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!
<table>
<thead>
<tr>
<th>Question</th>
<th>Available points</th>
<th>Your points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Axiomatic semantics</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2) Separation logic</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3) Data flow analysis</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4) Model checking</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5) Software model checking</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>6) Termination proofs</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>70</strong></td>
<td></td>
</tr>
</tbody>
</table>
1 Axiomatic semantics (9 points)

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

\[
\begin{align*}
\{ & y = n \} \\
\text{from} & \\
2 & z := 1 \\
3 & \text{until } y = 0 \text{ loop} \\
4 & y := y - 1 \\
5 & z := z \times x \\
6 & \text{end} \\
& \{ z = x^n \}
\end{align*}
\]

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

Solution:

\[
\begin{align*}
1 & \{ y = n \} \\
2 & \text{from} \\
3 & \{ 1 = x^{n-y} = x^0 \} \\
4 & z := 1 \\
5 & \{ z = x^{n-y} \} \\
6 & \text{until } y = 0 \text{ loop} \\
7 & \{ (z = x^{n-y}) \land \neg(y = 0) \} \\
8 & \{ z \cdot x = x^{n-(y-1)} = x^{n-y} \cdot x \} \\
9 & y := y - 1 \\
10 & \{ z \cdot x = x^{n-y} \} \\
11 & z := z \times x \\
12 & \{ z = x^{n-y} \} \\
13 & \text{end} \\
14 & \{ (z = x^{n-y}) \land (y = 0) \} \\
15 & \{ z = x^n \}
\end{align*}
\]
2 Separation Logic (13 points)

Consider the definition of the list binary predicate:

\[
\begin{align*}
\text{list } i & \equiv \text{empty} \land i = \text{nil} \\
\text{list } i \ (a : \sigma) & \equiv \exists j. \ (i \mapsto a, j) \ast (\text{list } j \ \sigma)
\end{align*}
\]

where \( \sigma \overset{\text{def}}{=} \[] | a : \sigma \) defines a sequence of integers.

2.1 States and semantics (7 points)

Consider the separation logic predicate \( P \), where

\[
P \overset{\text{def}}{=} 3 \mapsto 5, 8 \mapsto 7, 11 \mapsto 6, 1 \mapsto 3, \text{nil}
\]

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.

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

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

\textbf{Solution:} \\
\((\text{nil, } []), (1, 3:[]), (11, 6:3:], (8, 7:6:3:], (3, 5:7:6:3:])\). 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:

\[
\text{add_front ( list_pointer : INTEGER ; value: INTEGER ) : INTEGER}
\]

\[
\text{require list list_pointer } \sigma
\]

\[
\text{ensure list Result (value : } \sigma)\]

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

\[
\text{\text{\texttt{ConsAxiom}}}
\]

\[
\text{\text{\{empty\}x := cons(e_1,\ldots,e_n)\{x \mapsto e_1,\ldots,e_n\}}}
\]

provided that \(1 \leq n\) and \(x\) is not free in any of \(e_1,\ldots,e_n\).

\textbf{Solution:}

Result := \textit{cons}(value, list_pointer)

(2) Prove your routine body correct.

\textbf{Solution:}

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

\[
\{\text{list list_pointer } \sigma\}
\]

\[
\{\text{empty}\}
\]

\[
\text{Result := \textit{cons}(value, list_pointer)}
\]

\[
\{\text{Result } \mapsto \text{value, list_pointer}\} \quad \text{// By the axiom for cons.}
\]

\[
\{\text{Result } \mapsto \text{value, list_pointer } \circ\textit{list list_pointer } \sigma\} \quad \text{// By the frame rule.}
\]

\[
\{\text{list Result (value : } \sigma)\} \quad \text{// By the rule of consequence.}
\]

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

\textbf{Solution:}

The rule names may differ.

\[
\text{\text{\texttt{Frame}}}
\]

\[
\frac{\{P\}c\{Q\}}{\{P \circ R\}c\{Q \circ R\}}
\]

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

\[
\text{\text{\texttt{Consequence}}}
\]

\[
\frac{\{P\}c\{Q\}}{\{P'\}c\{Q'\}}
\]

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 $\text{AExp}_a$ denote the set of all non-trivial arithmetic expressions that occur in a given program fragment, and let $\text{AExp}(a)$ denote the set of all non-trivial arithmetic subexpressions of an expression $a$. Furthermore, let $\text{Vars}(a)$ denote the set of variables occurring in $a$.

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

$$AE_{\text{entry}}(\ell') = \left\{ \begin{array}{ll}
\emptyset & \text{if } \ell' \text{ is the initial label} \\
\bigcap_{(\ell,\ell') \in \text{CFG}} AE_{\text{exit}}(\ell) & \text{otherwise}
\end{array} \right.$$ $$AE_{\text{exit}}(\ell) = (AE_{\text{entry}}(\ell) \setminus \text{kill}(B)) \cup \text{gen}(B')$$

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

$$\text{kill}(B) = \{ a' \in \text{AExp}_a \mid x \in \text{Vars}(a') \}$$ $$\text{kill}(B) = \emptyset$$ $$\text{gen}(B) = \{ a' \in \text{AExp}(a) \mid x \notin \text{Vars}(a') \}$$ $$\text{gen}(B) = \text{AExp}(b)$$

Now consider the following program fragment:

1. $a := b * c$
2. $d := e + f$
3. $f := a - d$
4. if $f > 0$ then
5. $f := b * c$
6. else
7. from
8. $g := 1$
9. until $a * g > 10$ loop
10. $a := a + f$
11. $g := g + 1$
12. end
13. end
14. $b := a + b * c$

(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 \( b \times c \) 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:

\[
\begin{align*}
\text{tmp} & := b \times c \\
a & := \text{tmp} \\
d & := e + f \\
f & := a - d \\
\text{if } f > 0 \text{ then } \\
f & := \text{tmp} \\
\text{else} \\
g & := 1 \\
\text{end} \\
\text{from} \\
\text{until } a \times g > 10 \text{ do} \\
a & := a \times f \\
g & := g + 1 \\
\text{end} \\
\text{end} \\
b & := a + \text{tmp}
\end{align*}
\]
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 $|= $ is defined recursively as follows for $p,q \in \mathcal{P}$.

\begin{align*}
  w, i |= p & \text{ iff } p = w(i) \\
  w, i |= \neg \phi & \text{ iff } w, i \not|= \phi \\
  w, i |= \phi_1 \land \phi_2 & \text{ iff } w, i |= \phi_1 \text{ and } w, i |= \phi_2 \\
  w, i |= X \phi & \text{ iff } i < n \text{ and } w, i+1 |= \phi \\
  w, i |= \phi_1 U \phi_2 & \text{ iff } \text{there exists } i \leq j \leq n \text{ such that: } w, j |= \phi_2 \\
  & \quad \quad \quad \text{and for all } i \leq k < j \text{ it is the case that } w, k |= \phi_1 \\
  w, i |= \Diamond \phi & \text{ iff } \text{there exists } i \leq j \leq n \text{ such that: } w, j |= \phi \\
  w, i |= \Box \phi & \text{ iff } \text{for all } i \leq j \leq n \text{ it is the case that: } w, j |= \phi \\
  w |= \phi & \text{ iff } w, 1 |= \phi
\end{align*}

4.1 Automata and LTL formulas (6 points)

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

\begin{figure}[h]
  \centering
  \begin{tikzpicture}[node distance = 2cm, auto]
    \node (A) {$A$};
    \node (B) [right of=A] {$B$};
    \node (C) [right of=B] {$C$};
    \node (X) [below of=A] {$X$};
    \node (Y) [right of=X] {$Y$};
    \node (Z) [right of=Y] {$Z$};
    \path[->]
    (A) edge node {$p,q$} (B)
    (B) edge node {$p,q$} (C)
    (X) edge node {$p$} (Y)
    (Y) edge node {$q$} (Z)
    (X) edge [loop below] node {$q$} (X)
    (X) edge [loop above] node {$p$} (X)
    (Y) edge [loop below] node {$p$} (Y)
  \end{tikzpicture}
  \caption{Automata $T_A$ (top) and $T_X$ (bottom).}
\end{figure}

For each of the following LTL formulas say whether every run of $T_A$ or $T_X$ satisfies the formula. If it does, argue informally (but precisely) why this is the case; if it does not, provide a counterexample.
(1) $T_A \models \square(\Diamond p)$
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) $T_X \models \square(\Diamond p)$
No, with the same counterexample as in question (1).

(3) $T_A \models \Diamond (p \land X(p \lor q))$
Yes: every accepting run reaches the state C; to do so it must end with the events $p p$ or $p q$.

(4) $T_X \models \Diamond (p \land Xp)$
No: the word $w_2 = p q$ is clearly accepted but $w, 1 \not\models Xp$ because $w_2(1+1) = q \neq p$.

(5) $T_X \models p \lor q$
No: the word $w_3 = p q p q$ is a counterexample.

4.2 Automata-based model checking (4 points)
Let $\langle T_A \rangle$ and $\langle T_X \rangle$ respectively denote the set of all words accepted by $T_A$ and $T_X$. Show that $\langle T_A \rangle \not\subseteq \langle T_X \rangle$ by constructing the intersection automaton $T_A \cap X T_X$ of $T_A$ and the complement of $T_X$, and by showing that the intersection automaton accepts some word.

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

Solution:

![Diagram]

The accepting state $C, Y$ is reachable with the word $p p$ which is therefore in the intersection of $\langle T_A \rangle$ and $\neg \langle T_X \rangle$. 

11
5 Software model checking (13 points)

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

1. \texttt{assume } x + y > 0 \texttt{ end}
2. \texttt{x := x + y}

Remember that the Boolean abstraction of an \texttt{assume c end} statement is \texttt{assume not Pred (not c) end} followed by a parallel conditional assignment updating the predicates with respect to the original \texttt{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:

\begin{align*}
p &= x > 0 \\
q &= y > 0
\end{align*}

\textbf{Solution:}

The abstraction is:

1. \texttt{assume not (not p and not q) end}
2. \texttt{if (not p and not q) or p then p := True}
3. \texttt{elseif (not p and not q) or not p then p := False}
4. \texttt{else p := ? end}
5. \texttt{if (not p and not q) or q then q := True}
6. \texttt{elseif (not p and not q) or not q then q := False}
7. \texttt{else q := ? end}
8. \texttt{if p and q then p := True}
9. \texttt{elseif not p and not q then p := False}
10. \texttt{else p := ? end}
11. \texttt{if q then q := True}
12. \texttt{elseif not q then q := False}
13. \texttt{else q := ? end}

After simplifications, we get:

1. \texttt{assume p or q end}
2. \texttt{if not q then p := True end}
3. \texttt{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:**

1 \{p, \text{ not } q\}
2 \text{ assume } p \text{ or } q \text{ end}
3 \{p, \text{ not } q\}
4 \text{ if not } q \text{ then } p := \text{True} \text{ end}
5 \{p, \text{ not } q\}
6 \text{ if } p \text{ and } q \text{ then } p := \text{True} \text{ end}
7 \{p, \text{ not } q\}

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

1 \{x = 3, y = -1\}
2 \text{ assume } x + y > 0 \text{ end}
3
4
5 \{x = 3, y = -1\}
6 \text{ } x := x + y
7 \{x = 2, y = -1\}
6 Termination proofs (13 points)

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

```plaintext
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
    l := 1
    u := n
    Result := False
    until l > u
    loop
        m := (l + u) // 2
        if list [m] = v then
            −− Element found
            Result := True
            l := l + 1
        elseif list [m] > v then
            −− Continue search on left side
            u := m − 1
        else
            −− Continue search on right side
            l := l + 1
        end
    end
end
```

(1) Consider the loop invariant

\[ I = u - l + 1 \geq 0 \]

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

\[ V = u - l + 1 \]

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 \geq 0 \}
l := 1
u := n
Result := False
\{ u - l + 1 \geq 0 \}
until l > u
loop
\{ u - l + 1 \geq 0 \text{ and } l \leq u \}
m := (l + u) // 2
if list \[m\] = v then
  \{ u - l + 1 \geq 0 \text{ and } l \leq u \text{ and } list \[m\] = v \}
  \{ u - (u + 1) + 1 = 0 \geq 0 \}
Result := True
  \{ u - l + 1 \geq 0 \}
elseif list \[m\] > v then
  \{ m = (l + u) // 2 \text{ and } u - l + 1 \geq 0 \text{ and } l \leq u \text{ and } list \[m\] > v \}
  \{ m - 1 - l + 1 = m - l \geq 0 \}
u := m - 1
  \{ u - l + 1 \geq 0 \}
else
  \{ m = (l + u) // 2 \text{ and } u - l + 1 \geq 0 \text{ and } l \leq u \text{ and } list \[m\] > v \}
  \{ u - m - 1 + 1 = u - m \geq 0 \}
l := m + 1
  \{ u - l + 1 \geq 0 \}
end
end

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