Compiling Esterel

Stephen A. Edwards

Department of Computer Science
Columbia University
www.cs.columbia.edu/~sedwards

Outline

Introduction to Esterel and Existing Compilers

My Earlier Compiler [DAC 2000, TransCAD 2002]

New Compiler: ESUIF (work in progress [SLAP 2002])

The Esterel Language

Developed by Gérard Berry starting 1983

Originally for robotics applications

Imperative, textual language

Synchronous model of time like that in digital circuits

Concurrent

An Example

```
emit B;
present C then
   emit D end;
```

Force signal present in this cycle (→ emit B;)

Make D present if C is (→ present C then)

An Example

```
await A;
emit B;
present C then
   emit D end;
pause
```

Wait for next cycle where A is present

Wait for next cycle

An Example

```
loop          ← Infinite Loop
   await A;
   emit B;
   present C then
      emit D end;
   pause
end
```

An Example

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||    <---------- Run Concurrently
   loop
      present B then
         emit C end;
      pause
   end
end
```

An Example

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```

Restart on R

An Example

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```

Same-cycle bidirectional communication

An Example

```
every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end
```

Good for hierarchical FSMs

Bad at manipulating data

Hardware Esterel variant proposed to address this

Automata Compilers

Esterel is a finite-state language, so build an automata:

```
loop                    switch (s) {
  emit A; await C;      case 0: A = 1; s = 1; break;
  emit B; pause         case 1: if (C) B = 1; s = 0; break;
end                     }
```

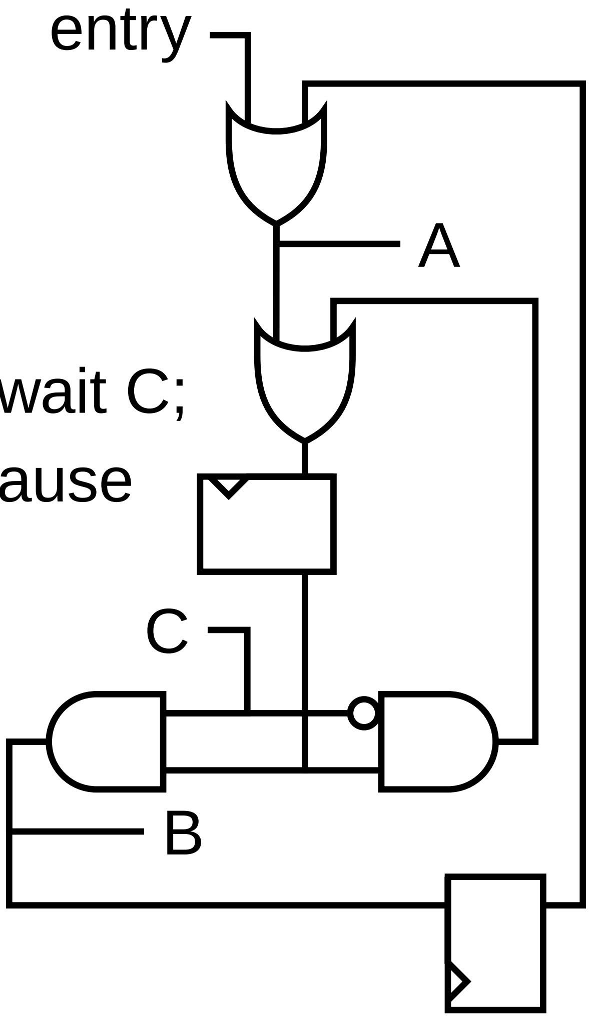
V1, V2, V3 (INRIA/CMA) [Berry, Gonthier 1992]

Fastest known code; great for programs with few states.

Does not scale; concurrency causes state explosion.

Netlist-based Compilers

```
loop
  emit A; await C;
  emit B; pause
end
```




```
A = entry || s2q;
cf = !C && s1q;
s1d = cf || A;
B = s2d = C && s1q;
```

Clean semantics, scales well, but inefficient.

Can be 100 times slower than automata code.

Other Esterel Compilers

Control-flow-graph based

- My work: EC [DAC 2000, TransCAD 2002]
- Produces very efficient code for acyclic programs only

Discrete-event based

- SAXO-RT [Weil et al. 2000]
- Produces very efficient code for acyclic programs only
- Being improved at Esterel Technologies?

Both proprietary; unlikely to be released.

Neither currently copes with statically cyclic programs.

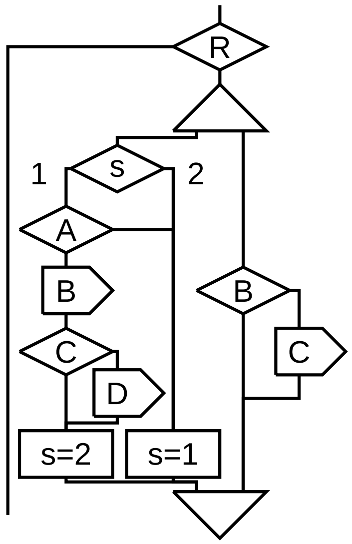
My Earlier Esterel Compiler

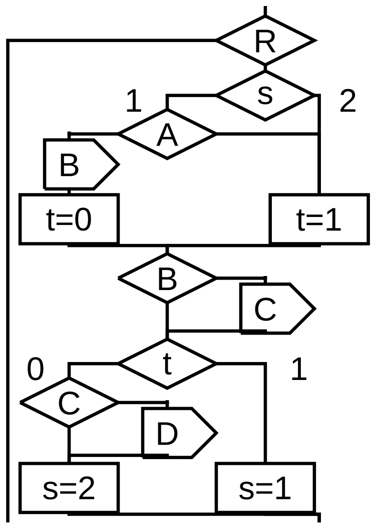
Presented at DAC 2000 (also TransCAD 2002)

Used inside Synopsys' CoCentric System Studio to generate control code

Outline

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```






```c
if ((s0 & 3) == 1) {
  if (S) {
    s3 = 1; s2 = 1; s1 = 1;
  } else
    if (s1 >> 1)
      s1 = 3;
    else {
      if ((s3 & 3) == 1) {
         s3 = 2; t3 = L1;
      } else {
         t3 = L2;
      }
```

Esterel Source

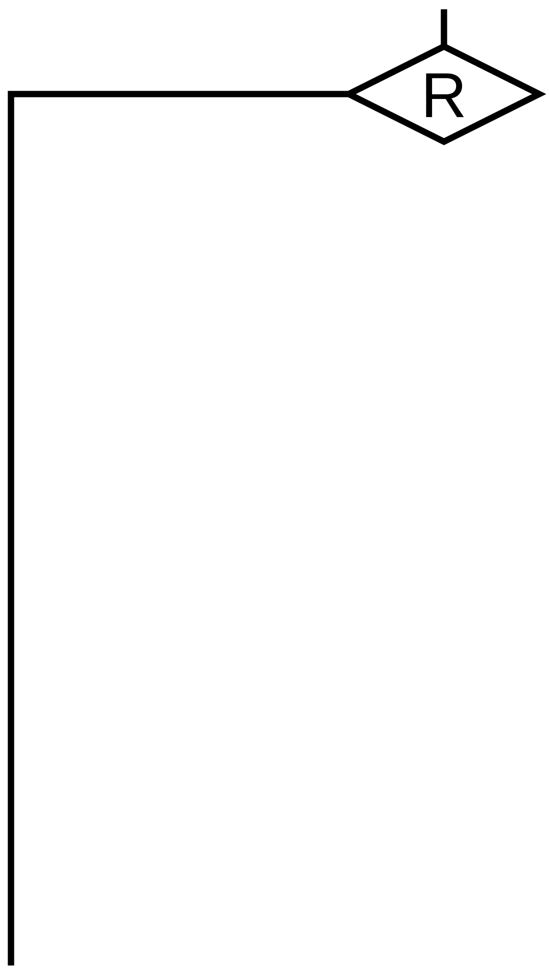
Concurrent CFG

Sequential CFG

C code

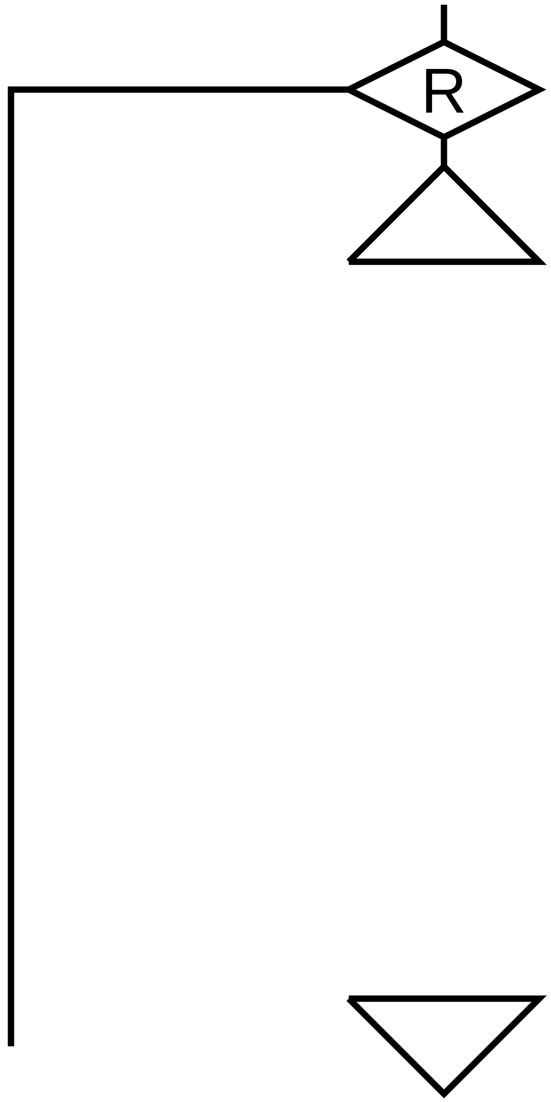
Translate every

```
every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end
```



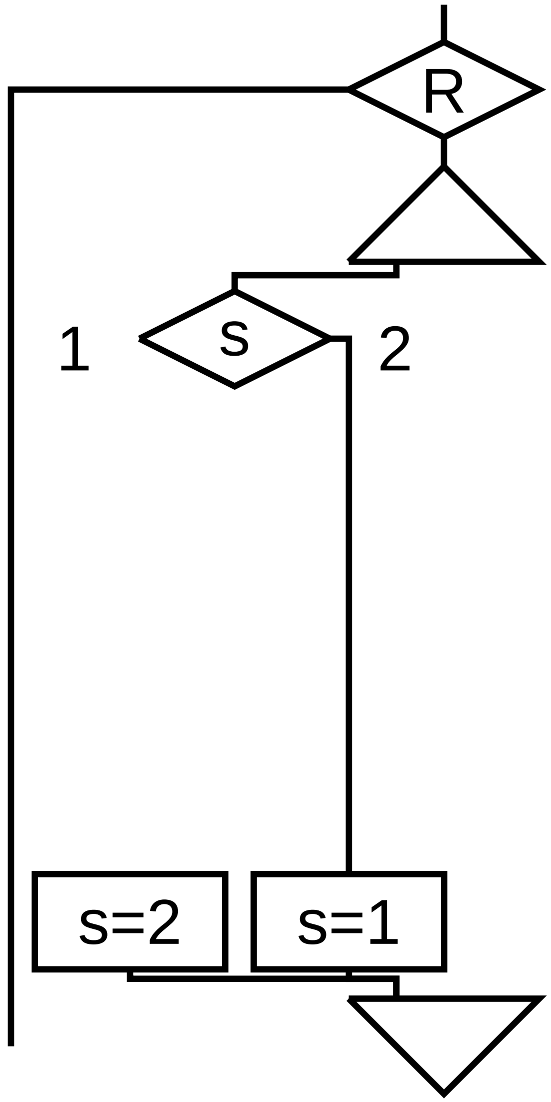
Add Threads

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```



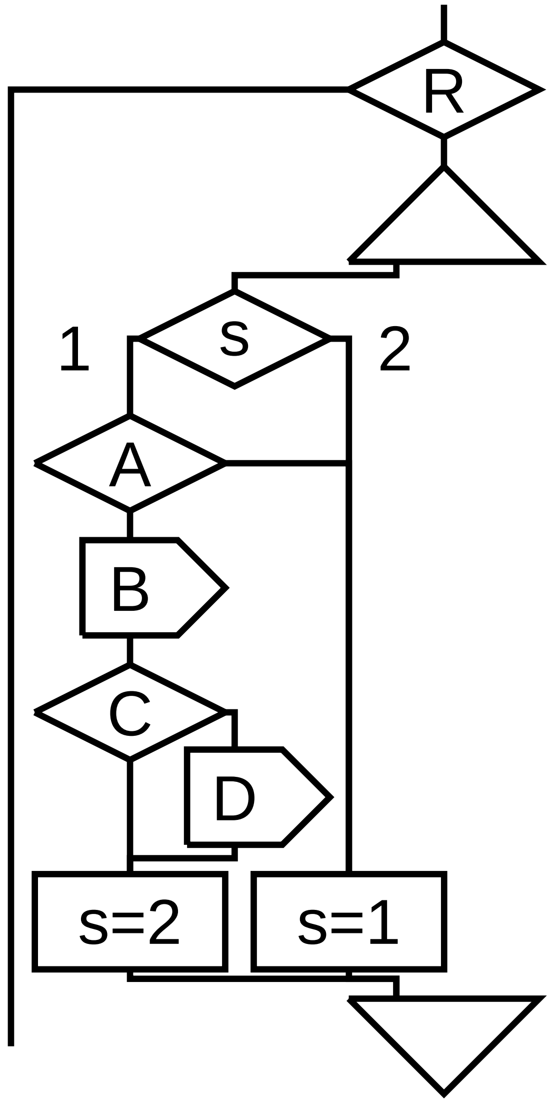
Split at Pauses

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```



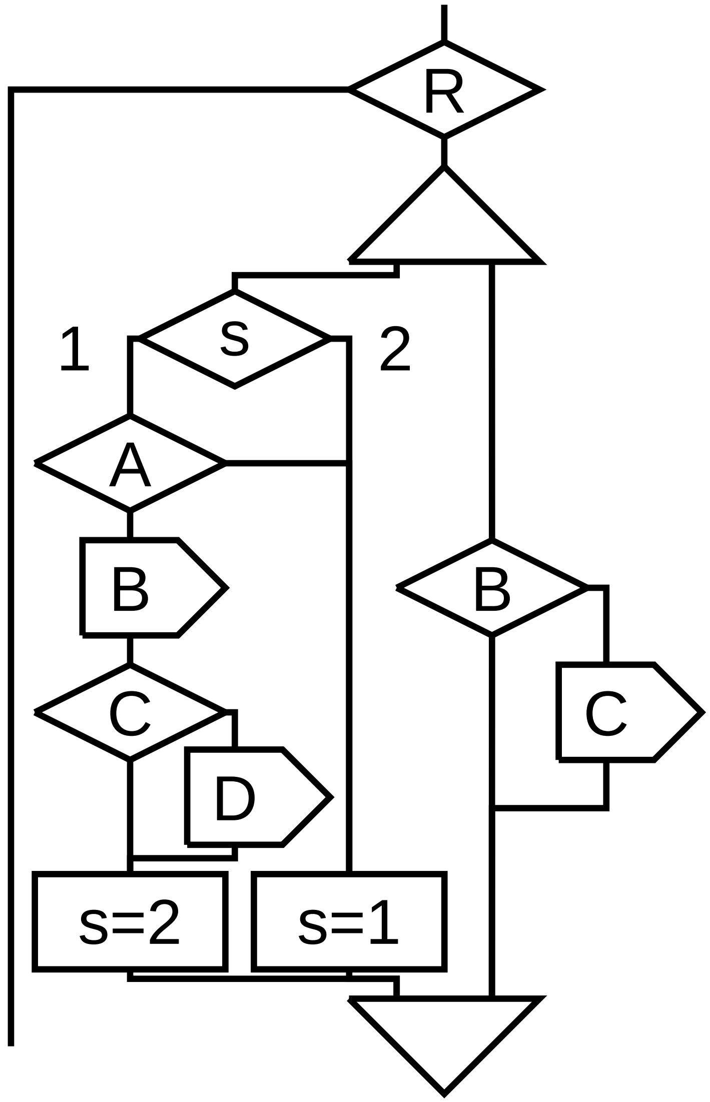
Add Code Between Pauses

```
every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end
```



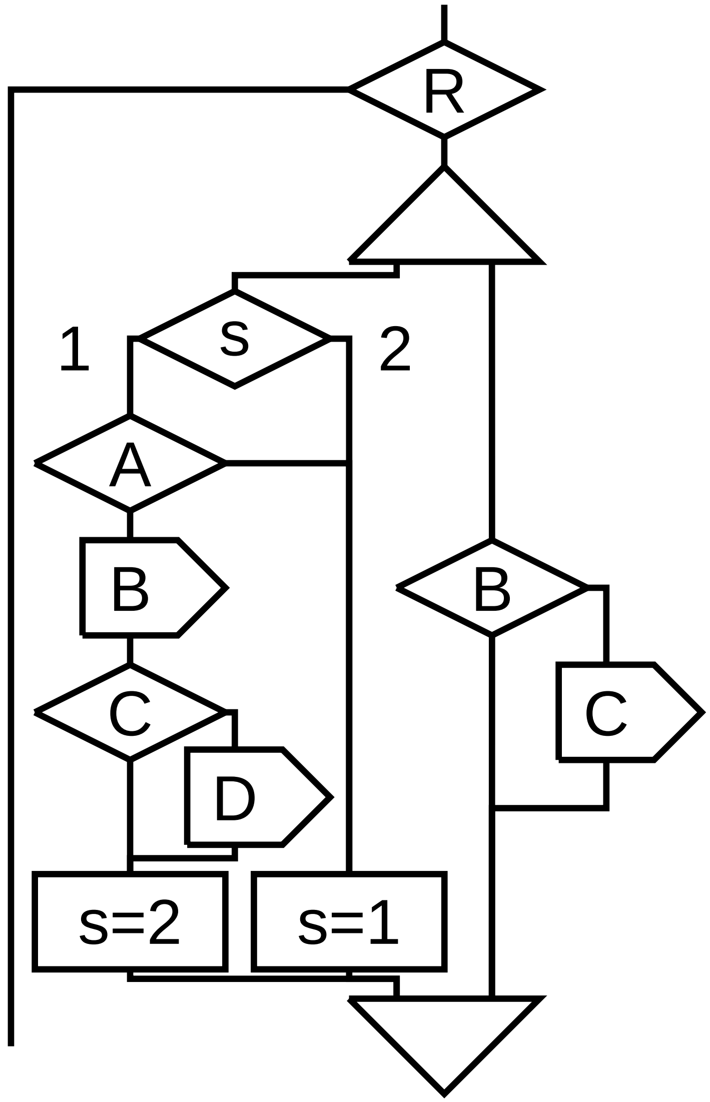
Translate Second Thread

```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```



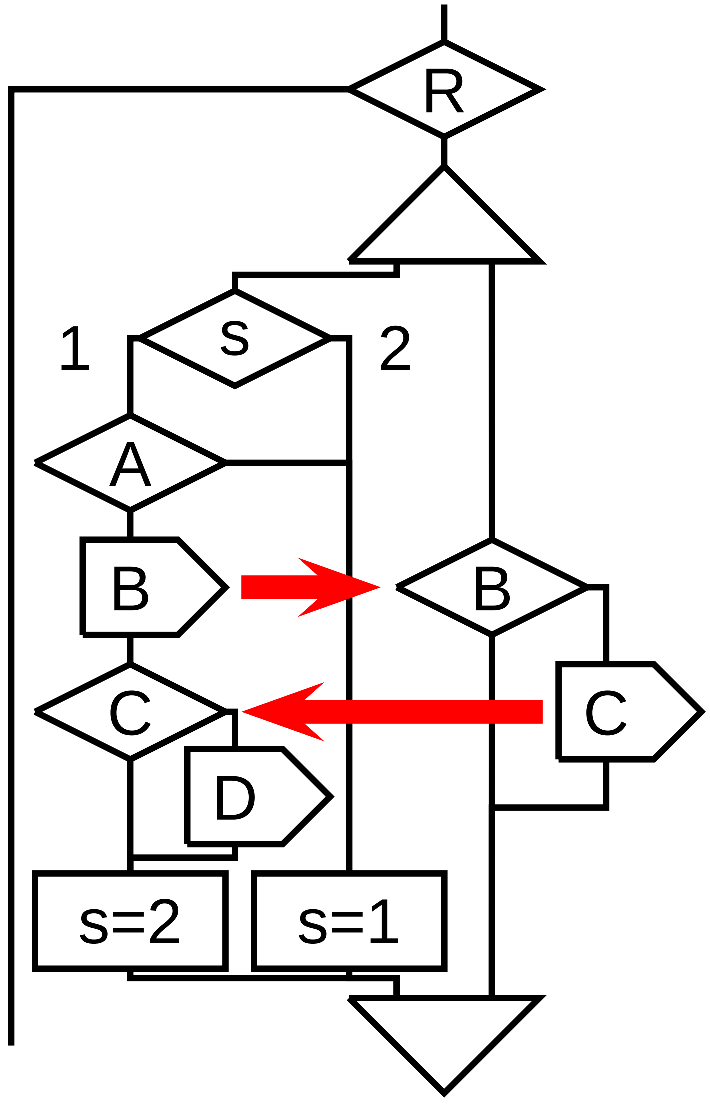
Finished Translating

```
every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end
```

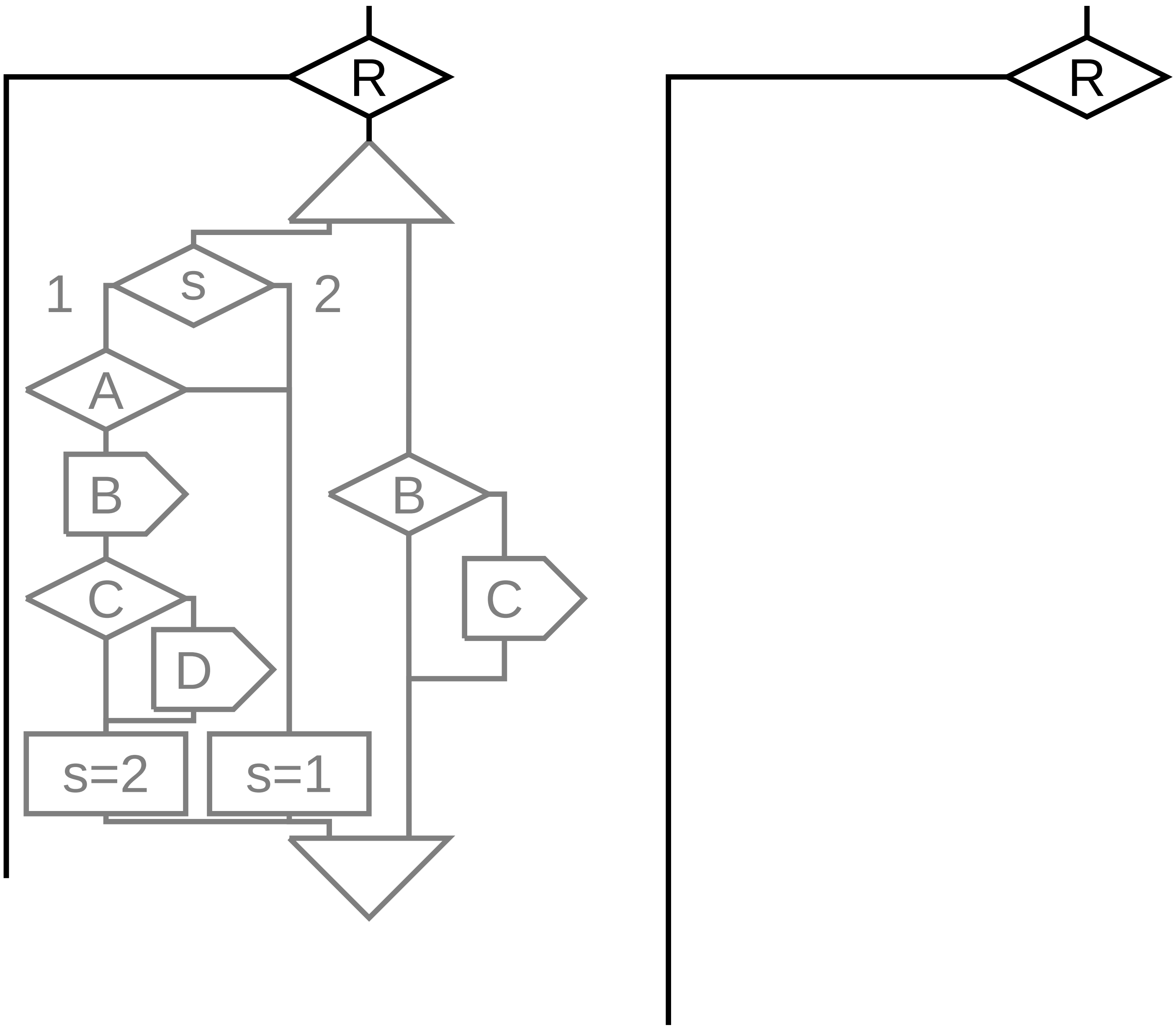


Add Dependencies and Schedule

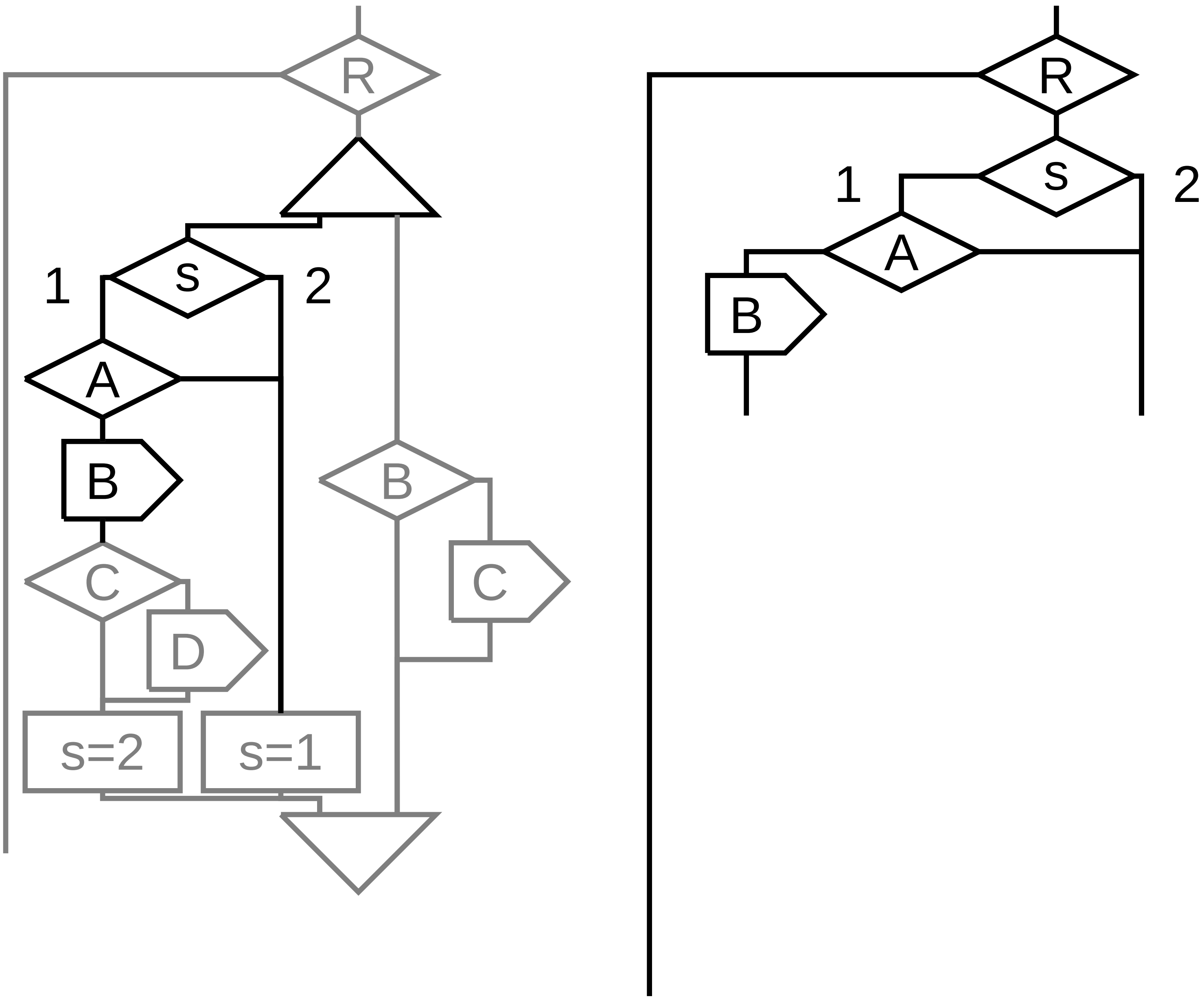
```
every R do
   loop
      await A;
      emit B;
      present C then
         emit D end;
      pause
   end
||
   loop
      present B then
         emit C end;
      pause
   end
end
```



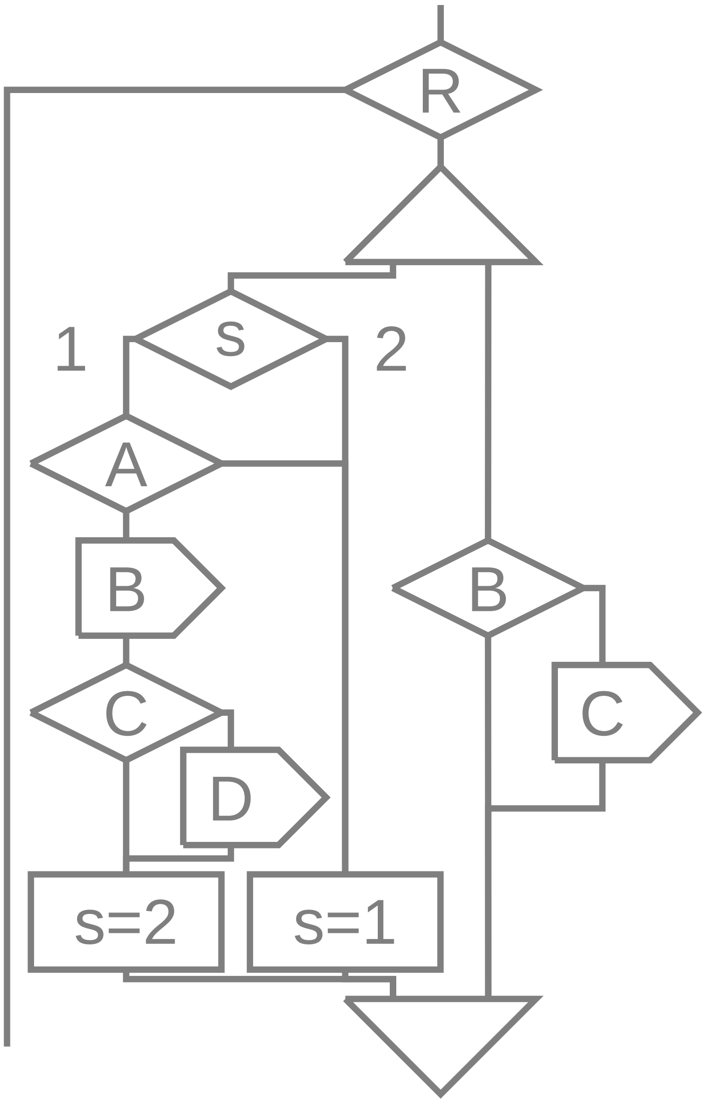
Run First Node

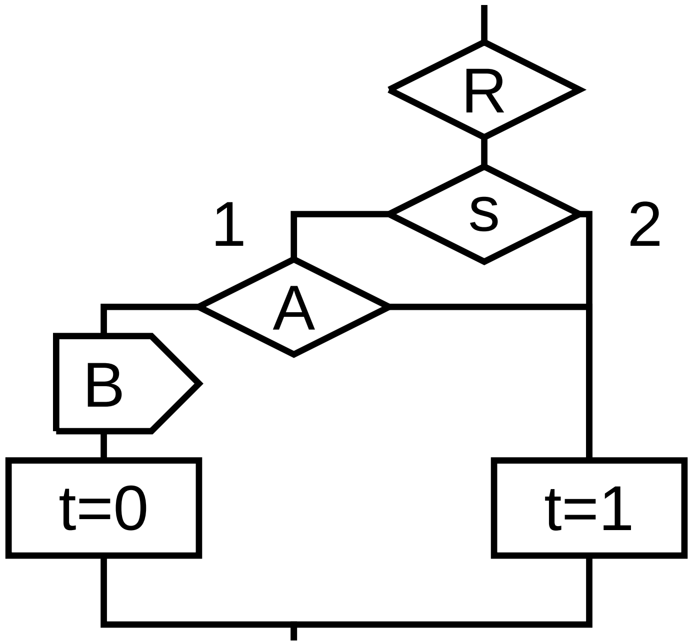


Run First Part of Left Thread

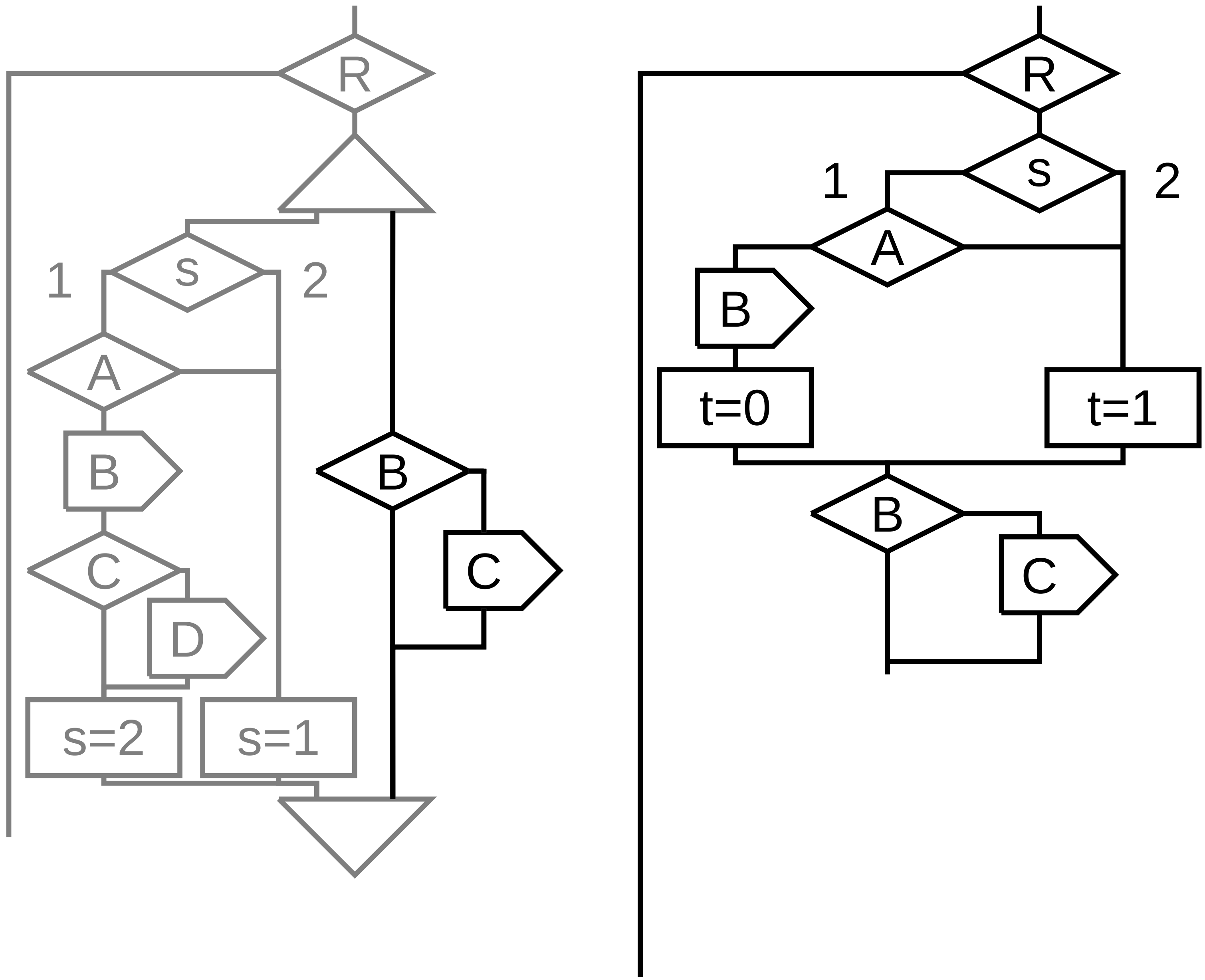


Context Switch

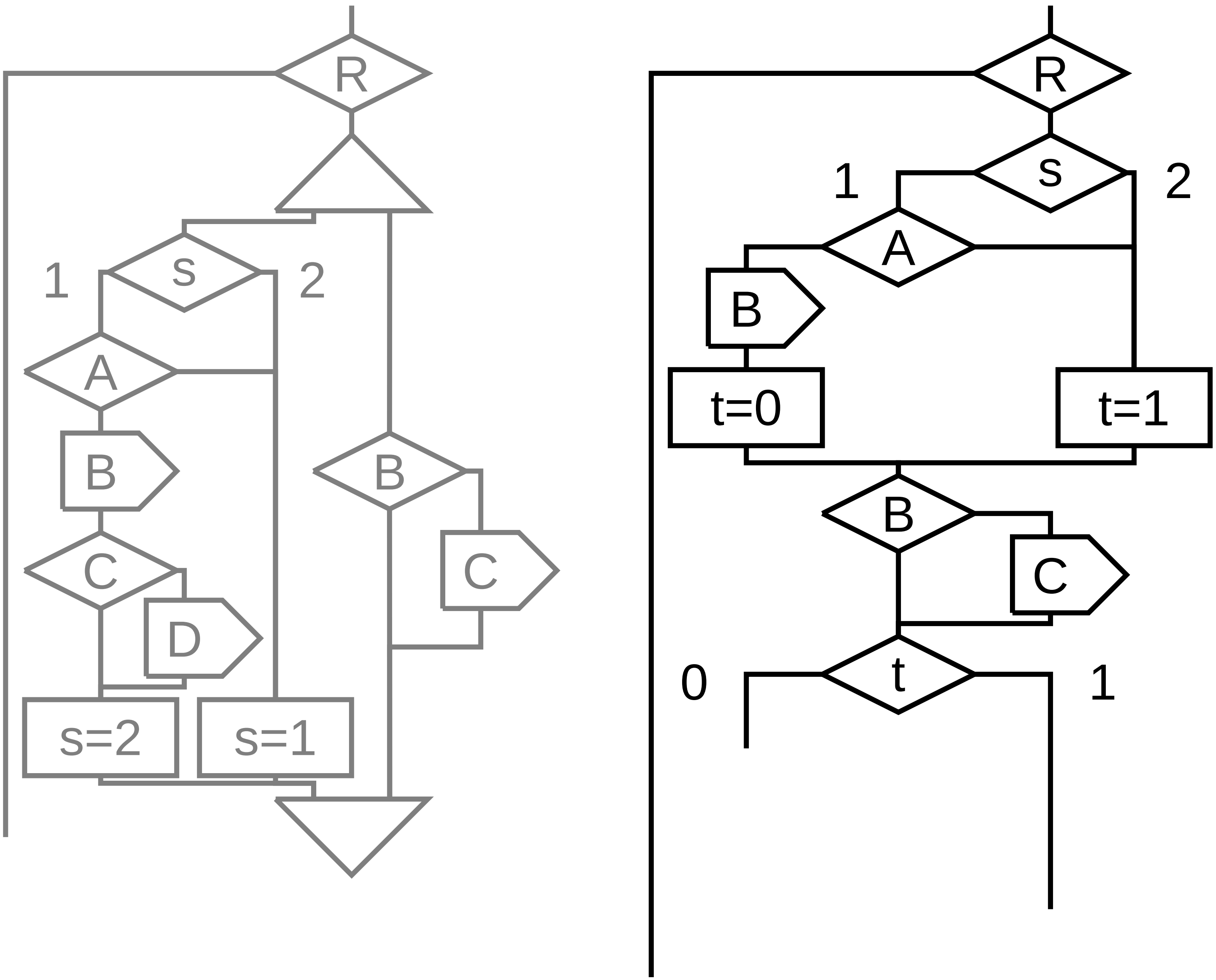




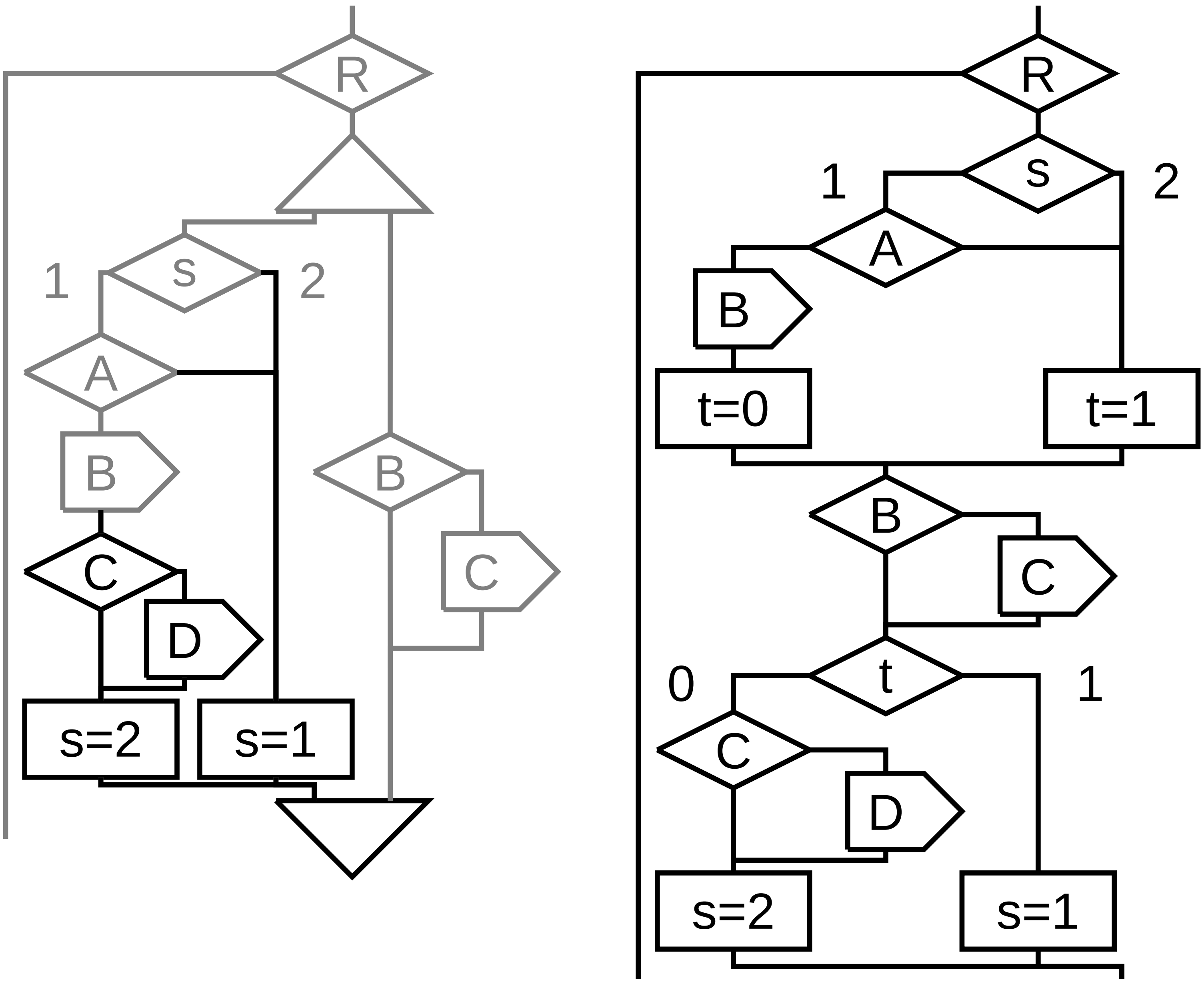
Run Right Thread



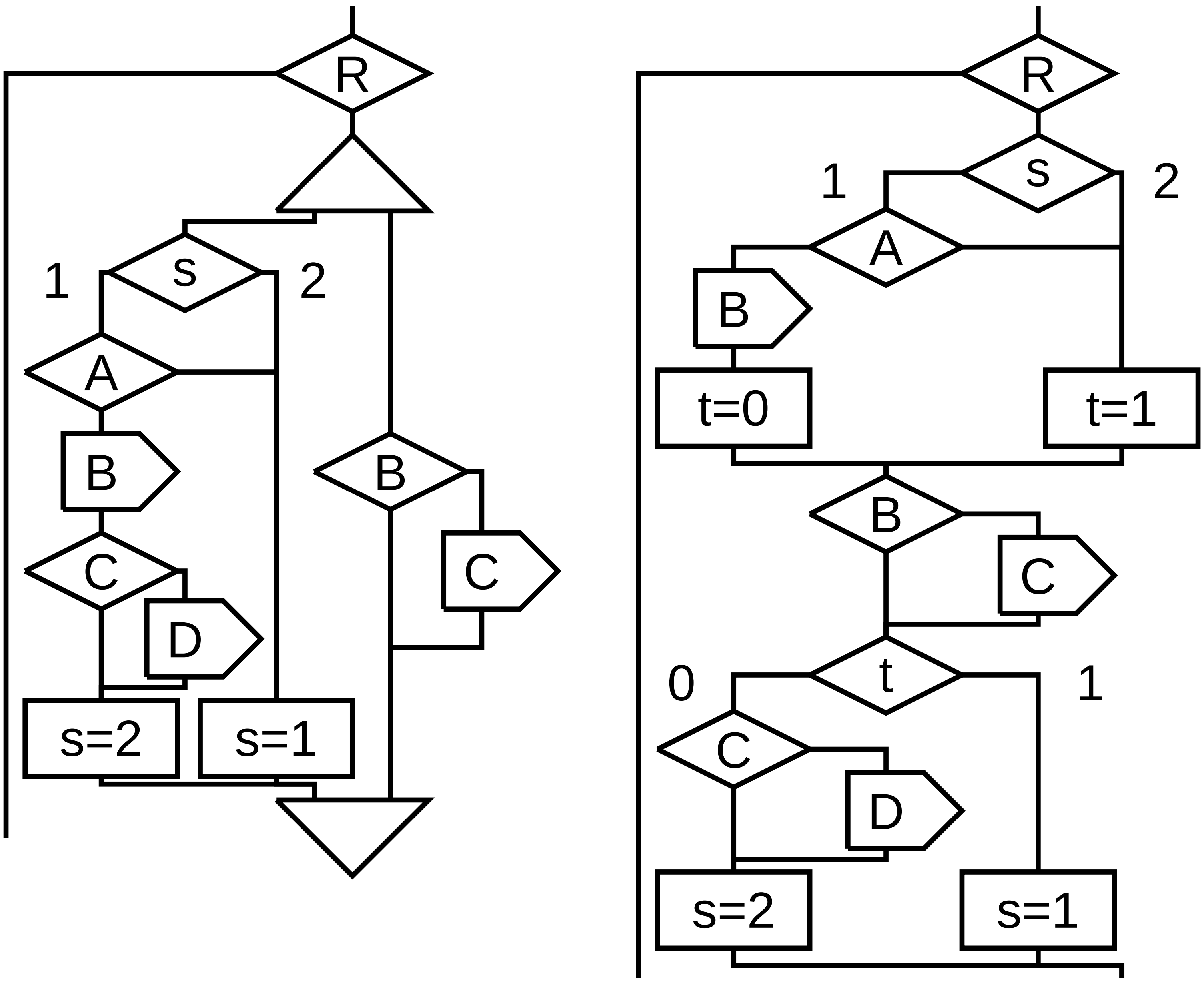
Context Switch



Finish Left Thread



Completed Example



ESUIF: An Esterel Compiler for Research

My goal is to improve Esterel compilation technology

We still don’t have a technique that builds fast code for large programs

No decent Esterel compiler available in source form

Being presented at SLAP 2002 (Grenoble, April)

ESUIF

New, open-source compiler being developed at Columbia

Based on SUIF 2 system from Stanford University

Much more modular: implemented as many little passes

Common database represents program throughout

SUIF 2 Database

Main component of the SUIF 2 system

User-customizable object-oriented database

Written in C++

Not highly efficient, but very flexible

SUIF 2 Database

Database schema written in their own “hoof” format

C++ implementation automatically generated

```
concrete MyClass {
  int x;
}
```

⇒

```
class MyClass : public SuifObject
{
public:
  int get_x();
  void set_x(int the_value);
  ~MyClass();
  void print(...);
  static const Lstring
    get_class_name();
}
```

Three Intermediate Representations

AST-like representation from front end

- Primitives: abort, emit, present, suspend, etc.

Lower-level “C-like” representation

- Primitives: if-then-else, try, resume, parallel, etc.

C code

- Primitives: if, goto, expressions
- SUIF 2 includes a complete C schema

My New Intermediate Representation

Intermediate Representation Goals

Linear, textual, imperative style fits the SUIF 2 philosophy

Gonthier's IC format used in V3–V5 is graph-based and difficult to visualize. Analysis requires depth-first search.

Straightforward translation into C code; simple semantics

IC format requires complicated depth-first search to linearize. Handling of "completion codes" is subtle.

Compound statements express traps, preemption, and concurrency

Tree structure present in IC, but must be rediscovered.

Intermediate Representation

```
var := expr
if (expr) { stmts } else { stmts }
Label:
goto Label

break n
continue
try { stmts } catch 2 { stmts } ...
resume { stmts } catch 1 { stmts } ...
parallel { resumes } catch 1 { stmts } ...

fork Label1, Label2, ...
join
```

Intermediate Representation

```
var := expr
if (expr) { stmts } else { stmts }
Label:
goto Label
```

Self-explanatory

Signals represented as variables.

Restrictions on where a goto may branch.

Intermediate Representation

`break` n

`continue`

`try { `*`stmts`*` } catch 2 { `*`stmts`*` } ...`

`resume { `*`stmts`*` } catch 1 { `*`stmts`*` } ...`

`parallel { `*`resumes`*` } catch 1 { `*`stmts`*` } ...`

Numerically-encoded “exceptions”

Based on Esterel’s completion codes

0=terminate 1=pause 2,3,. . . =exit

Implementing Exceptions

```
trap T1 in
  exit T1

handle T1 do
 c := 1
end
```

```
try {
  break 2

} catch 2 {
  c := 1
}
```

```
    goto Catch2;
    goto Catch0;
Catch2:
    c = 1;
Catch0:
```

try becomes a few labels.

break becomes a goto.

Resume/Continue

```
abort           resume {                     goto E
                                          C: switch (s) {
                                             case 0: goto St0;
                                             case 1: goto St1;
                                             }
  pause          break 1                  E: s = 0; goto Ca1; St0:
  pause          break 1                     s = 1; goto Ca1; St1:
                                             goto Ca0;
                } catch 1 {               Ca1:
                  break 1                   so = 0; goto Ca1o; St0o:
when A            if (!A) continue          if (!A) goto C;
                }                         Ca0:
```

`resume` becomes a multi-way branch plus some labels.

`continue` sends control to the multi-way branch.

Resume/Continue

First cycle:

```
    goto E
C:  switch (s) {
    case 0: goto St0;
    case 1: goto St1;
    }
E:  s = 0; goto Ca1; St0:
    s = 1; goto Ca1; St1:
    goto Ca0;
Ca1:
   so = 0; goto Ca1o;

St0o: if (!A) goto C;
Ca0:
```

Second cycle:

```
    goto E
C:  switch (s) {
    case 0: goto St0;
    case 1: goto St1;
    }
E:  s = 0; goto Ca1; St0:
    s = 1; goto Ca1; St1:
    goto Ca0;
Ca1:
   so = 0; goto Ca1o;

St0o: if (!A) goto C;
Ca0:
```

Parallel and Exit

```
trap T1 in
  trap T2 in



    exit T1
  ||
    exit T2


  handle T2 do emit B end
handle T1 do emit A end
```

```
try {
  try {
    parallel {
      resume {
        break 3 }
      resume {
        break 2 }
    } catch 1 {
      break 1; continue }
  } catch 2 { B := 1 }
} catch 3 { A := 1 }
```

Parallel

pause;
pause
||
pause

```
parallel {
  resume {
    break 1
    break 1
  }
  resume {
    break 1
  }
} catch 1 {
  break 1
  continue
}
```

Parallel Behavior

```
parallel {
  resume {
    break 1
    break 1
  }
  resume {
    break 1
  }
} catch 1 {
  break 1
  continue
}
```

```
parallel {
  resume {
    break 1
    break 1
  }
  resume {
    break 1
  }
} catch 1 {
  break 1
  continue
}
```

A Minor Point on Completion Codes

Berry’s encoding reduces the exit code if it is not handled.

```
try {
  break 5
} catch 2 { ... }
```

generates **`break 4`** in Berry’s encoding. I treat it as **`break 5`**.

I assign each trap its own completion code; they pass unchanged.

Simpler semantics vs. the danger of larger codes.

Irrelevant in HW, probably not a problem for SW.

Future Work on HW & SW Synthesis

- HW/SW synthesis from control dependence

  Clever concurrent representation produces efficient hardware and facilitates "sequentializing" SW.

- SW synthesis by static unrolling of cyclic programs

  Unrolling SW à la Bourdoncle coupled with constant propagation should quickly execute cyclic programs.

- SW synthesis with dynamic event-based scheduling

  Unrolling is expensive if done statically; a scheduler can do it dynamically with little overhead.

Summary

Introduction to Esterel and Existing Compilers

- Synchronous, Concurrent, Textual Language
- Automata, Netlist, and Control-based compilers

My Earlier Compiler [DAC 2000, TransCAD 2002]

- Translate to Concurrent CFG, schedule, then synthesize Sequential CFG

New Compiler: ESUIF (work in progress [SLAP 2002])

- Based on SUIF 2 infrastructure
- Open-source, under development
- Intermediate Representation
- Future Work