Synchronous Languages—Lecture 14

Prof. Dr. Reinhard von Hanxleden

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

15 Jan. 2019

Last compiled: January 29, 2019, 10:57 hrs

Sequentially Constructive Concurrency in Practice

The 5-Minute Review Session

1. What are goals and challenges in defining the SC MoC?
2. What is *confluence* in the SC MoC?
3. What is *thread reincarnation*?
4. In the SC MoC, when are threads considered *statically concurrent*?
5. What is a *thread tree*? How can it be used to define static concurrency?

The 5-Minute Review Session

1. How is *run-time concurrency* defined? How does it relate to static concurrency?
2. What is *SC-admissibility*?
3. When is a program *sequentially constructive*?
4. What is an *SC-schedule*? When is it *valid*?
5. What are conservative, practical approximations of sequential constructiveness?

References

Most of the material here draws from this reference [TECS]:

R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop.
Sequentially Constructive Concurrency – A Conservative Extension of the Synchronous Model of Computation.
ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design, July 2014, 13(4s).
https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/tecs14.pdf

Unless otherwise noted, the numberings of definitions, sections etc. refer to this.

There is also an extended version [TR]:

R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop.
Sequentially Constructive Concurrency – A Conservative Extension of the Synchronous Model of Computation.
Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1308, ISSN 2192-6247, Aug. 2013, 13(4s).
https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-1308.pdf

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

SC-Schedules [Def. 5.1, Lemma 5.3]

- Given: SCG $G = (N, E)$
- SC-schedule Σ is subset of G's instantaneous edges: $\Sigma \subseteq E_{ins}$
- E_{ins} is structural SC-schedule; derived solely by analysis of the program structure
- An SC-schedule Σ is valid if
  - for every macro tick R of G which can be reached and executed under the SC-admissibility rules,
  - if $(n_1, i_1) \rightarrow^R_\alpha (n_2, i_2)$ for some node instances $(n_{1,2}, i_{1,2})$ in R and some $\alpha \in \alpha_{ins}$,
  - then $(n_1 \rightarrow_\alpha n_2) \in \Sigma$.

Validity guarantees:

- If G is executed in an SC-admissible fashion,
- then static node relations $\rightarrow_\alpha$ of Σ are conservative over-approximation of dynamic relations $\rightarrow^R_\alpha$ on node instances

Lemma: E_{ins} is valid

Schedule order [Def. 5.2]

- Given: Valid SC-schedule Σ
- Schedule order: $n_1 \twoheadrightarrow^{\Sigma}_{ins} n_2$ iff
  1. $n_1 \parallel n_2$ and
  2. Σ contains a path from n_1 to n_2 that includes an iur-edge

To enforce the iur protocol among concurrent threads, it suffices to always execute $\twoheadrightarrow^{\Sigma}_{ins}$-minimal nodes

However: valid schedule may still contain conflicting orderings that cannot be satisfied or where it depends on the capabilities of the compiler or the run-time system whether it can be implemented

---

Note that (2) is conservative in that it may also impose a scheduling order between nodes if they are not run-time concurrent. We choose this conservative definition to be compatible with the priority-based scheduling scheme introduced in Sec. **??**.
A less conservative, *thread-instance aware* definition of schedule order would for example not consider paths that include $lcafork(n_1, n_2)$, since at run time, executing $lcafork(n_1, n_2)$ would preclude that the node instances corresponding to $n_{1,2}$ could be run-time concurrent.

Schedule / Program Classes [Def. 5.4]

Schedule properties

- acyclic: does not contain any cycle
- iur-acyclic: does not contain any cycle that contains edges induced by $\rightarrow_{iur}$

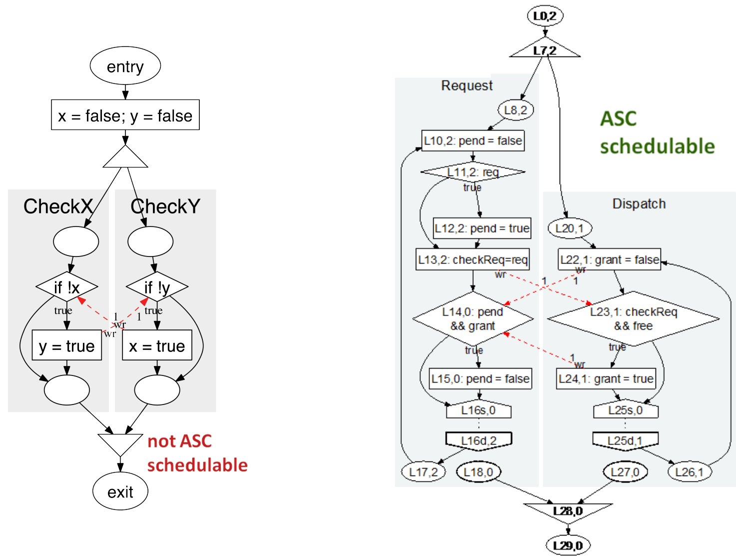
Program (SCG) properties

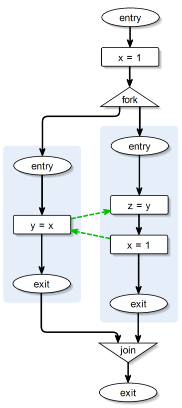
- Acyclic SC (ASC): $\exists$ valid acyclic SC-schedule
- Iur-acyclic SC (IASC): $\exists$ valid iur-acyclic SC-schedule
- Structurally acyclic SC (SASC): E_{ins} is acyclic
- Structurally iur-acyclic SC (SIASC): E_{ins} is iur-acyclic

Implications (see also Theorem 5.5):

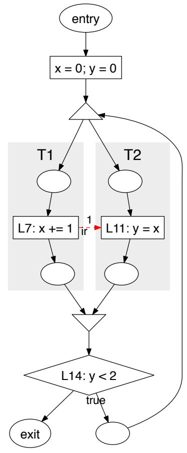
- SASC $\implies$ SIASC $\implies$ IASC $\implies$ SC
- SASC $\implies$ ASC $\implies$ IASC $\implies$ SC

May also relax the sequential order to only order non-confluent statements $\rightsquigarrow$ data-flow acyclic programs





ASC, and hence SC, but not SIASC, hence not SASC



SC, but not IASC, and hence not SIASC/ASC/SASC

Priorities [Def. 5.6, Lemma 5.7]

- Given: valid SC-schedule Σ
- Priority $n.pr$ of statement $n \in N$: maximal number of $\rightarrow_{iur}$ edges traversed by any path in Σ that originates in n

Lemma: Priorities implement the schedule order
Given:

- Priority assignment according to some SC-schedule Σ
- Run-time (and hence also statically) concurrent statements $n_{1,2} \in N$

Then: $n_1 \twoheadrightarrow^{\Sigma}_{ins} n_2$ implies $n_1.pr > n_2.pr$

Overview

Priority-Based Scheduler [Theorem 5.8]

Priority-based scheduler: always gives control to the thread with highest priority, chosen from the set of threads that are still active in the current tick

- Never allows a statement that is ready for execution to wait on another statement with lower priority
- Implements a valid schedule, as can be verified from the SCG construction
- For example $n_1 \rightarrow_{iu} n_2$ implies $n_1 \rightarrow_{iur} n_2$, which implies, by definition of priorities, $n_1.pr > n_2.pr$, which in turn implies that n_1 gets scheduled before n_2

Theorem
A program is IASC iff there exists a valid SC-schedule such that all statement priorities are finite

Computing Priorities for IASC Programs

- Given a valid SC-schedule Σ, can formulate the calculation of priorities as longest weighted path problem
- Assign to each edge $e \in \Sigma$ a weight $e.w$, with $e.w = 0$ iff $e.src \rightarrow_{seq} e.tgt$, and $e.w = 1$ iff $e.src \rightarrow_{iur} e.tgt$
- As relations $\rightarrow_{iur}$ and $\rightarrow_{seq}$ exclude each other, weight of each edge is uniquely determined
- $n.pr$ becomes maximal weight of any path originating in n
- Difficulty: want to handle (sequential) loops, *i. e.*, cyclic SCGs
- For arbitrary (*i. e.*, possibly cyclic) weighted graphs, the computation of the longest weighted path is NP-hard
- However, can exclude all graphs with a positive weight cycle

Algorithm for Computing Priorities I

1. Detect whether Σ has a positive weight cycle.
   We can do so by computing the Strongly Connected Components (SCCs), *e. g.*, by Tarjan's algorithm, and checking if any SCC contains a node that is connected to another node within the same SCC by a $\rightarrow_{iur}$ edge.
2. If a positive weight cycle exists, then Σ is not *iur*-acyclic; we then **reject** the program.
   Otherwise, we **accept** the program, and continue.
   Now nodes in the same SCC can reach each other, but only through paths with weight 0, and therefore must have the same priority.

Algorithm for Computing Priorities II

3. From the SCCs, construct the directed acyclic graph $G_{SCC} = (N_{SCC}, E_{SCC})$, where $N_{SCC} \subset N$ contains a representative node from each SCC of G (using *e. g.* the SCC roots computed by Tarjan's algorithm), and E_{SCC} contains an edge from one SCC representative to another iff the corresponding SCCs are connected in G.
   Here we assign an edge in E_{SCC} the maximum weight of the corresponding edges in Σ.
4. Compute for each $n_{SCC} \in N_{SCC}$ the maximum weighted length (priority) $n_{SCC}.pr$ of any path originating in n_{SCC}, *e. g.*, with a depth-first recursive traversal of all edges in the acyclic G_{SCC}.
5. Assign each statement $n \in N$ the priority computed for its SCC.

Complexity: linear in size of SCG

Overview

Summary I

Underlying idea of sequential constructiveness rather simple

- Prescriptive instead of descriptive sequentiality
- Thus circumventing "spurious" causality problems
- Initialize-update-read protocol

However, precise definition of SC MoC not trivial

- Challenging to ensure conservativeness relative to Berry-constructiveness
- Plain initialize-update-read protocol does not accomodate, *e. g.*, signal re-emissions
- Restricting attention to *concurrent*, *non-confluent* node instances is key

Summary II

ASC-schedulability

- Is conservative approximation to SC
- Basis for practical implementation

Future work

- Plenty of it (SC+, optimized code gen, improved SCCharts transformations, . . .)
- Talk to us if you want to be part of it