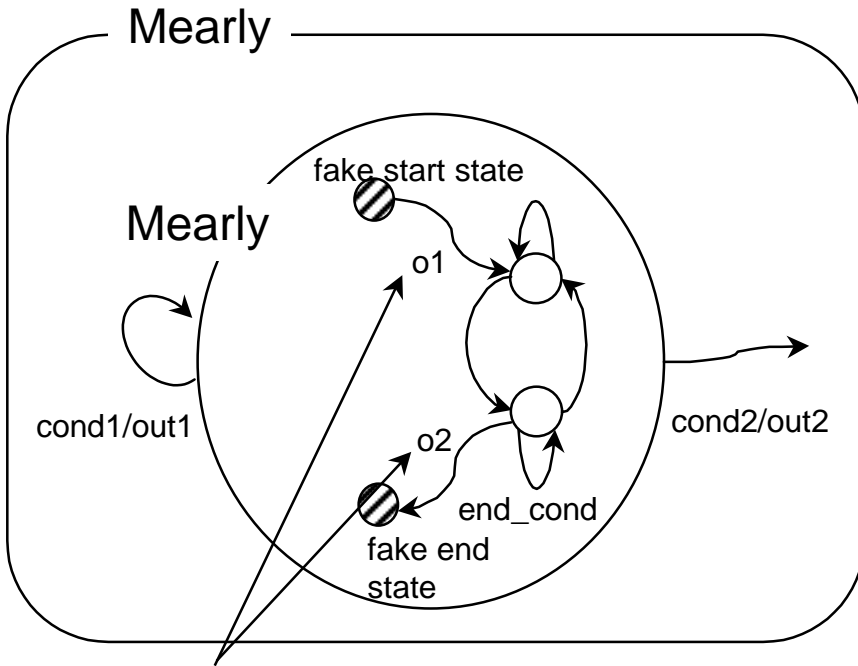
When branch condition is in boolean expression.
 $end_cond \ \& \ cond_tran = 1$
 $\Rightarrow end_cond = 1$

thus, we can distinguish between TOC and TOI.
 In general, it is tough task.
 Thus we need to introduce key word for distinguish
 between TOI and TOC.

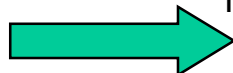
Even if we have such keywords,
 $end_cond \ \& \ cond_tran = 1$
 $\Rightarrow end_cond = 1$
 explain the fact that we can develop lint tool which
 check TOC by solving SAT problem.

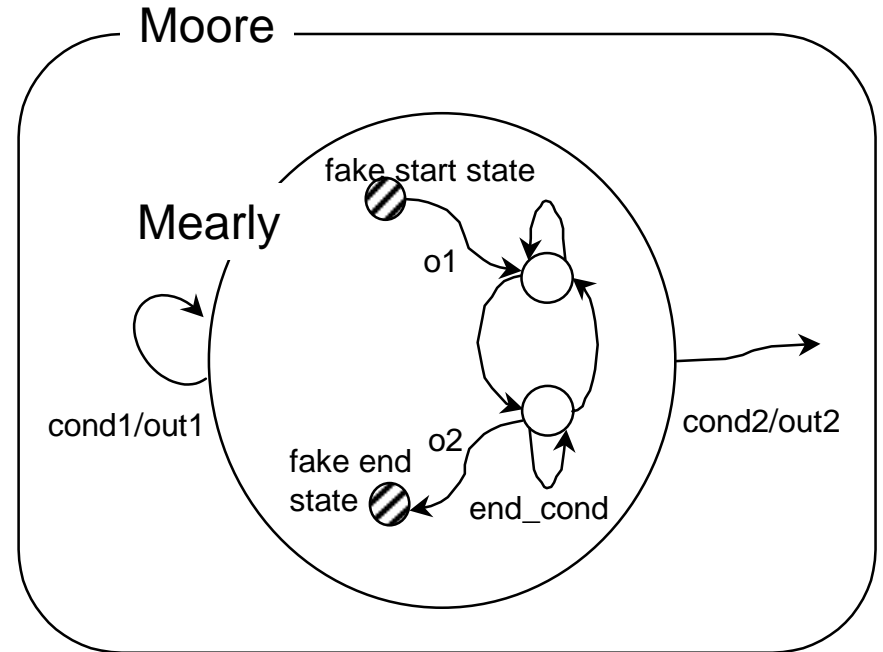
3. How to realize the hierarchy ? (cont'd)

But, Mearly type FSM with TOC semantics have some problems.



We have no output function.
How to determine it ?

 inheritance from parent hierarchy:
i.e, o1 = out1
o2 = oout2



Determine output function by holding previous value

We think it is reasonable. How do you think about it?


3. How to realize the hierarchy ? (cont'd)

For PSM we have below check items:

(1) Output function does not include event type argument.

(2) waitfor delay constraint check.

(3) fsm {

label: if ()


only one can have output function

(4) Transition condition check for checking TOC.

(5) Completeness check for PSM:

1) If TOC, sub FSM should have at least one fake start and end state.

2) If TOI , sub FSM should have fake start state.

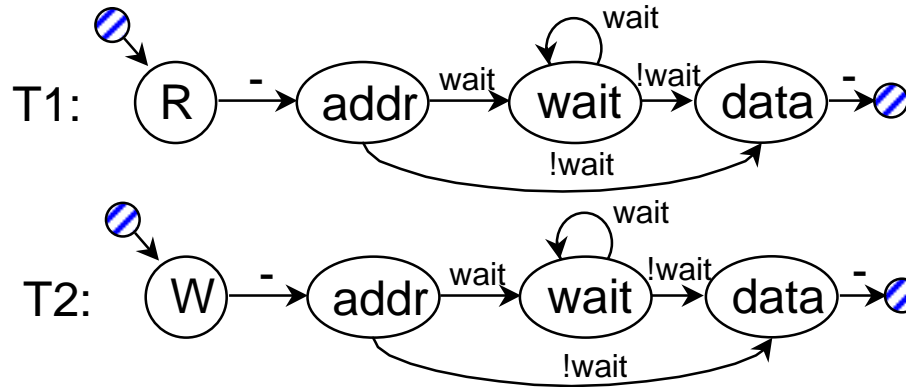
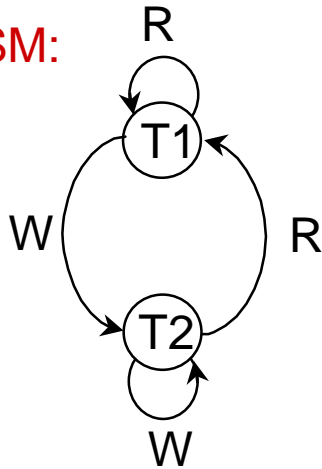
3) Transition conditions are exclusive (warning).

4) Same transition condition from same start state (error).

5) No dead lock state (state without output transition arc nor break).

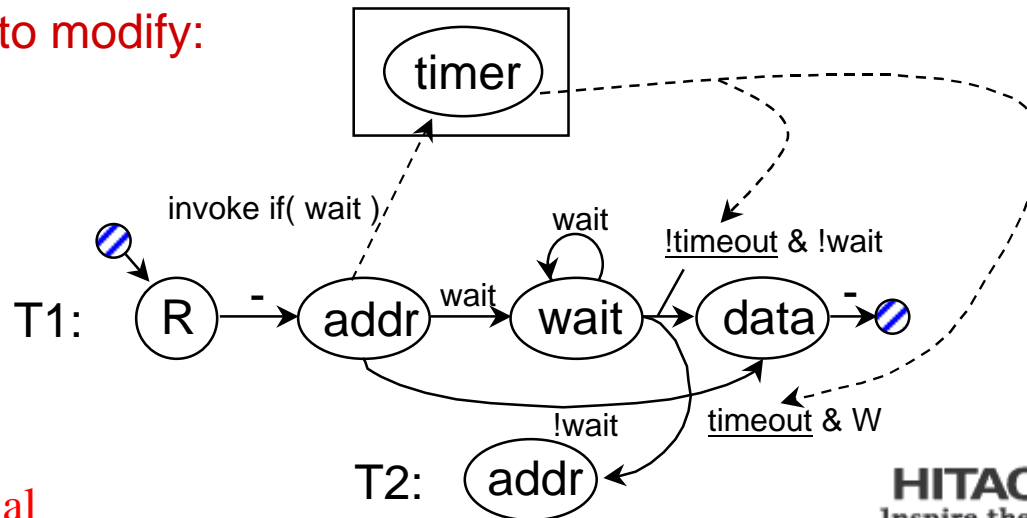
4. How to support for transition skipping hierarchy?

Initial FSM:



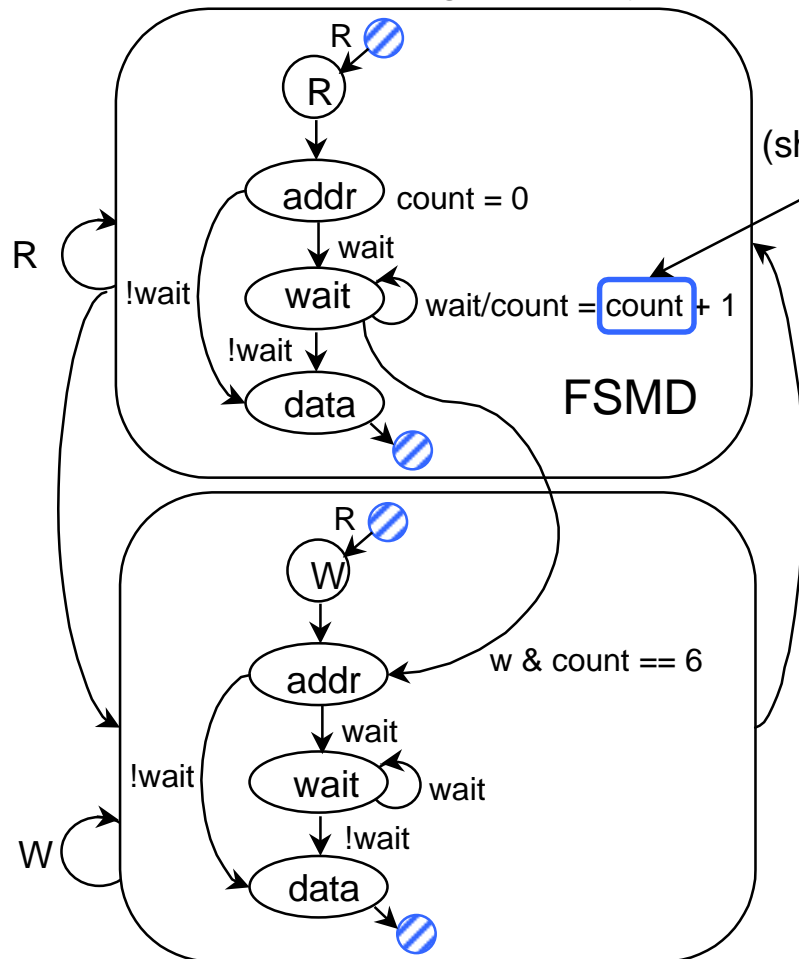
**Forget timeout control !!
We need to add it.**

Need to modify:



4. How to support for transition skipping hierarchy? (cont'd)

It is time consuming to modify this to PSM without transition skipping hierarchy.



local variable
(should not be port)

Supporting
"goto T2.addr"
enable us to refine this easily.
i.e instance.instance.state_name
↑
hierarchy defined by PSM

Compiler should check:
1) Completeness of modified PSM
2) Port matching and automatic refinement
Then simulate to check the behavior of PSM.

Ease the refinement task for designer.

5. What happen if parallel FSM?

End of fsm_2.main() Execution of fsm_1.main()

```
fsm {
  label: if (o2 == 0 & o1 == 1)
         goto label_2;
        else if (o1 == 0)
         goto label_3;
}
```

Explanation of the example:

- (1) If fsm_2 finishes before fsm_1, then the current state transits to label2.
- (2) If fsm_1 finishes before fsm_2, then the current state transits to label3. (But, in this case, we can allow fsm_1 and fsm_2 to finish simultaneously.)

Nearly equal wait event:

End of fsm_1.main()

```
fork
  wait();
  .....
  wait();
join
```

```
behavior( ) {
  f1 fsm_1 ( );
  f2 fsm_2 ( );
  par {
    o1 = fsm_1.main();
    o2 = fsm_2.main();
  }
}
```

In the light of TOI and TOC semantics, **transaction hardly occurs when transaction condition is satisfied.**

----> if transition conditions are not satisfied, substate is alive.

If we can connect two FSM with port, **two FSM's can control their transition each other.**