

Semantics and Verification of Software

Lecture 16: Provably Correct Implementation I (Abstract Machine)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)



noll@cs.rwth-aachen.de

<http://www-i2.informatik.rwth-aachen.de/i2/svsw11/>

Winter Semester 2011/12

EINLADUNG

Zeit: Mittwoch, 25. Januar 2012, 15:00 Uhr
Ort: Hörsaal AH 3, Ahornstr. 55
Referent: Dr. Thomas Noll
RWTH Aachen
Thema: Correctness, Safety and Fault Tolerance in
Aerospace Systems: The ESA COMPASS
Project

Building modern aerospace systems is highly demanding. They should be extremely dependable, offering service without failures for a very long time – typically years or decades. The need for an integrated system-software co-engineering framework to support the design of such systems is therefore pressing. However, current tools and formalisms tend to be tailored to specific analysis techniques and do not sufficiently cover the full spectrum of required system aspects such as safety, dependability and performance. Additionally, they cannot properly handle the intertwining of hardware and software operation. As such, current engineering practice lacks integration and coherence.

This talk gives an overview of the COMPASS project that was initiated by the European Space Agency to overcome this problem. It supports system-software co-engineering of real-time embedded systems by following a coherent and multidisciplinary approach. We show how such systems and their possible failures can be modeled in the Architecture and Analysis Design Language, how their behavior can be formalized, and how to analyze them by means of model checking and related techniques.

Es laden ein: Die Dozenten der Informatik

- To support **Compiler Construction** in Summer Semester
- Tasks:
 - evaluation of **exercises**
 - organizational **support**
- **12 hrs/week** contract
- Previous CC lecture **not** a prerequisite (but of course helpful)

- 1 Repetition: Semantics of Blocks and Procedures
- 2 Introduction
- 3 The Abstract Machine
- 4 Properties of AM

Procedure Environments

- **Procedure environments** now store **semantic** information:
 - So far: $PEnv := \{\pi \mid \pi : PVar \dashrightarrow Cmd \times VEnv \times PEnv\}$
 - Now: $PEnv := \{\pi \mid \pi : PVar \dashrightarrow (Sto \dashrightarrow Sto)\}$
- **Procedure declarations** (“`proc  $P$  is  $c$` ”) update procedure environment:

$$upd_p[\![\cdot]\!] : PDec \times VEnv \times PEnv \rightarrow PEnv$$

- **non-recursive** case: P not (indirectly) called within c
 $\Rightarrow \pi(P)$ immediately given by $\mathfrak{C}''[\![c]\!]\rho \pi$

$$upd_p[\![proc\ P\ is\ c;\ p]\!](\rho, \pi) := upd_p[\![p]\!](\rho, \pi[P \mapsto \mathfrak{C}''[\![c]\!]\rho \pi])$$

- **recursive** case: $\pi(P)$ must be a solution of equation $P = \mathfrak{C}''[\![c]\!]\rho \pi$
(cf. fixpoint semantics of **while** loop – Slide 5.15)

$$upd_p[\![proc\ P\ is\ c;\ p]\!](\rho, \pi) := upd_p[\![p]\!](\rho, \pi[P \mapsto \text{fix}(\Phi)])$$

where $\Phi : (Sto \dashrightarrow Sto) \rightarrow (Sto \dashrightarrow Sto) : f \mapsto \mathfrak{C}''[\![c]\!]\rho \pi[P \mapsto f]$

- $upd_p[\![\varepsilon]\!](\rho, \pi) := \pi$

Statement Semantics Including Procedures

So far: $\mathcal{E}'[\cdot] : \text{Cmd} \rightarrow \text{VEnv} \rightarrow (\text{Sto} \dashrightarrow \text{Sto})$

Definition (Denotational semantics with procedures)

$$\mathcal{E}''[\cdot] : \text{Cmd} \rightarrow \text{VEnv} \rightarrow \text{PEnv} \rightarrow (\text{Sto} \dashrightarrow \text{Sto})$$

is given by:

$$\begin{aligned}\mathcal{E}''[\text{skip}] \rho \pi &:= \text{id}_{\text{Sto}} \\ \mathcal{E}''[x := a] \rho \pi \sigma &:= \sigma[\rho(x) \mapsto \mathcal{A}[a](\text{lookup } \rho \sigma)] \\ \mathcal{E}''[c_1; c_2] \rho \pi &:= (\mathcal{E}''[c_2] \rho \pi) \circ (\mathcal{E}''[c_1] \rho \pi) \\ \mathcal{E}''[\text{if } b \text{ then } c_1 \text{ else } c_2] \rho \pi &:= \text{cond}(\mathcal{B}[b] \circ (\text{lookup } \rho), \\ &\quad \mathcal{E}''[c_1] \rho \pi, \mathcal{E}''[c_2] \rho \pi) \\ \mathcal{E}''[\text{while } b \text{ do } c] \rho \pi &:= \text{fix}(\Phi) \\ \mathcal{E}''[\text{call } P] \rho \pi &:= \pi(P) \\ \mathcal{E}''[\text{begin } v \text{ } p \text{ } c \text{ end}] \rho \pi \sigma &:= \mathcal{E}''[c] \rho' \pi' \sigma'\end{aligned}$$

where $\text{upd}_v[v](\rho, \sigma) = (\rho', \sigma')$

$\text{upd}_p[p](\rho', \pi) = \pi'$

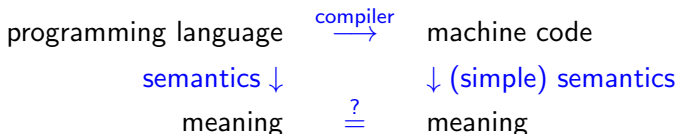
$\text{lookup } \rho \sigma := \sigma \circ \rho$

$\Phi(f) := \text{cond}(\mathcal{B}[b] \circ (\text{lookup } \rho), f \circ \mathcal{E}''[c] \rho \pi, \text{id}_{\text{Sto}})$

Summary: Blocks and Procedures

- **Blocks** allow to declare local variables and recursive procedures
- Requires concept of **locations** to support instantiation of variables
- **Static scoping**: meaning of identifier determined by declaration context (rather than calling context)
- Meaning of **variable declaration**: storage allocation
- Meaning of **procedure call**:
 - operationally: **execution** of procedure body
 - ⇒ procedure environment records statement (“symbol table”)
 - denotationally: **application** of procedure meaning
 - ⇒ procedure environment records (partial) store transformation
 - Recursive behavior again handled by **fixpoint approach**
- Further extensions:
 - **axiomatic semantics** (for `proc P is c`)
 - non-recursive:
$$(\text{call}) \frac{\{A\} c \{B\}}{\{A\} \text{call } P \{B\}}$$
 - recursive:
$$(\text{call}) \frac{\{A\} \text{call } P \{B\} \vdash \{A\} c \{B\}}{\{A\} \text{call } P \{B\}}$$
 - **procedure parameters**
 - **higher-order procedures**

- 1 Repetition: Semantics of Blocks and Procedures
- 2 Introduction
- 3 The Abstract Machine
- 4 Properties of AM



To do:

- 1 Definition of **abstract machine**
- 2 Definition (operational) **semantics of machine instructions**
- 3 Definition of **translation** WHILE $\rightarrow$ machine code (“compiler”)
- 4 **Proof:** semantics of generated machine code = semantics of original source code

- 1 Repetition: Semantics of Blocks and Procedures
- 2 Introduction
- 3 The Abstract Machine
- 4 Properties of AM

Definition 16.1 (Abstract machine)

The **abstract machine (AM)** is given by

- **configurations** of the form $\langle d, e, \sigma \rangle \in Cnf$ where
 - $d \in Code$ is the **sequence** of instructions (code) to be executed
 - $e \in Stk := (\mathbb{Z} \cup \mathbb{B})^*$ is the **evaluation stack** (top left)
 - $\sigma \in \Sigma := (Var \rightarrow \mathbb{Z})$ is the **(storage) state**

(thus $Cnf = Code \times Stk \times \Sigma$)

- **initial configurations** of the form $\langle d, \varepsilon, \sigma \rangle$
- **final configurations** of the form $\langle \varepsilon, e, \sigma \rangle$
- **code sequences** d and **instructions** i :
 $d ::= \varepsilon \mid i : d$
 $i ::= PUSH(z) \mid ADD \mid MULT \mid SUB \mid$
 $TRUE \mid FALSE \mid EQ \mid GT \mid AND \mid OR \mid NEG \mid$
 $LOAD(x) \mid STORE(x) \mid NOOP \mid BRANCH(d, d) \mid LOOP(d, d)$

(where $z \in \mathbb{Z}$ and $x \in Var$)

Definition 16.2 (Transition relation of AM)

The **transition relation** $\triangleright \subseteq \text{Cnf} \times \text{Cnf}$ is given by

$$\begin{aligned} \langle \text{PUSH}(z) : d, e, \sigma \rangle &\triangleright \langle d, z : e, \sigma \rangle \\ \langle \text{ADD} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 + z_2) : e, \sigma \rangle \\ \langle \text{MULT} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 * z_2) : e, \sigma \rangle \\ \langle \text{SUB} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 - z_2) : e, \sigma \rangle \\ \langle \text{TRUE} : d, e, \sigma \rangle &\triangleright \langle d, \text{true} : e, \sigma \rangle \\ \langle \text{FALSE} : d, e, \sigma \rangle &\triangleright \langle d, \text{false} : e, \sigma \rangle \\ \langle \text{EQ} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 = z_2) : e, \sigma \rangle \\ \langle \text{GT} : d, z_1 : z_2 : e, \sigma \rangle &\triangleright \langle d, (z_1 > z_2) : e, \sigma \rangle \\ \langle \text{AND} : d, t_1 : t_2 : e, \sigma \rangle &\triangleright \langle d, (t_1 \wedge t_2) : e, \sigma \rangle \\ \langle \text{OR} : d, t_1 : t_2 : e, \sigma \rangle &\triangleright \langle d, (t_1 \vee t_2) : e, \sigma \rangle \\ \langle \text{NEG} : d, t : e, \sigma \rangle &\triangleright \langle d, \neg t : e, \sigma \rangle \\ \langle \text{LOAD}(x) : d, e, \sigma \rangle &\triangleright \langle d, \sigma(x) : e, \sigma \rangle \\ \langle \text{STORE}(x) : d, z : e, \sigma \rangle &\triangleright \langle d, e, \sigma[x \mapsto z] \rangle \\ \langle \text{NOOP} : d, e, \sigma \rangle &\triangleright \langle d, e, \sigma \rangle \\ \langle \text{BRANCH}(d_{\text{true}}, d_{\text{false}}) : d, t : e, \sigma \rangle &\triangleright \langle d_t : d, e, \sigma \rangle \\ \langle \text{LOOP}(d_1, d_2) : d, e, \sigma \rangle &\triangleright \langle d_1 : \text{BRANCH}(d_2 : \text{LOOP}(d_1, d_2), \text{NOOP}) : d, e, \sigma \rangle \end{aligned}$$

Remark: more traditional machine architectures

- **Variables referenced by address** (and not by name)
 - configurations $\langle d, e, m \rangle$ with **memory** $m \in \mathbb{Z}^*$
 - $\text{LOAD}(x)/\text{STORE}(x)$ replaced by $\text{GET}(n)/\text{PUT}(n)$ (where $n \in \mathbb{N}$)
- **BRANCH** and **LOOP** instruction replaced by **code addresses** (labels) and **jumping instructions**
 - configurations $\langle pc, d, e, m \rangle$ with **program counter** $pc \in \mathbb{N}$
 - **BRANCH** and **LOOP** implemented by control flow, using $\text{JUMP}(l)$ and $\text{JUMPFALSE}(l)$ ($l \in \mathbb{N}$)
- **Registers** for storing intermediate values
(in place of evaluation stack e)

Definition 16.3 (AM computations)

- A **finite computation** is a finite configuration sequence of the form $\gamma_0, \gamma_1, \dots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \triangleright \gamma_i$ for each $i \in \{1, \dots, k\}$
- If, in addition, there is no γ such that $\gamma_k \triangleright \gamma$, then $\gamma_0, \gamma_1, \dots, \gamma_k$ is called **terminating**
- A **looping computation** is an infinite configuration sequence of the form $\gamma_0, \gamma_1, \gamma_2, \dots$ where $\gamma_i \triangleright \gamma_{i+1}$ for each $i \in \mathbb{N}$

Note: a terminating computation may end in a **final configuration** ($\langle \varepsilon, e, \sigma \rangle$) or in a **stuck configuration** (e.g., $\langle \text{ADD}, 1, \sigma \rangle$)

Example 16.4

For $d := \text{PUSH}(1) : \text{LOAD}(x) : \text{ADD} : \text{STORE}(x)$ and $\sigma(x) = 3$, we obtain the following terminating computation:

- $\langle \text{PUSH}(1) : \text{LOAD}(x) : \text{ADD} : \text{STORE}(x), \varepsilon, \sigma \rangle$
 - ▷ $\langle \text{LOAD}(x) : \text{ADD} : \text{STORE}(x), 1, \sigma \rangle$
 - ▷ $\langle \text{ADD} : \text{STORE}(x), 3 : 1, \sigma \rangle$
 - ▷ $\langle \text{STORE}(x), 4, \sigma \rangle$
 - ▷ $\langle \varepsilon, \varepsilon, \sigma[x \mapsto 4] \rangle$

Remark: implements statement $x := x + 1$

Example 16.5

The following computation loops:

$\langle \text{LOOP}(\text{TRUE}, \text{NOOP}), \varepsilon, \sigma \rangle$

- ▷ $\langle \text{TRUE} : \text{BRANCH}(\text{NOOP} : \text{LOOP}(\text{TRUE}, \text{NOOP}), \text{NOOP}), \varepsilon, \sigma \rangle$
- ▷ $\langle \text{BRANCH}(\text{NOOP} : \text{LOOP}(\text{TRUE}, \text{NOOP}), \text{NOOP}), \text{true}, \sigma \rangle$
- ▷ $\langle \text{NOOP} : \text{LOOP}(\text{TRUE}, \text{NOOP}), \varepsilon, \sigma \rangle$
- ▷ $\langle \text{LOOP}(\text{TRUE}, \text{NOOP}), \varepsilon, \sigma \rangle$
- ▷ ...

Remark: implements statement `while true do skip`

- 1 Repetition: Semantics of Blocks and Procedures
- 2 Introduction
- 3 The Abstract Machine
- 4 Properties of AM

A New Inductive Principle

Application: Finite computations (Def. 16.3)

Definition: a finite computation $\gamma_0, \gamma_1, \dots, \gamma_k$ has **length** k

Induction base: property holds for all computations of length 0

Induction hypothesis: property holds for all computations of length $\leq k$

Induction step: property holds for all computations of length $k + 1$

Lemma 16.6

If $\langle d_1, e_1, \sigma \rangle \triangleright^* \langle d', e', \sigma' \rangle$, then

$$\langle d_1 : d_2, e_1 : e_2, \sigma \rangle \triangleright^* \langle d' : d_2, e' : e_2, \sigma' \rangle$$

for every $d_2 \in \text{Code}$ and $e_2 \in \text{Stk}$.

Interpretation: both the code and the stack component can be extended without changing the behavior of the machine

Proof.

by induction on the length of the computation
(on the board)

