Synchronous Languages—Lecture 22

Prof. Dr. Reinhard von Hanxleden

Christian-Albrechts Universität Kiel
Department of Computer Science
Real-Time Systems and Embedded Systems Group

23 January 2014

Last compiled: January 31, 2014, 9:02 hrs

Sequentially Constructive Concurrency

Safety-Critical Embedded Systems

- Embedded systems often safety-critical
- Safety-critical systems must react deterministically
- Computations often exploit *concurrency*
- Key challenge: **Concurrency must be deterministic!**

Thanks to Michael Mendler (U Bamberg) for support with these slides

Implementing (Deterministic) Concurrency

- **C, Java, etc.:**
  - ☺ Familiar
  - ☺ Expressive sequential paradigm
  - ☹ Concurrent threads unpredictable in functionality and timing
- **Synchronous Programming**:
  - ☺ predictable by construction
    $\Longrightarrow$ Constructiveness
  - ☹ Unfamiliar to most programmers
  - ☹ Restrictive in practice

Aim: Deterministic concurrency with synchronous foundations, but without synchronous restrictions.

Comparing Both Worlds

Sequential Languages

- C, Java, ...
- Asynchronous schedule
  - o By default: Multiple concurrent readers/writers
  - o On demand: Single assignment synchronization (locks, semaphores)
- Imperative
  - o All sequential control flow prescriptive
  - o Resolved by programmer

Synchronous Languages

- Esterel, Lustre, Signal, SCADE, SyncCharts ...
- Clocked, cyclic schedule
  - o By default: Single writer per cycle, all reads initialized
  - o On demand: Separate multiple assignments by clock barrier (pause, wait)
- Declarative
  - o All micro-steps sequential control flow descriptive
  - o Resolved by scheduler

Comparing Both Worlds (Cont'd)

Sequential Languages

- Asynchronous schedule
  - ☹ No guarantees of determinism or deadlock freedom
  - ☺ Intuitive programming paradigm

Synchronous Languages

- Clocked, cyclic schedule
  - ☺ Deterministic concurrency and deadlock freedom
  - ☹ Heavy restrictions by constructiveness analysis

$\Longrightarrow$

Sequentially Constructive Model of Computation (SC MoC)

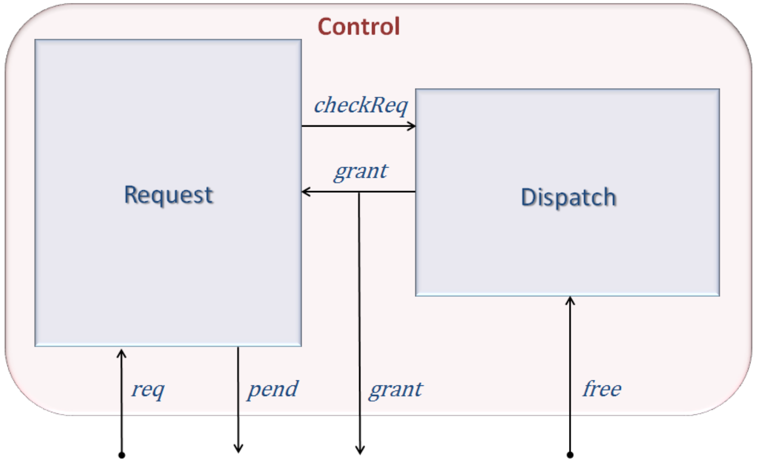
☺ Deterministic concurrency and deadlock freedom

☺ Intuitive programming paradigm

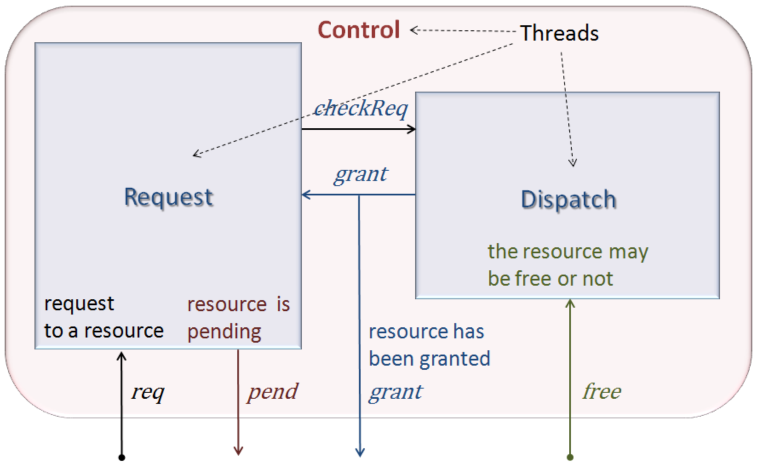
Implementing Deterministic Concurrency: SC MoC

- **Concurrent** micro-step control flow:
  - ☺ Descriptive
  - ☺ Resolved by scheduler
  - ☺ $\Longrightarrow$ Deterministic concurrency and deadlock freedom
- **Sequential** micro-step control flow:
  - ☺ Prescriptive
  - ☺ Resolved by the programmer
  - ☺ $\Longrightarrow$ Intuitive programming paradigm

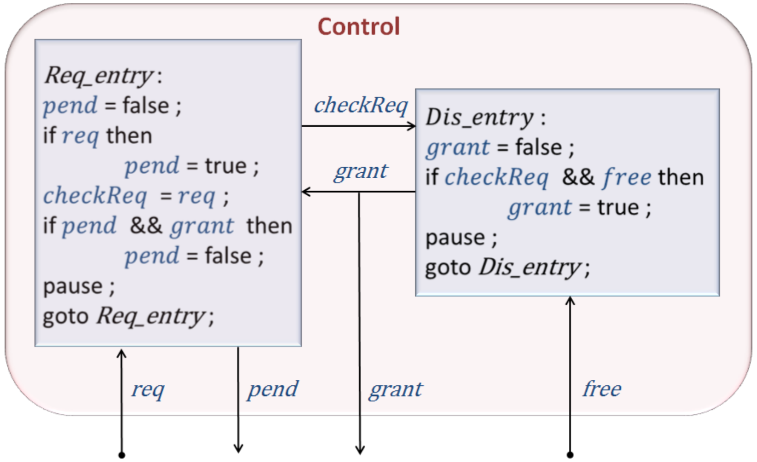
A Sequentially Constructive Program



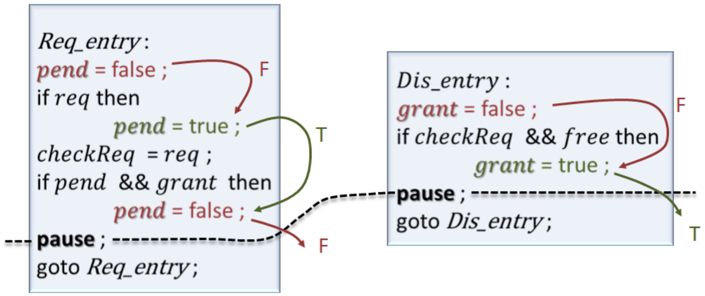
A Sequentially Constructive Program (Cont'd)



A Sequentially Constructive Program (Cont'd)



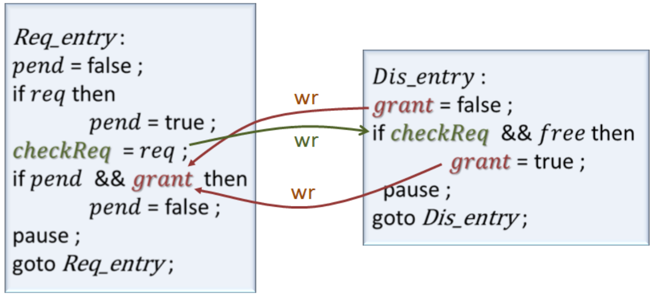
A Sequentially Constructive Program (Cont'd)



Imperative program order (sequential access to shared variables):

- "write-after-write" can change value sequentially
- Prescribed by programmer
  - ☺ Accepted in SC MoC
  - ☹ Not permitted in standard synchronous MoC

A Sequentially Constructive Program (Cont'd)

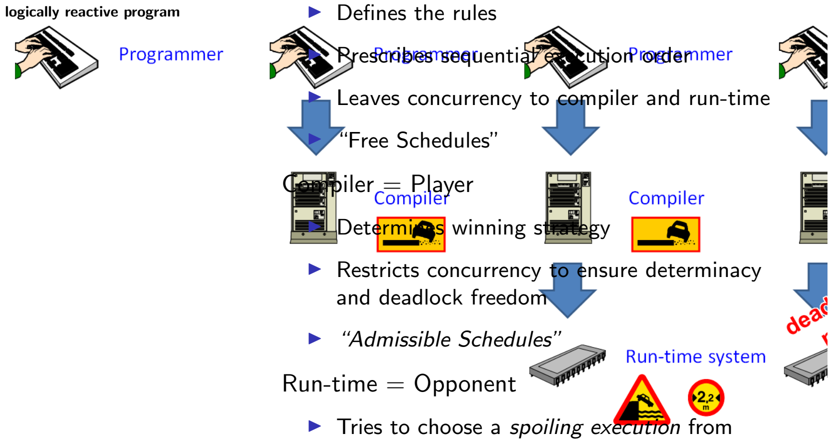


Concurrency scheduling constraints (access to shared variables):

- "write-before-read" for concurrent write/reads
- "write-before-write" (*i. e.*, conflicts!) for concurrent & non-confluent writes
- Micro-tick thread scheduling prohibits race conditions
- Implemented by the SC compiler

Overview

A Constructive Game of Schedulability



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

Organizing Concurrent Variable Accesses

- **SC Concurrent Memory Access Protocol (per macro tick)**

concurrent, multi-writer, multi-reader variables

concur

- **Confluent Statements (per macro tick)**

For all me
Mem, rea
in macro t

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 races
  - Must be resolved by the compiler
  - Can be ordered under multi-threading and run-time

The following applies to **concurrent** variable accesses only …

Types of Writes I

Given **two writes** to x, distinguish

- Confluent writes
  - Order of the writes does not matter
  - Precondition: No side effects
- Non-confluent writes
  - Order of the writes does matter

We also generalize the notion of confluence to pairs of arbitrary statements, if their execution order does not matter.

Also distinguish

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

Note: Given two identical writes $x = e; x = e$,

- these are confluent
- the 2nd write is ineffective

Combination Functions

Combination function f:

- $f(f(x, e_1), e_2) = f(f(x, e_2), e_1)$
  for all side-effect free expressions e_1, e_2
- Sufficient condition: f is *commutative* and *associative*
- Examples: *, $+$, $-$, max, and, or

Types of Writes II

Relative writes, of type f ("increment" / "modify"): $x = f(x, e)$

- f must be a combination function
- Evaluation of e must be free of side effects
- Thus, 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
- Thus, writes are confluent
- E.g., `x++`, `x = 5*x`, `x = x-10`

Absolute writes ("write" / "initialize"): $x = e$

- Writes that are not relative
- E.g., `x = 0`, `x = 2*y+5`, `x = f(z)`

Scheduling Relations I

For macro tick R, and **concurrent but not confluent** node instances (executed statements) ni_1, ni_2, define scheduling relations:

$ni_1 \rightarrow^R ni_2$: "happens before" (linear order)

- ni_1 occurs before ni_2 in R

$ni_1 \leftrightarrow^R_{ww} ni_2$: "write / write conflict"

- ni_1 and ni_2 both perform absolute writes on the same variable
- or both perform relative writes of different type on the same variable
- $\lightning$ **Impossible to find linear order!**

Scheduling Relations II

For macro tick R, and **concurrent but not confluent** node instances (executed statements) ni_1, ni_2, define scheduling relations:

$ni_1 \rightarrow^R_{wr} ni_2$: "write before read", or "initialize before read"

- ni_1 is absolute write
- ni_2 is read of the same variable

$ni_1 \rightarrow^R_{ir} ni_2$: "increment before read", or "update before read"

- ni_1 is relative write
- ni_2 is read of the same variable

$ni_1 \rightarrow^R_{wi} ni_2$: "write before increment", or "initialize before update"

- ni_1 is absolute write
- ni_2 is relative write of the same variable

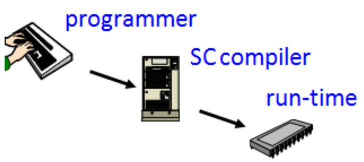
Sequential Admissibility

$ni_1 \rightarrow^R ni_2$: "happens before"
$ni_1 \leftrightarrow^R_{ww} ni_2$: "write / write"
$ni_1 \rightarrow^R_{wr} ni_2$: "write before read"
$ni_1 \rightarrow^R_{ir} ni_2$: "increment before read"
$ni_1 \rightarrow^R_{wi} ni_2$: "write before increment"

Definition: A run is SC-admissible iff
for all macro ticks R and all node instances ni_1, ni_2 in R:

$$\neg(ni_1 \leftrightarrow^R_{ww} ni_2) \wedge$$
$$((ni_1 \rightarrow^R_{wr} ni_2) \vee (ni_1 \rightarrow^R_{ir} ni_2) \vee (ni_1 \rightarrow^R_{wi} ni_2)) \Rightarrow ni_1 \rightarrow^R ni_2)$$

Sequential Constructiveness – Definition



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

1. There exists an SC-admissible run
2. Every SC-admissible run generates the same determinate sequence of macro responses

Overview

Conservative Static Approximation

In practice, a compiler must be conservative:

- Use a relation $n_1 | n_2$ to over-approximate $n_1 |_R n_2$, *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.
- May not recognize confluence
- May not recognize that writes are relative

Acyclic Sequential Constructiveness – Definition

- By *over-approximating* concurrency and confluence the static node relations

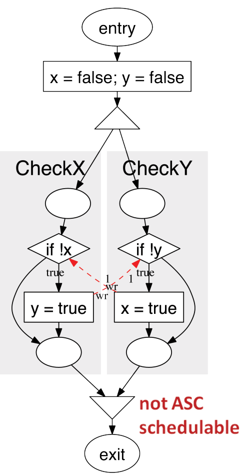
$$n_1 \leftrightarrow_{ww} n_2, \quad n_1 \rightarrow_{wr} n_2, \quad n_1 \rightarrow_{ir} n_2, \quad \text{and} \quad n_1 \rightarrow_{wi} n_2$$

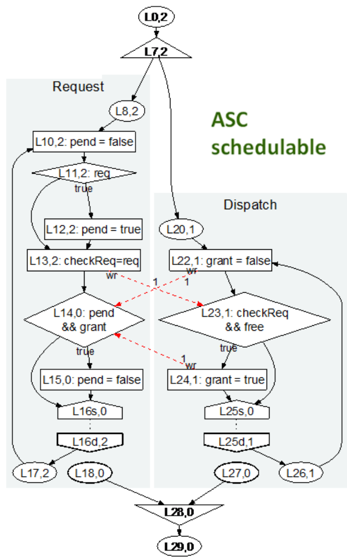
  are computed.
- A *suitable over-approximation* of $\rightarrow^R$ is the (transitive closure) of the static control flow relation $n_1 \rightarrow_{seq} n_2$ (program order).
- Let $\rightarrow$ be defined as the following union:

$$\rightarrow := \rightarrow_{seq} \cup \rightarrow_{wr} \cup \rightarrow_{ir} \cup \rightarrow_{wi} \cup \leftrightarrow_{ww}$$

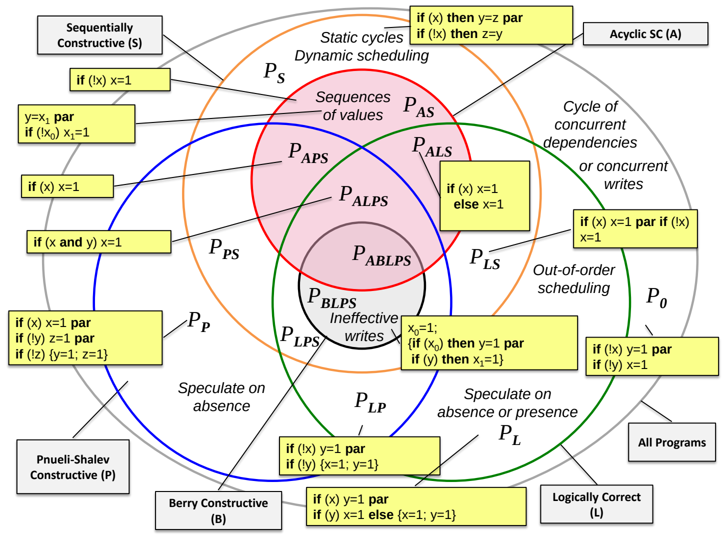
Definition: A program is acyclic SC (ASC) schedulable iff in its sequential-concurrent control flow graph (SCG) all $\rightarrow$ cycles consist entirely of $\rightarrow_{seq}$ edges.

Theorem: ASC schedulability $\Longrightarrow$ sequential constructiveness





Synchronous Program Classes



Conclusions

- Clocked, **synchronous model of execution** for **imperative, shared-memory multi-threading**
- Conservatively extends synchronous programming (Esterel) by **standard sequential control flow** (Java, C)
- $\Longrightarrow$ Deterministic concurrency with synchronous foundations, but without synchronous restrictions
  - ☺ Expressive and intuitive sequential paradigm
  - ☺ Predictable concurrent threads

To Go Further

- DFG-funded PRETSY Project: www.pretsy.org

- R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O'Brien. *Sequentially Constructive Concurrency – A conservative extension of the synchronous model of computation*. In Proc. Design, Automation and Test in Europe Conference (DATE'13), Grenoble, France, March 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/date13.pdf

- R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and Partha Roop. *Sequentially Constructive Concurrency – A Conservative Extension of the Synchronous Model of Computation*. Technical Report 1308, Christian-Albrechts-Universitaet zu Kiel, Department of Computer Science, Aug 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-13seqc.pdf

- G. Berry. *The foundations of Esterel*. In G. Plotkin, C. Stirling, and M. Tofte, editors, Proof, Language, and Interaction: Essays in Honour of Robin Milner, pages 425-454, Cambridge, MA, USA, 2000.