CS 766 : Analysis of Concurrent Programs

Krishna. S

Outline

Operational Semantics of SC

Formal Semantics

- Here we present (small-step) **operational semantics**
- Allowed behavior is captured as runs of a Labeled Transition System (LTS)

Formal Semantics

- Here we present (small-step) **operational semantics**
- Allowed behavior is captured as runs of a Labeled Transition System (LTS)
- The states of the LTS capture some state information about the program
    - pc, register storage, "shared memory"

Formal Semantics

- Here we present (small-step) **operational semantics**
- Allowed behavior is captured as runs of a Labeled Transition System (LTS)
- The states of the LTS capture some state information about the program
  - pc, register storage, "shared memory"
- The transitions of the LTS capture execution steps
  - E.g., a thread reading from a local register, or a memory update

A Minimal Concurrent Programming Language

Domains:

$r \in \mathsf{Reg}$	local registers
$x \in \mathsf{Loc}$	shared-memory locations
$v \in \mathsf{Data}$	data domain
$i \in \{1, \ldots, k\}$	thread identifiers

A Minimal Concurrent Programming Language

Domains:

$r \in \mathrm{Reg}$	local registers
$x \in \mathrm{Loc}$	shared-memory locations
$v \in \mathrm{Data}$	data domain
$i \in \{1, \ldots, k\}$	thread identifiers

Expressions and commands:

$$\begin{aligned} e &::= \; r \mid v \mid e + e \mid e \cdot e \mid \ldots \\ c &::= \; \textbf{skip} \mid \textbf{If } e \textbf{ then } c \textbf{ else } c \mid \textbf{While } e \textbf{ do } c \mid \textbf{fence()} \\ &\quad\;\; c; c \mid r := e \mid r := x \mid x := e \mid r := \textbf{ARW } (x, e, e) \end{aligned}$$

A Minimal Concurrent Programming Language

Domains:

$r \in \mathrm{Reg}$	local registers
$x \in \mathrm{Loc}$	shared-memory locations
$v \in \mathrm{Data}$	data domain
$i \in \{1, \ldots, k\}$	thread identifiers

Expressions and commands:

$$
\begin{aligned}
e &::= \; r \mid v \mid e + e \mid e \cdot e \mid \ldots \\
c &::= \; \mathbf{skip} \mid \mathbf{If}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{While}\ e\ \mathbf{do}\ c \mid \mathbf{fence()} \\
&\quad\;\; c; c \mid r := e \mid r := x \mid x := e \mid r := \mathbf{ARW}\ (x, e, e)
\end{aligned}
$$

Concurrent programs: $P \colon i \mapsto c_i$, maps a program c_i to each thread i

Structure of the LTS

- A single thread executes steps $c, s \xrightarrow{\ell} c', s'$

Structure of the LTS

- A single thread executes steps $c, s \xrightarrow{\ell} c', s'$
  - c, c' is the thread's program source
  - s, s' is the thread's register storage

Structure of the LTS

- A single thread executes steps $c, s \xrightarrow{\ell} c', s'$
  - c, c' is the thread's program source
  - s, s' is the thread's register storage
  - ℓ specifies the instruction executed
    - $\ell = \epsilon$ if the thread takes a silent step, not interacting with shared memory
    - $\ell = R(x, v)/W(x, v)/ARW(x, v_1, v_2)$ if the thread interacts with shared memory

Structure of the LTS

- A single thread executes steps $c, s \xrightarrow{\ell} c', s'$
  - c, c' is the thread's program source
  - s, s' is the thread's register storage
  - ℓ specifies the instruction executed
    - $\ell = \epsilon$ if the thread takes a silent step, not interacting with shared memory
    - $\ell = R(x, v)/W(x, v)/ARW(x, v_1, v_2)$ if the thread interacts with shared memory
- A concurrent program chooses a thread to execute a step: $P, S \xrightarrow{i:\ell} P', S'$
  - P, P' is the concurrent program source
  - S, S' is the register storage of each thread
  - $i : \ell$ specifies that thread i executes instruction ℓ

Structure of the LTS

- A single thread executes steps $c, s \xrightarrow{\ell} c', s'$
  - c, c' is the thread's program source
  - s, s' is the thread's register storage
  - ℓ specifies the instruction executed
    - $\ell = \epsilon$ if the thread takes a silent step, not interacting with shared memory
    - $\ell = R(x, v)/W(x, v)/ARW(x, v_1, v_2)$ if the thread interacts with shared memory
- A concurrent program chooses a thread to execute a step: $P, S \xrightarrow{i:\ell} P', S'$
  - P, P' is the concurrent program source
  - S, S' is the register storage of each thread
  - $i : \ell$ specifies that thread i executes instruction ℓ
- The shared memory is defined as an LTS with transitions $M \xrightarrow{i:\ell} M'$
  - M, M' are states of the shared memory
  - Transitions depend on the **memory model**

Structure of the LTS

- A single thread executes steps $c, s \xrightarrow{\ell} c', s'$
  - c, c' is the thread's program source
  - s, s' is the thread's register storage
  - ℓ specifies the instruction executed
    - $\ell = \epsilon$ if the thread takes a silent step, not interacting with shared memory
    - $\ell = R(x, v)/W(x, v)/ARW(x, v_1, v_2)$ if the thread interacts with shared memory
- A concurrent program chooses a thread to execute a step: $P, S \xrightarrow{i:\ell} P', S'$
  - P, P' is the concurrent program source
  - S, S' is the register storage of each thread
  - $i : \ell$ specifies that thread i executes instruction ℓ
- The shared memory is defined as an LTS with transitions $M \xrightarrow{i:\ell} M'$
  - M, M' are states of the shared memory
  - Transitions depend on the **memory model**
- The behavior of the concurrent program is an LTS with transitions $P, S, M \Rightarrow P', S', M'$, capturing either
  - ϵ transitions which are internal for P, S or M, or
  - $i : \ell$ transitions on which threads interact with the shared memory

Thread Operations

Store: $s\colon \text{Reg} \to \text{Data}$

Initial store: $s = \lambda r.0$ for all i

States: $\langle c, s\rangle \in \text{Command} \times \text{Store}$

Transitions:

$$\frac{}{\langle \textbf{skip}; c, s\rangle \xrightarrow{\epsilon} \langle c, s\rangle} \qquad \frac{\langle c_1, s\rangle \xrightarrow{\ell} \langle c_1', s'\rangle}{\langle c_1; c_2, s\rangle \xrightarrow{\ell} \langle c_1'; c_2, s'\rangle} \qquad \frac{s' = s[r \mapsto s(e)]}{\langle r := e, s\rangle \xrightarrow{\epsilon} \langle \textbf{skip}, s'\rangle}$$

$$\frac{\ell = R(x, v), s' = s[r \mapsto v]}{\langle r := x, s\rangle \xrightarrow{\ell} \langle \textbf{skip}, s'\rangle} \qquad \frac{\ell = W(x, s(e))}{\langle x := e, s\rangle \xrightarrow{\ell} \langle \textbf{skip}, s\rangle}$$

$$\frac{s(e) \neq 0}{\langle \textbf{If}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\rangle \xrightarrow{\epsilon} \langle c_1, s\rangle} \qquad \frac{s(e) = 0}{\langle \textbf{If}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\rangle \xrightarrow{\epsilon} \langle c_2, s\rangle}$$

$$\frac{}{\langle \textbf{While}\ e\ \textbf{do}\ c, s\rangle \xrightarrow{\epsilon} \langle \textbf{If}\ e\ \textbf{then}\ c; \textbf{While}\ e\ \textbf{do}\ c\ \textbf{else}\ \textbf{skip}, s\rangle}$$

Thread Operations (Cont'd)

$$\frac{\ell = R(x, v), v \neq s(e_r)}{\langle r := \textbf{ARW}\ (x, e_r, e_w), s\rangle \xrightarrow{\ell} \langle \textbf{skip}, s[r \mapsto 0]\rangle}$$

$$\frac{\ell = ARW(x, s(e_r), s(e_w))}{\langle r := \textbf{ARW}\ (x, e_r, e_w), s\rangle \xrightarrow{\ell} \langle \textbf{skip}, s[r \mapsto 1]\rangle}$$

$$\frac{}{\langle \text{fence}, s\rangle \xrightarrow{F} \langle \textbf{skip}, s\rangle}$$

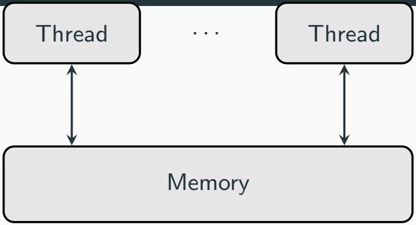
Concurrent Program Operations

State: $\langle P, S \rangle \in \text{Program} \times (\{1, \ldots, k\} \to \text{Store})$

Transition: Thread i makes a step

$$\frac{\langle P(i), S(i) \rangle \xrightarrow{\ell} \langle c, s \rangle}{\langle P, S \rangle \xrightarrow{i:\ell} \langle P[i \mapsto c], S[i \mapsto s] \rangle}$$

SC Memory

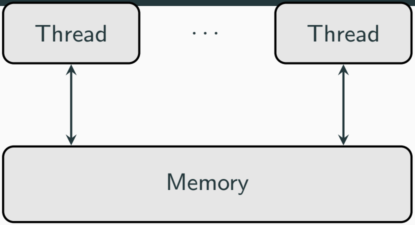


SC Memory State:

$M : \text{Loc} \rightarrow \text{Data}$

Initial state: $M_0 = \lambda x.0$

SC Memory



SC Memory State:

$M : \mathsf{Loc} \to \mathsf{Data}$

Initial state: $M_0 = \lambda x.0$

Transitions:

$$\textsc{Write}\quad \frac{\ell = W(x, v)}{M \xrightarrow{i:\ell} M[x \mapsto v]}$$

$$\textsc{Read}\quad \frac{\ell = R(x, v), M(x) = v}{M \xrightarrow{i:\ell} M}$$

$$\textsc{CAS}\quad \frac{\ell = ARW(x, v_r, v_w), M(x) = v_r}{M \xrightarrow{i:\ell} M[x \mapsto v_w]}$$

$$\textsc{Fence}\quad \frac{\ell = F}{M \xrightarrow{i:\ell} M}$$

Concurrent System Operations (Program + SC Memory)

State: $\langle P, S, M\rangle \in \text{Program} \times (\{1, \ldots, k\} \to \text{Store}) \times \text{Memory state}$

Transitions:

$$\textsc{Thread}\quad \frac{\langle P, S\rangle \xrightarrow{i:\epsilon} \langle P', S'\rangle}{\langle P, S, M\rangle \Rightarrow \langle P', S', M\rangle}$$

$$\textsc{Thread + Memory}\quad \frac{\langle P, S\rangle \xrightarrow{i:\ell} \langle P', S'\rangle, M \xrightarrow{i:\ell} M'}{\langle P, S, M\rangle \Rightarrow \langle P', S', M'\rangle}$$

Concurrent program and memory synchronize on non-silent transitions

Axiomatic Semantics of SC

Axiomatic Semantics General

- Axiomatic semantics are defined on program **executions**(traces)
- An execution contains sets of **events**, which capture the execution of program instructions
- Events are related with various **relations**, capturing, e.g., the order that they were executed, the write that a read is observing, etc.
- Axiomatic semantics are phrased as **rules (axioms)** that these relations must satisfy

Events

Labels

An event **label** is one of the following

$$R(x, v_r) \qquad W(x, v_w) \qquad ARW(x, v_r, v_w) \qquad F$$

where $x \in \mathrm{Loc}$ and $v_r, v_w \in \mathrm{Data}$

Events

Labels

An event **label** is one of the following

$$R(x, v_r) \qquad W(x, v_w) \qquad ARW(x, v_r, v_w) \qquad F$$

where $x \in \mathrm{Loc}$ and $v_r, v_w \in \mathrm{Data}$

Events

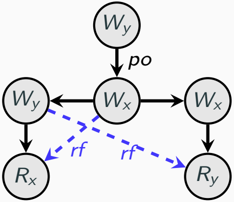
An **event** is a triple $\langle id, i, \ell \rangle$ where

- $id \in \mathbb{N}$ is an event id
- $i \in \{0, 1, \ldots, k\}$ is a thread id, and
- ℓ is a label

E.g., $\langle 47, 3, W(x, 1) \rangle$ means thread 3 executes event 47, which writes to shared location x the value 1

Execution Graphs

Program executions are represented as execution graphs



Execution Graphs

An **execution graph** is a tuple $G = \langle E, po, rf \rangle$, where:

- E is a set of events
- po is a partial order on E (called the program order)
- rf is a binary relation on E such that the following hold
  - For every $\langle w, r \rangle \in rf$
    - w has label $W(x, v)$ or $ARW(x, \cdot, v)$
    - r has label $R(x, v)$ or $ARW(x, v, \cdot)$
  - rf^{-1} is a function with domain all events labeled $R(\cdot, \cdot)$ or $ARW(\cdot, \cdot, \cdot)$

Modification and From-Read Orders

Modification Order

Given a variable x, a **modification order (on x)** mo_x is a total order on events $W(x, \cdot)$ and $ARW(x, \cdot, \cdot)$. The **modification order** is taken as $mo = \bigcup_x mo_x$.

Some papers call mo as coherence order and denote it by co

$$W_x \xrightarrow{mo} W_x \xrightarrow{mo} W_x$$

Modification and From-Read Orders

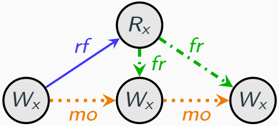
Modification Order

Given a variable x, a **modification order (on x)** mo_x is a total order on events $W(x, \cdot)$ and $ARW(x, \cdot, \cdot)$. The **modification order** is taken as $mo = \bigcup_x mo_x$.

Some papers call mo as coherence order and denote it by co

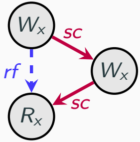
From-read Order

Given an mo, the **from-read** order is defined as $fr = rf^{-1}; mo \setminus [id]$.



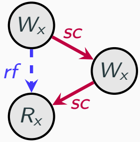
Axiomatic Semantics of SC

- SC-consistent executions admit a total ordering sc on $G.E$ such that
  - $(po \cup rf)^+ \subseteq sc$
  - The following pattern is not present



Axiomatic Semantics of SC

- SC-consistent executions admit a total ordering sc on $G.E$ such that
  - $(po \cup rf)^{+} \subseteq sc$
  - The following pattern is not present

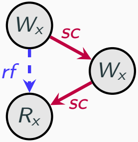


SC Consistency

An execution graph G is **SC-consistent** if there exists a modification order mo such that $po \cup rf \cup mo \cup fr$ is acyclic.

Axiomatic Semantics of SC

- SC-consistent executions admit a total ordering sc on $G.E$ such that
  - $(po \cup rf)^+ \subseteq sc$
  - The following pattern is not present



SC Consistency

An execution graph G is **SC-consistent** if there exists a modification order mo such that $po \cup rf \cup mo \cup fr$ is acyclic.

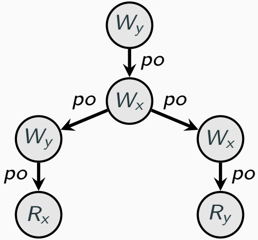
The two definitions are equivalent

Store Buffer Graphs under SC

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$ $a := R(x, \cdot)$	$\|\|$	$W(x, 1);$ $b := R(y, \cdot)$



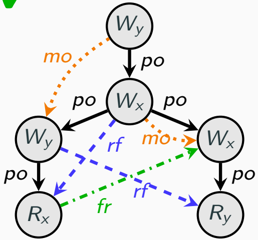
Store Buffer Graphs under SC

Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$	$\Vert$	$W(x, 1);$
$a := R(x, 0)$	$\Vert$	$b := R(y, 1)$

✓

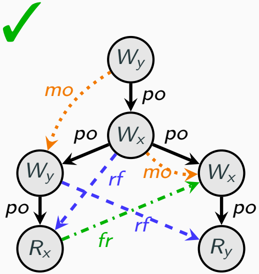


Store Buffer Graphs under SC

Store Buffer

$$x = 0, y = 0$$

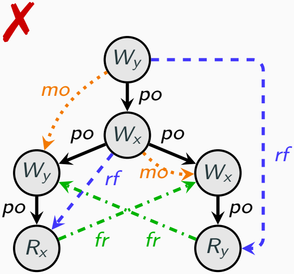
$W(y, 1);$ $a := R(x, 0)$	$\|\|$	$W(x, 1);$ $b := R(y, 1)$



Store Buffer

$$x = 0, y = 0$$

$W(y, 1);$ $a := R(x, 0)$	$\|\|$	$W(x, 1);$ $b := R(y, 0)$



Load Buffer

Load Buffer

$$x = 0, y = 0$$

$a := R(x, \cdot);$ $W(y, 1);$	$\|\|$	$b := R(y, \cdot);$ $W(x, 1);$

$$\varphi = (a = 1 \wedge b = 1)$$

Dekker's Protocol

Dekker's Protocol

$$x = 0, y = 0$$

$$\begin{array}{lcl} W(y,1); & & W(x,1); \\ a := R(x,\cdot); & \Big|\Big| & b := R(y,\cdot); \\ \text{if } (a = 0) & & \text{if } (b = 0) \\ CS_1 & & CS_2 \end{array}$$

$$\varphi = \neg(CS_1 \wedge CS_2)$$

2+2W

$$x = 0, y = 0$$

$$\begin{array}{lcl} W(x,1); & & W(y,1); \\ W(y,2); & \Big|\Big| & W(x,2); \\ a := R(y,\cdot); & & b := R(x,\cdot); \end{array}$$

$$\varphi = (a = 1 \land b = 1)$$

Oscillating

Oscillating

$$x = 0, y = 0$$

$$W(x,1); \quad \Bigg|\Bigg| \quad \begin{array}{l} a := R(x,\cdot); \\ b := R(x,\cdot); \\ c := R(x,\cdot); \end{array} \quad \Bigg|\Bigg| \quad W(x,2);$$

$$\varphi = (a = 1 \wedge b = 2 \wedge c = 1)$$

1R1W

1R1W

$$x = 0, y = 0$$

$$W(x, 1); \quad \Big|\Big| \quad \begin{array}{l} a := R(x, \cdot); \\ b := R(y, \cdot); \end{array} \quad \Big|\Big| \quad \begin{array}{l} c := R(y, \cdot); \\ d := R(x, \cdot); \end{array} \quad \Big|\Big| \quad W(y, 1);$$

$$\varphi = (a = 1 \wedge b = 0 \wedge c = 1 \wedge d = 0)$$

Runs and Execution Graphs

Given an execution graph $G = (E, po, rf, mo)$, construct a run ρ which "agrees" with G, and conversely.

Consistency Checking

Given a partial execution graph $\overline{G} = (E, po)$, synthesize the reads-from rf as well as modification order mo so that $\overline{G}$ can be extended to a full execution graph which is SC-consistent.

rf-Consistency Checking

Given a partial execution graph $\overline{G} = (E, po, rf)$, synthesize the modification order mo so that $\overline{G}$ can be extended to a full execution graph which is SC-consistent.

Runs and Execution Graphs

Given an execution graph $G = (E, po, rf, mo)$, construct a run ρ which "agrees" with G, and conversely.

Consistency Checking

Given a partial execution graph $\overline{G} = (E, po)$, synthesize the reads-from rf as well as modification order mo so that $\overline{G}$ can be extended to a full execution graph which is SC-consistent.

rf-Consistency Checking

Given a partial execution graph $\overline{G} = (E, po, rf)$, synthesize the modification order mo so that $\overline{G}$ can be extended to a full execution graph which is SC-consistent.

Runs and Execution Graphs

Given an execution graph $G = (E, po, rf, mo)$, construct a run ρ which "agrees" with G, and conversely.

Consistency Checking

Given a partial execution graph $\overline{G} = (E, po)$, synthesize the reads-from rf as well as modification order mo so that $\overline{G}$ can be extended to a full execution graph which is SC-consistent.

rf-Consistency Checking

Given a partial execution graph $\overline{G} = (E, po, rf)$, synthesize the modification order mo so that $\overline{G}$ can be extended to a full execution graph which is SC-consistent.