Software Verification – Exam

ETH Zürich

20 December 2010

Surname, first name: ....

Student number: ....

I confirm with my signature that I was able to take this exam under regular circumstances and that I have read and understood the directions below.

Signature: ....

Directions:

- Exam duration: 1 hour 45 minutes.
- Except for a dictionary you are not allowed to use any supplementary material.
- All solutions can be written directly on the exam sheets. If you need more space for your solution ask the supervisors for a sheet of official paper. You are **not** allowed to use other paper. Please write your student number on **each** additional sheet.
- Only one solution can be handed in per question. Invalid solutions need to be crossed out clearly.
- Please write legibly! We will only correct solutions that we can read.
- Manage your time carefully (take into account the number of points for each question).
- Please **immediately** tell the exam supervisors if you feel disturbed during the exam.

Good luck!

Question	Available points	Your points
1) Axiomatic semantics	9	
2) Separation logic	13	
3) Data flow analysis	12	
4) Model checking	10	
5) Software model checking	13	
6) Termination proofs	13	
Total	**70**	

[This page is intentionally left blank.]

1 Axiomatic semantics (9 points)

Consider the following Hoare triple (all variables of type $NATURAL$, assumed to describe mathematical natural numbers):

$\{\ y = n\ \}$
from
    $z := 1$
until $y = 0$ **loop**
    $y := y - 1$
    $z := z * x$
end
$\{\ z = x^n\ \}$

Prove that this triple is a theorem of Hoare's axiomatic system for partial correctness.

Solution:

$\{\ y = n\ \}$
from
$\{\ 1 = x^{n-y} = x^0 \}$
    $z := 1$
$\{\ z = x^{n-y}\ \}$
until $y = 0$ **loop**
$\{\ (z = x^{n-y})\ \wedge\ \neg(y = 0)\ \}$
$\{\ z \cdot x = x^{n-(y-1)} = x^{n-y} \cdot x\ \}$
    $y := y - 1$
$\{\ z \cdot x = x^{n-y}\ \}$
    $z := z * x$
$\{\ z = x^{n-y}\ \}$
end
$\{\ (z = x^{n-y})\ \wedge\ (y = 0)\ \}$
$\{\ z = x^n\ \}$

[This page is intentionally left blank.]

2 Separation Logic (13 points)

Consider the definition of the *list* binary predicate:

$$\begin{array}{lcl} \mathit{list}\ i\ [] & \equiv & \mathit{empty} \wedge i = \mathit{nil} \\ \mathit{list}\ i\ (a : \sigma) & \equiv & \exists j \cdot\ (i \mapsto a, j) * (\mathit{list}\ j\ \sigma) \end{array}$$

where $\sigma \stackrel{\text{def}}{=} [] \mid a : \sigma$ defines a sequence of integers.

2.1 States and semantics (7 points)

Consider the separation logic predicate P, where

$$P \stackrel{\text{def}}{=} 3 \mapsto 5, 8 * 8 \mapsto 7, 11 * 11 \mapsto 6, 1 * 1 \mapsto 3, \mathit{nil}$$

and answer the following questions:

(1) For every state $(s,\ h)$ that satisfies P, the heap component h will be the same. Write such a function h explicitly as a set of pairs.

Solution:
$h = \{$(1,3),(2,*nil*),(3,5),(4,8),(8,7),(9,11),(11,6),(12,1)$\}$.

(2) If $(s,h) \models P$, then $(s,h) \models \mathit{list}\ i\ \sigma * \mathit{true}$ for several values of i and σ. Provide all such pairs (i,σ).

Solution:
(nil, []), (1, 3:[]), (11, 6:3:[]), (8, 7:6:3:[]), (3, 5:7:6:3:[]). It is also fine to write [5,7,6,3] instead of 5:7:6:3:[], etc.

2.2 Separation logic and verification (6 points)

Consider the signature and separation logic specification for a routine that adds a value to the front of a linked list. It returns a pointer to the new head node by storing it in the **Result** variable:

add_front (*list_pointer* : *INTEGER* ; *value*: *INTEGER*): *INTEGER*
    require *list list_pointer* σ
    ensure *list* **Result** (*value* : σ)

(1) Write a body for the routine. Use the *cons* command, whose semantics is given by the axiom:

$$\textsc{ConsAxiom}\ \frac{}{\{\text{empty}\}\text{x} := \text{cons}(\text{e}_1,\ldots,\text{e}_n)\{\text{x} \mapsto \text{e}_1,\ldots,\text{e}_n\}}$$

provided that $1 \leq n$ and x is not free in any of $\text{e}_1,\ldots,\text{e}_n$.

Solution:
Result := cons(value, list_pointer)

(2) Prove your routine body correct.

Solution:
The proof looks as follows in outline form (other forms are also acceptable):

{*list list_pointer* σ}
    {*empty*}
        Result := *cons*(*value*, *list_pointer*)
    {**Result** $\mapsto$ *value*, *list_pointer*} // *By the axiom for cons.*
{**Result** $\mapsto$ *value*, *list_pointer* $*$ *list list_pointer* σ} // *By the frame rule.*
{*list* **Result** (*value* : σ)} // *By the rule of consequence.*

(3) Write down the schemas of all the inference rules that you used in the proof above.

Solution:
The rule names may differ.

$$\textsc{Frame}\ \frac{\{P\}\text{c}\{Q\}}{\{P * R\}\text{c}\{Q * R\}}$$

provided that no free variable of R is assigned by c.

$$\textsc{Consequence}\ \frac{\{P\}\text{c}\{Q\}}{\{P'\}\text{c}\{Q'\}}$$

provided that $P' \Rightarrow P$ and $Q \Rightarrow Q'$.

3 Data flow analysis (12 points)

An arithmetic expression is called *trivial* if it consists only of a single variable or constant; it is called *non-trivial* otherwise. Let $\mathbf{AExp}_\star$ denote the set of all non-trivial arithmetic expressions that occur in a given program fragment, and let $\mathbf{AExp}(a)$ denote the set of all non-trivial arithmetic subexpressions of an expression a. Furthermore, let $\mathit{Vars}(a)$ denote the set of variables occurring in a.

With this terminology, recall the definition of the *available expressions analysis* from the lecture

$$\begin{array}{rcl} AE_{entry}(\ell') & = & \begin{cases} \emptyset & \text{if } \ell' \text{ is the initial label} \\ \bigcap_{(\ell,\ell')\in CFG} AE_{exit}(\ell) & \text{otherwise} \end{cases} \\ AE_{exit}(\ell) & = & (AE_{entry}(\ell) \setminus \mathit{kill}_{AE}(B^\ell)) \cup \mathit{gen}_{AE}(B^\ell) \end{array}$$

where B is an elementary block of the form $[x := a]$ or $[b]$, and the *kill* and *gen* functions are given by

$$\begin{array}{rcl} \mathit{kill}_{AE}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}_\star \mid x \in \mathit{Vars}(a')\} \\ \mathit{kill}_{AE}([b]^\ell) & = & \emptyset \\ \mathit{gen}_{AE}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}(a) \mid x \notin \mathit{Vars}(a')\} \\ \mathit{gen}_{AE}([b]^\ell) & = & \mathbf{AExp}(b) \end{array}$$

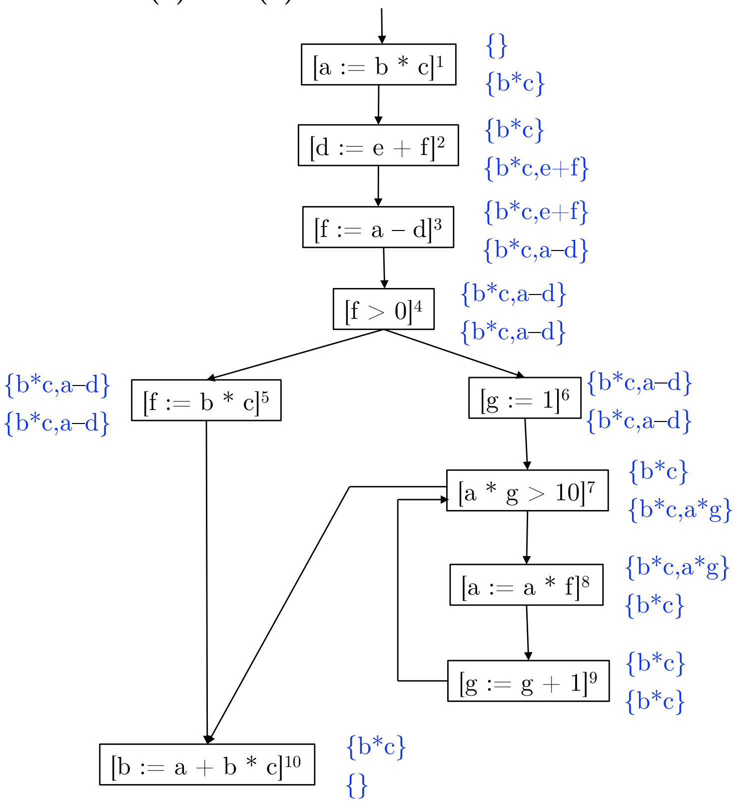
Now consider the following program fragment:

```
a := b * c
d := e + f
f := a - d
if f > 0 then
    f := b * c
else
    from
        g := 1
    until a * g > 10 loop
        a := a * f
        g := g + 1
    end
end
b := a + b * c
```

(1) Draw the control flow graph of the program fragment and label each elementary block. (3 points)

(2) Annotate your control flow graph with the analysis result of an available expressions analysis of the program fragment. (7 points)

Solution to (1) and (2):



(3) How can you use your analysis result to optimize the program fragment? (2 points)

Solution:
The analysis result can be used to eliminate common subexpressions, i.e. expressions which are always computed at least twice on a computation path.

As the expression $b * c$ is available at the entries to blocks 5 and 10 where it is also recomputed, it may be worth for optimization purposes to introduce a temporary variable tmp holding the computed value. The transformed code looks as follows:

```
tmp := b * c
a := tmp
d := e + f
f := a − d
if f > 0 then
    f := tmp
else
    from
        g := 1
    until a * g > 10 do
        a := a * f
        g := g + 1
    end
end
b := a + tmp
```

4 Model Checking (10 points)

Recall the semantics of LTL over finite words with alphabet $\mathcal{P}$. For a word $w = w(1)w(2)\cdots w(n) \in \mathcal{P}^*$ with $n \geq 0$ and a position $1 \leq i \leq n$ the satisfaction relation $\models$ is defined recursively as follows for $p, q \in \mathcal{P}$.

$w, i \models p$	iff	$p = w(i)$
$w, i \models \neg\phi$	iff	$w, i \not\models \phi$
$w, i \models \phi_1 \wedge \phi_2$	iff	$w, i \models \phi_1$ and $w, i \models \phi_2$
$w, i \models \mathsf{X}\phi$	iff	$i < n$ and $w, i+1 \models \phi$
$w, i \models \phi_1 \,\mathsf{U}\, \phi_2$	iff	there exists $i \leq j \leq n$ such that: $w, j \models \phi_2$ and for all $i \leq k < j$ it is the case that $w, k \models \phi_1$
$w, i \models \Diamond\, \phi$	iff	there exists $i \leq j \leq n$ such that: $w, j \models \phi$
$w, i \models \Box\, \phi$	iff	for all $i \leq j \leq n$ it is the case that: $w, j \models \phi$
$w \models \phi$	iff	$w, 1 \models \phi$

4.1 Automata and LTL formulas (6 points)

Consider the automata $T_{\mathcal{A}}$ (with states A, B, C) and $T_{\mathcal{X}}$ (with states X, Y, Z) in Figure 1, over the alphabet $\{p, q\}$. Notice that $T_{\mathcal{A}}$ is nondeterministic but $T_{\mathcal{X}}$ is deterministic.

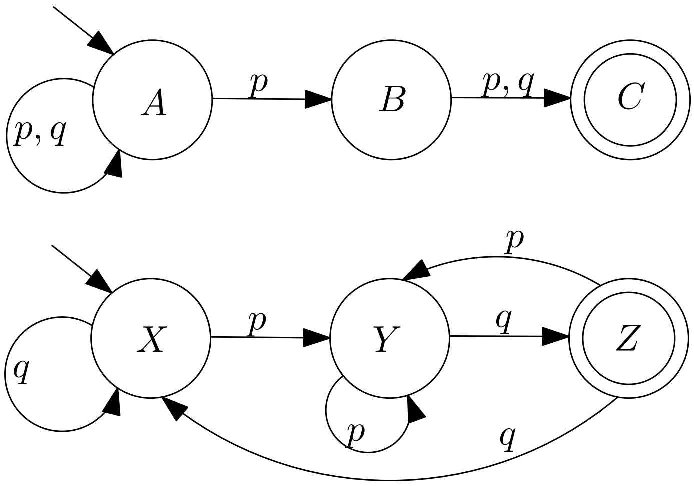


Figure 1: Automata $T_{\mathcal{A}}$ (top) and $T_{\mathcal{X}}$ (bottom).

For each of the following LTL formulas say whether every run of $T_{\mathcal{A}}$ or $T_{\mathcal{X}}$ satisfies the formula. If it does, argue informally (but precisely) why this is the case; if it does not, provide a counterexample.

(1) $T_{\mathcal{A}} \models \Box(\Diamond p)$

No: the word $w_1 = p\ q$ is a counterexample because $w_1, 2 \not\models p$ and hence $w_1, 2 \not\models \Diamond p$

(2) $T_{\mathcal{X}} \models \Box(\Diamond p)$

No, with the same counterexample as in question (1).

(3) $T_{\mathcal{A}} \models \Diamond\,(p \wedge \mathsf{X}(p \vee q))$

Yes: every accepting run reaches the state C; to do so it must end with the events $p\ p$ or $p\ q$.

(4) $T_{\mathcal{X}} \models \Diamond\,(p \wedge \mathsf{X}p)$

No: the word $w_2 = p\ q$ is clearly accepted but $w, 1 \not\models \mathsf{X}p$ because $w_2(1{+}1) = q \neq p$.

(5) $T_{\mathcal{X}} \models p\,\mathsf{U}\,q$

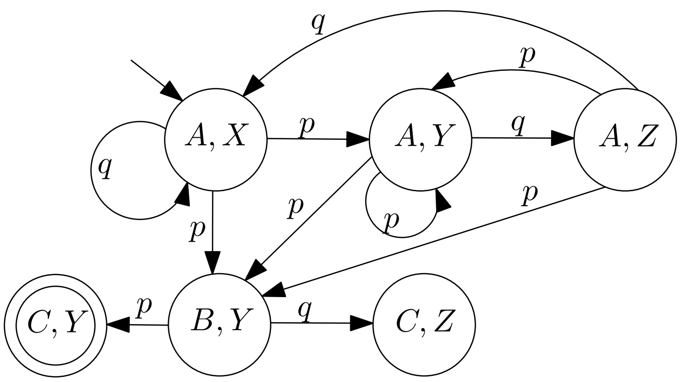
Yes: every accepted word begins with q or with $p^n q$, with $n \geq 1$, which satisfy $p\,\mathsf{U}\,q$.

4.2 Automata-based model checking (4 points)

Let $\langle T_{\mathcal{A}}\rangle$ and $\langle T_{\mathcal{X}}\rangle$ respectively denote the set of all words accepted by $T_{\mathcal{A}}$ and $T_{\mathcal{X}}$. Show that $\langle T_{\mathcal{A}}\rangle \not\subseteq \langle T_{\mathcal{X}}\rangle$ by constructing the intersection automaton $T_{\mathcal{A}} \times \neg T_{\mathcal{X}}$ of $T_{\mathcal{A}}$ and the *complement of* $T_{\mathcal{X}}$, and by showing that the intersection automaton accepts some word.

(Remember that the complement automaton of $T_{\mathcal{X}}$ is identical to $T_{\mathcal{X}}$ except for the accepting states which are X and Y in the complement, with Z becoming a rejecting state in the complement).

Solution:



The accepting state C, Y is reachable with the word $p\ p$ which is therefore in the interesection of $\langle T_{\mathcal{A}}\rangle$ and $\neg\langle T_{\mathcal{X}}\rangle$.

5 Software model checking (13 points)

Consider the following code snippet C, where x, y are integer variables.

```
assume x + y > 0 end
x := x + y
```

Remember that the Boolean abstraction of an **assume** c **end** statement is **assume not** *Pred* (**not** c) **end** followed by a parallel conditional assignment updating the predicates with respect to the original **assume** statement. *Pred* (f) denotes the weakest under-approximation of the expression f in terms of the given predicates.

5.1 Boolean abstractions (10 points)

Build the Boolean abstraction A of the code snippet C with respect to the following predicates:

$$p \quad = \quad x > 0$$
$$q \quad = \quad y > 0$$

Solution:
The abstraction is:

```
assume not (not p and not q) end
if (not p and not q) or p then p := True
elseif (not p and not q) or not p then p := False
else p := ? end
if (not p and not q) or q then q := True
elseif (not p and not q) or not q then q := False
else q := ? end

if p and q then p := True
elseif not p and not q then p := False
else p := ? end
if q then q := True
elseif not q then q := False
else q := ? end
```

After simplifications, we get:

```
assume p or q end
if not q then p := True end

if p and q then p := True end
```

5.2 Abstract and concrete traces (3 points)

Provide an annotated trace for the Boolean abstraction A, and a corresponding annotated trace for the concrete program C which is feasible. Note that in general there are multiple traces of C corresponding to the same trace of A: you must select one which is feasible.

The trace of A should be in the form of a valid sequence of statements and branch conditions in A which reaches the bottom of A. Each statement in the sequence must be preceded and followed by a complete description of the abstract program state in terms of values of the Boolean predicates p, q. Similarly, the trace of C should be in the form of a valid sequence of statements and branch conditions in C which reaches the bottom of C without violating any assertion. Each statement in the sequence must be preceded and followed by a concrete value for the variables $x,\ y$ which satisfies the corresponding state in the abstract trace of A.

Solution:

```
{p, not q}
  assume p or q end
{p, not q}
  if not q then p := True end
{p, not q}
  if p and q then p := True end
{p, not q}
```

A matching concrete trace which is feasible is, for example, the following.

```
{x = 3, y = −1}
  assume x + y > 0 end
{x = 3, y = −1}
  x := x + y
{x = 2, y = −1}
```

6 Termination proofs (13 points)

Consider the following implementation of binary search, where // denotes integer division.

```
binary_search (v: G ; list : LIST [G] ; n: INTEGER): BOOLEAN
  -- Is 'v' contained in ' list ' in the range [1..' n']?
  require n > 0 and list.is_sorted
  do
    from
      l := 1
      u := n
      Result := False
    until l > u
    loop
      m := (l + u) // 2
      if list [m] = v then
        -- Element found
        Result := True
        l := u + 1
      elseif list [m] > v then
        -- Continue search on left side
        u := m - 1
      else
        -- Continue search on right side
        l := m + 1
      end
    end
end
```

(1) Consider the loop invariant

$$I \triangleq u - l + 1 \geq 0$$

Find a suitable *variant* function V which decreases along all branches of the loop body, and describe how V and I can be combined to prove that the loop always terminates. You do not have to provide a formal proof, but only to outline a termination argument for the given program with a suitable variant V. (7 points)

Solution:
A termination proof can be carried out using the variant

$$V \triangleq u - l + 1$$

Termination can be established from the observation that V decreases along each branch, because either u is decreased and l stays the same, or l is increased and u stays the same. The invariant I then guarantees that V has a lower bound, hence the loop must terminate when V reaches the lower bound.

(2) Provide a proof that I is an invariant of the loop. For full credit, it is enough if you consider only the **else** branch of the conditional and prove invariance (consecution) along it. (6 points)

Solution:

```
from
  { n > 0 }
  { n − 1 + 1 = n ≥ 0 }
  l := 1
  u := n
  Result := False
  { u − l + 1 ≥ 0 }
until l > u
loop
  { u − l + 1 ≥ 0 and l ≤ u }
  m := (l + u) // 2
  if list [m] = v then
    { u − l + 1 ≥ 0 and l ≤ u and list [m] = v }
    { u − (u + 1) + 1 = 0 ≥ 0 }
    Result := True
    l := u + 1
    { u − l + 1 ≥ 0 }
  elseif list [m] > v then
    { m = (l + u) // 2 and u − l + 1 ≥ 0 and l ≤ u and list [m] > v }
    { m − 1 − l + 1 = m − l ≥ 0 }
    u := m − 1
    { u − l + 1 ≥ 0 }
  else
    { m = (l + u) // 2 and u − l + 1 ≥ 0 and l ≤ u and list [m] > v }
    { u − m − 1 + 1 = u − m ≥ 0 }
    l := m + 1
    { u − l + 1 ≥ 0 }
  end
end
```

To discharge the verification condition in the first branch of the **elseif**, notice that $m = (l{+}u)//2$ implies $u \geq 2{*}m - l$, which combined with $u - l + 1 \geq 0$ implies $(2{*}m - l) - l + 1 = 2{*}(m - l){+}\ 1 \geq 0$. The latter also implies $m - l \geq 0$ because m, l are of integer type.

A similar reasoning discharges the verification condition in the second branch of the **elseif**: $m = (l{+}u)//2$ implies $-l \leq u - 2{*}m$, which combined with $u - l + 1 \geq 0$ implies $u + u - 2{*}m + 1 = 2{*}(u - m){+}\ 1 \geq 0$. The latter also implies $u - m \geq 0$ because m, u are of integer type.