TECHNISCHE UNIVERSITÄT DRESDEN

Constructing and Verifying Cyber Physical Systems

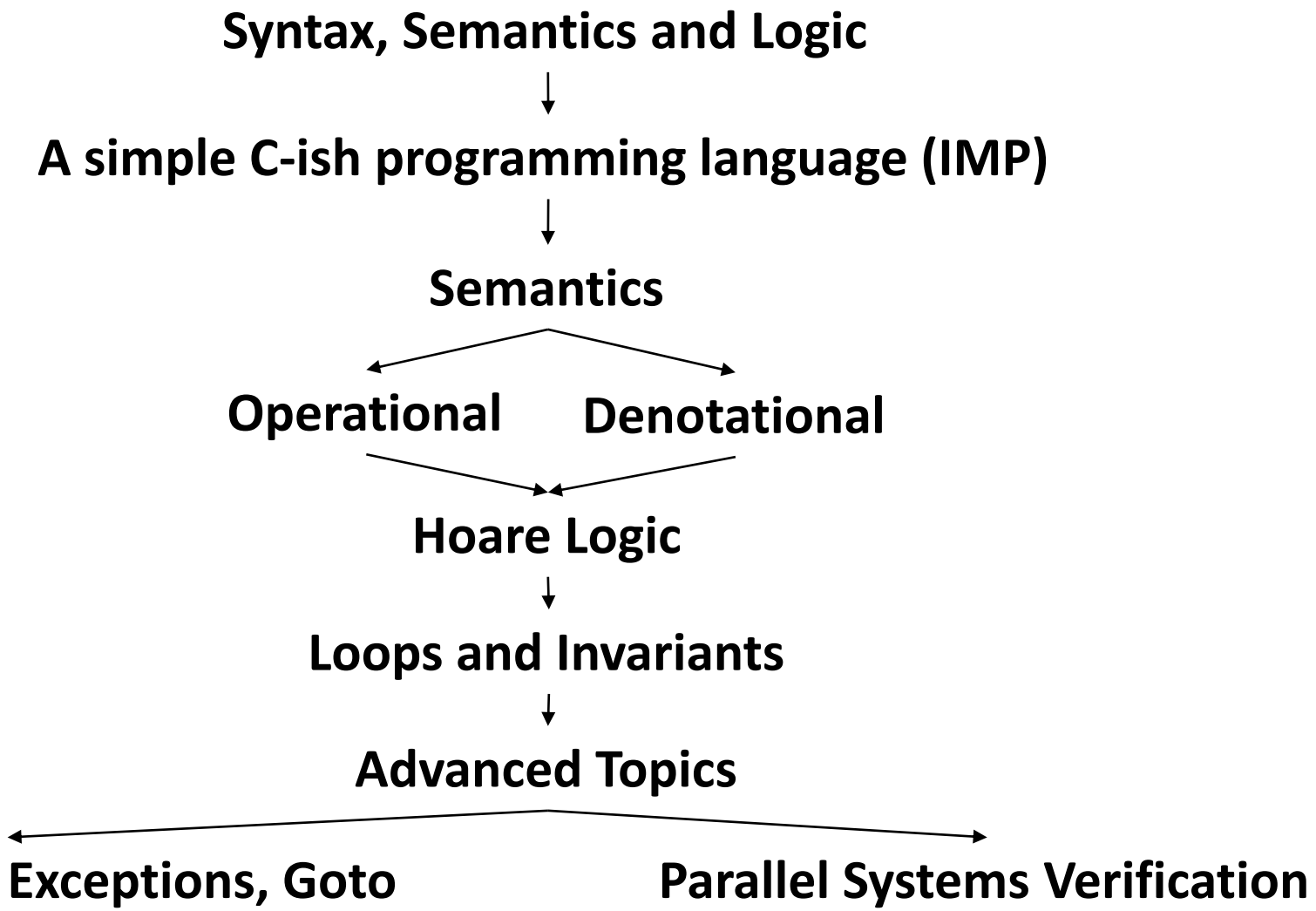
Program Verification

Marcus Völp

Overview

Introduction	
Mathematical Foundations (Differential Equations and Laplace Transformation)	Math
Control and Feedback	
Transfer Functions and State Space Models	Physics
Poles, Zeros / PID Control	Feedback Control
Stability, Root Locust Method, Digital Control	
Mixed-Criticality Scheduling and Real-Time Operating Systems (RTOS)	RTOS
Coordinating Networked Cyber-Physical Systems	CPS
Program Verification	
Differential Dynamic Logic and KeYmaera X	Verification
Differential Invariants	

Overview



Glynn Winskel: The Formal Semantics of Programming Languages

Example: ArduPilot Scheduler

```
typedef void (*task_fn_t)(void);

struct Task {
        task_fn_t function;
        uint16_t interval_ticks;
        uint16_t max_time_micros;
};

Task _tasks[num_tasks];

/* Scheduler
 *===========
 * run one tick; this will run as many scheduler tasks as we can in the specified time
 */
void AP_Scheduler::run(uint16_t time_available)
{
    uint32_t run_started_usec = hal.scheduler->micros();
    uint32_t now = run_started_usec;

    for (uint8_t i=0; i<_num_tasks; i++) {
        uint16_t dt = _tick_counter - _last_run[i];
        uint16_t interval_ticks = pgm_read_word(&_tasks[i].interval_ticks);
        if (dt >= interval_ticks) {
            // this task is due to run. Do we have enough time to run it?
            _task_time_allowed = pgm_read_word(&_tasks[i].max_time_micros);

            if (dt >= interval_ticks*2) {
                // we've slipped a whole run of this task!
                ...
                }
            }
```

```
            if (_task_time_allowed <= time_available) {
                // run it
                _task_time_started = now;
                task_fn_t func = (task_fn_t)pgm_read_pointer(&_tasks[i].function);
                current_task = i;
                func();
                current_task = -1;

                // record the tick counter when we ran. This drives
                // when we next run the event
                _last_run[i] = _tick_counter;


                // work out how long the event actually took
                now = hal.scheduler->micros();
                uint32_t time_taken = now - _task_time_started;

                if (time_taken > _task_time_allowed) {
                    // the event overran!
                    ...
                }
                if (time_taken >= time_available) {
                    ...
                }
                time_available -= time_taken;
            }
        }
    }
```

Syntax, Semantics and Logic

Syntax

Rules how to write down the program text. Often expressed as grammar.

Grammar: (Σ, N, P, S)

- finite set of terminal symbols: $\Sigma \coloneqq \{0|1|\ldots|\text{true}|\text{false}|\text{x}|\ldots|==|=|+|-|*|!|;|\text{if}|\text{while}|)|(|'\{'|'\}'\}$
- finite set of nonterminal symbols: $N \coloneqq \{B, I, S\}$
- finite set of production rules: P
  - $B \rightarrow true \mid false \mid !B \mid B == B \mid I == I$
  - $I \rightarrow no^* \mid I + I \mid -I \mid (I) \mid I * I \mid var$
  - $S \rightarrow var = I \mid \text{S; S} \mid \text{if (B)\{S\}} \mid \text{while(B)\{S\}}$
- distinguished start symbol: S

Parsing: the determining of production rules that have lead to a given program.

Example

$B \rightarrow true \mid false \mid !B \mid B == B \mid I == I$

$I \rightarrow no^* \mid I + I \mid -I \mid (I) \mid I * I \mid var$

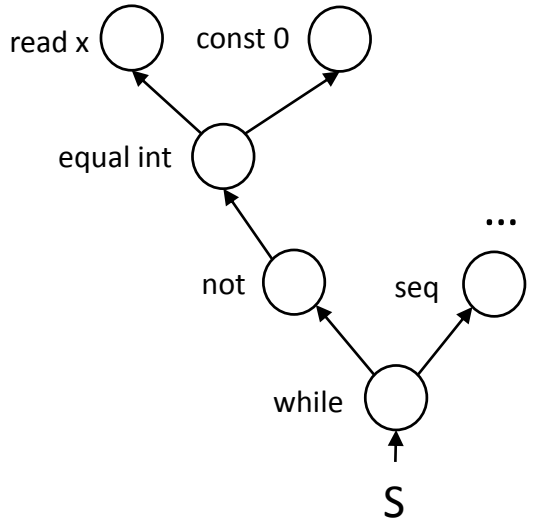
$S \rightarrow var = I \mid S; S \mid \text{if } (B)\{S\} \mid \text{while}(B)\{S\}$

$$\dfrac{\text{read x}\ \dfrac{*}{I \rightarrow x} \quad \dfrac{*}{I \rightarrow 0}}{\text{equal int}\ \dfrac{}{B \rightarrow x==0}} \qquad \dfrac{\dfrac{*}{I \rightarrow y} \quad \dfrac{*}{I \rightarrow 1}}{I \rightarrow y + 1} \qquad \dfrac{\dfrac{*}{I \rightarrow x} \quad \dfrac{\dfrac{*}{I \rightarrow 1}}{I \rightarrow -1}}{I \rightarrow x + (-1)}$$

$$\dfrac{\text{not}\ \dfrac{}{B \rightarrow !x==0} \qquad \dfrac{}{S \rightarrow y = y + 1} \quad \dfrac{}{S \rightarrow x = x + (-1)}}{\text{while}\ \dfrac{}{S \rightarrow y = y + 1;\quad x = x + (-1)}}$$

$$S \rightarrow \text{while } (!x==0)\ \{\ \ y = y + 1;\quad x = x + (-1)\ \}$$

Syntax Tree



Semantics

$B \rightarrow true \mid false \mid !B \mid B == B \mid I == I$

$$[\![B]\!]_{bexp} \rightarrow Bool$$

$$[\![true]\!]_{bexp} = true$$
$$[\![false]\!]_{bexp} = false$$
$$[\![!\,B]\!]_{bexp} = \begin{cases} true & if\ [\![B]\!]_{bexp} \rightarrow false \\ false & otherwise \end{cases}$$

...

Semantics

$$I \rightarrow no^* \mid I + I \mid -I \mid (I) \mid I * I \mid var$$

$$[\![I]\!]_{exp} \rightarrow \mathbb{Z}$$

$$[\![42]\!]_{exp} = 42$$
$$[\![42 + 23]\!]_{exp} = [\![42]\!]_{exp} + [\![23]\!]_{exp} \ \% \ MAX_{INT}$$
$$\dots$$

State $s : var \rightarrow \mathbb{Z}$

$$[\![I]\!]_{exp} \rightarrow State \rightarrow \mathbb{Z}$$

$$[\![x]\!]_{exp}(s) = s(x)$$

S

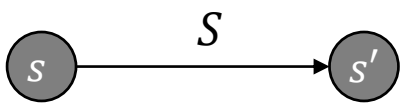
Syntax, Semantics and Logic

Semantics

$B \rightarrow true \mid false \mid !B \mid B == B \mid I == I$

$I \rightarrow no^* \mid I + I \mid -I \mid (I) \mid I * I \mid var$

$S \rightarrow var = I \mid S; S \mid \text{if (B)\{S\}} \mid \text{while(B)\{S\}}$



$$[\![S]\!] \rightarrow State \rightarrow State$$

$$[\![x = I]\!](s) = s \mapsto x = [\![I]\!]_{exp}(s)$$

$$[\![S_0; S_1]\!](s) = [\![S_1]\!]\big([\![S_0]\!](s)\big)$$

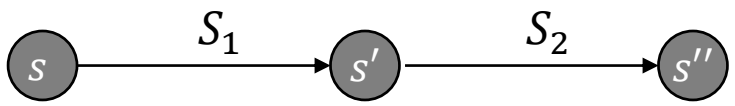
$$[\![if\,(B)\{S\}]\!](s) = \begin{cases} [\![S]\!](s) & if\,[\![B]\!]_{exp}(s) = true \\ s & otherwise \end{cases}$$

...

Semantics: Meaning or behavior of the program (or part of it)

(Program) Logic

Means of reasoning about the behavior of programs.



P $\qquad\qquad\qquad\qquad$ Q

$$seq\ \frac{\{P\}S_1\{R\},\quad \{R\}S_2\{Q\}}{\{P\}S_1;S_2\{Q\}}$$

$$assign\ \frac{}{\{P[Exp/x]\}\ x = Exp\ \{P\}}$$

$$consequence\ \frac{P \Longrightarrow P',\quad \{P'\}S\{Q'\},\quad Q' \Longrightarrow Q,}{\{P\}S\{Q\}}$$

(Program) Logic

Means of reasoning about the behavior of programs.

$$\dfrac{\dfrac{\dfrac{*}{\{y>1\wedge x=5\}\Rightarrow\{y*x>1\wedge x=5\wedge y*x>x\}}\quad\dfrac{*}{\{y*x>1\wedge x=5\wedge y*x>x\}\ y=y*x\ \{y>1\wedge x=5\wedge y>x\}}\quad\dfrac{*}{\{y>1\wedge x=5\wedge y>x\}\Rightarrow\{y>x\}}}{\{y>1\ \wedge x=5\}\ y=y*x\ \{y>x\}}}{\dots}$$

$$\dfrac{\dfrac{\dfrac{*}{\{y>1\}\Rightarrow\{y>1\wedge 5=5\}}\quad\dfrac{*}{\{y>1\wedge 5=5\}\ x=5\ \{y>1\wedge x=5\}}}{\{y>1\}\ x=5\ \{y>1\ \wedge x=5\}}\quad\dfrac{*}{\{y>1\wedge x=5\}\Rightarrow\{y>1\wedge x=5\}}\quad,\quad\dfrac{\dots}{\{y>1\ \wedge x=5\}\ y=y*x\ \{y>x\}}}{\{y>1\}\ x=5;y=y*x\ \{y>x\}}$$

$$seq\ \frac{\{P\}S_1\{R\},\quad\{R\}S_2\{Q\}}{\{P\}S_1;S_2\{Q\}}$$

$$\text{assign}\ \frac{*}{\{P[Exp/x]\}\ x=Exp\ \{P\}}$$

$$consequence\ \frac{P\Longrightarrow P',\quad\{P'\}S\{Q'\},\quad Q'\Longrightarrow Q,}{\{P\}S\{Q\}}$$

IMP: a C-ish language

TECHNISCHE UNIVERSITÄT DRESDEN

Why Imp?

- C blurs the concepts of expressions and statements:
  - Statement: control flow and side effects
  - Expression: side-effect free execution
  - But:
    - x = 5 is an expression in C (e.g., if (x = 5 < 6) …)
    - y = x <5 ? -1 : 3 * x; is control flow inside an expression
- Starting easy with State s : Var -> Value
  - C memory model:
    - State s: Address -> Byte
    - Interpretation of bytes when reading with a certain type
      e.g., read_int(Address a) = int_from_bytes(s[a], s[a+1], s[a+2], s[a+3])
- No structure type
  - C allows complicated to formalize type construction using incomplete structs
    - e.g. struct A; struct B {A * a;}; struct A {B *b;};

IMP

Glynn Winskel: The Formal Semantics of Programming Languages

Types:

- numbers $\mathbb{Z}$: positive and negative integers, including zero (m,n, l, r)
- truth values $\boldsymbol{\mathcal{B}} = \{\text{true}, \text{false}\}$
- variables / locations **Loc** (x, y)
- arithmetic expressions **Aexp** $(a_0, a_1, \ldots, a_n)$
- boolean expressions **Bexp** $(b_0, b_1, \ldots, b_n)$
- statements / commands **Com** $(c_0, c_1, \ldots, c_n)$

Arithmetic expressions (Aexp): $a ::= n \mid X \mid a_0 + a_1 \mid -a_0 \mid a_0 * a_1$

Boolean expressions (Bexp): $b ::= true \mid false \mid a_0 = a_1 \mid a_0 < a_1 \mid \neg b_0 \mid b_0 \wedge b_1$

Commands (Com): $c ::= skip \mid x := a \mid c_0; c_1 \mid \textbf{if } (b) \textbf{ then } \{c_0\} \textbf{ else } \{c_1\} \mid \textbf{while } (b) \{c\}$

State: s : Loc -> Value vs. Loc -> Type x Value

(here: not required; cannot store Booleans)

Denotational vs. Operational Semantics

Denotational Semantics:

Construct mathematical objects (denotations) that describe the meaning of expressions

$$[\![n]\!]: \mathbb{Z} = n \qquad [\![a_0 + a_1]\!]: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} = \lambda l.\, \lambda r.\, l + r$$

$$[\![3 + 5]\!] = [\![3]\!] + [\![5]\!] = 3 + 5 = 8$$

Operational Semantics

Describe how programs compute

- Structural operational semantics (small step): how to compute individual steps

  e.g., $\dfrac{[\![C_1,s]\!] \to s'}{[\![C_1;C_2,s]\!] \to [\![C_2,s']\!]} \qquad \dfrac{[\![C_1,s]\!] \to [\![C'_1,s']\!]}{[\![C_1;C_2,s]\!] \to [\![C'_1;C_2,s']\!]} \qquad \dfrac{}{[\![skip,s]\!] \to s}$

- Natural semantics (big step): how to obtain the overall result

  e.g., $\dfrac{[\![C_1,s]\!] \to s', \quad [\![C_2,s']\!] \to s''}{[\![C_1;C_2,s]\!] \to s''}$

Operational Semantics

Natural Semantics (Big Step):

	natural semantics	
const (bool / arith):	$\frac{}{[\![c,s]\!]\to c}$	where $c \in \{n, true, false\}$
unary expression (bool / arith):	$\frac{[\![b,s]\!]\to n}{[\![\circ b,s]\!]\to \circ n}$	where $\circ \in \{-, \neg\}$ (side effects: $\frac{[\![b,s]\!]\to (n,s')}{[\![\circ b,s]\!]\to (\circ n,s')}$)
binary expression (bool / arith):	$\frac{[\![a,s]\!]\to m \quad [\![b,s]\!]\to n}{[\![a \circ b,s]\!]\to m \circ n}$	where $\circ \in \{+, \wedge, *, =, <\}$
read:	$\frac{s(x)=n}{[\![x,s]\!]\to n}$	(! Reading device registers may cause side effects)
skip:	$\frac{}{[\![skip,s]\!]\to s}$	
assign:	$\frac{[\![a,s]\!]\to m}{[\![x:=a,s]\!]\to s[x \backslash \mathrm{m}]}$	
seq:	$\frac{[\![c_0,s]\!]\to s', [\![c_1,s']\!]\to s''}{[\![c_0;c_1, s]\!]\to s''}$	
if:	$\frac{[\![b,s]\!]\to true, \quad [\![c_0,s]\!]\to s'}{[\![\textbf{if } (b)\{c_0\} \textbf{ else } \{c_1\},s]\!]\to s'}$ $\frac{[\![b,s]\!]\to false, \quad [\![c_1,s]\!]\to s'}{[\![\textbf{if } (b)\{c_0\} \textbf{ else } \{c_1\},s]\!]\to s'}$	
while: (later)		

Operational Semantics

Structural Operational Semantics (Small Step):

	natural semantics	small step
const (bool / arith):	$\frac{}{[\![c,s]\!]\to c}$	(same)
unary expression (bool / arith):	$\frac{[\![b,s]\!]\to n}{[\![\circ b,s]\!]\to \circ n}$	$\frac{}{[\![\circ b,s]\!]\to[\![b;\lambda n.\circ n,s]\!]}$, …, $\frac{\circ n=m}{[\![\circ n,s]\!]\to(s,m)}$
binary expression (bool / arith):	$\frac{[\![a,s]\!]\to m \quad [\![b,s]\!]\to n}{[\![a\circ b,s]\!]\to m\circ n}$	$\frac{}{[\![a\circ b,s]\!]\to[\![a;\lambda n.n\circ b,s]\!]}$, …, $\frac{}{[\![n\circ b,s]\!]\to[\![b;\lambda m.n\circ m,s]\!]}$
read:	$\frac{s(x)=n}{[\![x,s]\!]\to n}$	(same) !atomicity of reads
skip:	$\frac{}{[\![skip,s]\!]\to s}$	(same)
assign:	$\frac{[\![a,s]\!]\to m}{[\![x:=a,s]\!]\to s[x\backslash \mathrm{m}]}$	(same) !atomicity
seq:	$\frac{[\![c_0,s]\!]\to s', [\![c_1,s\prime]\!]\to s''}{[\![c_0;c_1,s]\!]\to s''}$	$\frac{[\![c_0,s]\!]\to s'}{[\![c_0;c_1,s]\!]\to[\![c_1,s\prime]\!]}$, $\frac{[\![c_0,s]\!]\to[\![c_0',s\prime]\!]}{[\![c_0;c_1,s]\!]\to[\![c_0\prime;c_1,s\prime]\!]}$, $\frac{[\![a_0,s]\!]\to(s',m)}{[\![a_0;\lambda\, n.a_1(n),s]\!]\to[\![a_1(m),s\prime]\!]}$
if:	$\frac{[\![b,s]\!]\to true, \quad [\![c_0,s]\!]\to s\prime}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to s\prime}$ $\frac{[\![b,s]\!]\to false, \quad [\![c_1,s]\!]\to s\prime}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to s\prime}$	$\frac{}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to[\![\mathrm{b};\ \boldsymbol{\lambda}\ \mathbf{r}.\mathbf{if}\ (r)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]}$, …, $\frac{r=true}{[\![\mathbf{if}\ (r)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to[\![c_0,s]\!]}$ $\frac{}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to[\![\mathrm{b};\ \boldsymbol{\lambda}\ \mathbf{r}.\mathbf{if}\ (r)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]}$, …, $\frac{r=false}{[\![\mathbf{if}\ (r)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to[\![c_1,s]\!]}$
while: (later)		

Denotational Semantics

Denotational

	natural semantics	small step	denotational
const (bool / arith):	$\frac{}{[\![c,s]\!]\to c}$	(same)	$[\![c]\!](s) = c$
unary expression (bool / arith):	$\frac{[\![b,s]\!]\to n}{[\![\circ b,s]\!]\to \circ n}$	$\frac{}{[\![\circ b,s]\!]\to[\![b;\lambda n.\circ n,s]\!]}$	$[\![\circ\, b]\!](s) = \circ\, [\![b]\!](s)$
binary expression (bool / arith):	$\frac{[\![a,s]\!]\to m \quad [\![b,s]\!]\to n}{[\![a\circ b,s]\!]\to m\circ n}$	$\frac{}{[\![a\circ b,s]\!]\to[\![a;\lambda n.n\circ b,s]\!]}$	$[\![a \circ b]\!](s) = [\![a]\!](s) \circ [\![b]\!](s)$
read:	$\frac{s(x)=n}{[\![x,s]\!]\to n}$	(same)	$[\![x]\!](s) = s(x)$
skip:	$\frac{}{[\![skip,s]\!]\to s}$	(same)	$[\![skip]\!] = \{(s,s)\}$
assign:	$\frac{[\![a,s]\!]\to m}{[\![x:=a,s]\!]\to s[x\backslash \mathrm{m}]}$	(same)	$[\![x := a]\!] = \{(s,s') \mid s' = s\,[x\backslash [\![a]\!](s)]\}$
seq:	$\frac{[\![c_0,s]\!]\to s',\ [\![c_1,s\prime]\!]\to s''}{[\![c_0;c_1,s]\!]\to s''}$	$\frac{[\![c_0,s]\!]\to s'}{[\![c_0;c_1,s]\!]\to[\![c_1,s\prime]\!]},\ \frac{[\![c_0,s]\!]\to[\![c_0',s\prime]\!]}{[\![c_0;c_1,s]\!]\to[\![c_0\prime;c_1,s\prime]\!]}$	$[\![c_0; c_1]\!] = [\![c_1]\!] \circ [\![c_0]\!]$
if:	$\frac{[\![b,s]\!]\to true,\ [\![c_0,s]\!]\to s\prime}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to s\prime}$ $\frac{[\![b,s]\!]\to false,\ [\![c_1,s]\!]\to s\prime}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to s\prime}$	$\frac{}{[\![\mathbf{if}\ (b)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]\to[\![\mathrm{b};\ \boldsymbol{\lambda}\ \mathbf{r}.\mathbf{if}\ (r)\{c_0\}\ \mathbf{else}\ \{c_1\},s]\!]}$	$[\![if\,(b)\{c_0\}\ else\ \{c_1\}, s]\!] =$ $\{(s, s') \in [\![c_0]\!] \mid [\![b]\!](s) = t\} \cup$ $\{(s, s') \in [\![c_1]\!] \mid [\![b]\!](s) = f\}$
while: (later)			

Hoare Logic

Logic for reasoning about program safety / lifeness:

1969 C.A.R. Hoare

precondition: What holds initially?

postcondition: What holds afterwards?

$$\{P\}\ C\ \{Q\}$$

program

skip: $\frac{}{\{P\}\ skip\ \{P\}}$

seq: $\frac{\{P\}\ c_0\ \{Q\},\{Q\}\ c_1\ \{R\}}{\{P\}\ c_0; c_1\ \{R\}}$

cond: $\frac{\{P \wedge B\}\ c_0\ \{Q\},\{P \wedge \neg B\}\ c_1\ \{Q\}}{\{P\}\ if\ (B)\ \{c_0\}\ else\ \{c_1\}\ \{Q\}}$

assign: $\frac{}{\{P[x\backslash \mathrm{E}]\}\ x := E\ \{P\}}$ ⬅ if P also holds if we replace x with E, we can "ignore" the assignment and use P[x\E]

consequence: $\frac{P \Rightarrow P', \{P\prime\}\ c_0\ \{Q\prime\}\ Q' \Rightarrow Q}{\{P\}\ c_0\ \{Q\}}$

Soundness:

$$\forall\ s, s', c, P, Q.\ if\ \underbrace{\{P\}c\{Q\}}\ and\ [\![c, s]\!] = s'\ then\ P(s) \Rightarrow Q(s')$$

all goals close when applying hoare rules.

Loops and Invariants

Structural Operational Semantics

```
while(cond) {                    if (cond) {
    stmt                             stmt; while (cond) { stmt }
}                                }
```

But what if the loop never terminates (i.e., cond never becomes false)?

There is no final state s' and we cannot ask whether the predicate Q is true on s'.

1) assume termination and show correctness under this assumption: partial correctness
2) prove termination to obtain total correctness

Operational Semantics (SOS)

$$\frac{}{[\![while(B)\{c\}]\!] \rightarrow [\![if\ (B)\ \{c\ ;while(B)\{c\}\}\ else\ \{skip\}]\!]}$$

How to capture the behavior of while in one step?

Loops and Invariants

Natural Semantics (Operational big step semantics):

while(cond) {

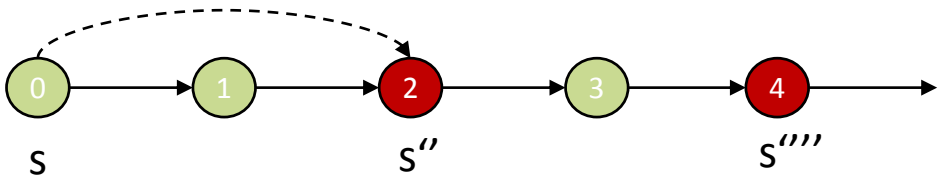
    stmt

}

How to capture the behavior of while in one step?

helper function:

$$iterate\ while(c,n)(s) = \begin{cases} c; iterate\ while(c,n-1)(s) & if\ n > 0 \\ s & otherwise \end{cases}$$



$$[\![while(B)\{c\}, s]\!] = \begin{cases} \min_n\{s' \mid s' = iterate\ while(C,n)(s), [\![B,s']\!] \to false\} & if\ \exists\, n. [\![B, iterate\ while(C,n)(s)]\!] \to false \\ hang & otherwise \end{cases}$$

Loops and Invariants

Denotational semantics

while(cond) {

    stmt

}

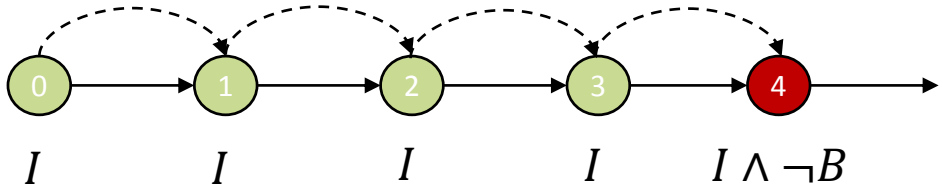
$$[\![while(B)\{c\}]\!] = fix(\Gamma(\varphi)) \text{ where}$$

$fix(\Gamma(\varphi))$ is the least fixed point of $\Gamma(\varphi)$, i.e., where $\varphi = \Gamma(\varphi)$

$$\Gamma(\varphi) = \{(s,s') | [\![B]\!]s = true \wedge (s,s') \in \varphi \circ [\![c]\!]\} \cup \{(s,s) | [\![B]\!]s = false\}$$

Loops and Invariants

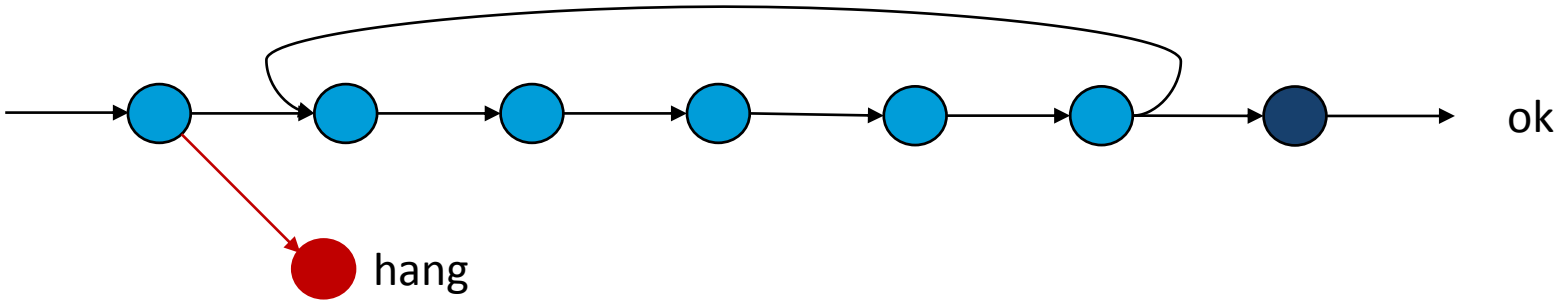
Hoare Rule and Invariants



iteration: $\dfrac{\{I \wedge B\}\, c_0\, \{I\}}{\{I\}\, while(B)\{c_0\}\, \{I \wedge \neg B\}}$

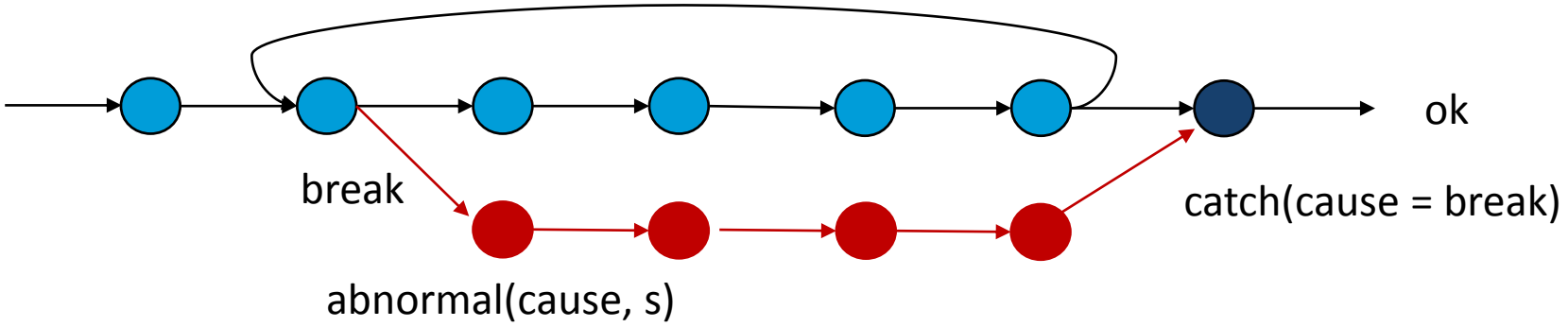
Break, Continue, Exceptions, Goto

Normal execution



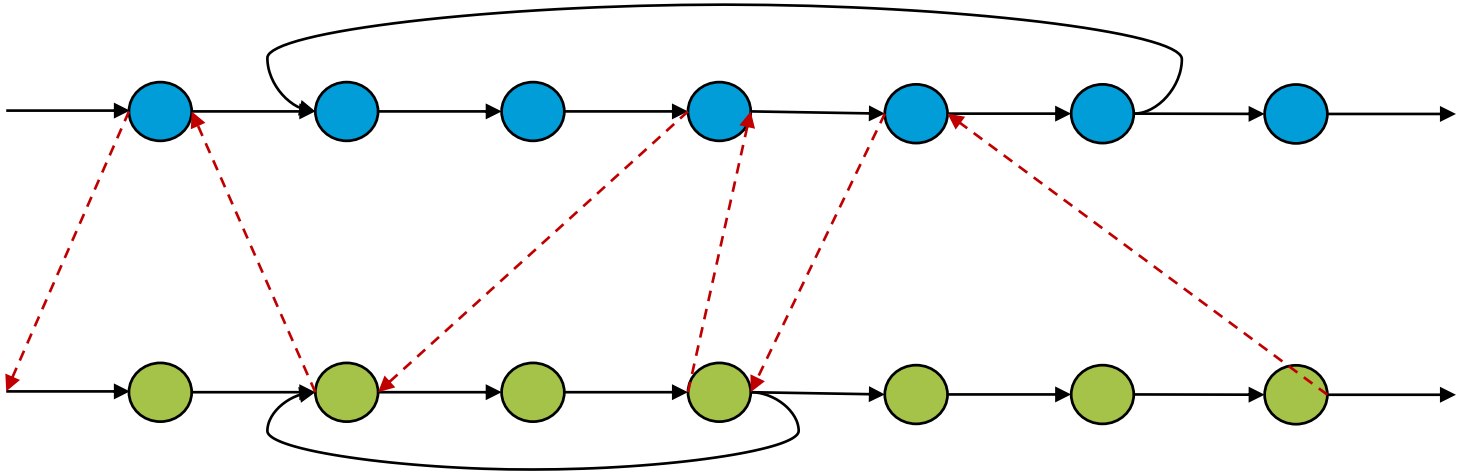
$$[\![while(B)\{c\}, s]\!] = \begin{cases} \min_n\{s' \mid s' = iterate\ while(C,n)(s), [\![B, s']\!] \to false\} & if\ \exists\, n.\, [\![B, iterate\ while(C,n)(s)]\!] \to false \\ hang & otherwise \end{cases}$$

Abnormal execution



Parallel Systems Verification

in a microscopic nutshell



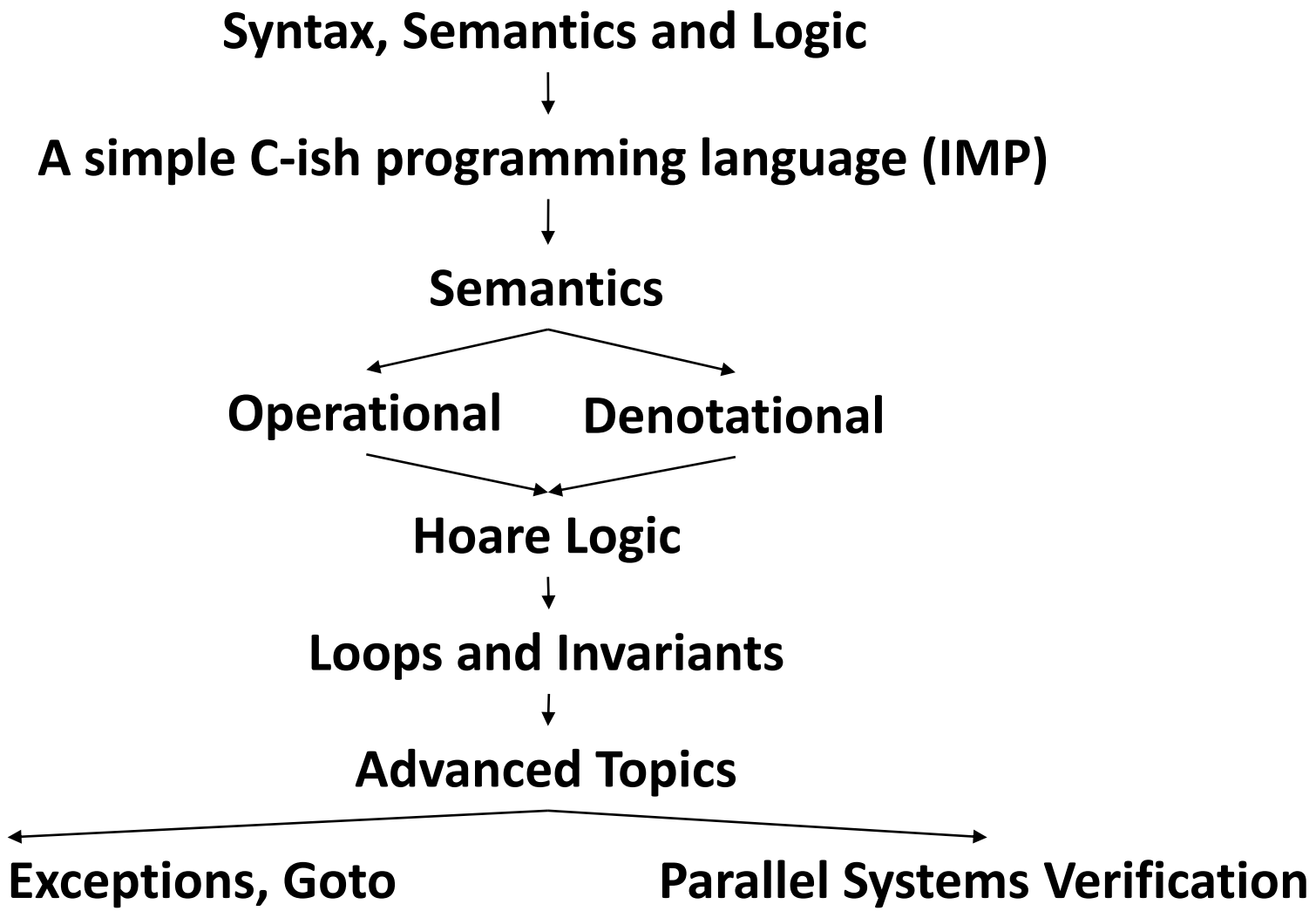
interference free: don't write variables read by other process
$\Rightarrow$ all interleavings are equivalent (in particular to seq. execution)

$$\frac{A: \{P_A, R_A\} c_A \{G_A, Q_A\}, G_A \Rightarrow P_B \quad B: \{P_B, R_B\} c_B \{G_B, Q_B\}, G_B \Rightarrow P_A}{\{P_A \wedge P_B, R_A \wedge R_B\} c_A || c_B \{G_A \wedge G_B, Q_A \wedge Q_B\}}$$

rely guarantee:
A: I rely on B not to do bad things. I guarantee B not to do bad things.
B: I rely on A not to do bad things. I guarantee A not to do bad things.

rely guarantee

Overview



Glynn Winskel: The Formal Semantics of Programming Languages

Overview

Introduction	
Mathematical Foundations (Differential Equations and Laplace Transformation)	Math
Control and Feedback	
Transfer Functions and State Space Models	Physics
Poles, Zeros / PID Control	Feedback Control
Stability, Root Locust Method, Digital Control	
Mixed-Criticality Scheduling and Real-Time Operating Systems (RTOS)	RTOS
Coordinating Networked Cyber-Physical Systems	CPS
Program Verification	
Differential Dynamic Logic and KeYmaera X	Verification
Differential Invariants	

KeYmaeraX.org (+ see Email)