Sequentially Constructive Concurrency*

A conservative extension of the Synchronous Model of Computation

Reinhard v. Hanxleden[1], Michael Mendler[2], J. Aguado[2],
Björn Duderstadt[1], Insa Fuhrmann[1], Christian Motika[1],
Stephen Mercer[3] and Owen Brian[3]

* to appear at DATE, Grenoble, March 2013

[1] University of Kiel, [2] University of Bamberg, [3] National Instruments

Motivation (Taming Concurrency)

Synchronous Languages
Esterel, Lustre, Signal, ...

Clocked, cyclic schedule

- by default: single writer per cycle, all reads initialised
- on demand: separate multiple assignments by clock barrier (pause, wait)

Declarative

- all *micro-step* sequential control flow descriptive
- resolved by scheduler

Sequential Languages
C, Java, ...

Asynchronous schedule

- by default: multiple concurrent readers/writers
- on demand: single assignment synchronisation (locks, semaphores)

Imperative

- all sequential control flow prescriptive
- resolved by programmer

Motivation (Taming Concurrency)

Synchronous Languages
Esterel, Lustre, Signal, ...

Clocked, cyclic schedule

ü deterministic concurrency and deadlock freedom

✘ Heavy restrictions by constructiveness analysis

Sequential Languages
C, Java, ...

Asynchronous schedule

✘ No guarantees of determinism or deadlock freedom

ü Intuitive programming paradigm

Sequentially Constructive Model of Computation (SC MoC)

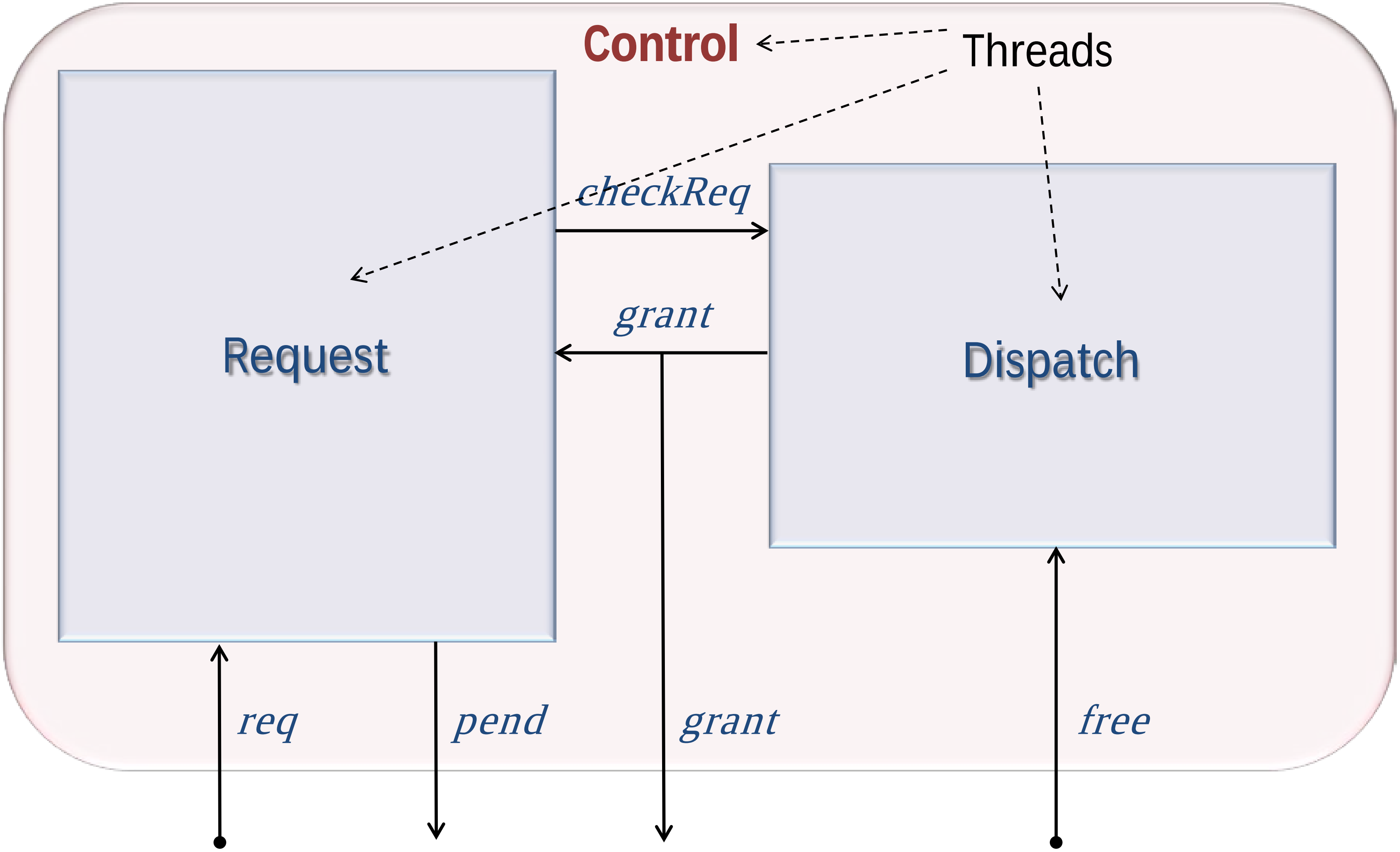
- all *micro-step* concurrent control flow descriptive
- resolved by scheduler

- all *micro-step* sequential control flow is prescriptive
- resolved by programmer

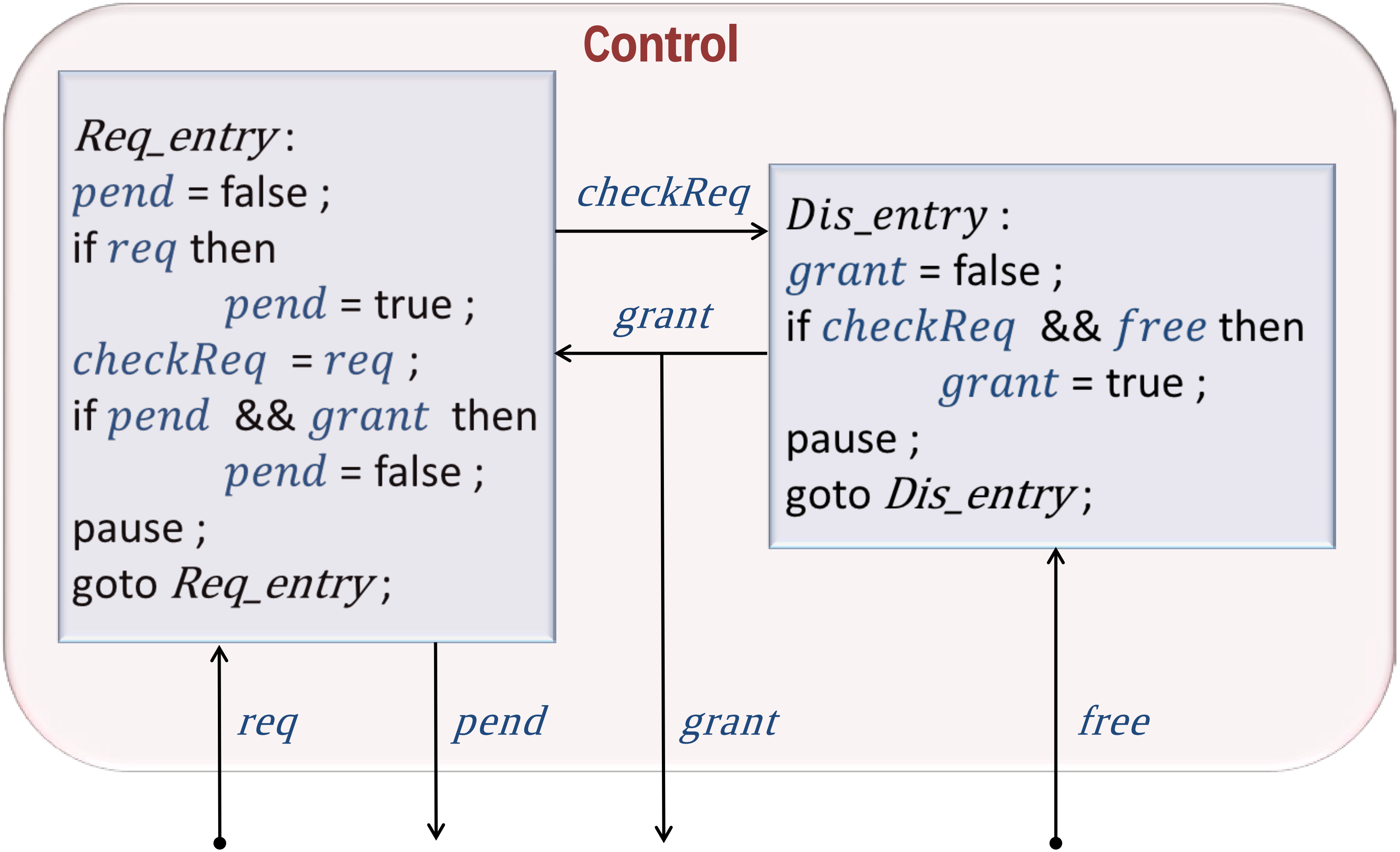
Outline

1. **Example**
2. Threads and Concurrency
3. Sequential Constructiveness (SC)
4. Analysing SC
5. Notions of Constructiveness

A Sequentially Constructive Program



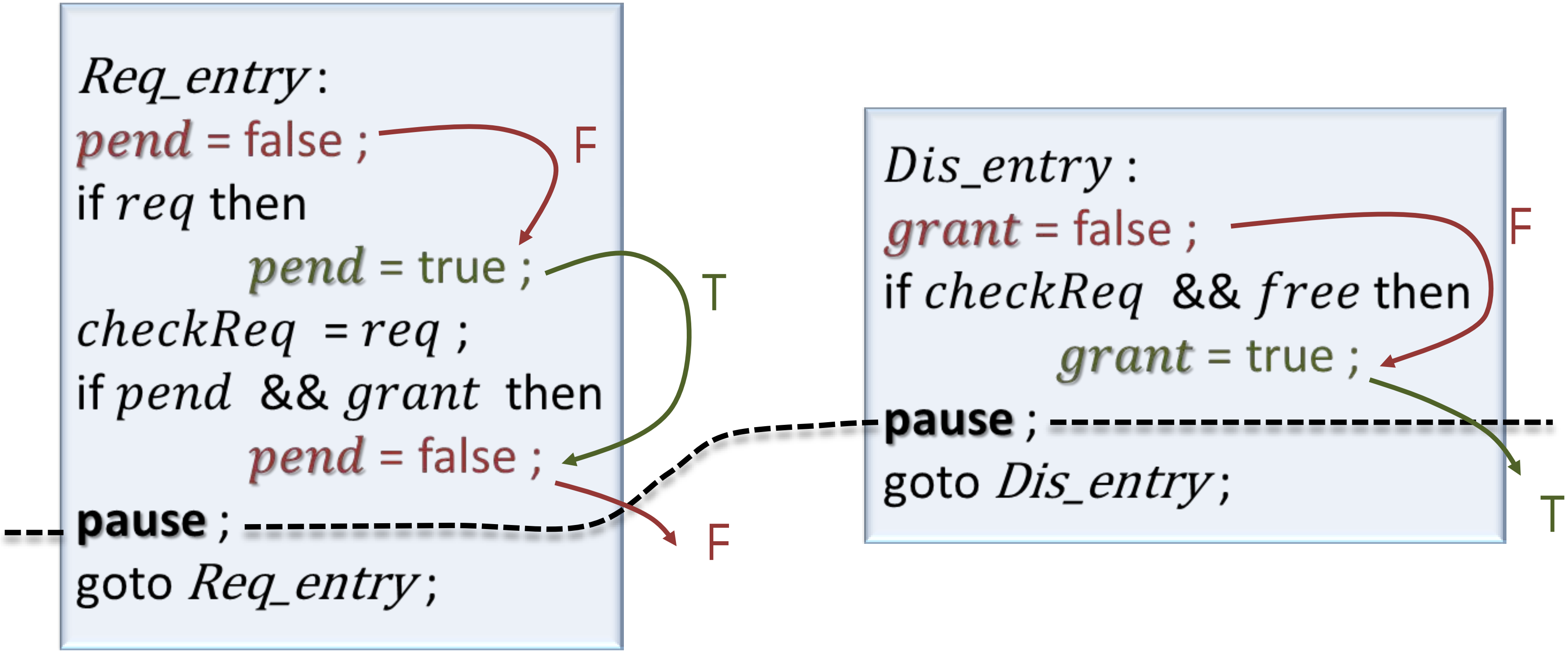
A Sequentially Constructive Program



```
Req_entry:
pend = false ;
if req then
        pend = true ;
checkReq = req ;
if pend && grant then
        pend = false ;
pause ;
goto Req_entry;
```

```
Dis_entry :
grant = false ;
if checkReq && free then
        grant = true ;
pause ;
goto Dis_entry;
```

A Sequentially Constructive Program



```
Req_entry:
pend = false ;
if req then
        pend = true ;
checkReq = req ;
if pend && grant then
        pend = false ;
pause ;
goto Req_entry;
```

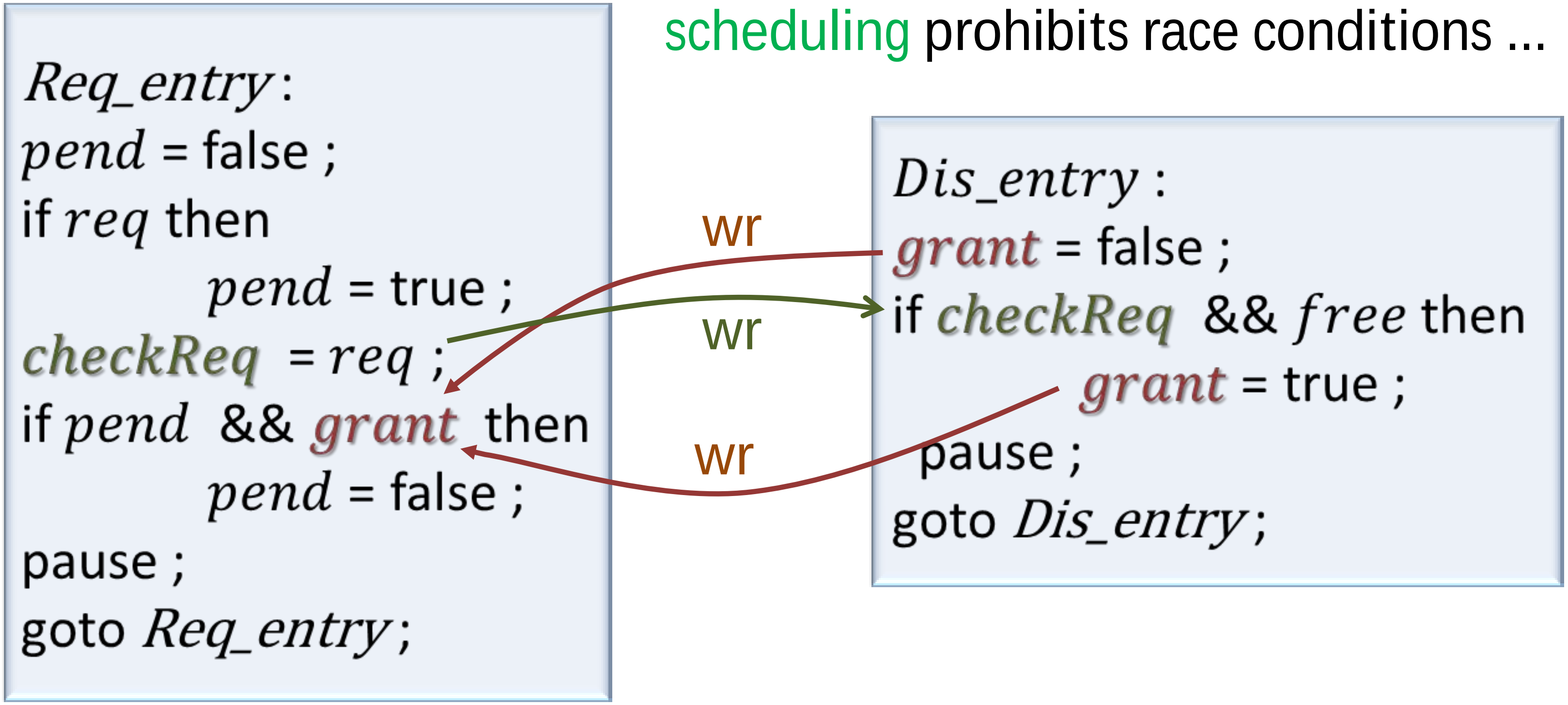
```
Dis_entry :
grant = false ;
if checkReq && free then
        grant = true ;
pause ;
goto Dis_entry;
```

Imperative Program Order (Sequential access to share variables):

- „write-after-write“ can change value sequentially (multi-writer)
- fully deterministic at thread level
- but not permitted in standard synchronous MoC

A Sequentially Constructive Program

SC MoC: Intra-instant (micro-step) thread scheduling prohibits race conditions ...



```
Req_entry:
pend = false ;
if req then
        pend = true ;
checkReq = req ;
if pend && grant then
        pend = false ;
pause ;
goto Req_entry;
```

```
Dis_entry :
grant = false ;
if checkReq && free then
        grant = true ;
 pause ;
goto Dis_entry;
```

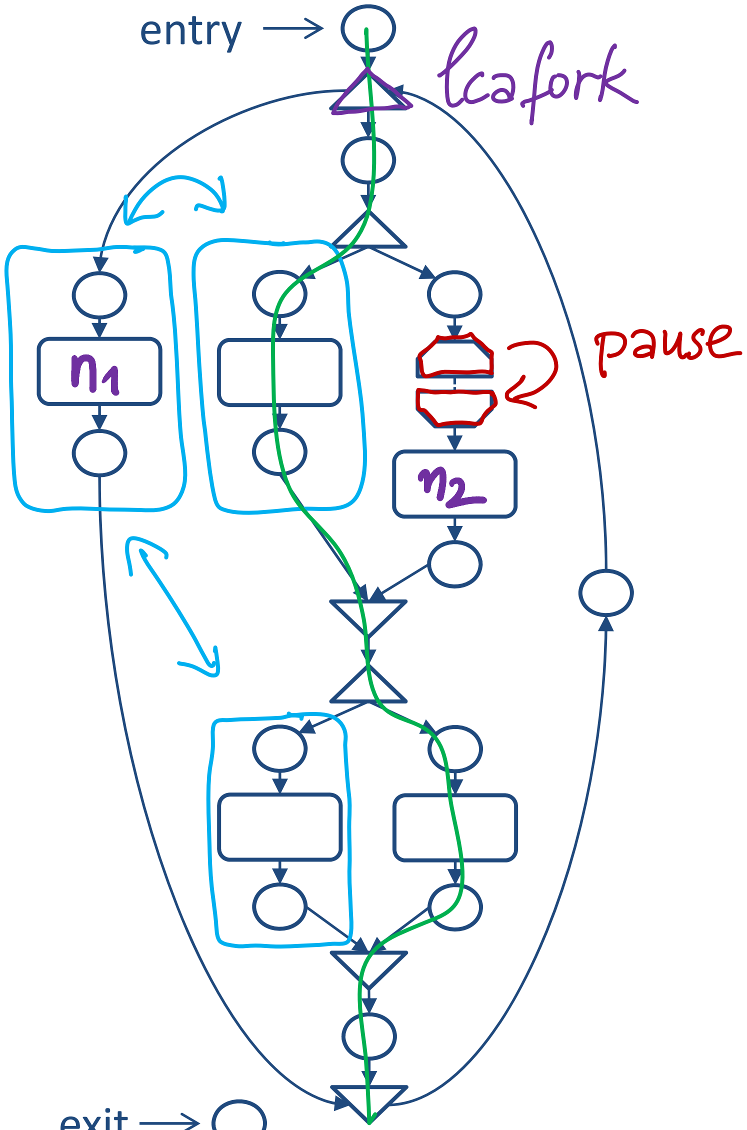
Concurrency Scheduling Constraints (access to shared variables):

- “write-before-read” for concurrent write/reads
- „write-before-write“ for concurrent & conflicting writes (see later)

Outline

1. Example
2. **Threads and Concurrency**
3. Sequential Constructiveness (SC)
4. Analysing SC
5. Notions of Constructiveness

Sequential-Concurrent Program Graph (SCG)



prescribes the static topology of the computation:

sequential edges $\longrightarrow_{seq}$

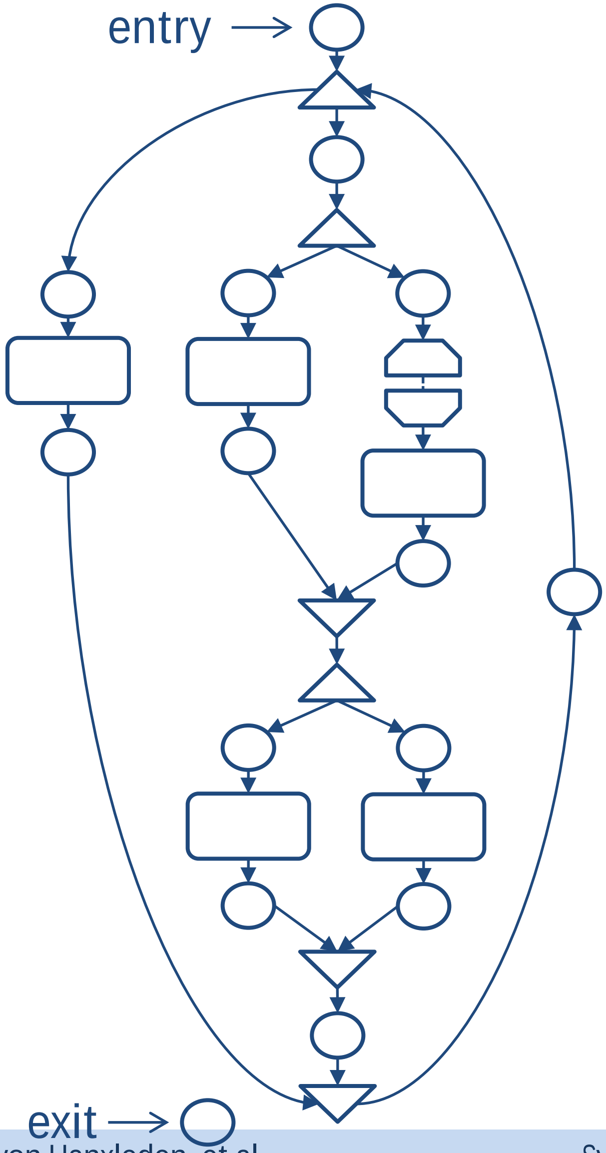
tick edges $\longrightarrow_{tick}$

concurrent nodes $\longleftrightarrow$ $\parallel$

least common ancestor fork

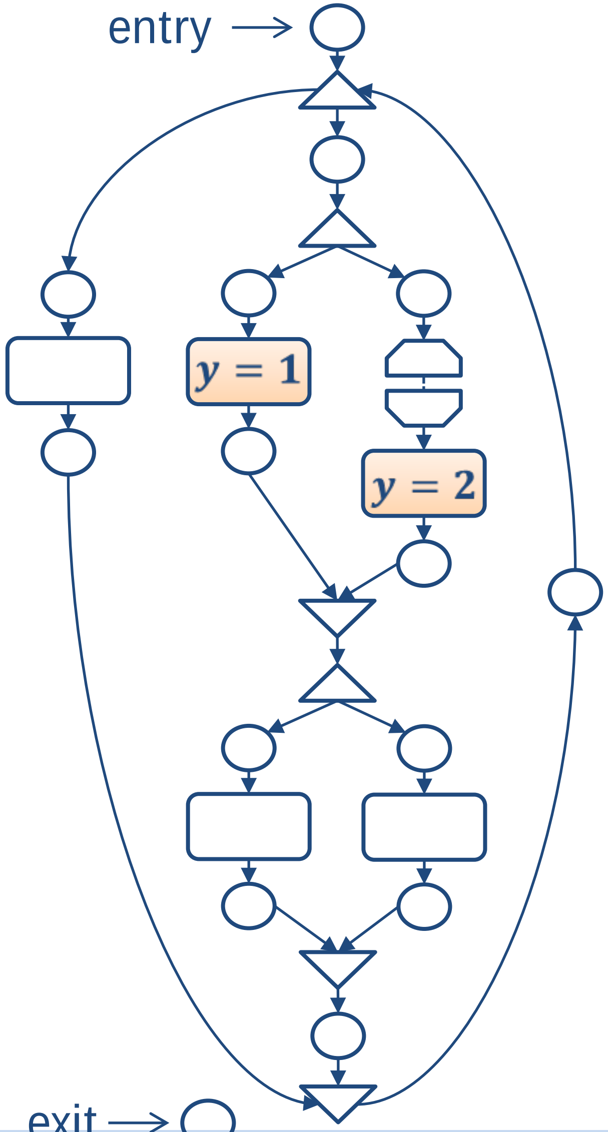
$lcafork(n_1, n_2)$

Intra-Instant Concurrency



Static thread concurrency is not sufficient to capture *run-time concurrency*!

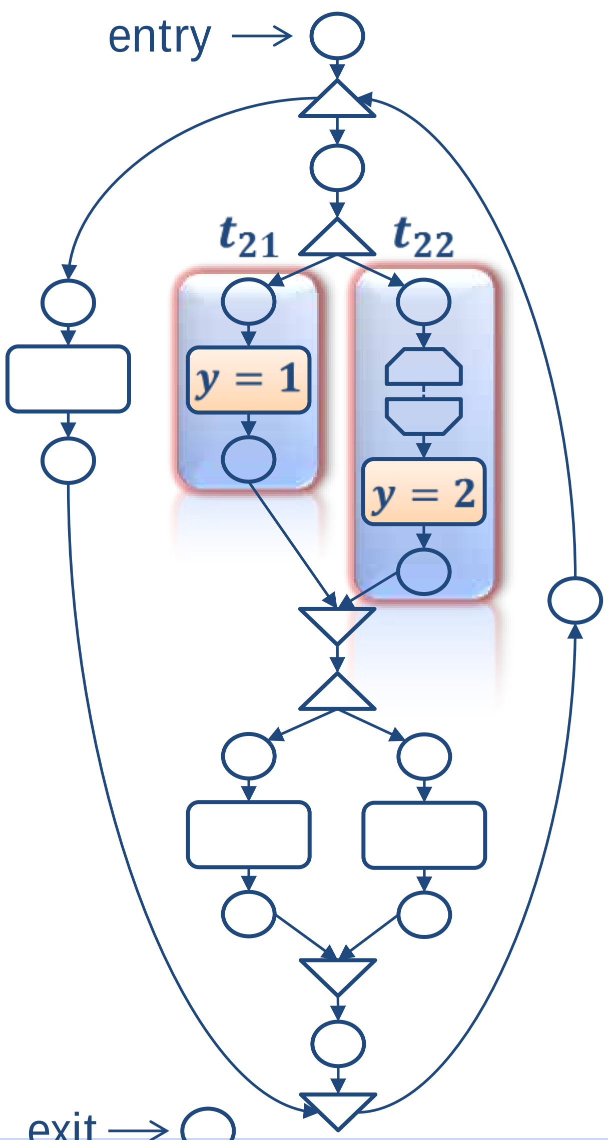
Intra-Instant Concurrency



Static thread concurrency is not sufficient to capture ***run-time concurrency***!

Consider the assignments $y = 1$ and $y = 2$ in the SCG.

Intra-Instant Concurrency

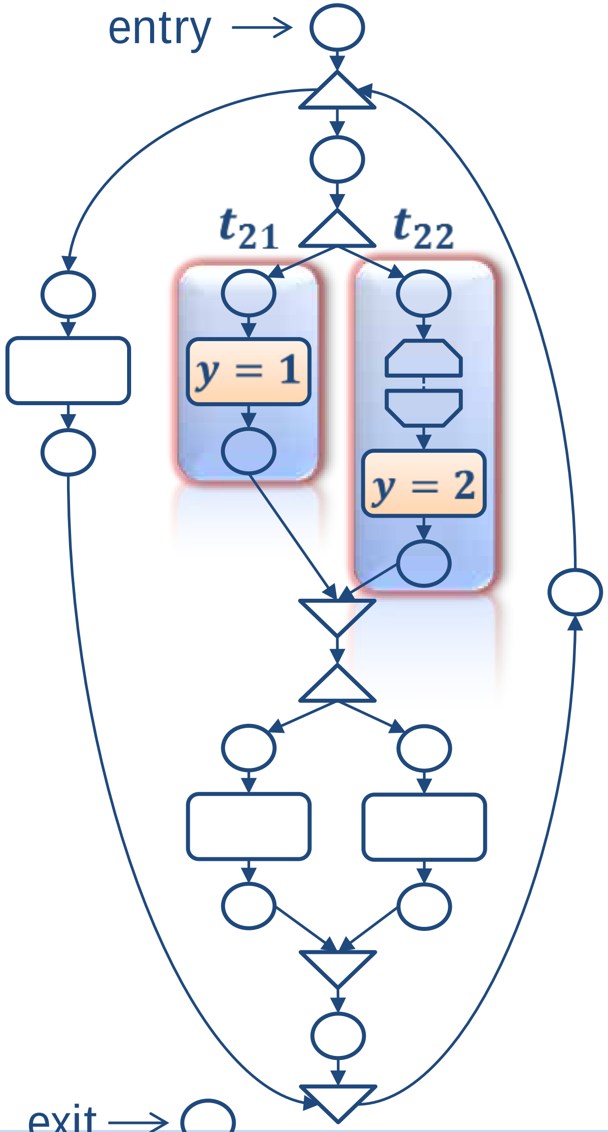


Static thread concurrency is not sufficient to capture ***run-time concurrency***!

Consider the assignments $y = 1$ and $y = 2$ in the SCG.

These are in threads t_{21} and t_{22}, and can be activated in the same tick.

Intra-Instant Concurrency



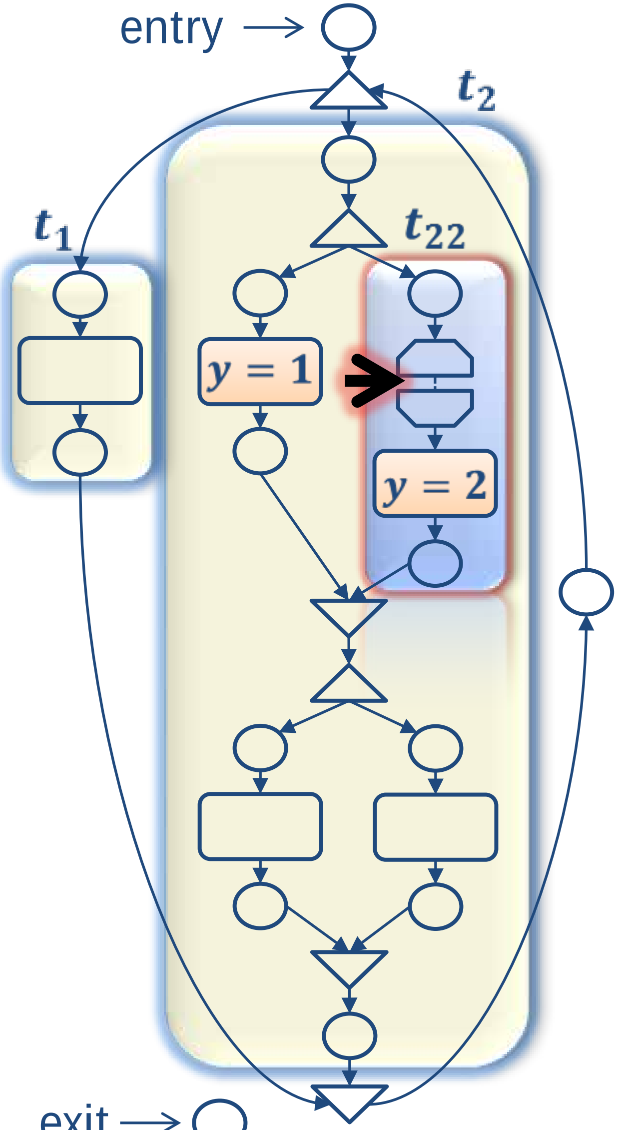
Static thread concurrency is not sufficient to capture ***run-time concurrency***!

Consider the assignments $y = 1$ and $y = 2$ in the SCG.

These are in threads t_{21} and t_{22}, and can be activated in the same tick.

But they are still **sequentially ordered** and thus not run-time concurrent.

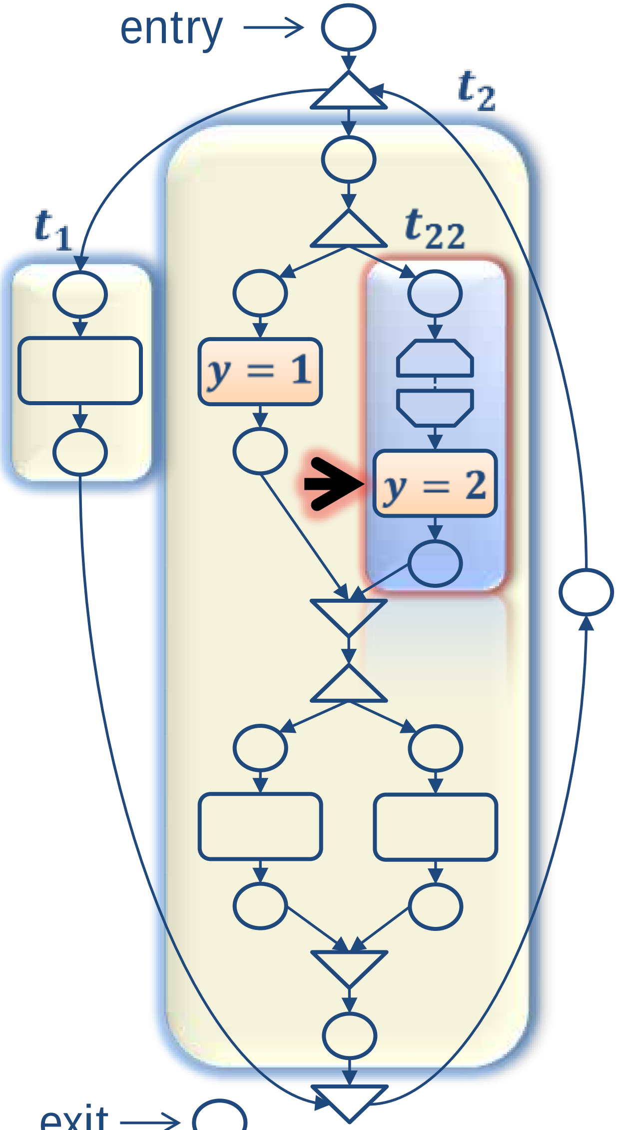
Intra-Instant Concurrency



Static thread concurrency is not sufficient to capture ***run-time concurrency***!

After the initial tick t_1 and t_2 have terminated, and control rest at the pause of t_{22}.

Intra-Instant Concurrency

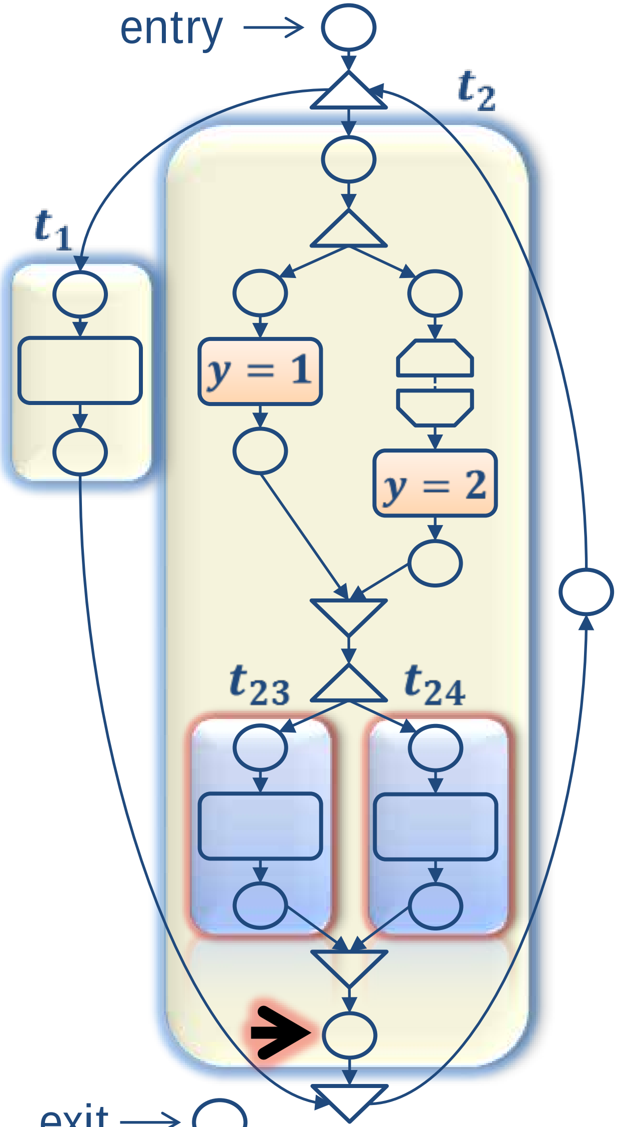


Static thread concurrency is not sufficient to capture *run-time concurrency*!

After the initial tick t_1 and t_2 have terminated, and control rest at the pause of t_{22}.

In the next instant, $y = 2$ gets executed and t_{22} terminates.

Intra-Instant Concurrency



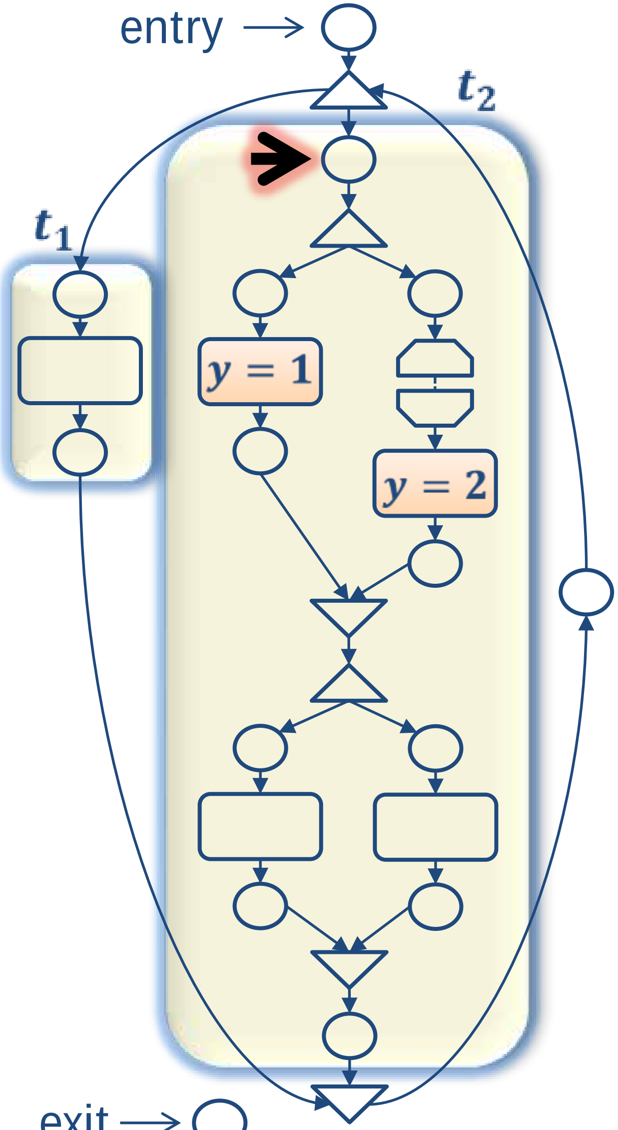
Static thread concurrency is not sufficient to capture ***run-time concurrency***!

After the initial tick t_1 and t_2 have terminated, and control rest at the pause of t_{22}.

In the next instant, $y = 2$ gets executed and t_{22} terminates.

Also t_{23} and t_{24} are executed; at the end, t_2 terminates.

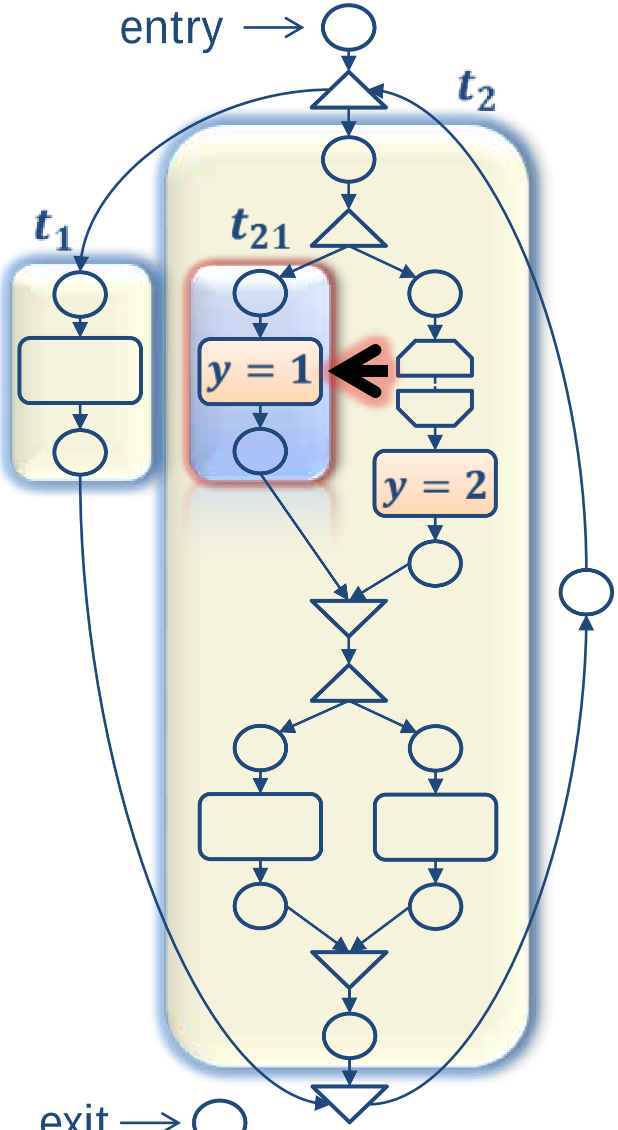
Intra-Instant Concurrency



Static thread concurrency is not sufficient to capture ***run-time concurrency***!

Then, after the loop, t_2 gets started again.

Intra-Instant Concurrency

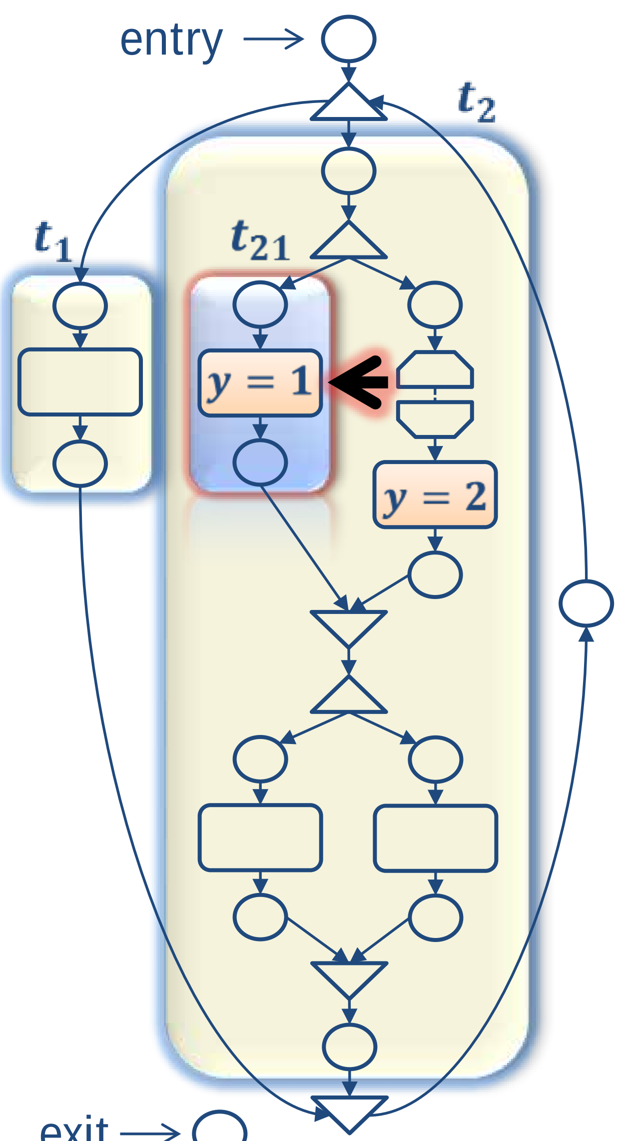


Static thread concurrency is not sufficient to capture *run-time concurrency*!

Then, after the loop, t_2 gets started again.

Finally, t_{21} gets to executed $y = 1$.

Intra-Instant Concurrency



Static thread concurrency is not sufficient to capture *run-time concurrency*!

Then, after the loop, t_2 gets started again.

Finally, t_{21} gets to executed $y = 1$.

The fact that $y = 1$ and $y = 2$ are not run-time concurrent is because their executions go back to different instances of t_{21}.

Intra-Instant Concurrency

Definition: Two node instances $ni_1 = (n_1, i_1)$ and $ni_2 = (n_2, i_2)$ are *concurrent* in a macro tick R, denoted $ni_1 \mid_R ni_2$, iff

- they appear in the micro ticks of R
- they belong to statically concurrent threads
- their threads have been instantiated by the same instance of the associated least common ancestor fork.

$$last(n, i_1) = last(n, i_2)$$
$$n = lcafork(n_1, n_2)$$

Outline

1. Example
2. Threads and Concurrency
3. **Sequential Constructiveness (SC)**
4. Analysing SC
5. Notions of Constructiveness

Sequential Admissibility

Remember

Sequentially ordered variable accesses

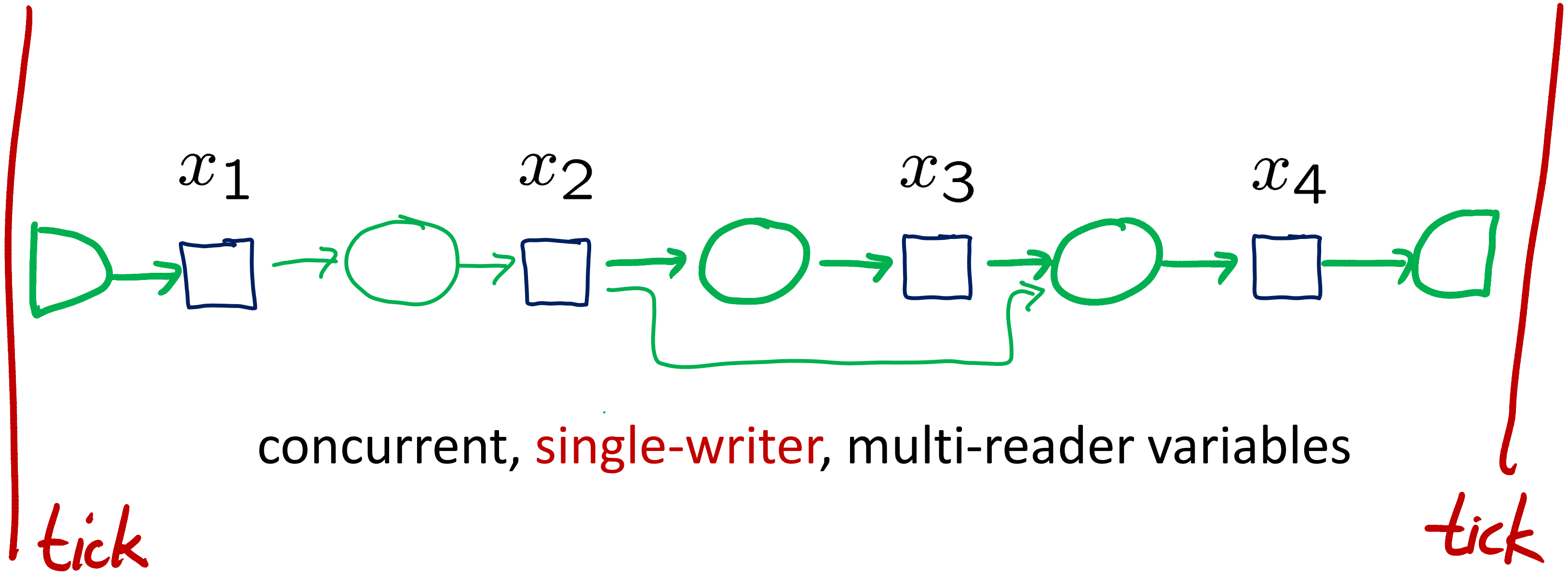
- exhibit *no races*
- *cannot be reordered* by the compiler

Only ***concurrent writes*** to the same variable

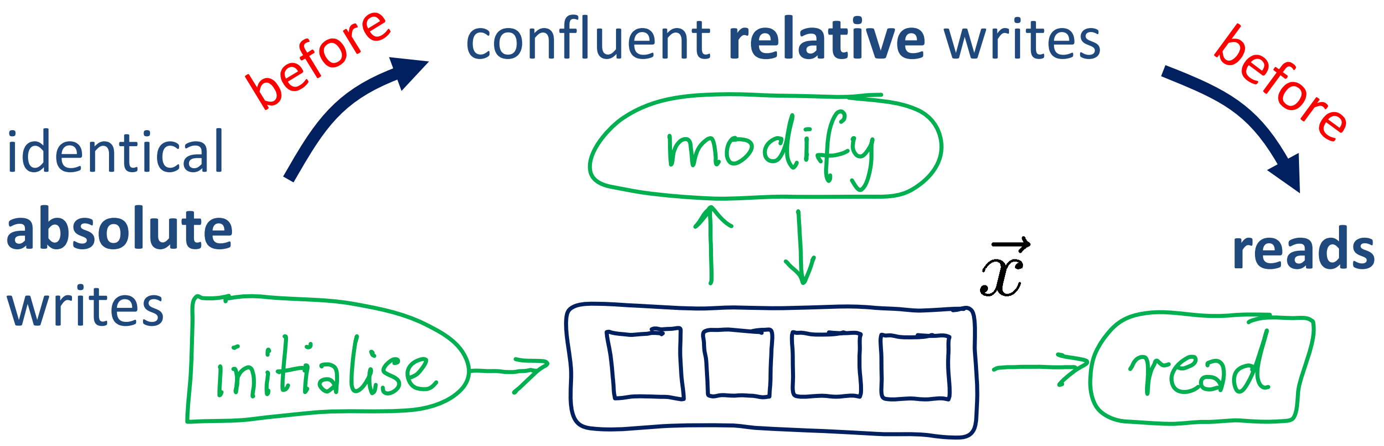
- generate potential data *data races*
- *must be resolved* by the compiler
- can be ordered under multi-threading

The following applies to concurrent variable accesses only ...

Organising Concurrent Variable Accesses



concurrent, single-writer, multi-reader variables



identical **absolute** writes

before

confluent **relative** writes

before

reads

Types of Writes

Given two writes to x, distinguish

- *Confluent writes*, where the order of the writes does not matter
  - This implies that there are no side effects
- *Non-confluent writes*, where the order of the writes matters

Given one write to x, distinguish

- *Absolute writes ("initialisation")*
  - x = e
  - Expression e does not constitute relative write (see below)
  - Eg, x = 0, x = 2*y + 5, x = f(z)
- *Relative writes ("increments")*
  - x = $f(x, e)$
  - *Combination function* f such that $f(f(x, e_1), e_2) = f(f(x, e_2), e_1)$
  - Hence schedules "x = $f(x, e_1)$; x = $f(x, e_2)$" and "x = $f(x, e_2)$; x = $f(x, e_1)$" yield same result for x – the writes are confluent
  - Sufficient condition: f is a commutative and associative
  - Eg, x++, x = 5*x, x = x – 10

Also distinguish

- *Effective writes*, which change value of x
- *Ineffective writes*, that do not change value of x

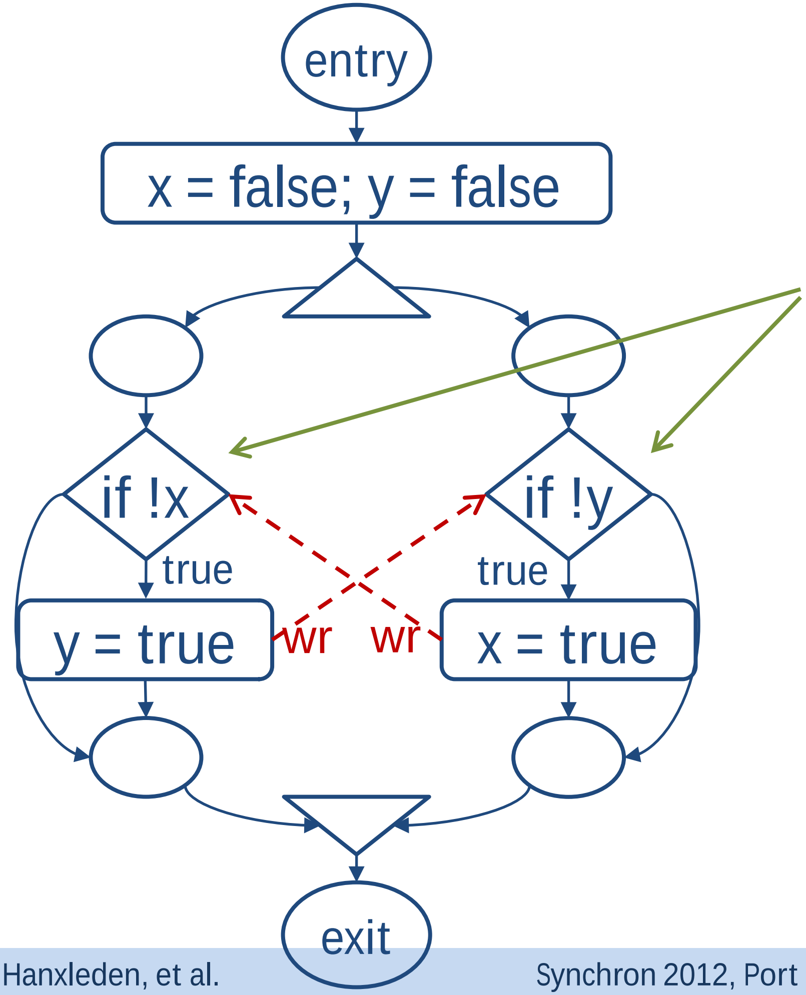
Sequential Admissibility

Definition: A *run* for a SCG $G = (N,E)$ is *S-admissible* if, for all ticks in this run, and for all concurrent node instances (n_1, i_1), (n_2, i_2), with $i_1 \cdot i_2$ and $n_1 |_R n_2$ **none** of the following occurs:

- n_1 and n_2 perform non-confluent writes on the same variable
- n_1 reads a variable, on which
  n_2 then performs an effective write
- n_1 performs a relative write to a variable, on which
  n_2 then performs an absolute write.

Sequential Constructiveness

The existence of an S-admissible run does not guarantee by itself determinism!



This program has two S-admissible runs.

Depending on which conditional is scheduled first,

The resulting memory would be either:

[x=true, y=false]
or
[x=false, y=true]

Sequential Constructiveness

Definition:
A program is *sequentially constructive (SC)* if
for each initial configuration and input:

1. there exists an S-admissible run
2. every S-admissible run generates the same, determinate sequence of macro responses in bounded time.

Outline

1. Example
2. Threads and Concurrency
3. Sequential Constructiveness (SC)
4. **Analysing SC**
5. Notions of Constructiveness

Conservative Static Approximation

- Use a relation $n_1\, \mathbf{j}\, n_2$ to over-approximate $n_1\, \mathbf{j}_R\, n_2$, i.e., what statements are concurrently invoked in the same tick,

  - by considering only static control flow, or
  - ignoring dependency on initial conditions, or
  - by falsely considering nodes to be in the same tick.

- Over-approximate what writes are
  - relative and confluent
  - absolute and confluent

  by not evaluating expressions (combination function).

Analysing Sequential Constructiveness

In addition to $\rightarrow_{seq}$ and $|$ the following static node relations are introduced:

$n_1 \leftrightarrow_{ww} n_2$ iff $n_1 \mid n_2$ and there exists a variable on which n_1 and n_2 perform **non-confluent** writes (e.g., non-identical absolute writes or relative writes with different combination function).

$n_1 \rightarrow_{wr} n_2$ iff $n_1 \mid n_2$ and n_1 performs an absolute write to a variable that is read by n_2.

$n_1 \rightarrow_{wi} n_2$ iff $n_1 \mid n_2$ and n_1 performs an absolute write to a variable on which n_2 performs a relative write.

Analysing Sequential Constructiveness

$n_1 \rightarrow_{ir} n_2$ iff $n_1 \mid n_2$ and n_1 performs an relative write to a variable that is read by n_2.

$n_1 \rightarrow_{wir} n_2$ iff $n_1 \rightarrow_{wr} n_2$ or $n_1 \rightarrow_{wi} n_2$ or $n_1 \rightarrow_{ir} n_2$. This contains the constraints induced by concurrent write/increment/read accesses.

$n_1 \rightarrow n_2$ iff $n_1 \rightarrow_{seq} n_2$ or $n_1 \rightarrow_{wir} n_2$ that is, if there is any control-flow or concurrent-access-induced ordering constraints.

Analysing Sequential Constructiveness

Definition: A program is ***acyclic SC (ASC) schedulable*** if in its SCG:

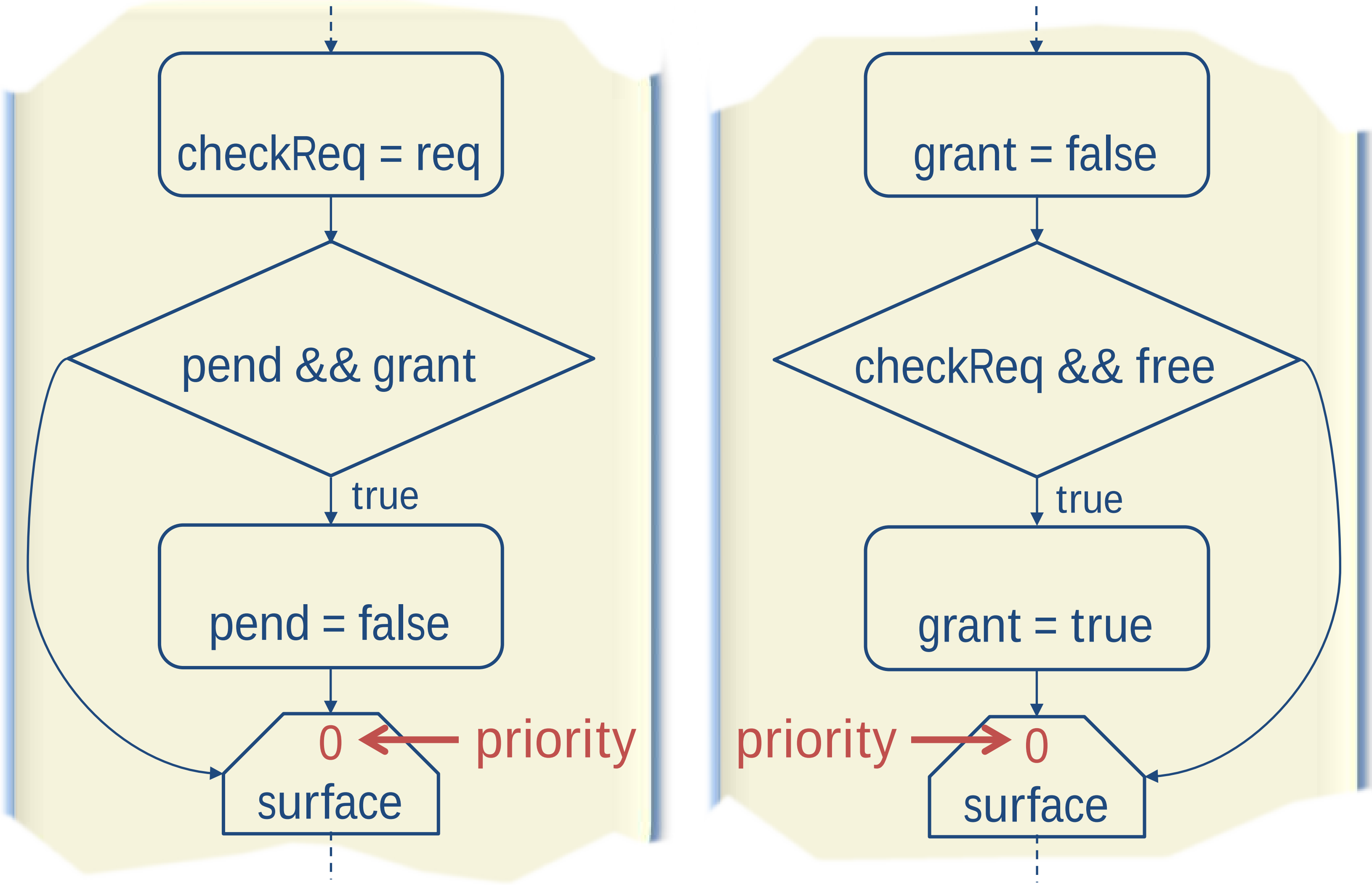
1. There are no statement nodes n_1, n_2 with $n_1 \leftrightarrow_{ww} n_2$
2. There is no $\rightarrow$ cycle that contains edges induced by $\rightarrow_{wir}$.

Lemma: Every ASC schedulable program is sequentially constructive.

For a ASC program, an *S-admissible schedule* is one which executes concurrent statements in the order induced by ! . Such schedule may be implemented by associating a priority with each statement node ...

Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule:



checkReq = req

pend && grant

true

pend = false

0 priority

surface

grant = false

checkReq && free

true

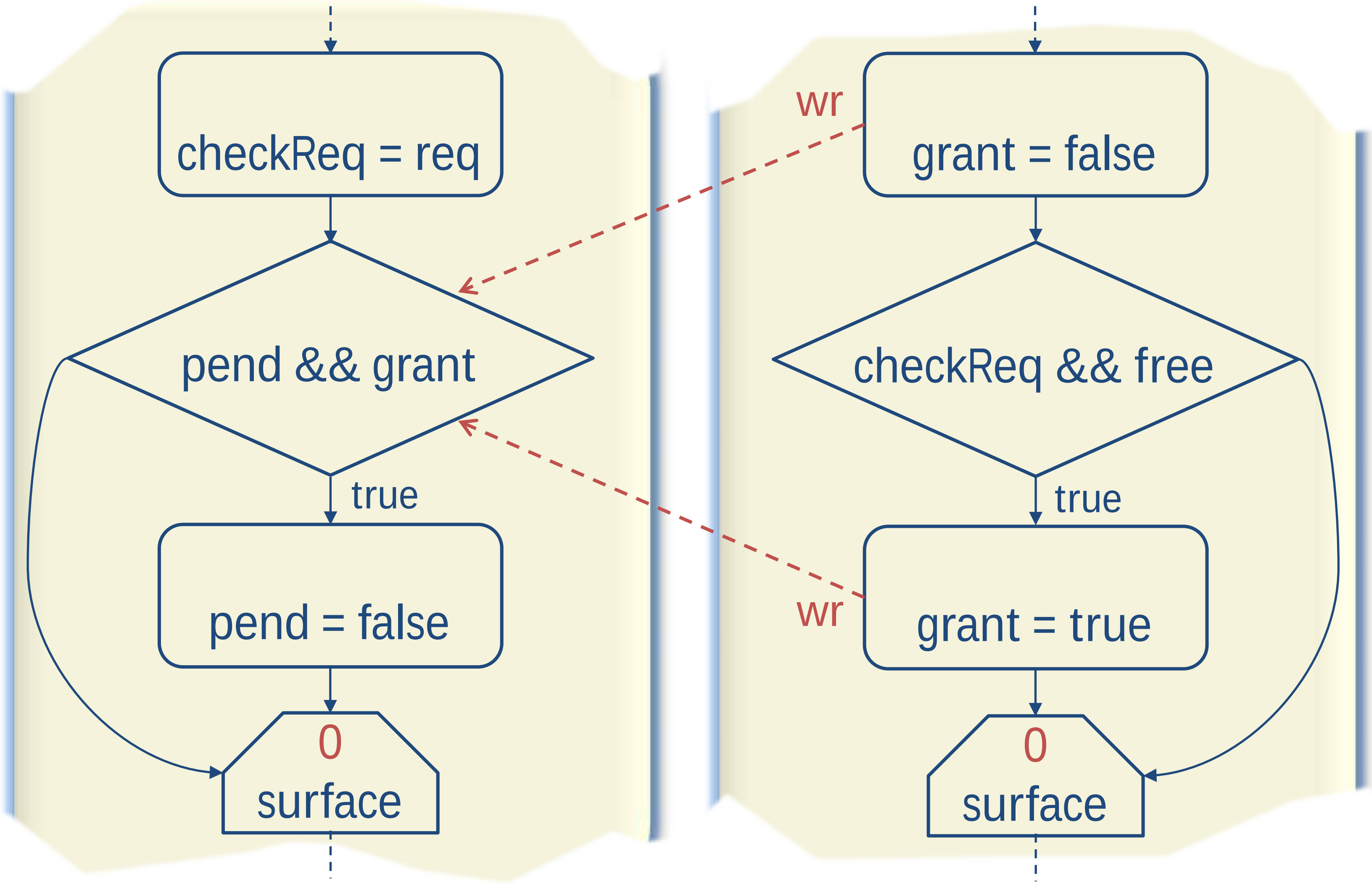
grant = true

priority 0

surface

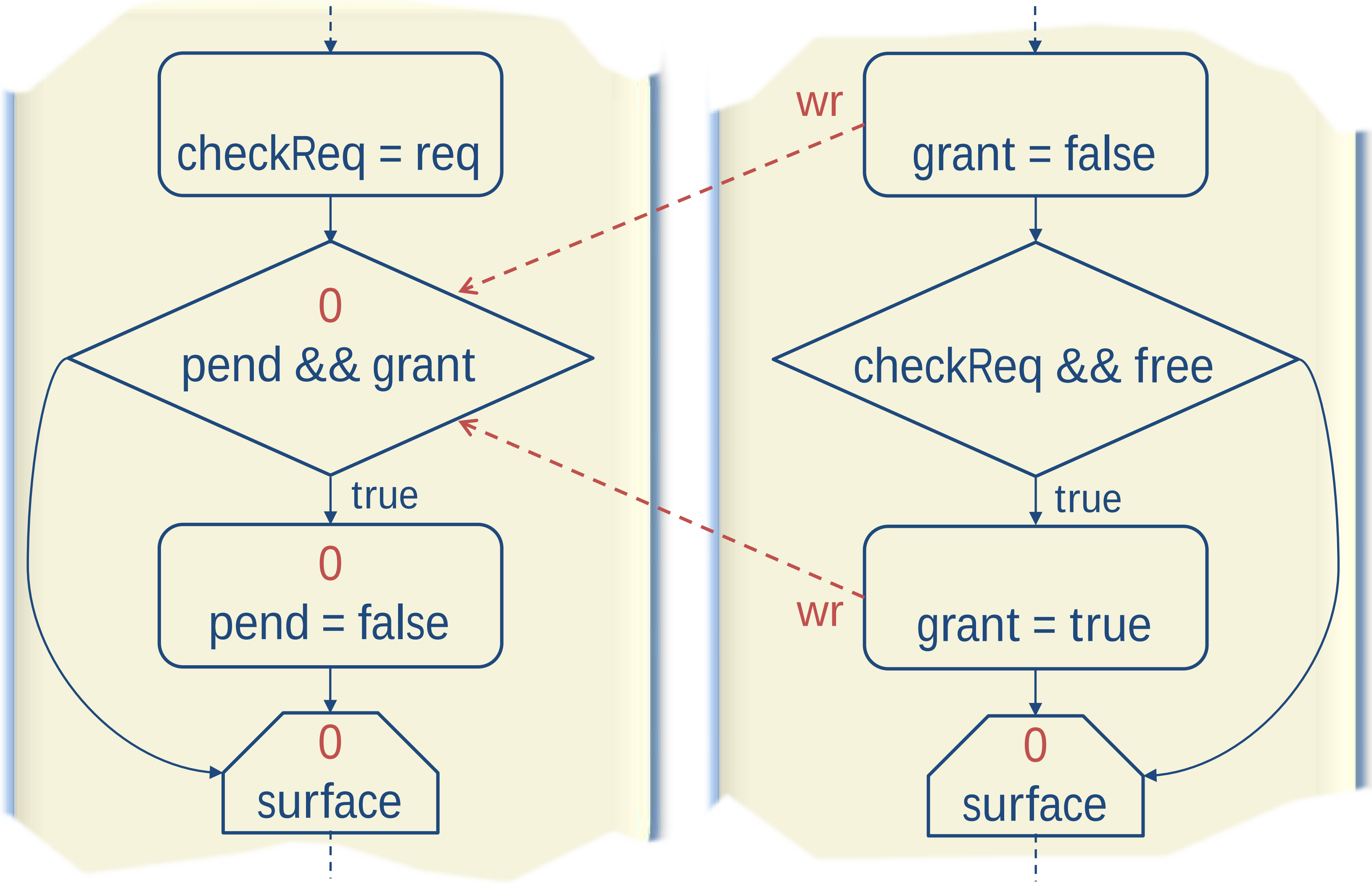
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule:



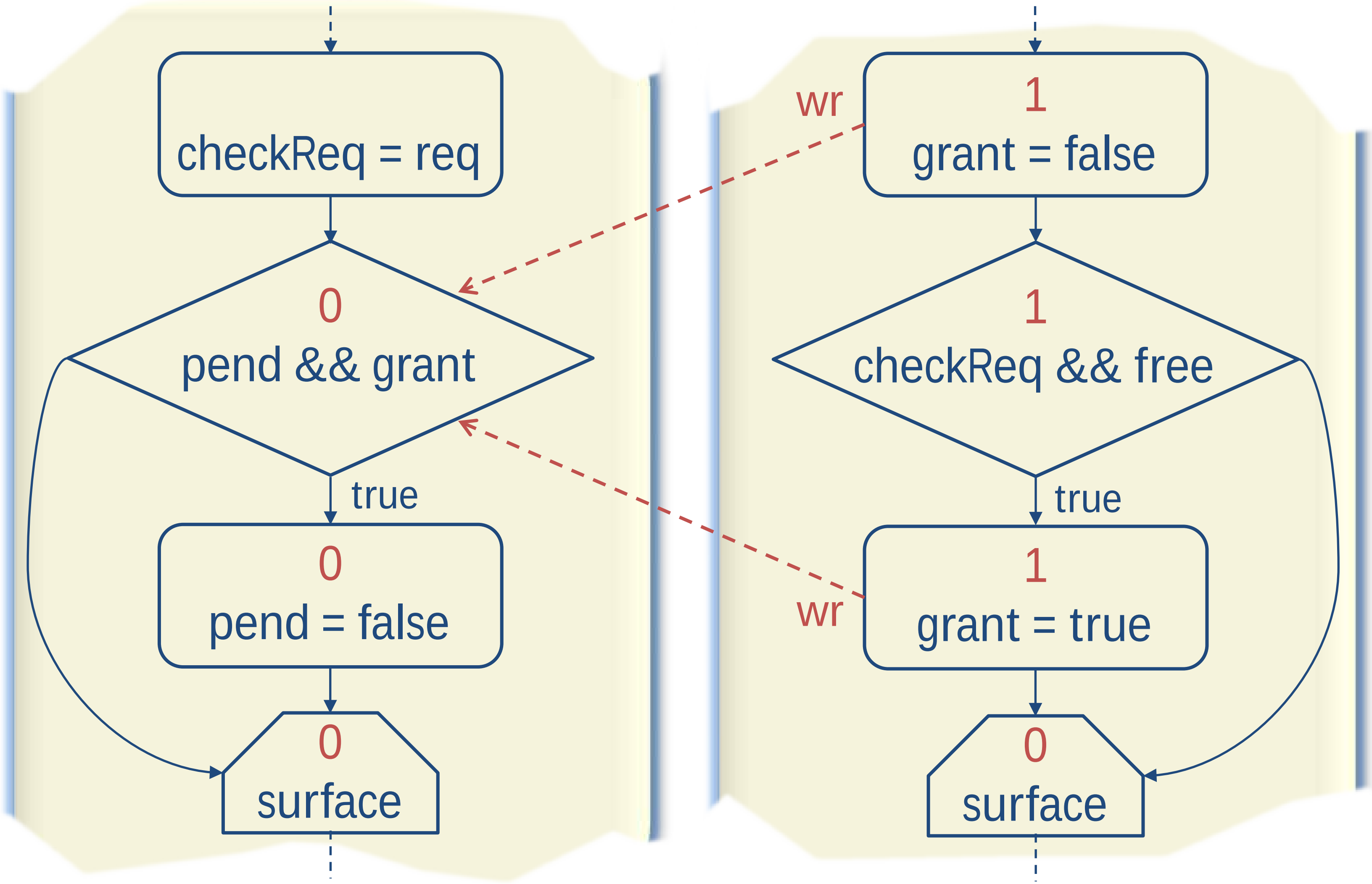
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule:



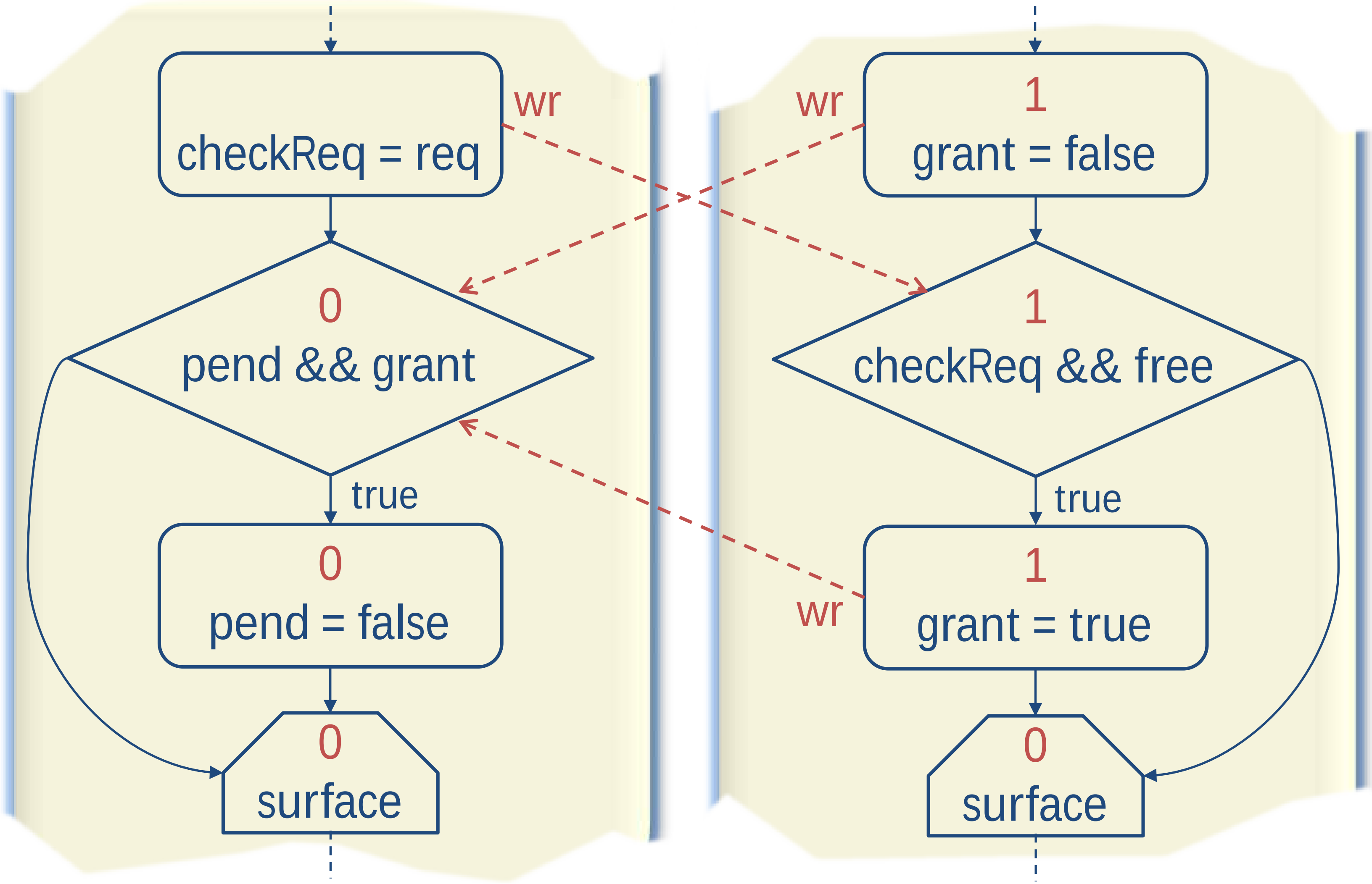
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule:



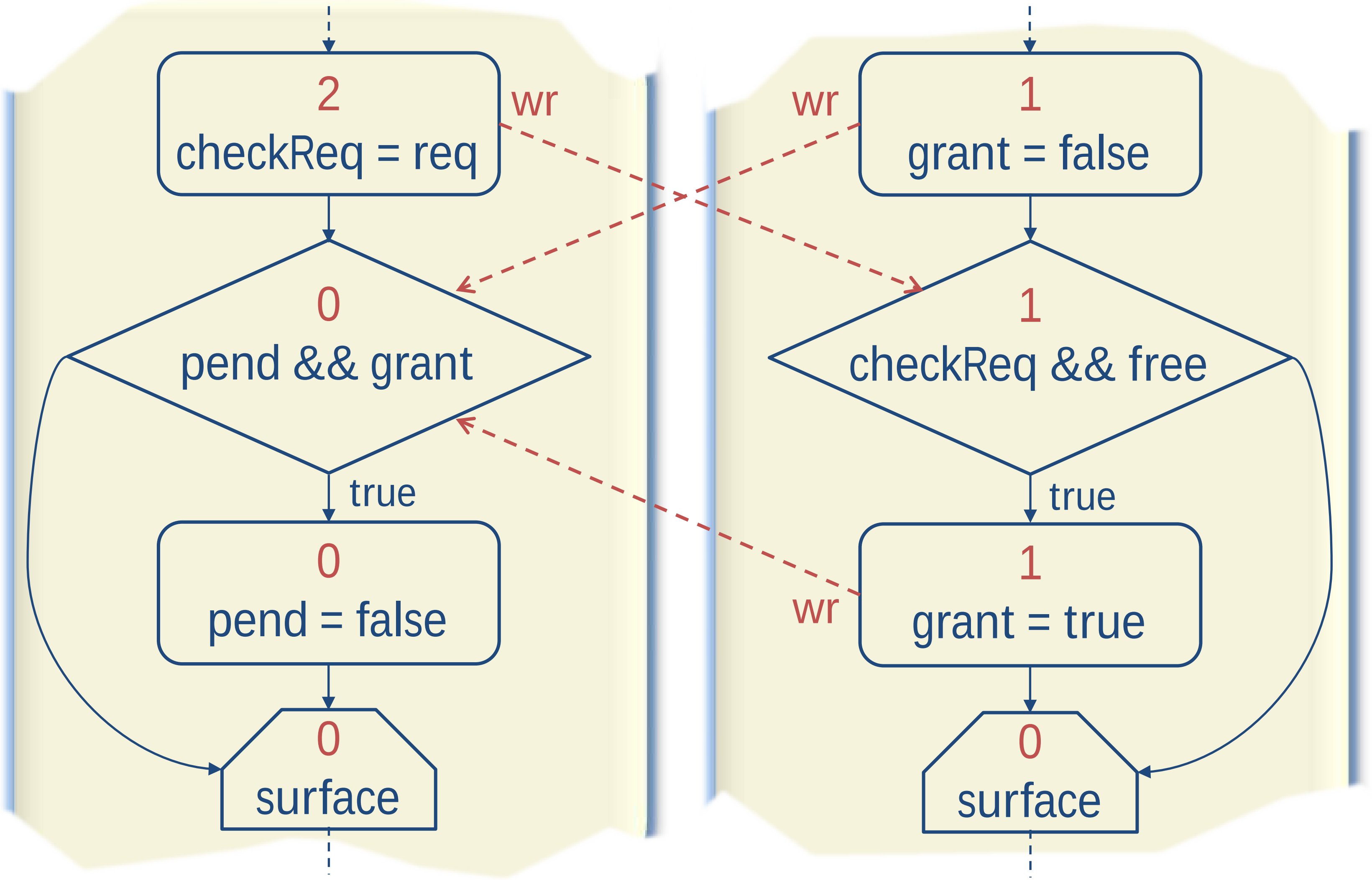
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule:



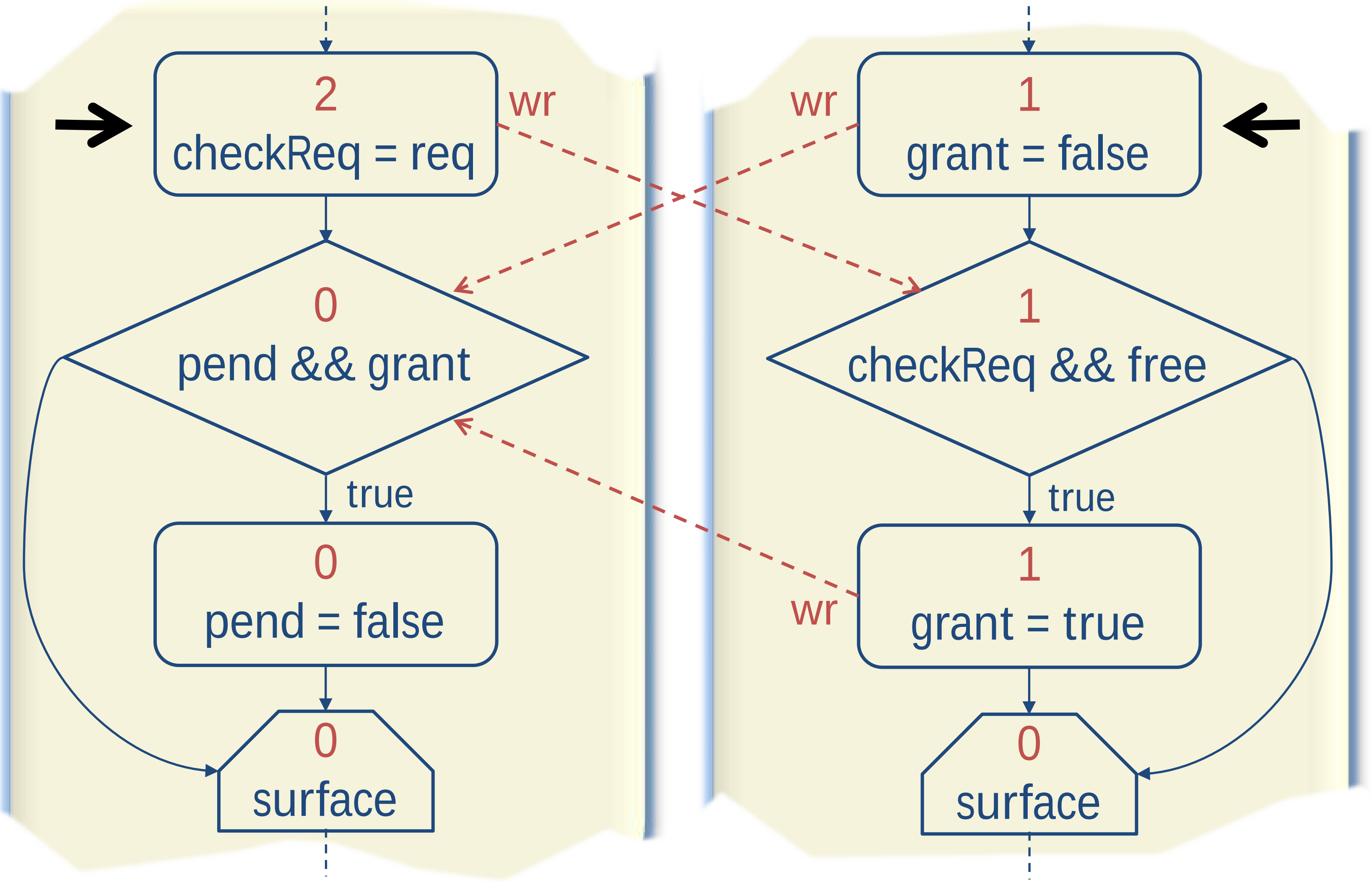
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule:



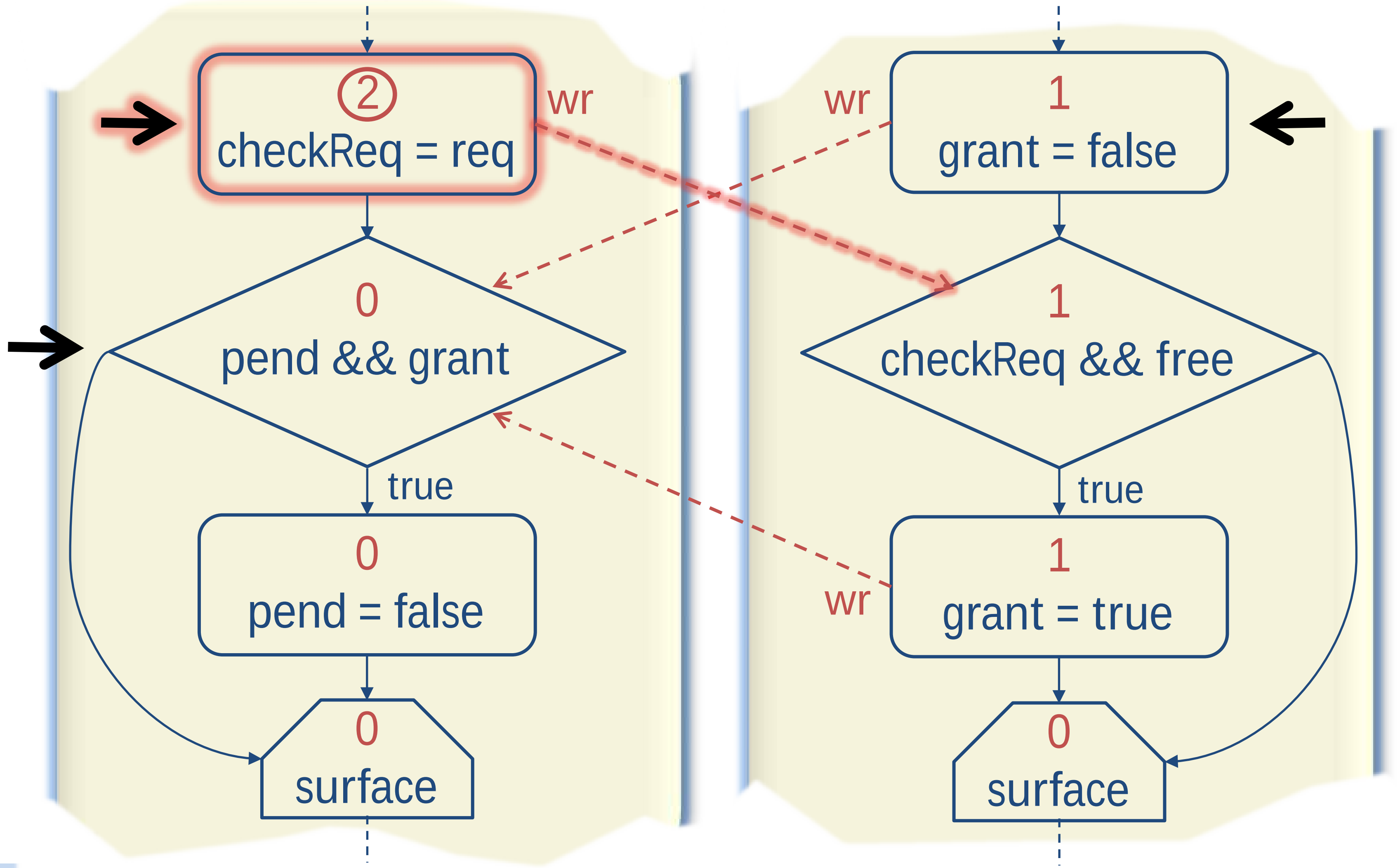
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: (**variables are true initially)**



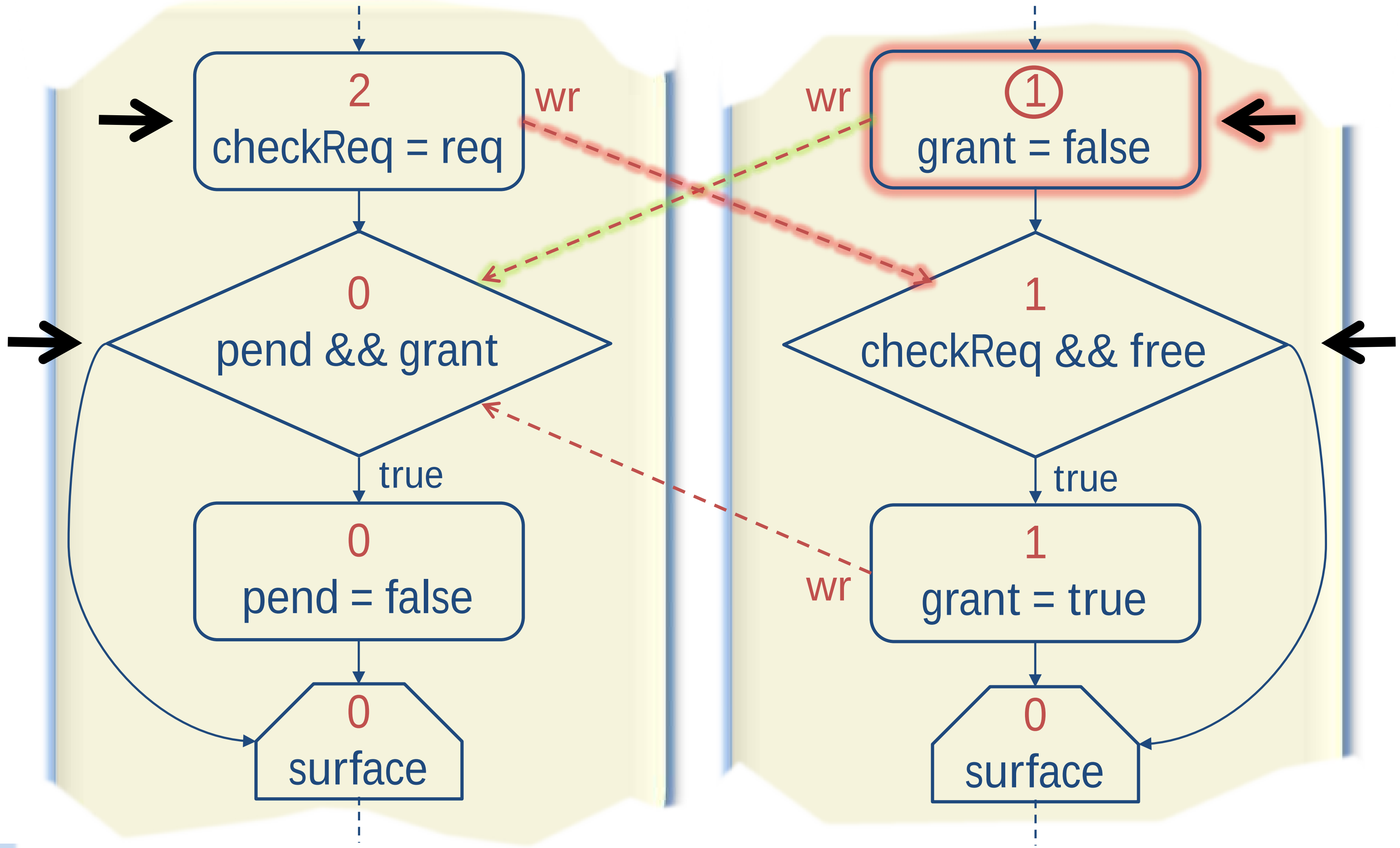
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: (**variables are true initially)**



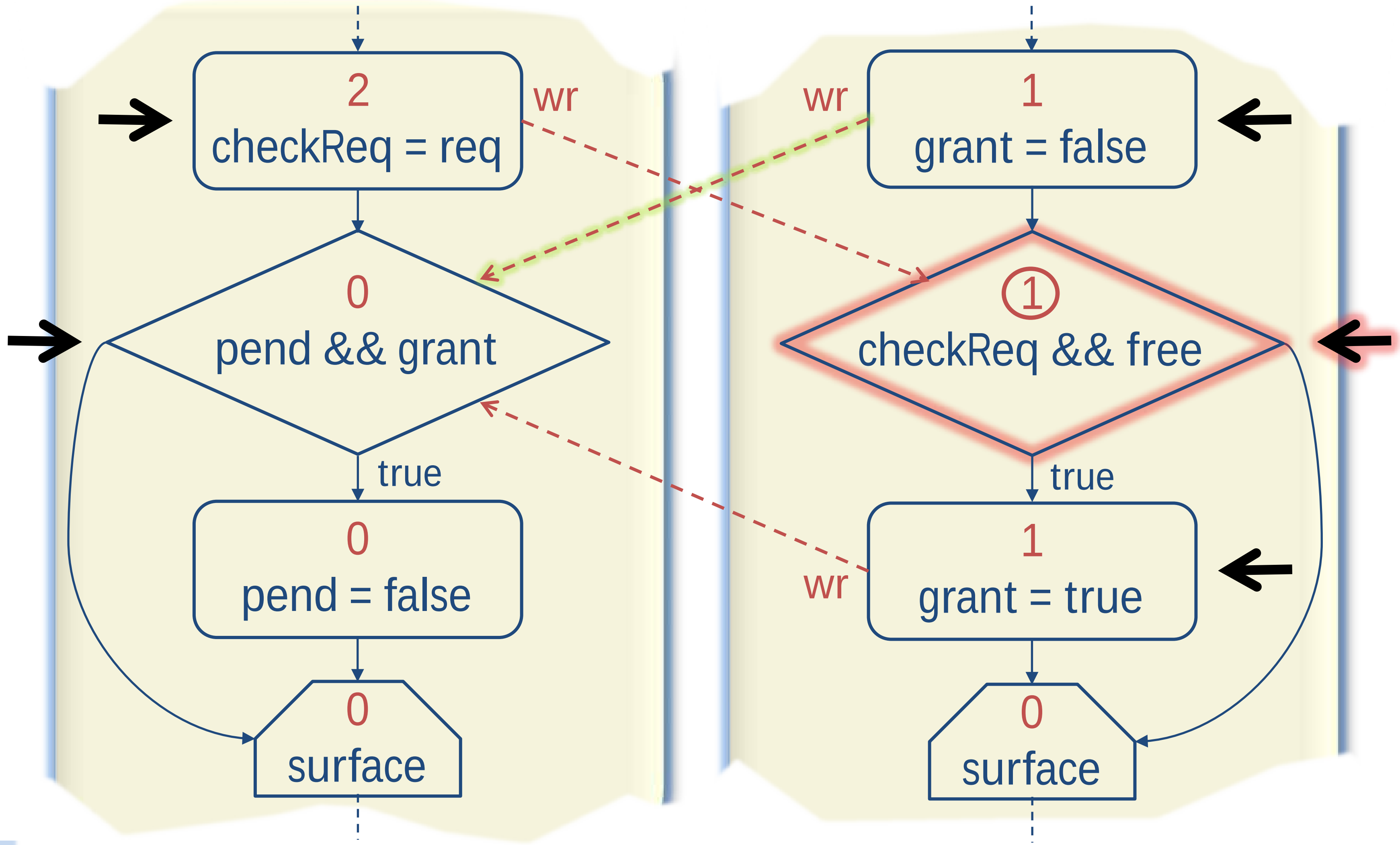
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: (**variables are true initially)**



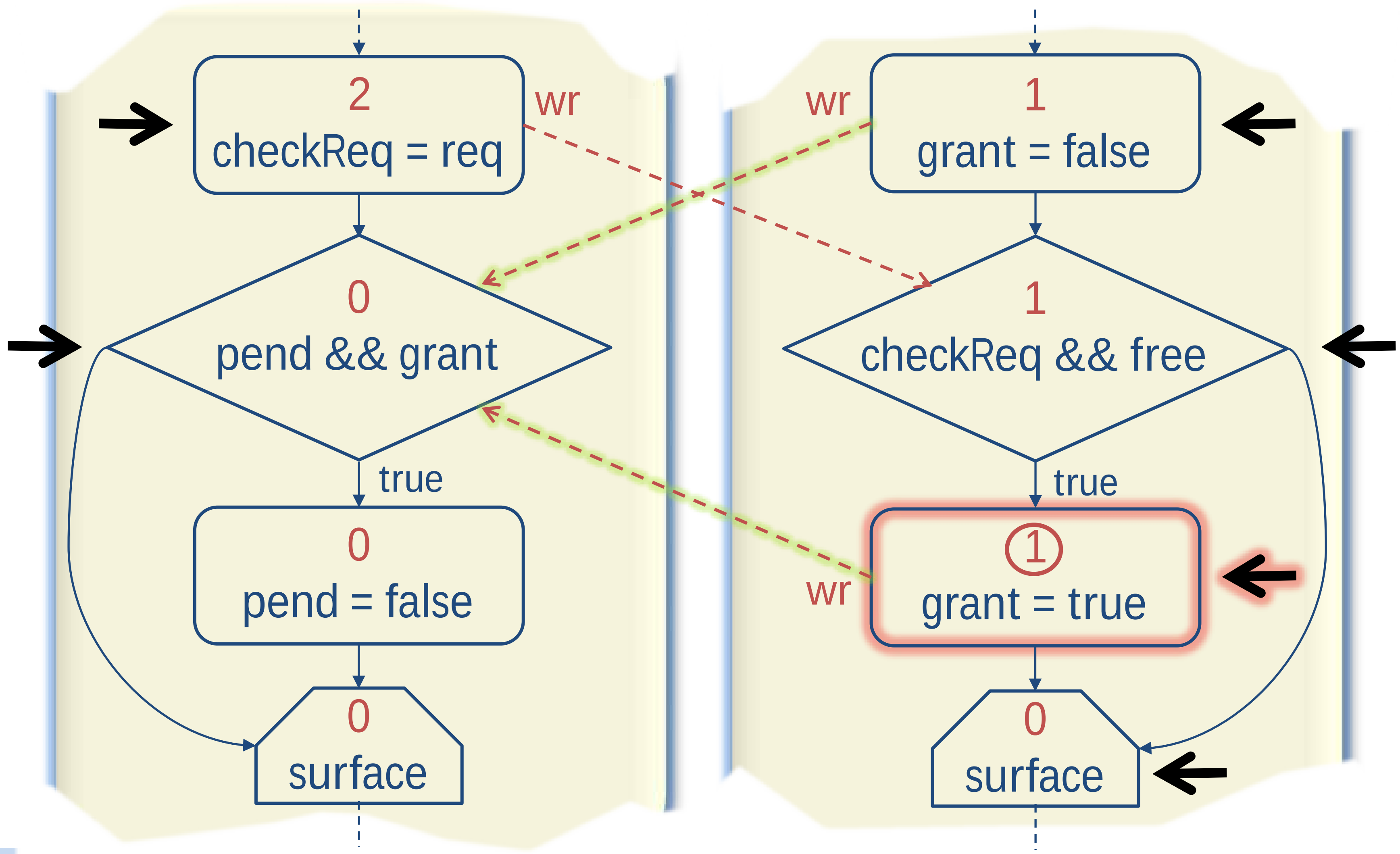
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: **(variables are true initially)**



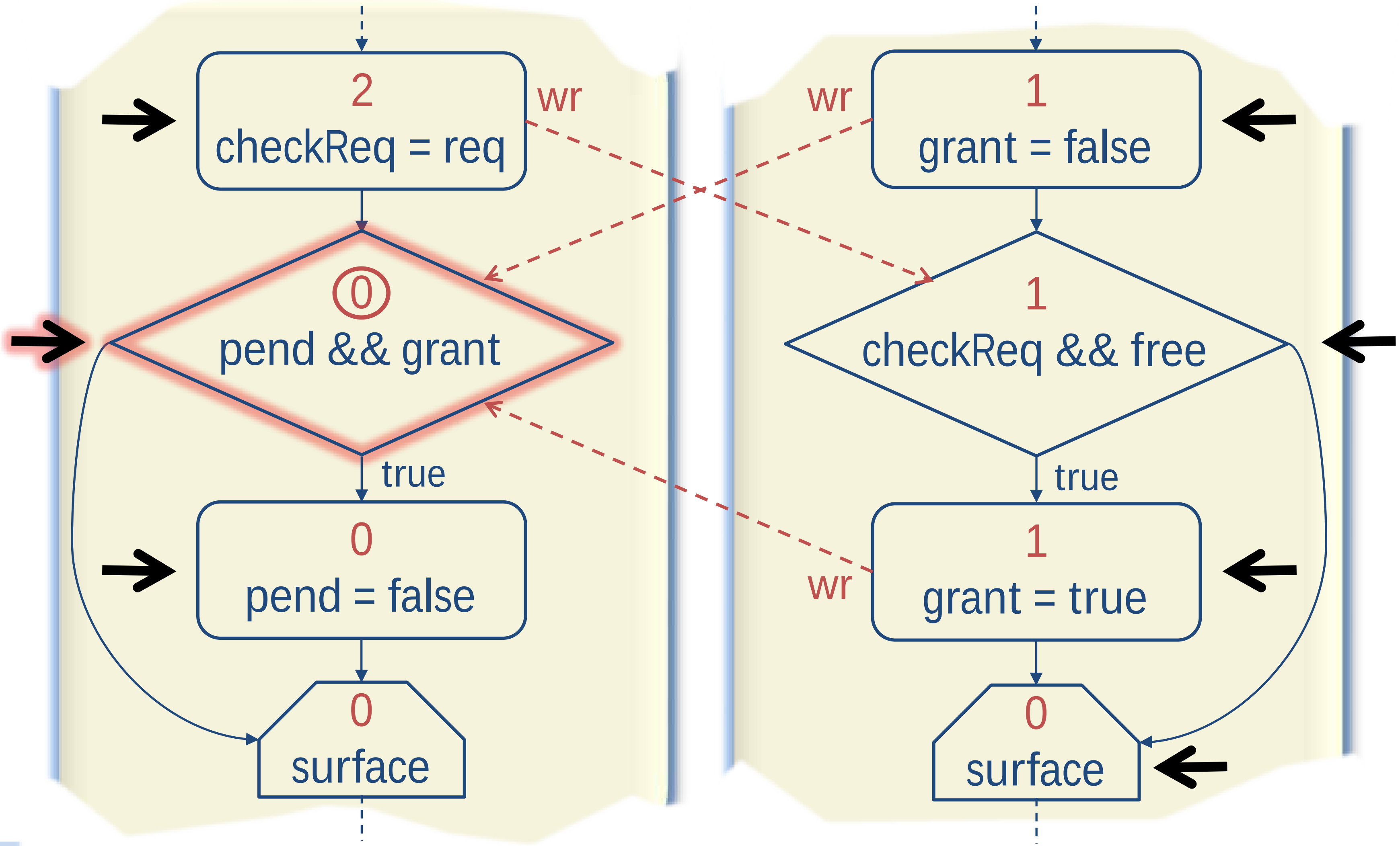
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: (**variables are true initially)**



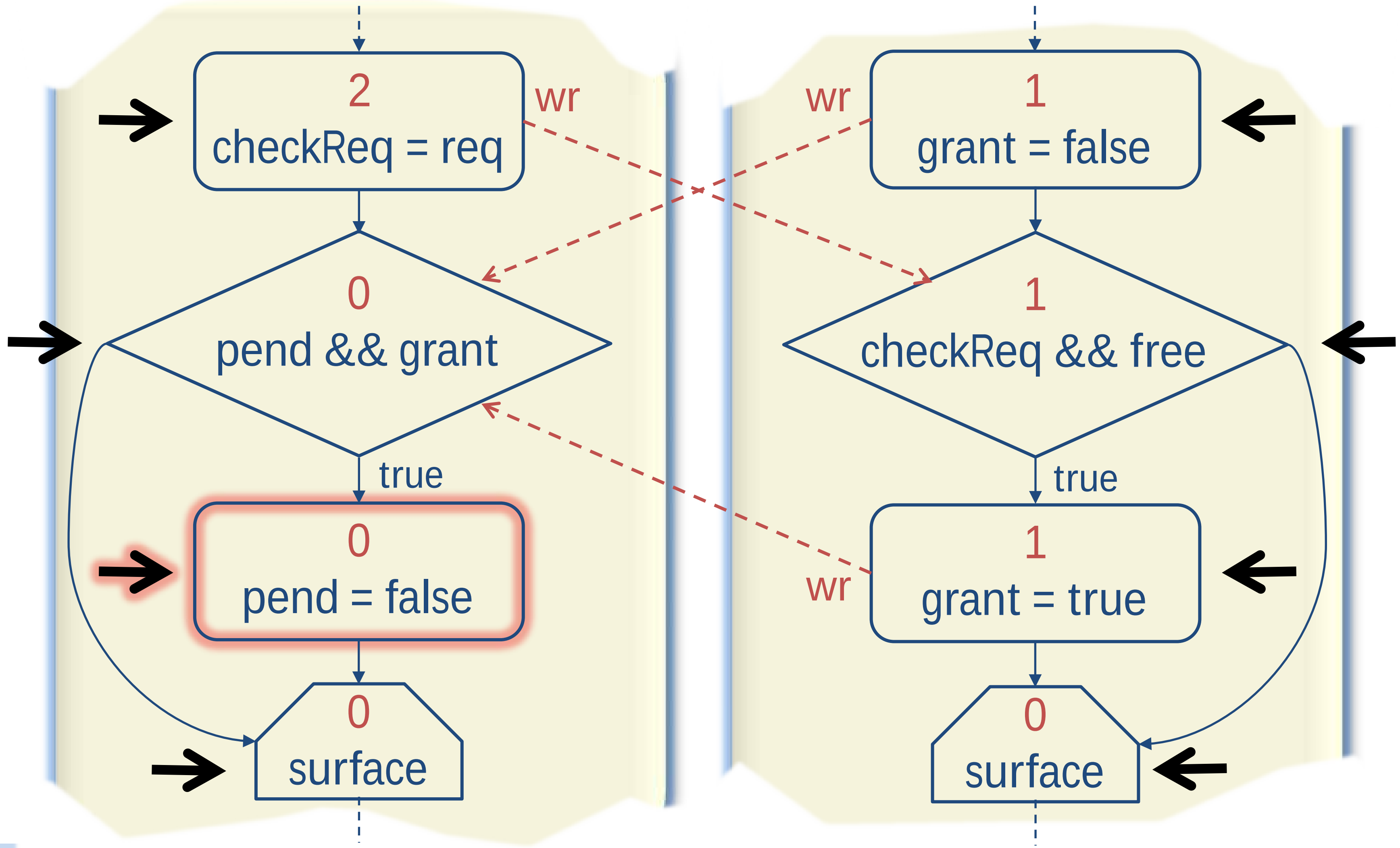
Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: (**variables are true initially)**



Analysing Sequential Constructiveness

Priorities and *S-admissible* schedule: (**variables are true initially)**



Analysing Sequential Constructiveness

Lemma: A program is *ASC schedulable* if in its SCG:

1. There are no statement nodes n_1, n_2 with $n_1 \leftrightarrow_{ww} n_2$.
2. All statement priorities are finite.

) *Longest Weighted Path Problem*

- NP hard in presence of non-zero weighted cycles
- However:
  - non-zero cycles indicate causality problem (reject)
  - ASC constructive programs have zero cycles
- factorises: (a) Strongly Connected Components,
  (b) Max Path in DAG

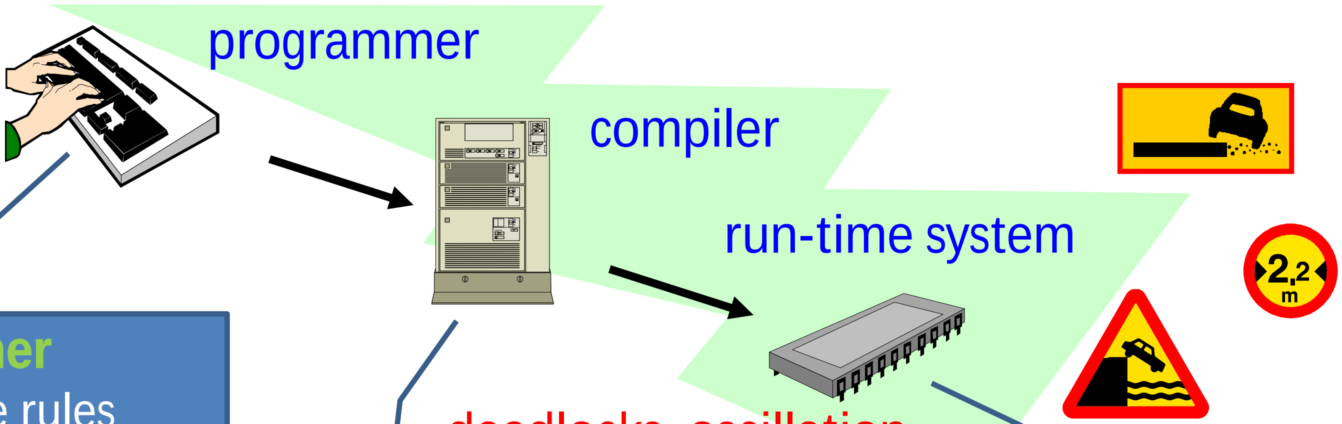
) linear complexity

Outline

1. Example
2. Threads and Concurrency
3. Sequential Constructiveness (SC)
4. Analysing SC
5. **Notions of Constructiveness**

A Game of Constructiveness and Schedulability

logically reactive program



deadlocks, oscillation, non-determinism, metastability

Programmer
- defines the rules
- prescribes sequential execution order
- leaves concurrency to compiler and run time
- „Free Schedules“

Compiler = Player
- determines winning strategy
- restricts concurrency to ensure determinacy and deadlock freedom
- „Admissible Schedules“

Run-time = Opponent
- tries to choose a spoiling execution from admissible schedules

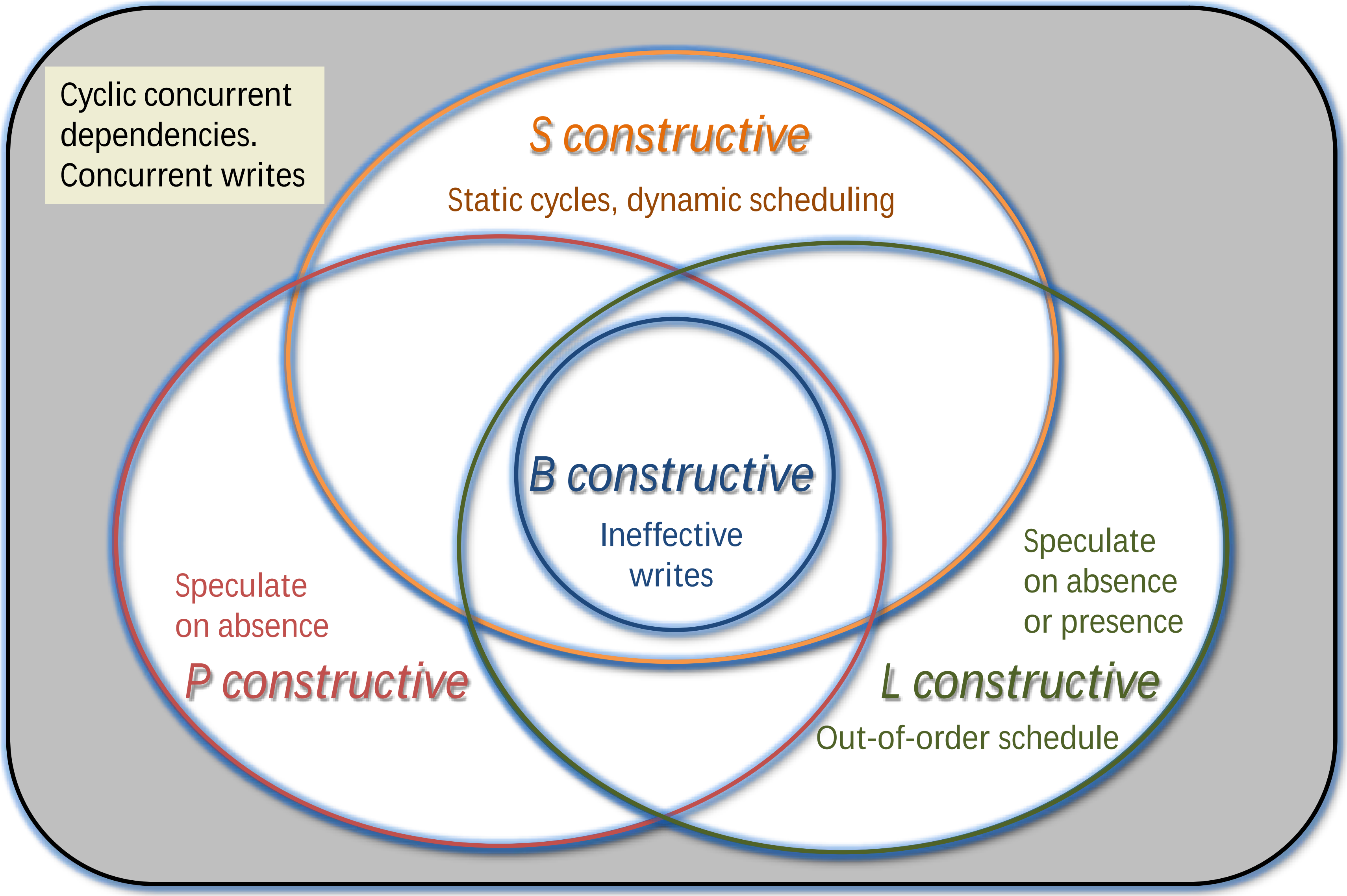
X-Constructiveness

Definition:

A program is *X-constructive (XC)* if
for each initial configuration and input:

1. there exists an X-admissible run
2. every X-admissible run generates the same, determinate sequence of macro step responses in bounded time.

Alternative Notions of Constructiveness



Cyclic concurrent
dependencies.
Concurrent writes

S constructive

Static cycles, dynamic scheduling

B constructive

Ineffective
writes

Speculate
on absence

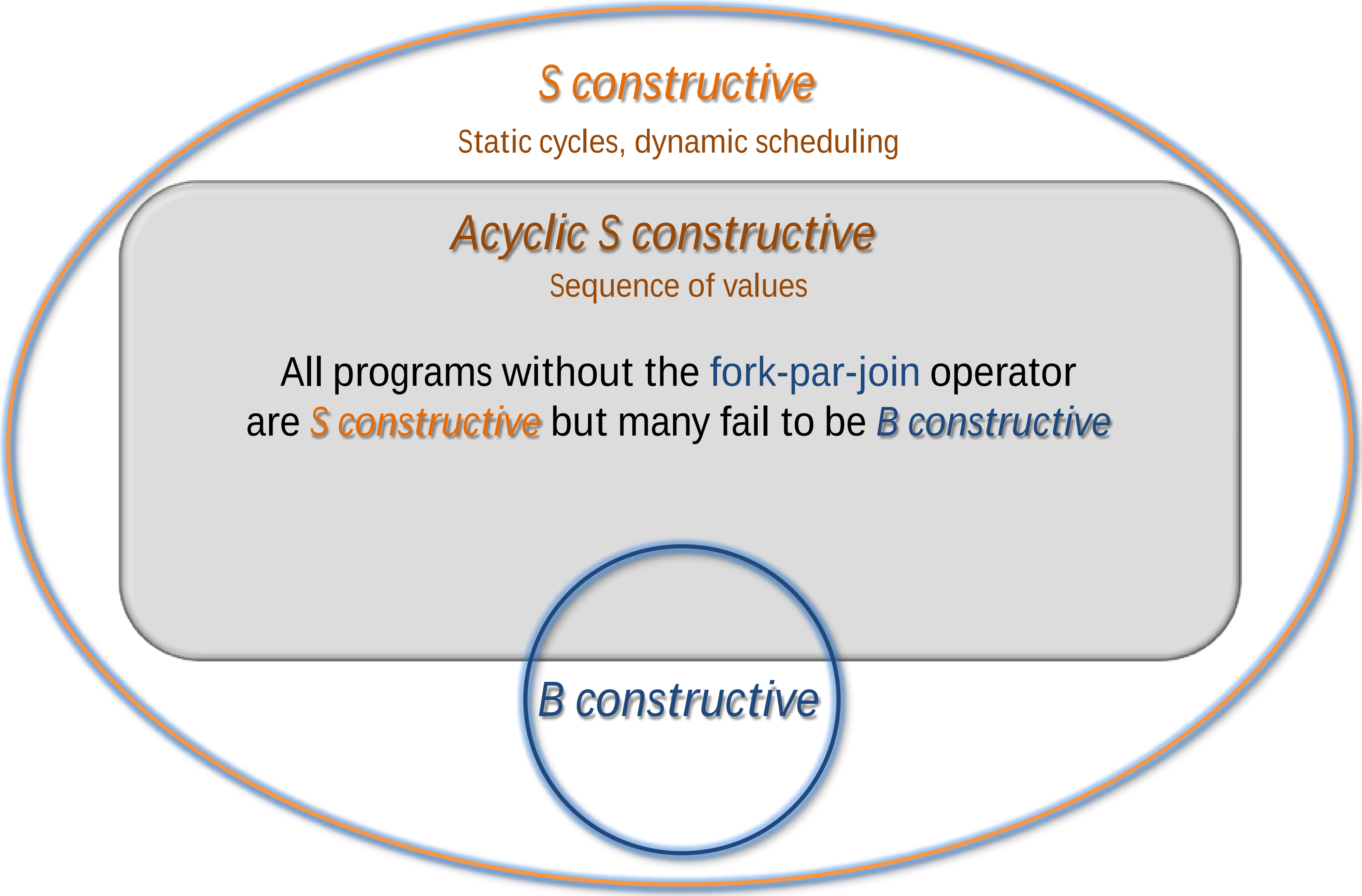
P constructive

Speculate
on absence
or presence

L constructive

Out-of-order schedule

Alternative Notions of Constructiveness



S constructive

Static cycles, dynamic scheduling

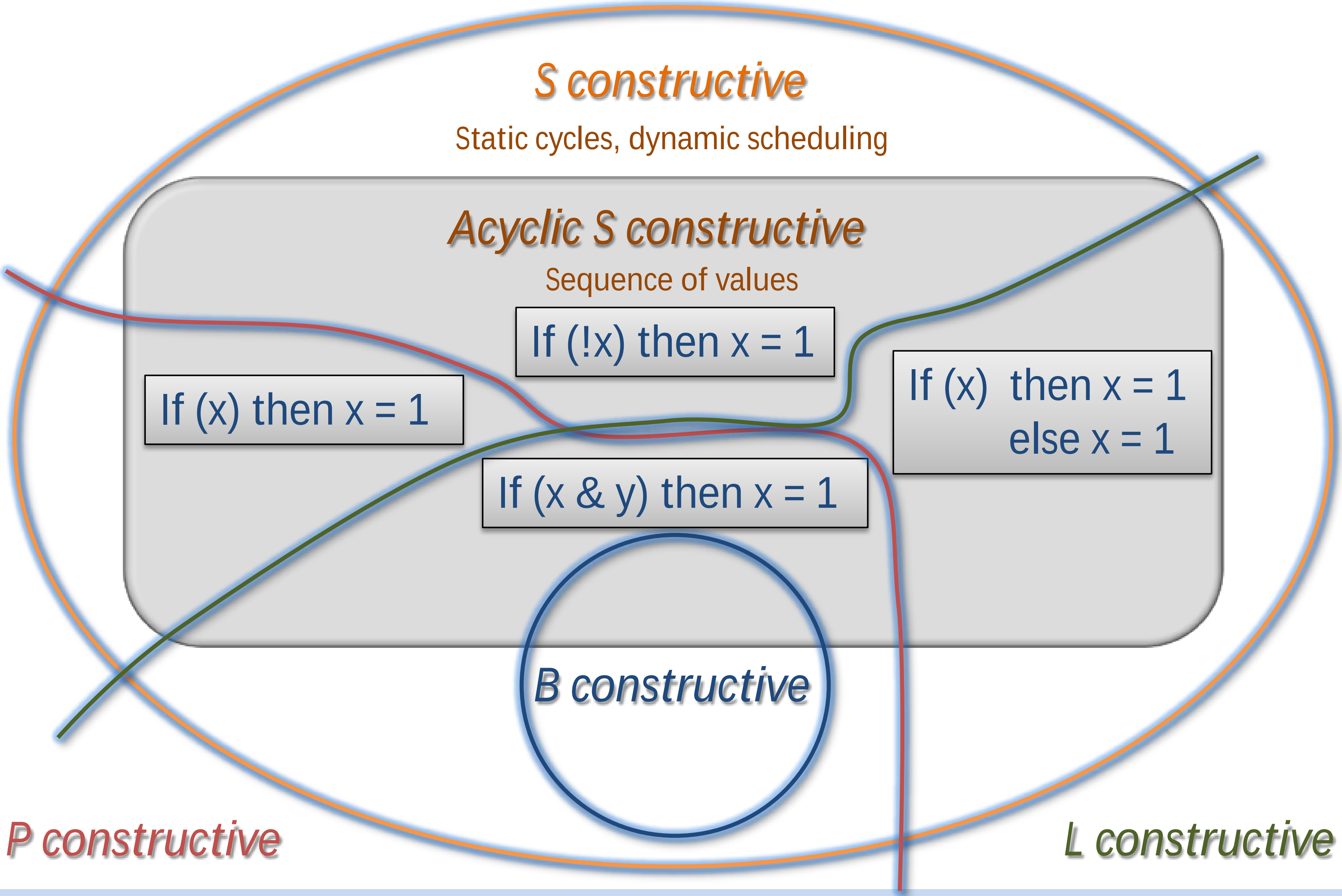
Acyclic S constructive

Sequence of values

All programs without the fork-par-join operator
are *S constructive* but many fail to be *B constructive*

B constructive

Alternative Notions of Constructiveness



S constructive

Static cycles, dynamic scheduling

Acyclic S constructive

Sequence of values

If (!x) then x = 1

If (x) then x = 1

If (x) then x = 1
else x = 1

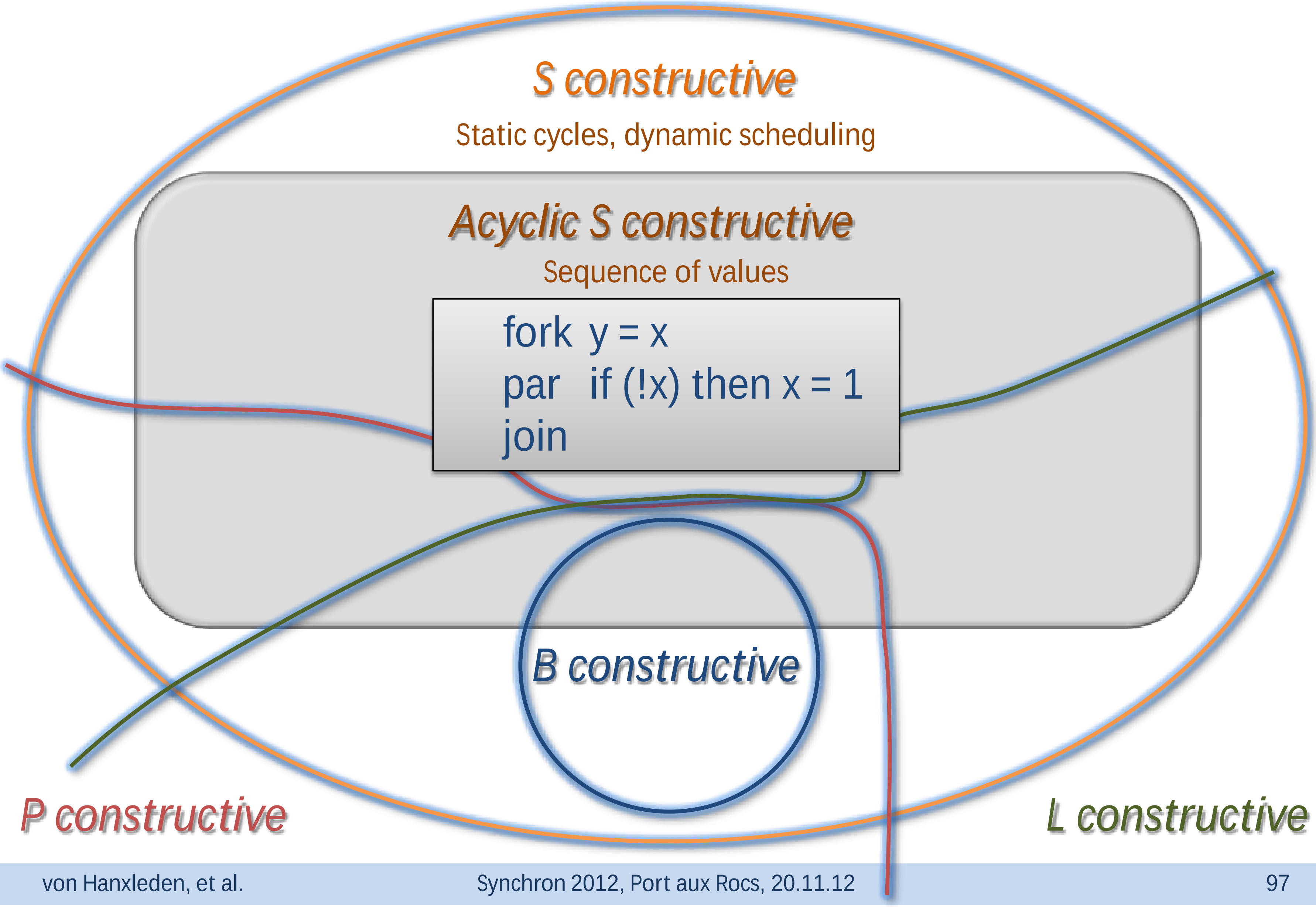
If (x & y) then x = 1

B constructive

P constructive

L constructive

Alternative Notions of Constructiveness



S constructive

Static cycles, dynamic scheduling

Acyclic S constructive

Sequence of values

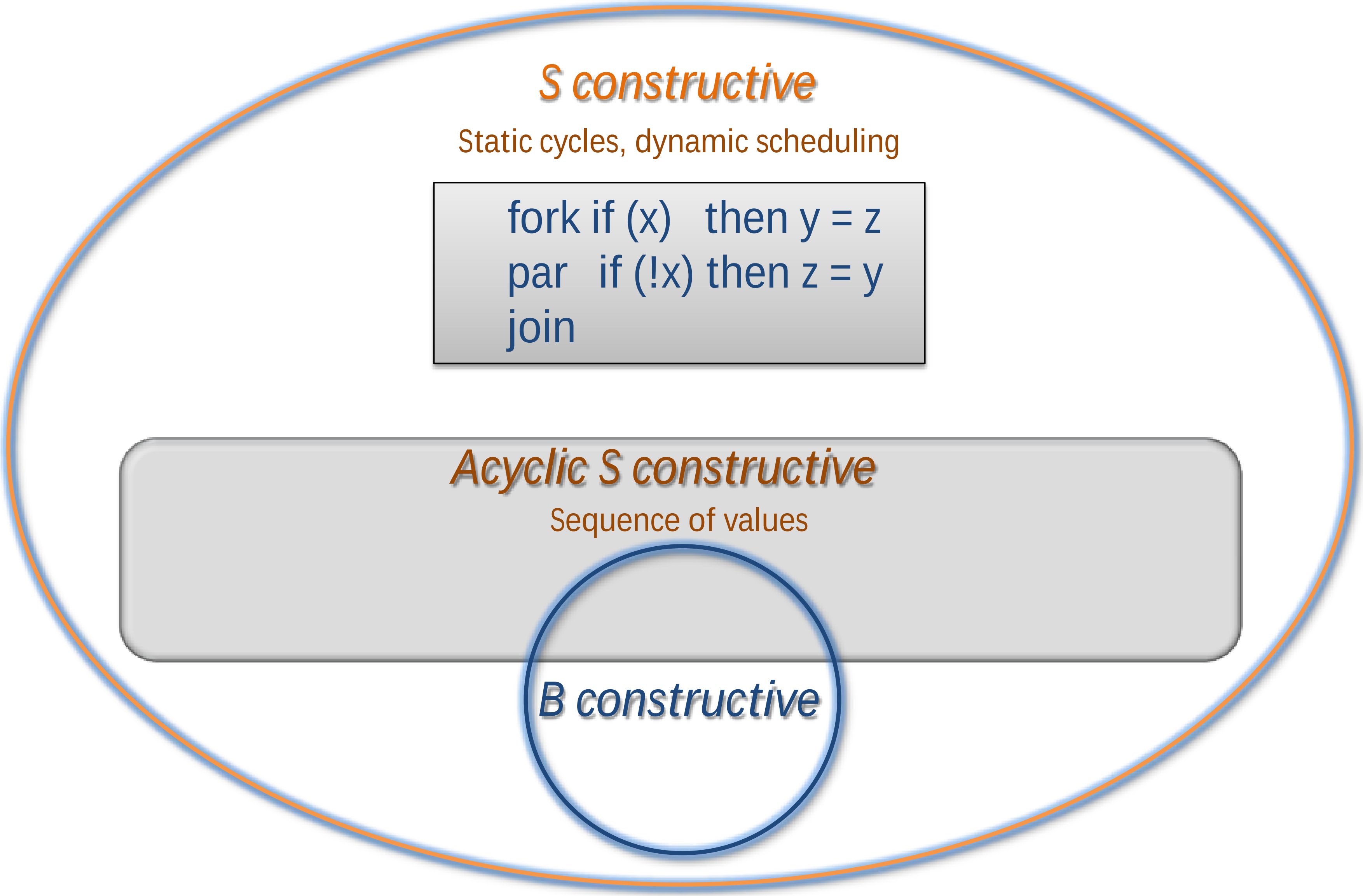
```
fork  y = x
par   if (!x) then x = 1
join
```

B constructive

P constructive

L constructive

Alternative Notions of Constructiveness



S constructive

Static cycles, dynamic scheduling

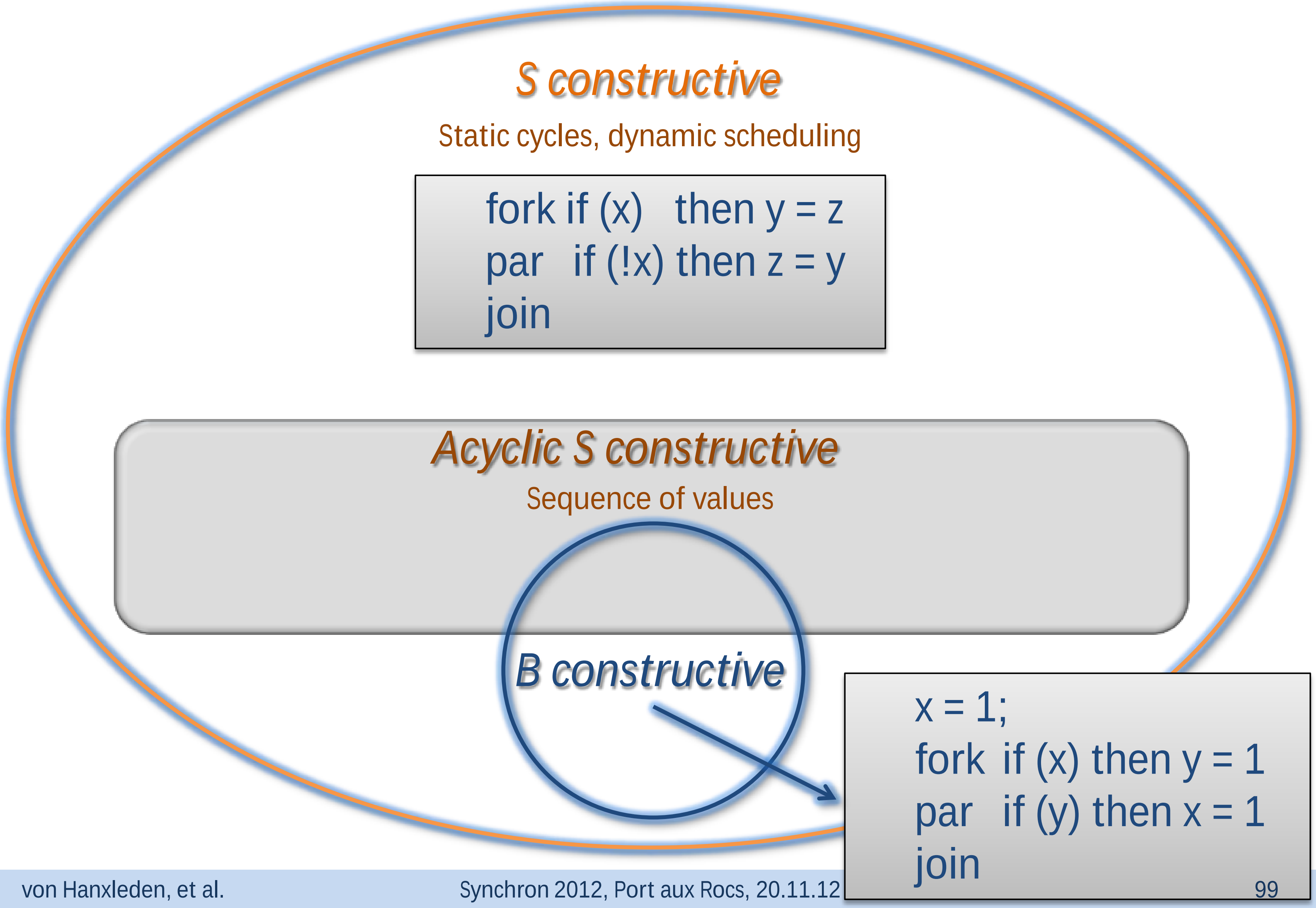
```
fork if (x)   then y = z
par   if (!x) then z = y
join
```

Acyclic S constructive

Sequence of values

B constructive

Alternative Notions of Constructiveness



S constructive

Static cycles, dynamic scheduling

```
fork if (x)   then y = z
par  if (!x) then z = y
join
```

Acyclic S constructive

Sequence of values

B constructive

```
x = 1;
fork  if (x) then y = 1
par   if (y) then x = 1
join
```

Conclusion

This Talk

- Clocked synchronous model of execution for imperative, shared-memory multi-processing
- Recovers and relaxes Esterel-style synchrony

Future Plans

- Full-scale implementation within PRETSY (Precision-timed Synchronous Processing)
- Develop approximating algorithms for SC-analysis: Constructiveness + WCRT
- Detailed semantical study of the class of SC programs vis-a-vis other classes (Pnueli & Shalev, Berry, Signal, …)

Questions

Thank you !