Combining ACL2 and Mathematica for the symbolic simulation of digital systems

AL SAMMANE Ghiath, BORRIONE Dominique,
OSTIER Pierre, SCHMALTZ Julien, TOMA Diana
TIMA Laboratory -VDS Group, Grenoble, France

ACL2 Workshop 2003 Boulder,CO

Symbolic simulation

- Symbolic simulation - semi-formal technique based on numeric simulation :
  - some inputs are valued, others stay symbolic
  - results: arithmetic and logical expressions
- Problems:
  - Symbolic expressions become very large with the number of simulation cycles
  - The simulation tree grows exponentially (with conditional statements when the condition is a symbolic term)
  - The results are unreadable

Constrained Symbolic Simulation

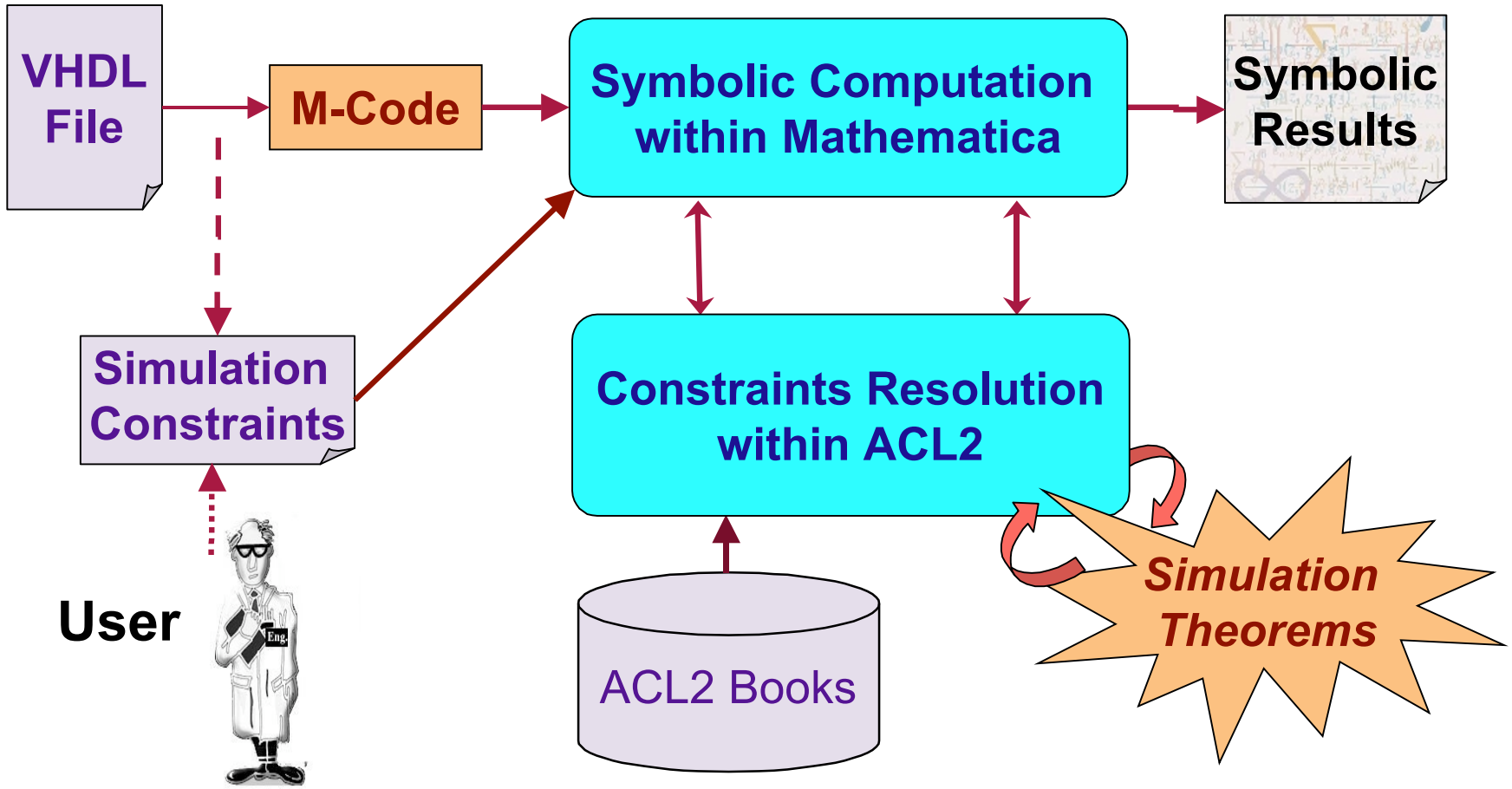
Two aspects of the simulation:

algebraic computation -> Mathematica

branching decision -> ACL2

- Constraints:
  - data type restrictions,
  - inequalities and equalities between expressions composed of design variables or input signals and arithmetic operators

Constrained Symbolic Simulation



Branch Decision Scheme

Mathematica | **ACL2**

Call of ACL2 to check consistency of constraints

Consistency L_h?

Check_consistency (L_h)

I_h or t

If I_h is returned, show it to the user, else L_h implies branch condition **B**?

L_h B

Prove (L_h implies **B**)

answer

If answer is Q.E.D simulate ***true*** branch else L_h implies **not B**?

L_h (not B)

Prove (L_h implies (**not B**))

answer

If answer is Q.E.D. simulate ***false*** branch else ask the user to add constraints or fork

Checking constraints consistency

- *check_consistency* function
  - Uses :
    - *tool1-fn* from /books/misc/expander
    - *consistency* function

```
(defun check-consistency (Lh state)
   (if (true-listp Lh)
       (cond ((endp Lh) (value  nil))
             (t (mv-let (erp val state)
                    (tool1-fn Lh state nil t nil t t)
                    (if erp
                       (value nil)
                       (if (nth 1 val)
                           (value t)
                           (consistency Lh nil 1 state))))))
       (value nil)))
```

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

I_h

nil

L_h

$$(\underbrace{C_1}_{t} \ldots C_{k-1}\ C_k\ C_{k+1} \ldots C_n)$$

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

I_h

nil

L_h

$$(\underbrace{C_1 \ldots C_{k-1}}_{t}\ C_k\ C_{k+1} \ldots C_n)$$

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

I_h

nil

L_h

$$(\underbrace{C_1 \ldots C_{k-1}\ C_k}_{f}\ C_{k+1} \ldots C_n)$$

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

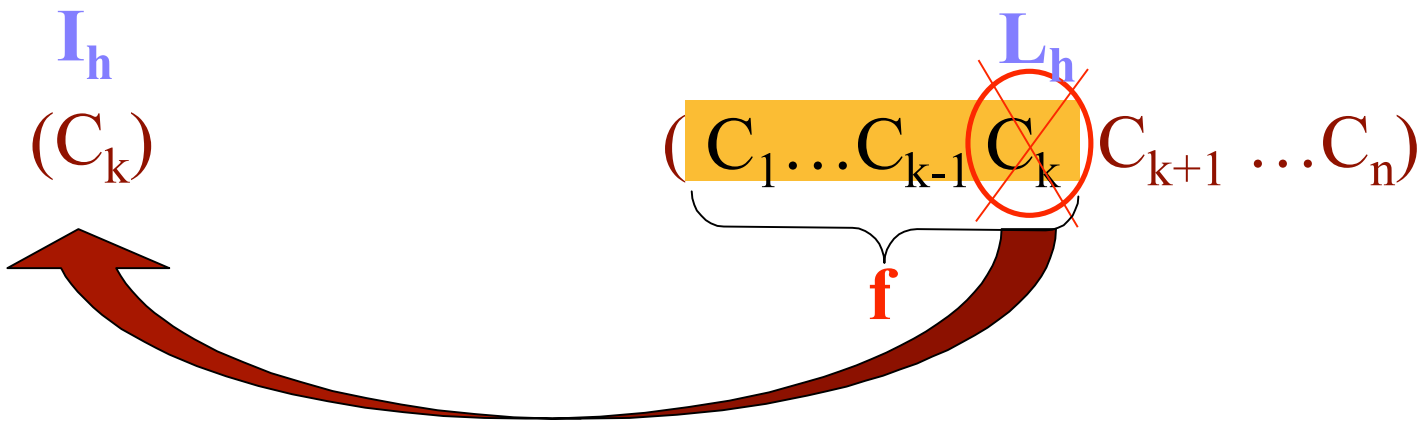
I_h

nil

$$\underbrace{(C_1 \ldots C_{k-1} \overset{L_h}{C_k}}_{f} C_{k+1} \ldots C_n)$$

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)



Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

I_h

(C_k)

f -> stop

L_h

$(C_1 \ldots C_{k-1}\ C_{k+1} \ldots C_n)$

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

I_h

$(\underbrace{C_k}_{t})$

L_h

$(C_1 \ldots C_{k-1}\ C_{k+1} \ldots C_n)$

Checking constraints consistency

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints***
      (*minimal = any strict subset is satisfiable*)

$$\underbrace{\overset{I_h}{(C_k)} \qquad\qquad\qquad\qquad \overset{L_h}{(C_1 \ldots C_{k-1}\ C_{k+1} \ldots C_n)}}_{t \quad \ldots}$$

Checking constraints consistency Example

- *consistency* function
  - Returns, for an inconsistent set of constraints
    - ***nil*** in case of errors
    - otherwise a ***minimal set of contradictory constraints*** (*minimal = any strict subset is satisfiable*)

L_h= ((< a b) (integerp a) (< c a) (< c d) (< b c))

I_h= ((< b c) (< c a) (< a b))

ACL2 - Mathematica communication

- Communication via a pipe
- Initialized by Mathematica with *callAcl2* function
- Mathematica gets the last line of the ACL2 response
- One ACL2 session during the whole simulation

```
callAcl2["(defthm foo (equal x x) : rule-classes nil)"]
FOO

callAcl2["(defthm foo (not (equal x x) ): rule-classes nil)"]
******** FAILED ******** See :DOC failure ******** FAILED
   ****
```

ACL2 - Mathematica communication

- Theorems:

```
callAcl2[“(mv-let (erp val state)
                  (defthm foo (implies L B))
                  (declare (ignore val))
                  (if erp
                     (value nil)
                     (value T)))”]
```

- Functions:

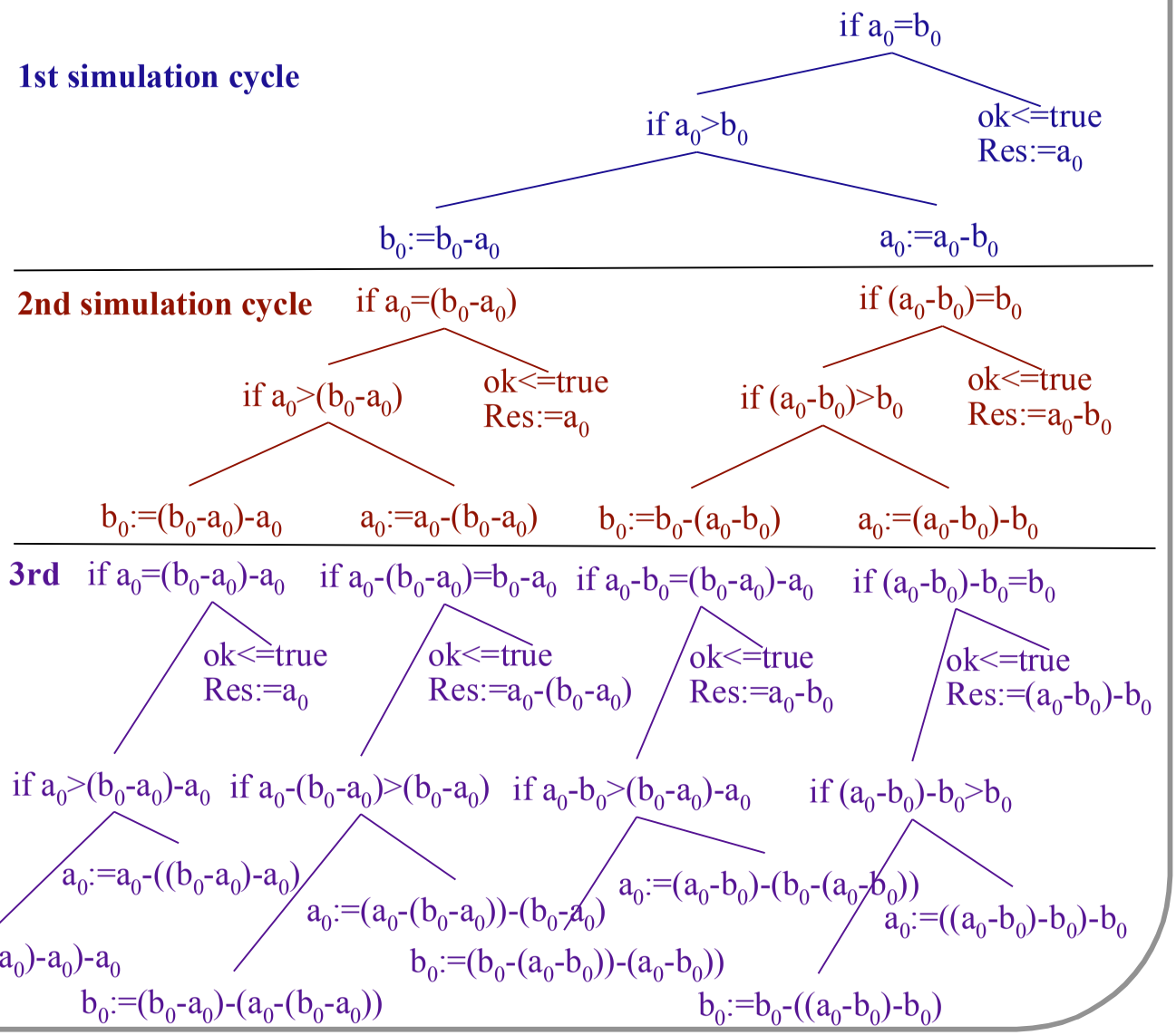
```
callAcl2[“(mv-let (erp val state)
                  (check_consistency L)
                  (if erp
                     (value nil)
                     (value val)))”]
```

Euclid's GCD Algorithm

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

Euclid's GCD Algorithm

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

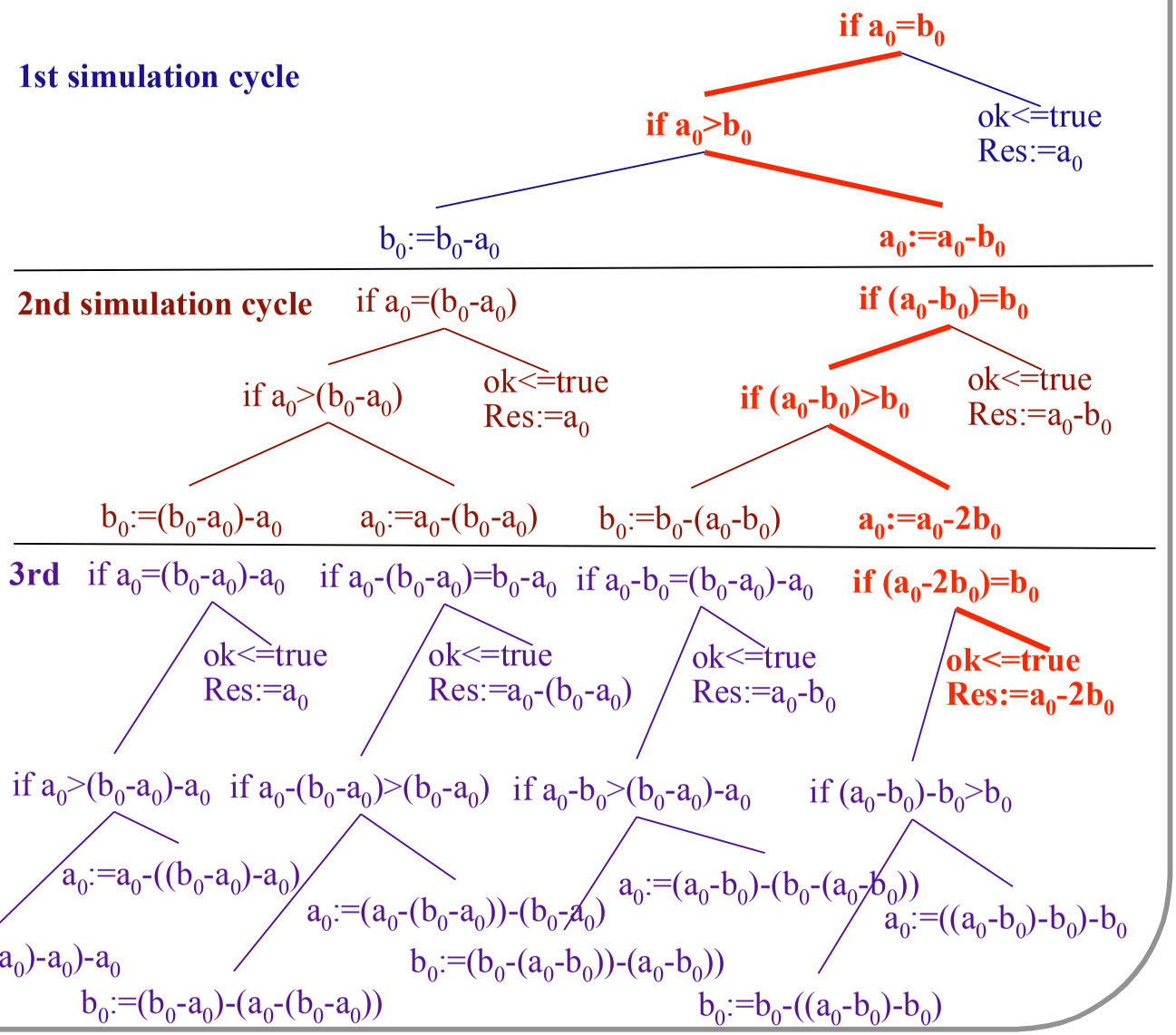


Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```



Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle

if $a_0=b_0$

```
callAcl2["(mv-let (erp val state)
          (check_consistency
            ((integerp n) (< 0 n)))
          (if erp (value nil)
              (value val)))"]
```

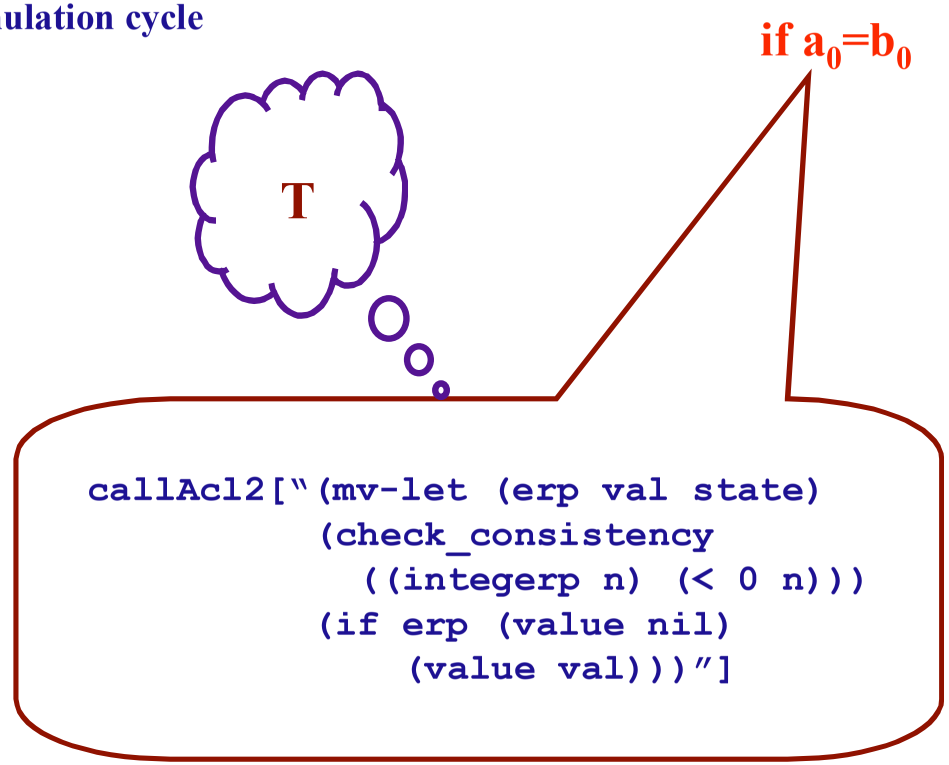
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle



Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle

if $a_0=b_0$

```
callAcl2["(mv-let (erp val state)
          (defthm thm1
             (implies (and (integerp n)(< 0 n))
                      (equal (* 3 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

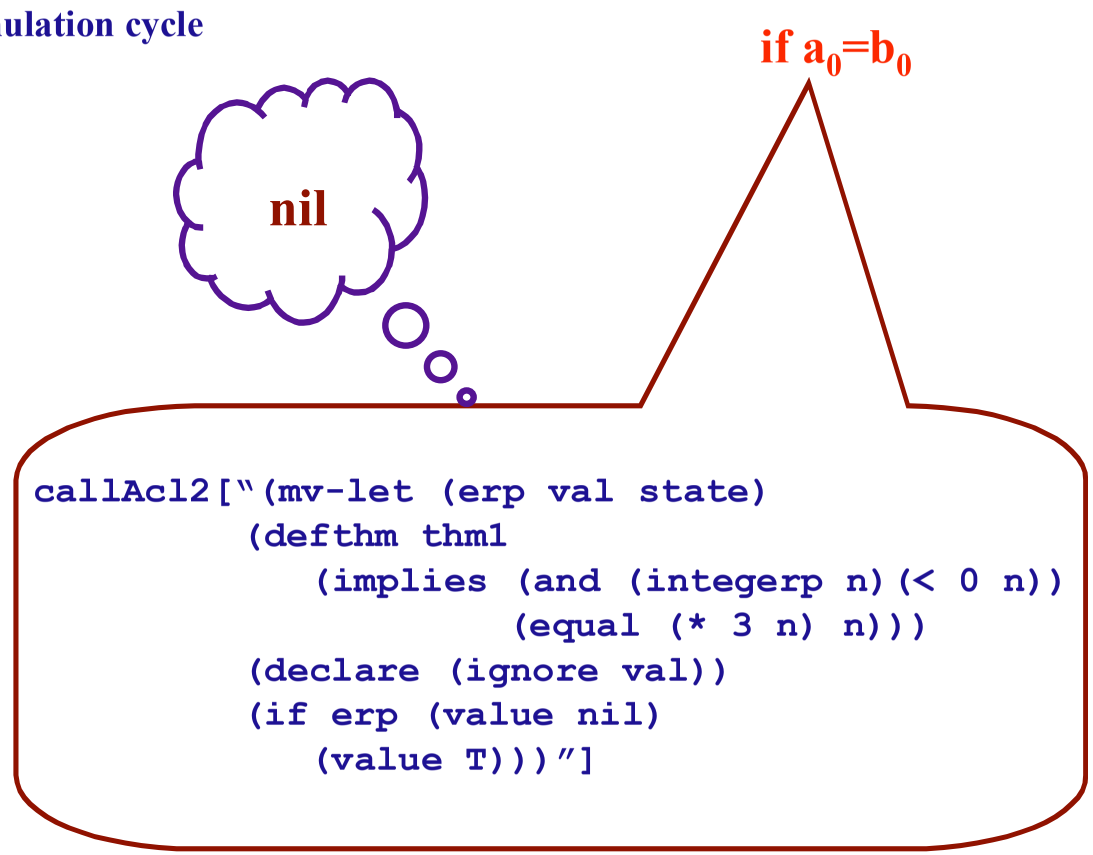
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle



Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle

if $a_0=b_0$

```
callAcl2["(mv-let (erp val state)
          (defthm thm1-neg
             (implies (and (integerp n)(< 0 n))
                      (not (equal (* 3 n) n))))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

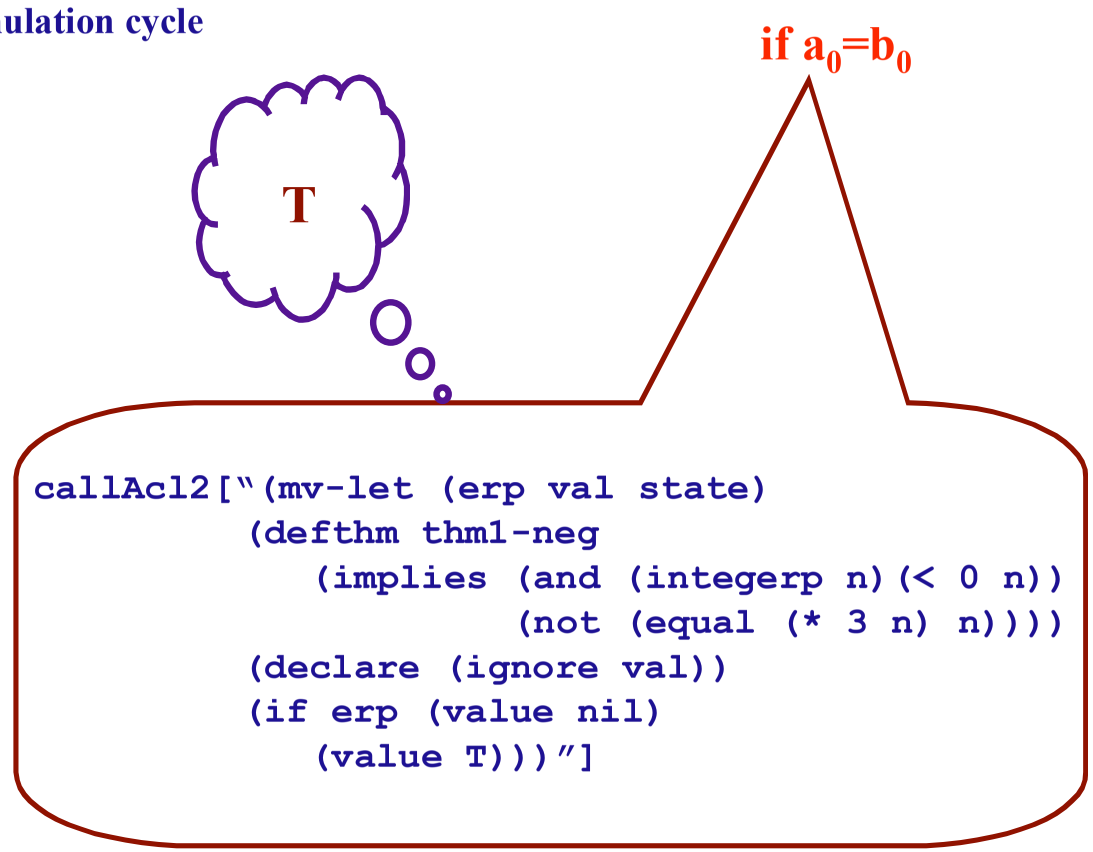
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle



Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle

if $a_0=b_0$

if $a_0>b_0$

```
callAcl2["(mv-let (erp val state)
          (defthm thm2
             (implies (and (integerp n)(< 0 n))
                      (> (* 3 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

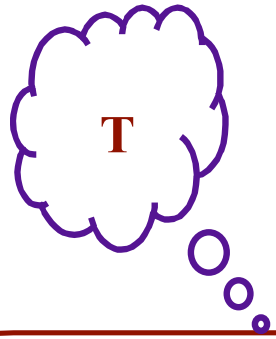
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle



if $a_0=b_0$

if $a_0>b_0$

```
callAcl2["(mv-let (erp val state)
          (defthm thm2
             (implies (and (integerp n)(< 0 n))
                      (> (* 3 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

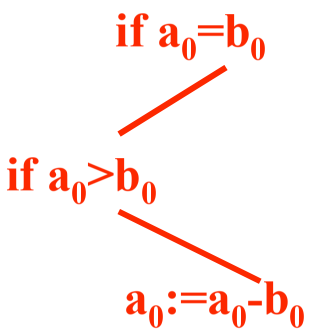
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

1st simulation cycle



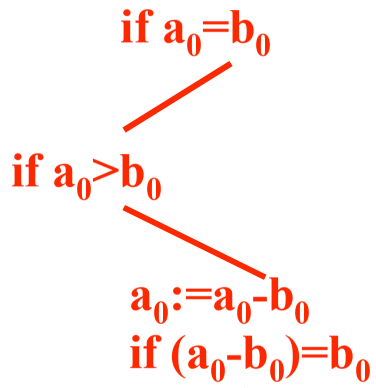
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle




```
callAcl2["(mv-let (erp val state)
          (defthm thm3
             (implies (and (integerp n)(< 0 n))
                      (equal (* 2 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

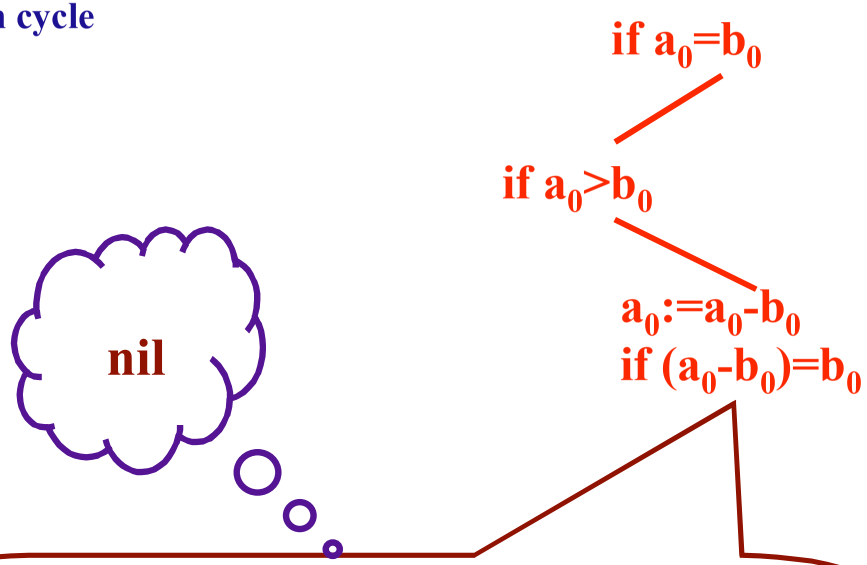
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle




```
callAcl2["(mv-let (erp val state)
          (defthm thm3
             (implies (and (integerp n)(< 0 n))
                      (equal (* 2 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

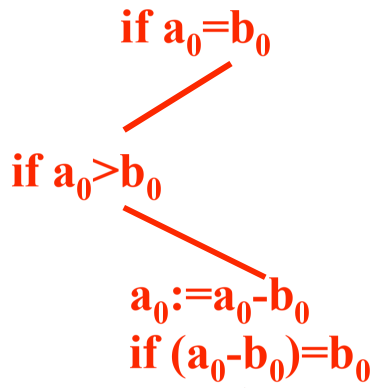
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle




```
callAcl2["(mv-let (erp val state)
          (defthm thm3-neg
             (implies (and (integerp n)(< 0 n))
                      (not (equal (* 2 n) n))))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

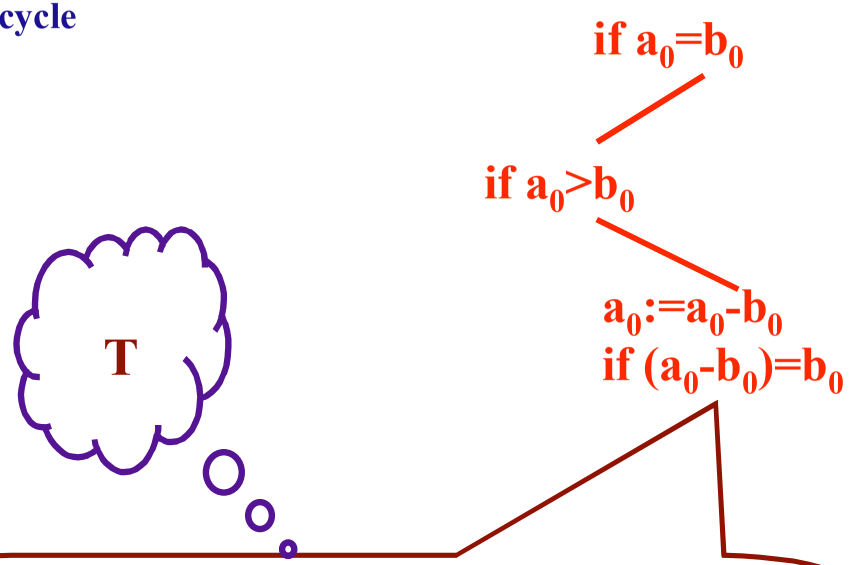
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle




```
callAcl2["(mv-let (erp val state)
          (defthm thm3-neg
             (implies (and (integerp n)(< 0 n))
                      (not (equal (* 2 n) n))))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

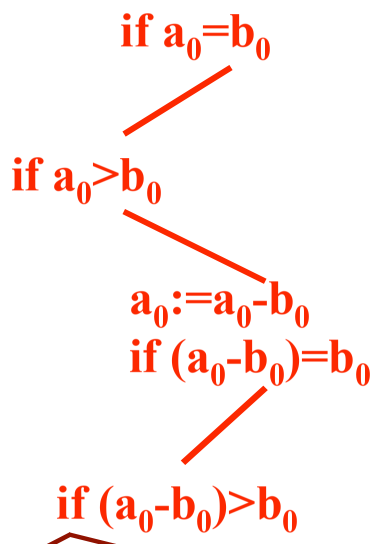
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle




```
callAcl2["(mv-let (erp val state)
          (defthm thm4
             (implies (and (integerp n)(< 0 n))
                      (> (* 2 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

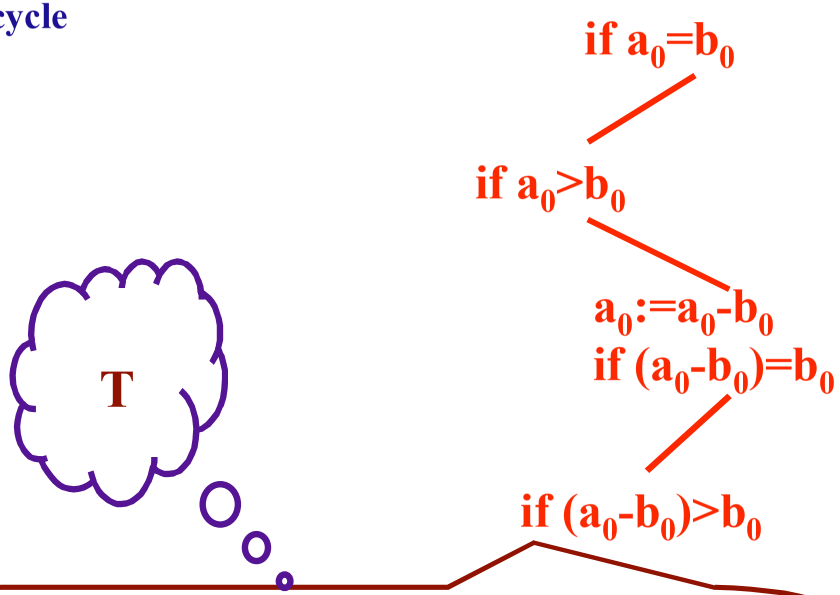
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle



T

```
callAcl2["(mv-let (erp val state)
          (defthm thm4
             (implies (and (integerp n)(< 0 n))
                      (> (* 2 n) n)))
          (declare (ignore val))
          (if erp (value nil)
             (value T)))"]
```

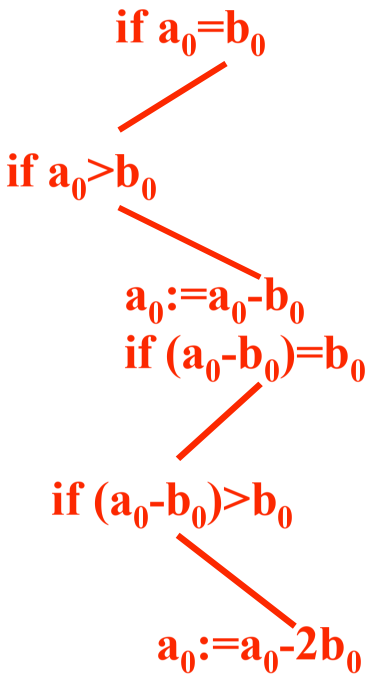
Reduction of the execution tree

Constraints:

$a=3n,\ b=n,\ n \in \mathcal{N}^*$

```
P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

2nd simulation cycle

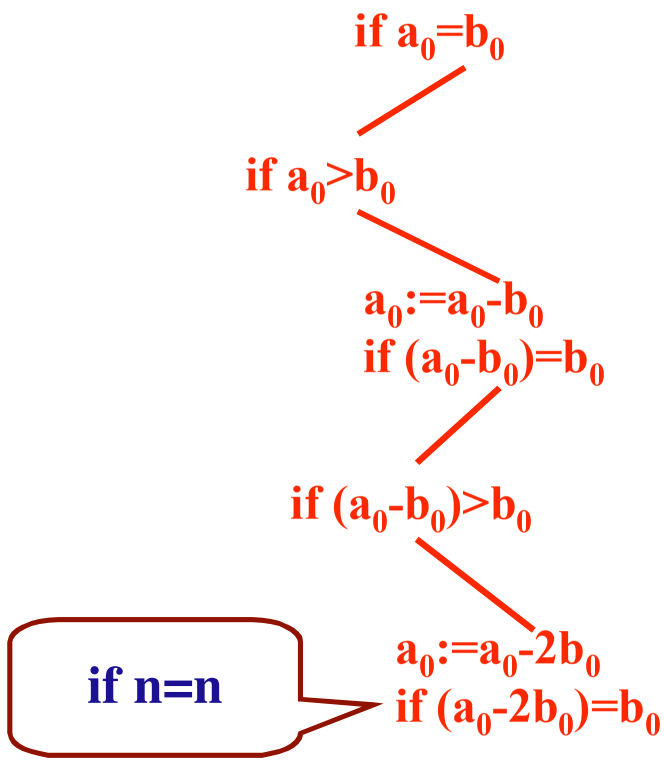


Reduction of the execution tree

```
Constraints:
    a=3n, b=n, n∈N*

P1: process begin
    wait until clk='1';
    if RST='1' then
        a0:=a;
        b0:=b;
        ok<=False;
    elsif a0=b0 then
            ok<=True;
            res<=a0;
    elsif a0>b0 then
            a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

3rd simulation cycle

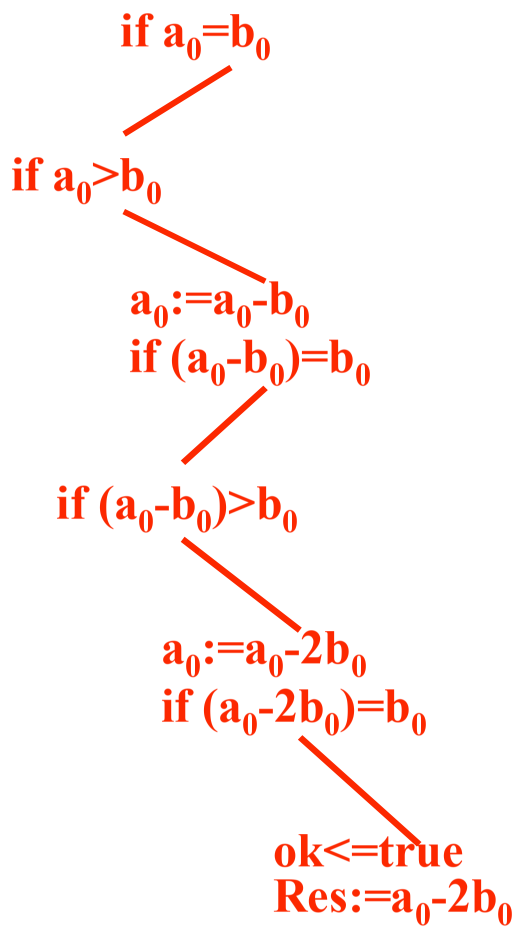


Reduction of the execution tree

```
Constraints:
    a=3n, b=n, n∈N*

P1: process begin
    wait until clk='1';
    if RST='1' then
       a0:=a;
       b0:=b;
       ok<=False;
    elsif a0=b0 then
          ok<=True;
          res<=a0;
    elsif a0>b0 then
          a0:=a0-b0;
    else b0:=b0-a0;
    end if;
end process P1;
```

3rd simulation cycle



Symbolic evaluations of assertions

- the assert statement assures that *bool_expr* is never violated
- *Label1* is translated into a variable that can remain symbolic during one or more simulation cycles
- If it evaluates to false at cycle C, the simulation path is a counter example

VHDL	Mathematica *if* function
Label1: assert bool_expr	If[bool_expr
report "message"	,ChangeVar[Label1,True]
severity severitylevel;	,ChangeVar[Label1,False]
	,If[CallACL2[bool_expr]
	,ChangeVar[Label1,True]
	,ChangeVar[Label1,False]
	,Label1]]

Conclusion

- A new approach for the symbolic simulation of high level circuits specifications
- Use of typing information and user constraints to prune the execution tree
- Use of two powerful automatic systems: Mathematica and ACL2

Future works:

- Validate the approach on industrial circuits
- Extend to new VHDL subset for the system-level synthesis and SystemC

Thank you

Foreseen VHDL subset

- Level 1 synthesizable subset
  - Concurrent statements:
    - Signal assignment
    - Component instantiation
    - Processes : single clock synchronized processes
  - Sequential statements:
    - Variable and signal assignments
    - *if-then-else*, *case* conditionals, *for-loop* statements
  - Types
    - Scalar data types: integer, bit, boolean, character
    - Subtypes defined on integer, bit vector type
  - Hierarchy: components do not contain combinatorial processes

Simulation cycle=clock cycle