Universität Bremen

dfki Deutsches Forschungszentrum für Künstliche Intelligenz
German Research Center for Artificial Intelligence

Systems of High Safety and Security

Lecture 10 from 07.01.2026: Overall Correctness

Winter term 2025/26

Christoph Lüth

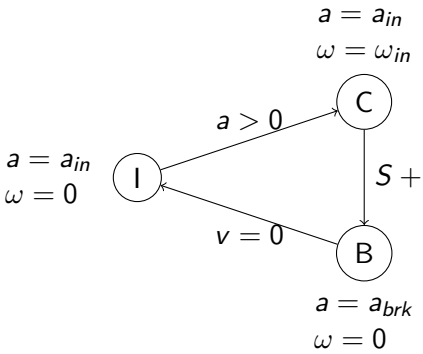
Frohes Neues Jahr!

Roadmap

- Introduction
- Legal Requirements - Norms and Standards
- The Development Process
- Hazard Analysis
- The Big Picture: Hybrid Systems
- Temporal Logic with LTL and CTL
- Operational Semantics
- Axiomatic Semantics - Specifying Correctness
- Floyd-Hoare Logic - Proving Correctness
- A Simple Compiler and its Correctness
- TinyRV32I
- Hardware Verification & Conclusions

General Correctness

Abstract Model



- System state $\mathcal{X} = \langle x, y, v, a, \phi, \omega \rangle$
- Input $I = \langle a_{in}, \omega_{in}, L \rangle$
- State transition: $\varphi : \mathcal{X} \times I \to \mathcal{X}$

- New state $\varphi(\mathcal{X}, I)$ given by

$$\begin{aligned} \phi' &= \phi + \omega \\ x' &= x + s \cdot \cos(\phi') \\ y' &= y + s \cdot \sin(\phi') \\ v' &= v + a \cdot \Delta T \end{aligned}$$

- Safety invariant SI:
  $S + s_{brk}(v) < \min(L)$

Concrete Model I

The state in C:

```
enum mode_t {STATE_INITIAL, STATE_CRUISING,

struct state_t {
  enum mode_t mode;
  double x;
  double y;
  double v;
  double phi;
  double omega;
  double a;
  };

struct input_t {
  double a_in;
  double w_in;
  double L[NUM_DIST];
};
```

The transition function:

```
struct state_t state =
{STATE_INITIAL, 0, 0, 0, 0, 0, 0};

void tick(struct input_t i)
{
  if (state.mode == STATE_INITIAL) {
    ...
  } else if (state.mode == STATE_CRUISING) {
    ...
  } else if (state.mode == STATE_BRAKING) {
    ...
  }
  /* update state */
  ...
}
```

Concrete Model II

```
void loop()
{
  struct input_t i;

  while (1) {
    i.a_in= read_acceleration();
    i.w_in= read_steering();
    read_sensors(i.L);
    tick(i);
    set_acceleration(state.a);
    set_steering(state.phi);
    sleep_ms(DELTA_T);
   }
}
```

- Problem?

Concrete Model II

```
void loop()
{
  struct input_t i;

  while (1) {
    i.a_in= read_acceleration();
    i.w_in= read_steering();
    read_sensors(i.L);
    tick(i);
    set_acceleration(state.a);
    set_steering(state.phi);
    sleep_ms(DELTA_T);
   }
}
```

- Problem? Non-Terminating Loop.
- Want to show: Safety invariant *SI* **always** holds

Formal Proof

- FSM $\mathcal{M} = \langle \Sigma, \rightarrow, I \rangle$ and $\phi : \Sigma \rightarrow \mathbb{B}$ is a **Kripke Structure**.
- Want to show: $\mathcal{M} \models \Box\phi$ for a state-dependent propositiopnal formula $\phi : \Sigma \rightarrow \mathbb{B}$
- Show:
  1. If $\sigma \in I$ then $\phi(\sigma)$
  2. If $\phi(\sigma)$ and $\sigma \rightarrow \sigma'$ then $\phi(\sigma')$
- In our case:
  1. $SI(s)$ for initial state s.
  2. $\vdash \{SI\}\ tick(I)\ \{SI\}$

Compiling Programs

A Simple Stack Machine

Instr ::= LOADI $\mathbb{Z}$ | LOAD **Var** | ADD | STORE **Var** | JMP $\mathbb{Z}$ | JMPLE $\mathbb{Z}$ | JMPGE $\mathbb{Z}$

- A **program** is a list (sequence) of instructions Seq(**Instr**)
- The **state** is a finite map $\Sigma = \mathbf{Var} \rightharpoonup \mathbb{Z}$ (as before)
- A **configuration** has a program counter, the state, and a stack: $\mathit{Config} = \mathbb{Z} \times \Sigma \times \mathit{Stack}$

Semantics for the Stack Machine

Single instructions:

$$
\begin{aligned}
\text{iexec}(\text{LOADI } n, \langle i, \sigma, s\rangle) &= \langle i+1, \sigma, \text{push}(n, s)\rangle \\
\text{iexec}(\text{LOAD } x, \langle i, \sigma, s\rangle) &= \langle i+1, \sigma, \text{push}(\sigma(x), s)\rangle \\
\text{iexec}(\text{ADD}, \langle i, \sigma, s\rangle) &= \langle i+1, \sigma, \text{push}(\text{top}(\text{pop}(s)) + \text{top}(s), \text{pop}(\text{pop}(s)))\rangle \\
\text{iexec}(\text{STORE } n, \langle i, \sigma, s\rangle) &= \langle i+1, \sigma[n \mapsto \text{top}(s)], \text{pop}(s)\rangle \\
\text{iexec}(\text{JMP } n, \langle i, \sigma, s\rangle) &= \langle i+1+n, \sigma, s\rangle \\
\text{iexec}(\text{JMPLE } n, \langle i, \sigma, s\rangle) &= \begin{cases} \langle i+1+n, \sigma, s\rangle & \text{top}(\text{pop}(s)) < \text{top}(s) \\ \langle i+1, \sigma, s\rangle & \textit{otherwise} \end{cases}
\end{aligned}
$$

Semantics: Instruction Sequences

- One instruction in a sequence:

$$\langle P, \langle i, \sigma, s \rangle \rangle \to c' \iff c' = \mathsf{iexec}(P!!i, \langle i, \sigma, s \rangle) \wedge 0 \leq i < \mathsf{len}(P)$$

- Whole programs: **reflexive-transitive closure**

$$\langle P, c \rangle \twoheadrightarrow c' \iff \langle P, c \rangle \to^* c'$$

Compilation

- Translating C programs to stack machine programs:

$$\text{scomp} : \textbf{Stmt} \to \text{Seq}(\textbf{Instr})$$

- Auxiliary functions:

$$\text{acomp} : \textbf{AExp} \to \text{Seq}(\textbf{Instr})$$
$$\text{bcomp} : \textbf{BExp} \times \ldots \to \text{Seq}(\textbf{Instr})$$

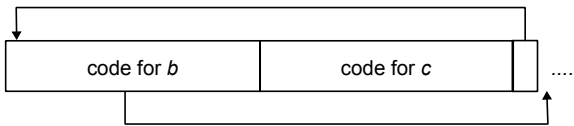
Compilation of Arithmetic Expressions

$$
\begin{aligned}
\text{acomp}(n) &= [\text{LOADI } n] \\
\text{acomp}(x) &= [\text{LOAD } x] \\
\text{acomp}(e_1 + e_2) &= \text{acomp}(e_1) +\!\!+ \text{acomp}(e_2) +\!\!+ [\text{ADD}]
\end{aligned}
$$

Compilation of Boolean Expressions

$$\text{bcomp} : \textbf{BExp} \times \mathbb{B} \times \mathbb{Z} \times \text{Seq}(\textbf{Instr})$$

- Observation: boolean expressions **always** evaluated for control flow
- bcomp(b, f, n): jump to offset n if b evaluates to given value f
- Example: compilation of **while** (b) c



- Also allows short-circuit evaluation

Compilation of Boolean Expressions

$$\text{bcomp}(\mathit{true}, f, n) = \begin{cases} [\text{JMP } n] & f = \mathit{true} \\ [] & \mathit{otherwise} \end{cases}$$

$$\text{bcomp}(\mathit{false}, f, n) = \begin{cases} [\text{JMP } n] & f = \mathit{false} \\ [] & \mathit{otherwise} \end{cases}$$

$$\text{bcomp}(!\, e, f, n) = \text{bcomp}(e, \neg f, n)$$

$$\text{bcomp}(e_1 \;\&\&\; e_2, f, n) = c_1 +\!\!+\, c_2 \quad \text{where } c_2 = \text{bcomp}(e_2, f, n)$$

$$m = \begin{cases} \text{len}(c_2) & f = \mathit{true} \\ \text{len}(c_2) + n & \mathit{otherwise} \end{cases}$$

$$c_1 = \text{bcomp}(e_1, \mathit{false}, m)$$

$$\text{bcomp}(e_1 < e_2, f, n) = \begin{cases} \text{acomp}(e_1) +\!\!+ \text{acomp}(e_2) +\!\!+ [\text{JMPLE } n] & f = \mathit{true} \\ \text{acomp}(e_1) +\!\!+ \text{acomp}(e_2) +\!\!+ [\text{JMPGE } n] & f = \mathit{false} \end{cases}$$

Compilation of Statements

$$
\begin{aligned}
\text{scomp}(\textbf{skip}) &= [] \\
\text{scomp}(x := e) &= \text{acomp}(e) +\!\!+ [\text{STORE } x] \\
\text{scomp}(c_1; c_2) &= \text{scomp}(c_1) +\!\!+ \text{scomp}(c_2) \\
\text{scomp}(\textbf{if } (b) \textbf{ then } c_1 \textbf{ else } c_2) &= bc +\!\!+ s_1 +\!\!+ [\text{JMP len}(s_2)] +\!\!+ s_2 \\
&\quad \text{where } s_1 = \text{scomp}(c_1) \\
&\quad\quad\; s_2 = \text{scomp}(c_2) \\
&\quad\quad\; bc = \text{bcomp}(b, \mathit{false}, \text{len}(s_1) + 1) \\
\text{scomp}(\textbf{while } (b)\ c) &= bc +\!\!+ s +\!\!+ [\text{JMP } (-(\text{len}(bc) + \text{len}(s) + 1))] \\
&\quad \text{where } s = \text{scomp}(c) \\
&\quad\quad\; bc = \text{bcomp}(b, \mathit{false}, \text{len}(s) + 1)
\end{aligned}
$$

Correctness of Compilation

- Correctness of the translation:

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \tau \iff \langle \mathsf{scomp}\, c, \langle 0, \sigma, s \rangle \rangle \twoheadrightarrow \langle \mathsf{len}(\mathsf{scomp}\, c), \tau, s \rangle$$

  - Why both directions?

Correctness of Compilation

- Correctness of the translation:

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \tau \iff \langle \text{scomp } c, \langle 0, \sigma, s \rangle \rangle \rightarrow\!\!\!\!\rightarrow \langle \text{len}(\text{scomp } c), \tau, s \rangle$$

  - Why both directions?
  - Compiled program terminates iff source program does.
- Sketch of proof:
  - $\Leftarrow$: Rule induction on operational semantics.
  - $\Rightarrow$: Structural induction on c, induction on n for **while** (b) c
  - Need corresponding lemmas for acomp, bcomp
- Correctness proof completely formalized in Isabelle