Efficient Compilation of Cyclic Esterel Programs

Jan Lukoschus
Reinhard von Hanxleden

Christian-Albrechts Universität Kiel
Faculty of Engineering
Dept. of Computer Science and Applied Mathematics
Real-Time Systems and Embedded Systems Group
www.informatik.uni-kiel.de/~{jlu|rvh}

SYNCHRON 2003, December 2003
Overview

Classes of Cyclic Esterel Programs

Existing solutions

Proposal 1: Runtime solution

Proposal 2: Static Partial Evaluation

Proposal 3: Esterel preprocessing for cyclic signals
Classes of Cyclic Esterel Programs

Esterel programs with cyclic dependencies can be differentiated into these categories:

▶ Non constructive programs
  ▶ Non-deterministic programs
  ▶ Non-reactive programs
▶ Constructive programs
  ▶ Statically schedulable programs
  ▶ Only dynamically schedulable programs
Non Constructive Programs

Two possible solutions → not deterministic

No stable solution → not reactive
Constructiveness may depend on environment

```plaintext
present [ S or A ] then
  emit B
end
||
present [ T or B ] then
  emit A
end
```

- Presence of A executes emit B which implies emit A
- Presence of B executes emit A which implies emit B
- If neither S or T is present then the presence of A and B is undefined

This program is constructive only if we can assure that S or T is present in each instant at runtime
Classes of Cyclic Esterel Programs

Existing solutions
Proposal 1: Runtime solution
Proposal 2: Static Partial Evaluation
Proposal 3: Esterel preprocessing for cyclic signals

Non Constructive Programs
Conditionally constructive programs
Static cycles
Dynamic cycles

Cyclic Gate Representation

module cycle_guarded:

present S then
  present A then
    emit B
  end
else
  present B then
    emit A
  end
end

No fixed execution order is valid for these two assignments
Cyclic Gate Representation

\[
\begin{align*}
B &:= S \land A \\
A &:= \neg S \land B
\end{align*}
\]

State exploration of $S$ results in static solution:

\[
\begin{align*}
B &:= 0 \\
A &:= 0
\end{align*}
\]
Static Schedule for a Cyclic Program

```
present S then
  present A then
    emit B
  end
else
  present B then
    emit A
  end
end
```

```
if (S) {
  if (A) {
    B = 1;
  }
} else {
  if (B) {
    A = 1;
  }
}
```

If cyclic dependent blocks are mutually exclusively executed, a static schedule is possible.
Cyclic program without a static schedule

```plaintext
present [ S or A] then
  emit B
end
||
present [not S or B] then
  emit A
end
```

- Presence of S executes emit B which enables emit A
- Absence of S executes emit A which enables emit B

If blocks can activate each other mutually, a static schedule may not be derived directly.
Existing solutions for constructive cyclic programs

Two main principles to compile Esterel programs:

- Automata code (Esterel v3)
  - Execution of a flat automaton
  - Fast
  - Handles constructive cyclic programs
  - State explosion may happen

- Netlist code (Esterel v5)
  - Emulation of a logic circuit
  - Slow
  - Constructive cyclic programs require explicit synthetisation
  - Linear in size to Esterel source
  - Also used for hardware synthesis

Gerard Berry.  
*The Constructive Semantics of Pure Esterel.*  
Other Esterel Compilation Schemes

- EC (Stephen E. Edwards)
  Based on traversing an control flow graph in each instant

- SAXO-RT (Weil, Bertin, Closse, Poize, Venier, Pulou)
  Ordered execution of basic statement blocks controlled by activation bit masks

Stephen A. Edwards.
An Esterel compiler for large control-dominated systems.

Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil.
SAXO-RT: Interpreting esterel semantic on a sequential execution structure.
Classes of Cyclic Esterel Programs

Existing solutions
Proposal 1: Runtime solution
Proposal 2: Static Partial Evaluation
Proposal 3: Esterel preprocessing for cyclic signals

Other approaches

EC and SAXO-RT

Common properties of both compilers:

- Generated code size is nearly proportional to the size of Esterel code (no state explosion like v3)
- Even big programs are compileable
- Program size has not much impact on execution speed (no execution of inactive program parts like v5)
- *No support for constructive cyclic programs* because of fixed order of execution for basic blocks
A Synchronous Block Diagram Language

Handles cyclic dependencies at runtime

- Three-valued signals
- Fixpoint iteration at runtime to determine signal values
- Elaborated scheme to minimize the number of iterations in each instant
- Implemented in Ptolemy Classic

Proposal 1: Extend Static Approach with Dynamic Scheduler for Cyclic Parts

Outline:

- Per default, use static scheduling approach (e.g., SAXO-RT)
- Identification of basic blocks which are strongly connected via cyclic signal and control dependencies ("Cycle")
- Remaining blocks, which are not part of a cycle, are either unrelated, predecessor, or successor of these cyclic blocks
- Execution of cyclic and remaining blocks in order of signal dependency
- Execution of cycles enclosed in individual fixpoint iteration loops
Proposal 1: Static Approach + Selective Dynamic Scheduling

Implementation:

- New bit map for signals which are part of a cyclic dependency
- Additional bit map indicates known presence/absence status of signals
- Prepend each basic block with an additionally synthesized predicate
- Predicate checks if execution of the block could deliver only known signal statuses for the cyclic dependent signals
- Another bit map stores successful execution of blocks to inhibit repeated execution in the same instant
- Iteration until all activated blocks of a cycle have been executed
Proposal 1: Static Approach + Selective Dynamic Scheduling

Drawbacks:
- Slow
  - Looping over all active blocks in the cycle
  - Evaluation of the additional predicate
- Big
  - Storage for additional bitmaps
  - Control statements
Proposal 2: Static Partial Evaluation of Cycles

Idea: Application of netlist synthetization from v5

- Determine the sets of blocks which are strongly connected by cyclic signal and control dependencies
- Consider all combinations of presence/absence for all input signals relevant to one block
- Compute all signal emissions by this set via fixpoint iteration
  - If output signals with unknown presence status remain, perform reachability analysis for respective input signal combinations
  - If problematic input signal combinations are reachable, then the program is not constructive and therefore rejected
Proposal 2: Static Partial Evaluation of Cycles

- Compute netlist expressions for each signal in the cycle
- Each netlist expression contains only input signals
- Isolate one block in each strongly connected cycle
- Replace each cycle signal occurrence inside that block with netlist expressions for each output signal
- Now the cycle is cut and a static schedule is possible

A possible scheme for this task is described in:

Pros and Cons of Proposal 2:

Benefits:

- Speed improvement: blocks in the cycles are only executed once in each instance
- Size: No additional bitmaps or predicates

Disadvantages:

- Implementation is compiler specific
Proposal 3: Esterel preprocessing for cyclic signals

Idea: Cutting of cyclic dependencies at Esterel source code level

Such a preprocessor would:

- read an Esterel file
- identify cycles
- perform partial evaluation of cycles
- selective replacement of expressions
- write the result as an Esterel program

Now other compilers can read the new (now non-cyclic) program to produce efficient C code
Additional tasks for the preprocessor

Since a standalone preprocessor cannot use the infrastructure of an existing compiler, an implementation must provide:

- Scanner/parser for Esterel to interpret the tree structure of the program
- Expansion of run modules
- Identification of signal and control dependencies between the statements
- Identification of cyclic dependencies
- Fixpoint iteration for all input signals for that block including reachability analysis
- Isolate one signal in the cycle and replace with respective netlist expression
- Write modified Esterel file
Esterel resynthesis for cyclic signals

Benefits:

- Compiler source code is not needed
- Solution is not compiler specific

Disadvantages:

- Duplication of effort for program analysis
Example: (Simplified) Token Ring Arbiter

module STATION:

input Request; output Grant;
input Pass; output PassNext;
input Token; output TokenNext;

loop
    present [Token or Pass] then
    present Request then
        emit Grant
    else
        emit PassNext
    end
    end present ;
    pause
end loop

||

loop
    present Token then
    pause ;
    emit TokenNext
else
    pause
end
end

One station of the token ring arbiter

Gerard Berry. 
*The Esterel v5 Language Primer.*
Token Ring Arbiter: Network Structure

- The network stations are connected in a circle via their Token (T<sub>n</sub>) and Pass (P<sub>n</sub>) input/output signals.
- The Request (R<sub>n</sub>) and Grant (G<sub>n</sub>) signals are local on the network stations.
- One station gets the token (not shown here) at system startup.
Token Ring Arbiter: Network Structure in Esterel

module BUS:

input Request1, Request2, Request3;
output Grant1, Grant2, Grant3;

signal Pass1, Pass2, Pass3,
    Token1, Token2, Token3
in
    emit Token1
    ||
    run Station1/STATION
        [ signal Request1 / Request,
           Grant1 / Grant,
           Pass1 / Pass,
           Pass2 / PassNext,
           Token1 / Token,
           Token2 / TokenNext
        ]
    ||
    run Station2/STATION
        [ signal Request2 / Request,
          ...
        ]
Token Ring Arbiter: Expanded Run Modules

Before we can start the cycle analysis, the run modules must be expanded

```plaintext
module BUS:

    input Request1, Request2, Request3;
    output Grant1, Grant2, Grant3;

    signal Pass1, Pass2, Pass3,
        Token1, Token2, Token3

in
    emit Token1

|| % Station1
loop
    present [ Token1 or Pass1 ] then
        present Request1 then
            emit Grant1
        else
            emit Pass2
    end
end;
    pause
end

|| % Station2
loop
    present [ Token2 or Pass2 ] then
        present Request2 then
            emit Grant2
        else
            emit Pass3
    end
end;
    pause
end
```
Token Ring Arbiter: Cyclic Dependencies

This is the cyclic dependency found in the token ring arbiter:

- Emission of Pass2 depends on Pass1
- Pass1 depends on Pass3
- Pass3 depends on Pass2

This non determinism will be broken up by the presence of one of the Token<sub>n</sub> signals.

```plaintext
present [ Token1 or Pass1 ] then
  present Request1 else
    emit Pass2
  end
end;
||
present [ Token2 or Pass2 ] then
  present Request2 else
    emit Pass3
  end
end;
||
present [ Token3 or Pass3 ] then
  present Request3 else
    emit Pass1
  end
end;
```
Problem with static scheduling

- In the cycle one "or" block gets a present token value which sets its output value.
- Now the following blocks can be executed in the order of the cyclic dependency.
- But this execution order cannot be done if the compiler only supports a fixed schedule of execution.
Problem with static scheduling

Solution:

- Computation of all possible signal values in the cycle from the input signals via fixpoint iteration
- Replacement of cyclic signals by their input signal expression
- Now a static schedule is possible
- **Optimization:** It is sufficient to replace only one cyclic signal to break the cycle
### Fixpoint Iteration

<table>
<thead>
<tr>
<th>Token</th>
<th>Request</th>
<th>Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>X X X</td>
<td>⊥ ⊥ ⊥</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 0 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 0 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 X X</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1 X</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1 0 0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>X 1 X</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>X 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 X 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>X X 1</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
Synthesis of new netlists

State exploration for Token1 to Token3 delivers the unreachability of 0, 2, or 3 tokens in the system. Therefore those input assignments are ignored, when writing the netlists for the Pass signal:

\[
\begin{align*}
\text{Pass1} & := (\overline{R1} \land \overline{R2} \land \overline{R3}) \lor (T1 \land \overline{R1}) \lor (T3 \land \overline{R1} \land \overline{R3}) \\
\text{Pass2} & := (\overline{R1} \land \overline{R2} \land \overline{R3}) \lor (T2 \land \overline{R2}) \lor (T1 \land \overline{R2} \land \overline{R1}) \\
\text{Pass3} & := (\overline{R1} \land \overline{R2} \land \overline{R3}) \lor (T3 \land \overline{R3}) \lor (T2 \land \overline{R3} \land \overline{R1})
\end{align*}
\]

Now the cyclic Pass signals can be computed in every valid instant without another cyclic signal on the right hand side.

Application of new netlists

```
|| % Station1
loop
  present [ Token1 or
    ((not Request1 and not Request2
     and not Request3)
    or (Token3 and not Request3)
    or (Token2 and not Request3
     and not Request2))
  ] then
    present Request1 then
      emit Grant1
    else
      emit Pass2
  end
end
pause
```

- Pass1 is replaced by expression
- Pass2, Pass3 are not changed
- Emission of Pass1 becomes superfluous
Summary

- Cyclic dependencies in Esterel programs should be handled at compile time to save CPU and Memory resources.
- There is no compelling reason to implement the cycle transformation in the compiler.
- Constructive cyclic Esterel programs could be transformed by a preprocessor into acyclic ones.
- This in effect extends the applicability of known, efficient compilation approaches to the domain of cyclic Esterel programs.
Addendum

This talk resulted in several fruitful comments and discussions. In particular, it was noted that

- the concepts proposed here should be applicable to synchronous languages in general (including, e.g., Lustre)
- it is not obvious how to apply the concept of partially evaluating cycles in the presence of registers (pause statements), for example in module `pause1` (slide 38), as the state of registers is not directly accessible from within the Esterel program

Regarding the second point, one idea (Stephen Edwards) to make the register state accessible was to augment a program with auxiliary signals that would indicate the current state module, as in module `pause1a` (slide 39)
Addendum

In the following, we will

- illustrate the case of cycles broken by registers
- give an example of a program enhanced to make register state accessible
- outline a more general scheme for dealing with cyclic Esterel programs at the source code level
Classification of cycles

1. True cycles
   - All dependencies can be present at same instant
   - Constructiveness may depend on inputs
   - Example: Token Ring Arbiter (slide 23)

2. False cycles
   - Not all dependencies are active at the same instant
     2.1 Cycles are broken by guards
        - Example: cycle_guarded (slide 6)
     2.2 Cycles broken by registers
        - These correspond to pause statements
        - Example: pause1 (slide 38)
Classification of cycles

- In a certain sense, true cycles can be considered particularly difficult:
  - to prove their constructiveness one has to analyze the reachable input space (e.g., the presence of a token in the case of the bus arbiter)
  - there does not exist a fixed order for evaluating the cycle (e.g., for the bus arbiter the required order of evaluation depends on which station has the token)

- However, even false cycles pose a problem for compilers that try to construct a statical evaluation order by ordering the signal dependencies, without taking into account guards or registers (which would in fact make the evaluation order irrelevant)
False cycle broken by register

- This example contains a cycle involving signals A and B
- Both blocks are executed in different instances separated with a “pause” statement, hence this is a false cycle, broken by a register
- **Question:** how to apply partial evaluation to remove the cycle?
- **One apparent problem:** State of registers is not accessible as input value to computed function
Accessing register states

**Idea:** Make register states visible by extra signals

```plaintext
module pause1a:
    input A, B;

    signal E, F in
        emit E;
        present A then
            emit B
        end;
    pause;
    emit F;
    present B then
        emit A
    end
end signal
```

- The emission of signals E and F takes place before and after the “pause” statement.
- So we found a way to access the state—but are not finished yet.
Partial evaluation of cycle

Try to remove dependency on signal A to break cycle, analogous to Token Ring Arbiter:

```plaintext
module pause1b:
  input A, B;

  signal E, F in
    emit E;
    present F and B then
      emit B
    end;
  pause;
  emit F;
  present B then
    emit A
  end
end signal
```

- A is emitted when F is emitted and B is present—hence replace test on A by test on F and B
- However, there is still a static cycle, on signal B!
- It does not appear obvious how to apply the partial evaluation concept here
- In the following, will provide alternative scheme
Break cycles

Other idea: Break cycles with auxiliary signals

1. Identify cyclic dependency broken by register; assume signal B carries this dependency
2. Within cycle, replace “emit B” by “emit B_” (B_ is a new signal)
3. Outside of the cycle, replace all tests for B by tests for “(B or B_)”
Example for auxiliary signal

To illustrate, modify pause1 by adding some usage of B outside of the cycle, resulting in pause2

\begin{verbatim}
module pause2: % Cyclic
input A, B;

  present A then
    emit B
  end;
  pause;
  present B then
    emit A
  end
||
  present B then
    something
  end
\end{verbatim}

\begin{verbatim}
module pause2a: % Acyclic
input A, B;

  signal B_ in
    present A then
      emit B_
    end;
  pause;
  present B then
    emit A
  end
||
  present [B or B_] then
    something
  end
end signal
\end{verbatim}
A General Code Transformation Scheme (for pure signals)

1. Select a dependency (edge) in the cycle, carried by some signal \( S \); the dependency has the form

\[
\text{emit } S \quad \text{% dependency source}
\]

\[
\ldots
\]

\[
\text{present } f(S) \text{ then } \ldots \quad \text{% dependency sink}
\]

where \( f(S) \) is an expression involving \( S \)

2. Replace dependency source with “\text{emit } S_-”, where \( S_- \) is a fresh signal

3. Replace dependency sink with “\text{present } f(S \text{ or expr})”, where “\text{expr}” is the result of evaluating the dependency (emission of \( S \) by dependency source in the untransformed program) at the current instant

4. Replace other tests for \( S \) by tests for “\((S \text{ or } S_-)\)”

5. If \( S \) is an output signal, add \( S_- \) to list of output signals
General Code Transformation—Notes

1. Instead of being tested by a present statement at the dependency sink, S might also be tested by a suspend statement (or any derived statement)

2. Evaluating the cycle at the current instant implies the detection of false cycles, for example, by tracing the inputs back to a register

3. If the dependency is a (false) dependency, broken by a register or a guard, “expr” becomes empty (as in pause2a)

4. If S in the current instant is not tested by any statement other than the dependency sink, do not need to emit the auxiliary replacement signal S_ (as in Token Ring Arbiter)

5. If S in the current instant is not emitted by any statement other than the dependency source (before transformation), can set S to false (absent) in f(S) in the dependency sink—which may lead to f(S) becoming empty (as in Token Ring Arbiter)
Application to Token Ring Arbiter

The first 3 steps of the general code transformation:

1. The cycle involves signals Pass1, Pass2, and Pass3
   Select arbitrarily the dependency carried by Pass1 to be broken
   f(Pass1) is “Token1 or Pass1”

2. As Pass1 is not tested by any statement other than the dependency sink, do not need auxiliary signal (see Note 4)
Application to Token Ring Arbiter

3. As Pass1 is not emitted by any statement other than the dependency source, can set Pass1 to absent in f(Pass1)—which then simplifies to Token1 (see Note 5). The replacement expression “expr” for the signal Pass1 is:

\[
\begin{align*}
( & \text{not Request1 and not Request2 and not Request3) } \\
or & (\text{Token3 and not Request3}) \\
or & (\text{Token2 and not Request3 and not Request 2}) \\
\end{align*}
\]

As there is no auxiliary signal, steps 4 and 5 of the general transformation become superfluous.
General Code Transformation—An Optimization

To avoid the need to always test for two signals ($S$ and $S_-$) as prescribed in Step 4, one may use another auxiliary signal:

- Introduce fresh auxiliary signal $S_{--}$
- Modify step 4: Instead of replacing other tests for $S$ by “($S$ or $S_-$)”, replace them by solely $S_{--}$
- Add globally parallel statement:
  “every [ $S$ or $S_-$ ] do emit $S_{--}$ end”

Note: This is akin to Common Subexpression Elimination (CSE), which does not necessarily need to be done at the source code level, but may also be performed during compilation/synthesis
Example of Optimization

module pause2b:
  input A, B;
  signal B_, B__ in
    present A then
      emit B_
    end;
  pause;
  present B then
    emit A
  end
  ||
  present B__ then
    something
  end
  ||
  every [B or B_] do
    emit B__
  end
end signal

- The new signal B__ is present if B or B_ are present
- Outside the cycle the test for “(B or B_)” is replaced by a simple test for B__
Example of cycle on output signal

- This example models a simple line repeater. The direction of information flow changes every clock tick.
- Depending on the clock state, signal A depends on signal B or vice versa.
- That static cycle is not removable by any means without changing the interface, since that cyclic dependency is specified as the interface behaviour.

```plaintext
module pause3: % Cyclic
inputoutput A, B;
loop
    present A then
        emit B
    end;
    pause;
    present B then
        emit A
    end;
    pause;
end
```

This example models a simple line repeater. The direction of information flow changes every clock tick.
- Depending on the clock state, signal A depends on signal B or vice versa.
- That static cycle is not removable by any means without changing the interface, since that cyclic dependency is specified as the interface behaviour.
Example of cycle on output signal

module pause3a: % Acyclic
inputoutput A;
input B;
output B_;

loop
  present A then
    emit B_;
  end;
  pause;
  present B then
    emit A
  end;
  pause;
end

▶ Have added auxiliary signal B_
▶ As B is not emitted by this module any more, changed it from inputoutput to just input
▶ Users of this module must replace tests for B by tests for (B or B_)
▶ If there is no other module that can still emit B, these tests simplify to just B_
Example of cycle on output signal

```plaintext
module pause3b: % Acyclic
inputoutput A;
input B; output B_;

signal B_ in
loop
  present A then
    emit B_;
  end;
  pause;
  present B then
    emit A
  end;
  pause;
end
||
  every [ B or B_ ] then
    emit B__ end
end signal
```

- Module pause3a after application of CSE optimization
- Have added auxiliary signals B_ and B__
- Users of this module must replace tests for B by tests for B__
Example of cycle broken by guard

```plaintext
module reg1:

input A, B, S;
output X, Y;

present S then
  present A then
    emit B     % dependency source
  end
else
  present B then % dependency sink
    emit A
  end
end;

present A then emit X end;
present B then emit Y end
```

- Module contains (false) dependencies, broken by guard $S$
- Select dependency carried by $B$ to break
Example of cycle broken by guard

```plaintext
module reg1_acyclic:

    input A, B, S;
    output X, Y;

    present S
        % Now empty then-branch
    else
        present B then emit A end
    end;

    present A then emit X end;
    present [ B or (S and A) ]
        then emit Y end
```

- As B is not used elsewhere in module and is not output, do not need auxiliary signal
- Emission of B becomes superfluous, hence the then-branch disappears
Example of cycle broken by guard

```plaintext
module reg1_acyclic_opt:

input A, B, S;
output X, Y;

% Deleted check altogether

present [ A or ((not S) and B) ]
   then emit X end;

present [ B or (S and A) ]
   then emit Y end
```

- Applied forward substitution again, now on signal A
- Net benefit depends on subsequent compilation/synthesis process
Example of cycle with `suspend`

```plaintext
module suspend_cyclic:
  output T1, T2;
  output P1, P2;

  suspend
    sustain P2
    when immediate [not (T1 or P1)]
  ||
  suspend
    sustain P1
    when immediate [not (T2 or P2)]
  ||
  loop
    emit T1; pause;
    emit T2; pause
end
```

- The signals P1 and P2 are connected cyclically because the “suspend” would cut the emission of each signal if the other one is not present.
- The cycle is dynamically resolved by the signals T1 and T2.
- Select dependency carried by P1 to break the cycle statically.
Example of cycle with suspend

```esterel
module suspend_acyclic:
output T1, T2;
output P1, P2;
output P1;

suspend
   sustain P2
   when immediate [not (T1 or 1)]
||
   suspend
   sustain P1_
   when immediate [not (T2 or P2)]
||
   loop
      emit T1; pause;
      emit T2; pause
   end
```

- Replace P1 by P1_ in the dependency source
- Compute partial evaluation “expr” for P1
- Here it results in P1 always present (symbolized by “1”)
- Replace P1 by “expr”
- Add P1_ to output
Example of cycle with suspend

```esterel
module suspend_acyclic:
output T1, T2;
output P2;  % P1 eliminated
output P1_;

  % suspend ... eliminated
sustain P2
||
suspend
  sustain P1_
  when immediate [not (T2 or P2)]
||
loop
  emit T1; pause;
  emit T2; pause
end
```

- Propagation of “always present” leads to elimination of the “suspend” wrapper of the “sustain P2”
Example with cyclic valued signals

```plaintext
module reg_value:

input S;
input A : integer, B : integer;
output X : integer, Y : integer;

present S then
    present A then
        emit B(?A) % dependency source
    end
else
    present B then % dependency sink
        emit A(?B)
    end
end;

present A then emit X(?A) end;
present B then emit Y(?B) end
```

- This program is a copy of “reg1” (slide 52) but with valued signals.
- If signals A and B can both be present on the input then a combine function is needed.
- Select dependency carried by B to break.
Example with cyclic valued signals

module reg_value_acyclic:
input S;
input A : integer, B : integer;
output X : integer, Y : integer;

signal B_:integer, B__:integer in
present S then
  present A then emit B_:?A end
else
  present B then emit A:?B end
end;
present A then emit X:?A end;
present B__ then emit Y:?B__ end
||
every B emit B__:?B
||
every B_ emit B__:?B_
end signal

- Added local auxiliary signals B_ to break the cycle via B
- Additional parallel statements transfer the values from B and B_ to B__
- Can construct other programs with dependencies on values of signals where partial evaluation is not obvious.
Conclusions

- Have proposed a general scheme for eliminating (true and false) cycles from Esterel programs

Remaining issues:

- Validate general applicability of translation scheme
- How to choose the dependencies to break (could look for, e.g., minimal resulting expressions)
- How to compute efficient replacement expressions for signals carrying dependencies at current instant
- Extending this to full Esterel, including valued signals
- Implementation and experimental evaluation