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 O'Brien[3]

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

PRETSY Project

*Full version (+ corrections) Tech Rep. Univ. of Kiel, March 2013, ISSN 2192-6247

Aim of this Work

- Motivating application: programming safety-critical embedded systems
- Key challenge: deterministic concurrency
- C, Java et. al.: familiar sequential paradigm, but concurrent constructs (threads) unpredictable in functionality and timing
- Synchronous Programming: predictable by construction (constructiveness), but unfamiliar to most programmers, restrictive in practice
- Aim of this work: concurrency with synchronous foundations, without synchronous restrictions

SC in a Nutshell: 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 con-current readers/writers
- on demand: single assign-ment synchronisation (locks, semaphores)

Imperative

- all sequential control flow prescriptive
- resolved by programmer

SC in a Nutshell: Taming Concurrency

Synchronous Languages
Esterel, Lustre, Signal, …

Clocked, cyclic schedule

✓ deterministic concurrency and deadlock freedom

✗ Heavy restrictions by constructiveness analysis

Sequential Languages
C, Java, …

Asynchronous schedule

✗ No guarantees of determinism or deadlock freedom

✓ Intuitive programming paradigm

Sequentially Constructive Model of Computation (SC MoC)

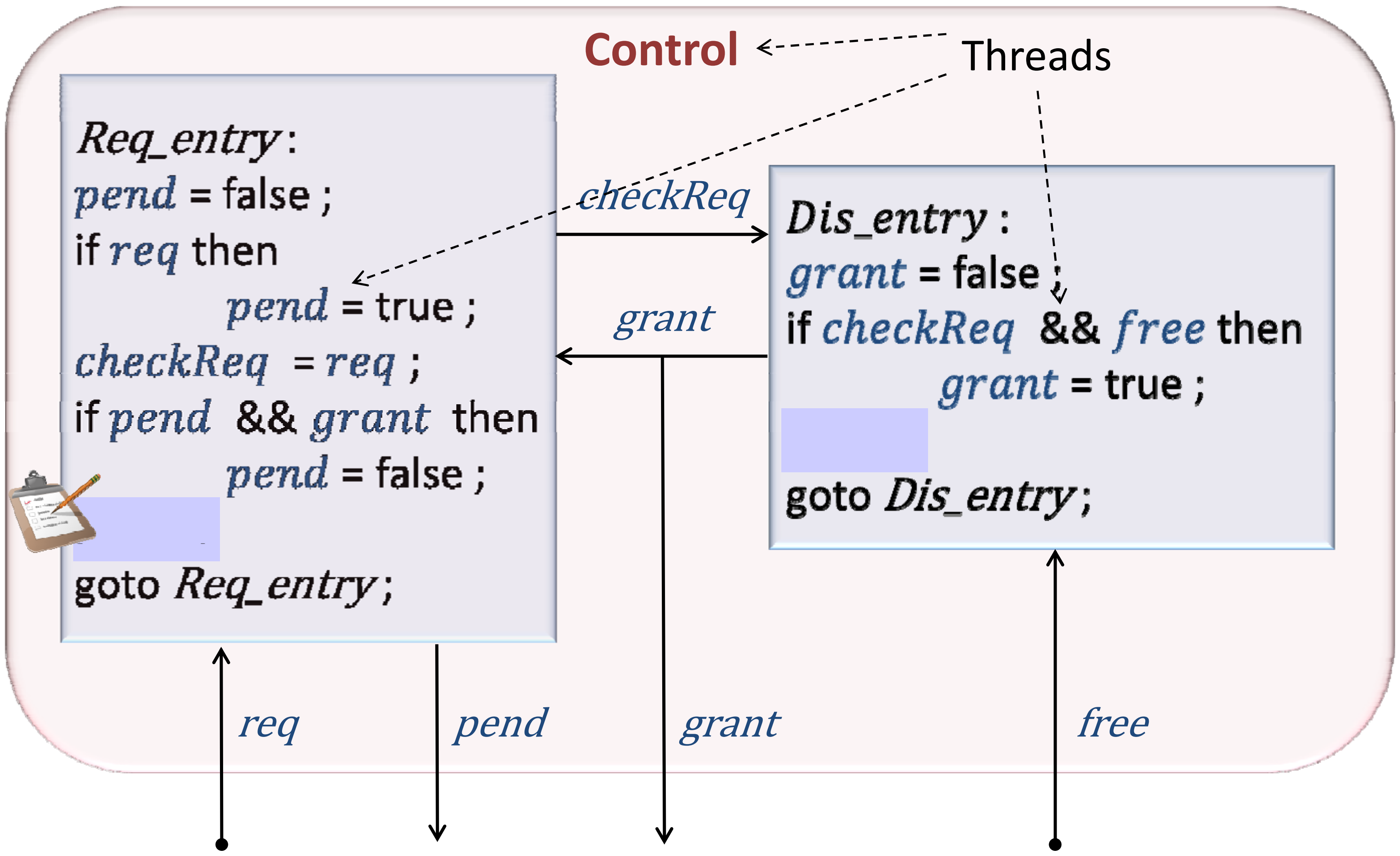
- all micro-step concurrent control flow descriptive
- resolved by scheduler

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

Outline

1. **Example**
2. Sequential Constructiveness (SC)
3. Analysing SC (ASC)
4. Conclusion

A Sequentially Constructive Program



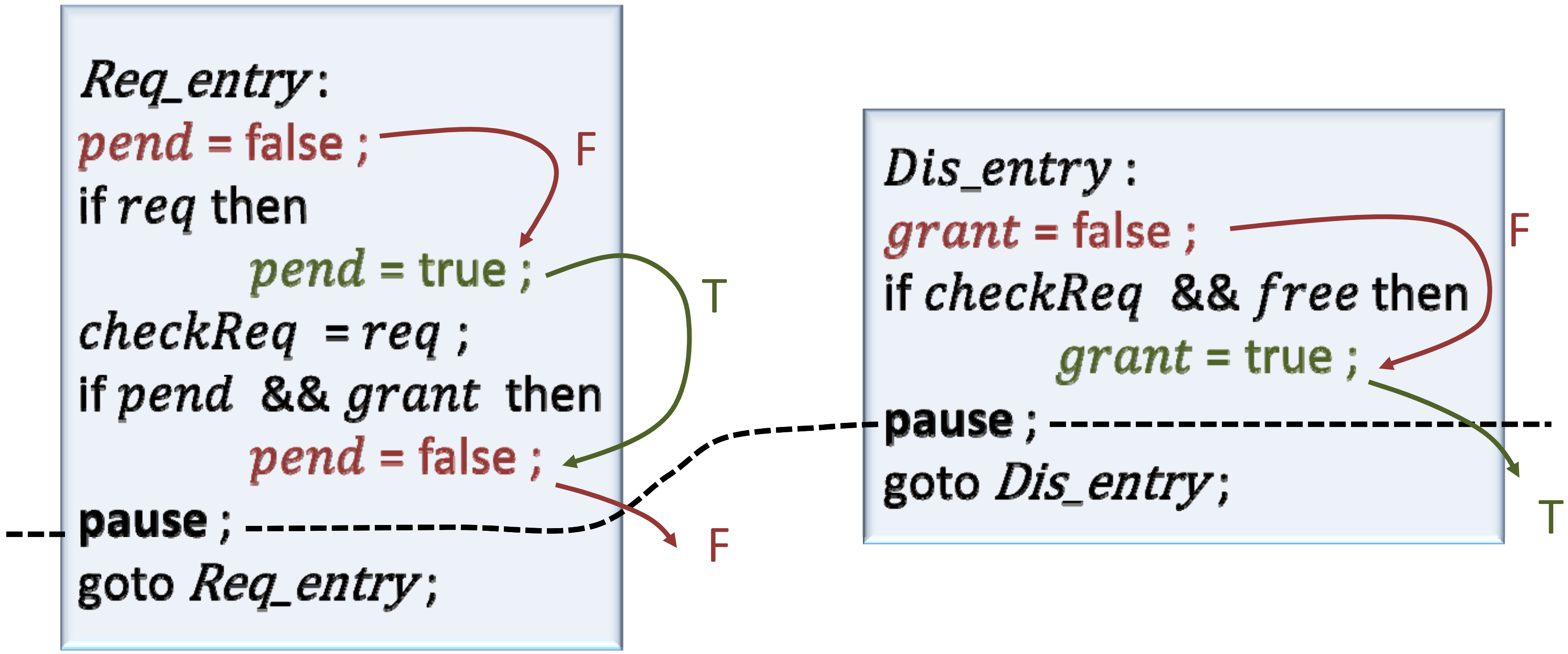
```
Req_entry:
pend = false;
if req then
        pend = true;
checkReq = req;
if pend && grant then
        pend = false;

goto Req_entry;
```

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

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;
```

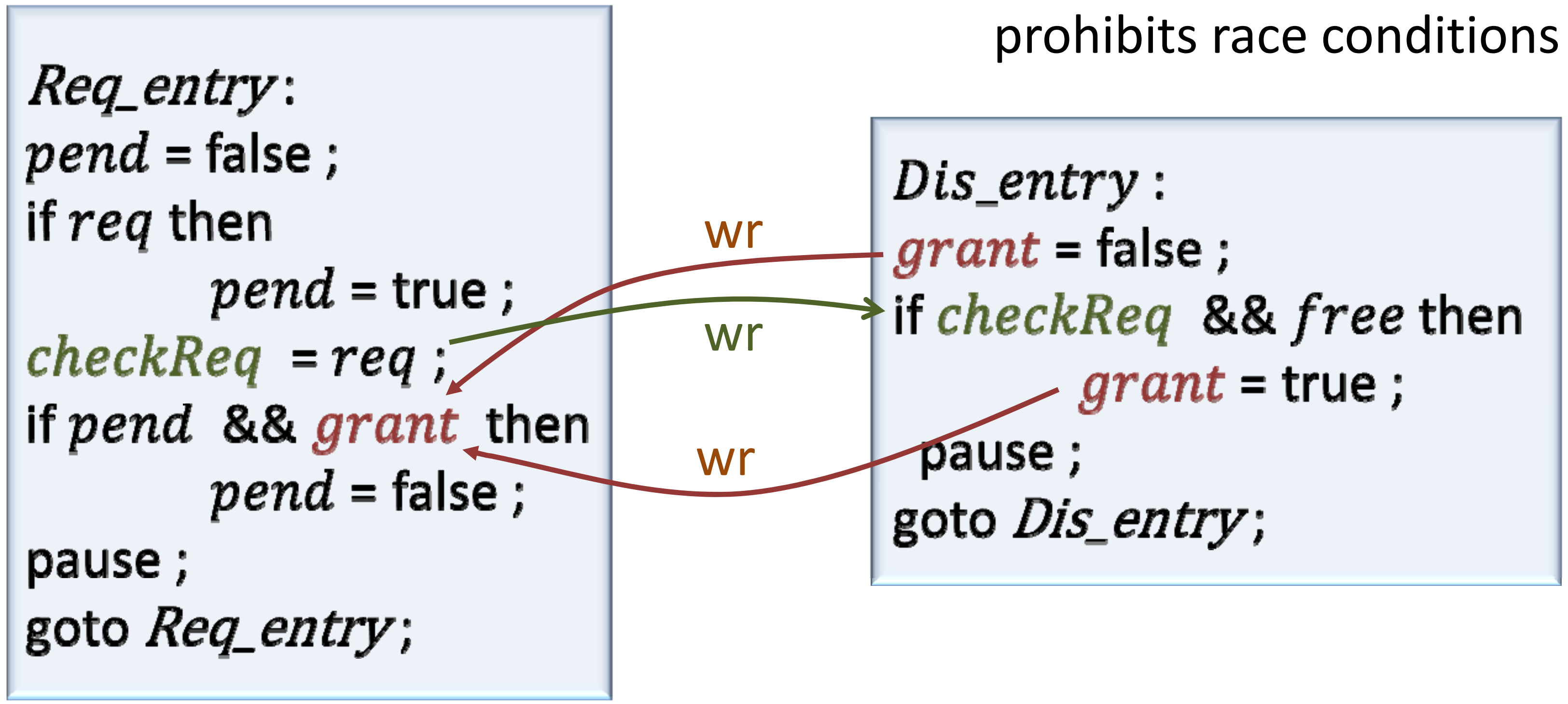
```
Dis_entry :
grant = false ;
if checkReq  && free then
        grant = true ;
pause ;
goto Dis_entry;
```

Imperative Program Order (sequential access to shared variables):

- “write-after-write” can change value sequentially (prescriptive)
- but not permitted in standard synchronous MoC

A Sequentially Constructive Program

SC MoC: micro-tick 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 & non-confluent writes
- to be implemented by the compiler ...

Outline

1. Example

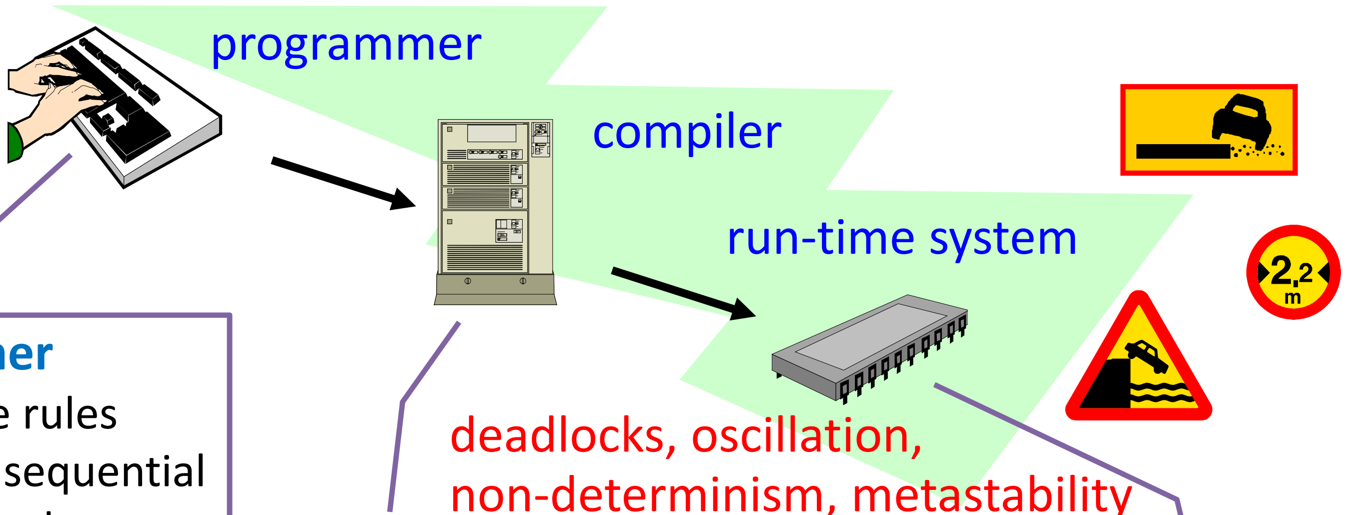
2. Sequential Constructiveness (SC)

3. Analysing SC (ASC)

4. Conclusion

A Constructive Game of 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

Sequential Admissibility

Basic Idea:

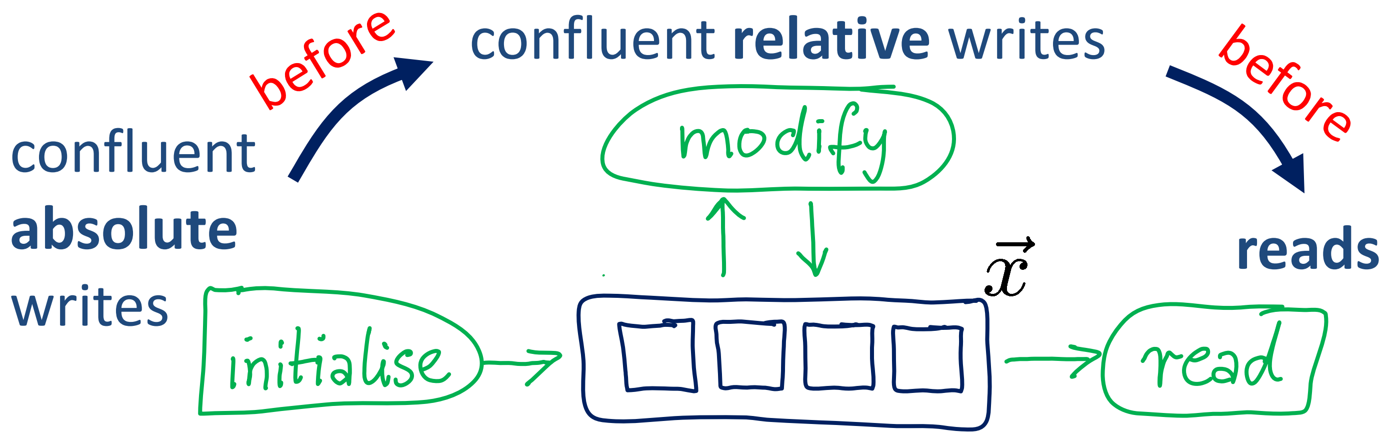
Sequentially ordered variable accesses
- are enforced by the programmer
- cannot be reordered by compiler or run-time platform
- exhibit no races

Only ***concurrent*** writes/reads to the same variable
- generate potential data data races
- must be resolved by the compiler
- can be ordered under multi-threading and run-time

The following applies to concurrent variable accesses only ...

Organising Concurrent Variable Accesses

SC Concurrent Variable Access Protocol:

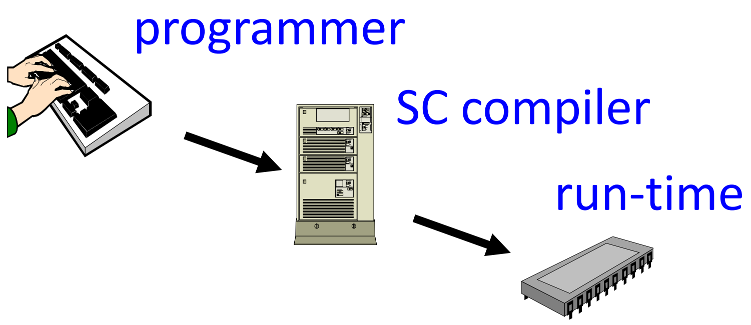


concurrent, multi-writer, multi-reader variables

Definition: A *run* for a SCG *G = (N,E)* is *SC-admissible* if, for all ticks in this run, and for all concurrent node instances [TODO]...

[TODO: what's „absolute", „relative", examples, „confluent" ?]

Sequential Constructiveness



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

1. there exists an SC-admissible run

2. every SC-admissible run generates the same determinate sequence of macro responses (in bounded time)

Outline

Analysing Sequential Constructiveness

By over-approximating concurrency and confluence the following static node relations are introduced:

$n_1 \rightarrow_{ww} n_2$ concurrent, **non-confluent absolute** writes

$n_1 \rightarrow_{wr} n_2$ n_1 absolute write and n_2 conccurent, non-confluent read

$\rightarrow_{wi}$ absolute write, concurrent non-confluent relative write.

$\rightarrow_{ir}$

$\rightarrow_{seq}$

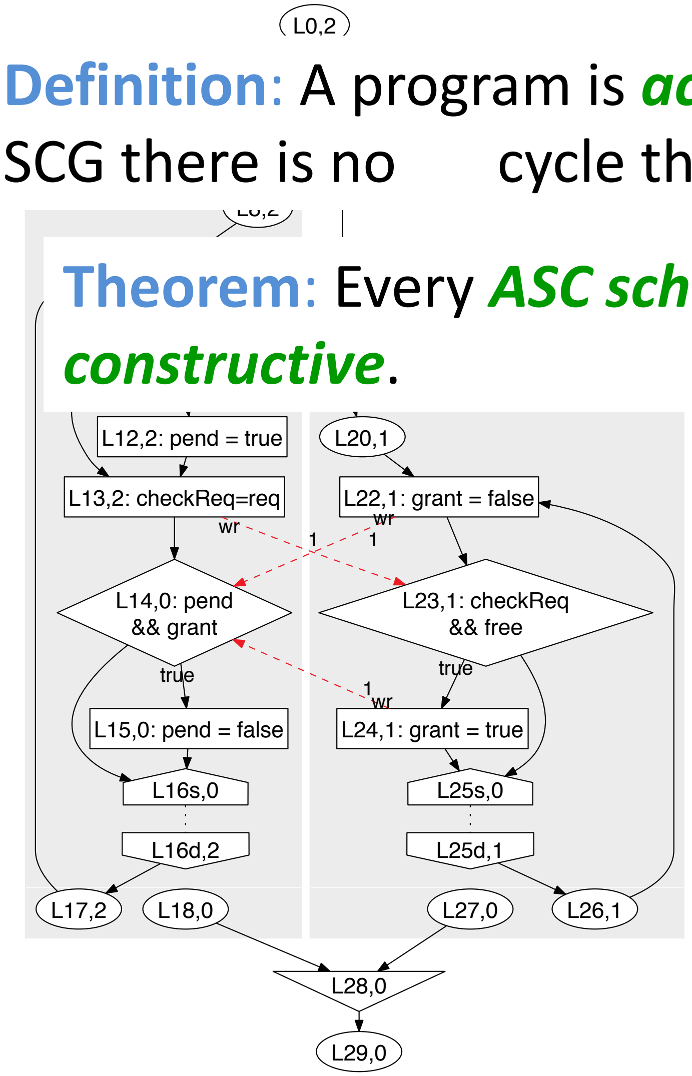
$\rightarrow_{wwir}$

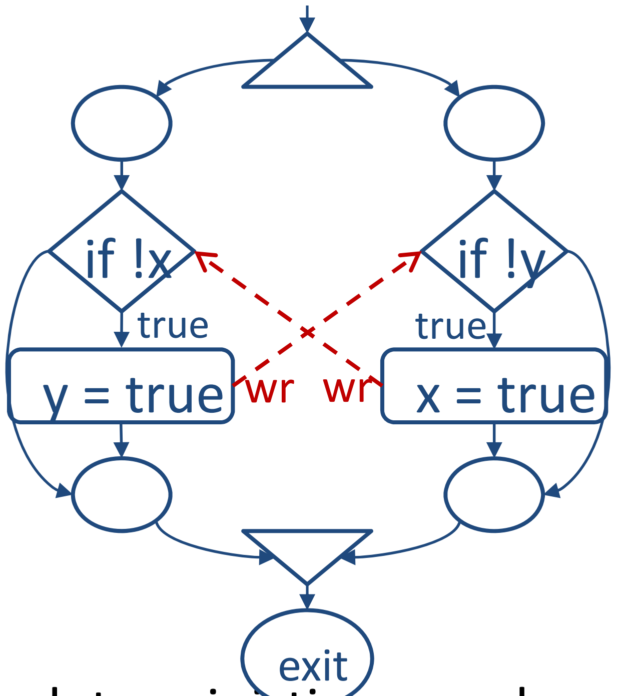
$\rightarrow$

Analysing Sequential Constructiveness

Definition: A program is ***acyclic SC (ASC) schedulable*** if in its SCG there is no cycle that contains edges induced by $\rightarrow_{wwir}$

Theorem: Every ***ASC schedulable*** program is ***sequentially constructive***.





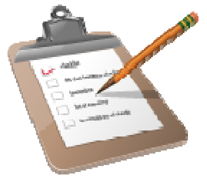
[TODO: running example, no cycle, non-deterministic example: cycle!

Outline

Conclusion

This Talk

- Clocked, synchronous model of execution for imperative, shared-memory multi-threading
- Conservatively extends synchronous programming (Esterel) by standard sequential control flow (Java, C)



Future Plans

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

PRETSY Project: www.pretsy.org

Questions

Thank you !