DD2452 Formal Methods

TAKE-HOME EXAMINATION PROBLEMS
25 October 2018

Dilian Gurov, KTH EECS
08-790 81 98

Give cleanly written solutions in English or Swedish, each problem beginning on a new page. Write your name on all sheets. The maximal number of points is given for each problem. Up to two bonus points from the homework assignments will be taken into account for each level (E, C, and A). Upload your solutions on CANVAS, as a PDF file, no later than 10:00 on Monday, 29 October 2018. The exam is strictly individual. KTH's rules for cheating and plagiarism apply.

---

1 Level E

For passing level E you need 8 (out of 10) points from this section.

1. Consider **Dilian's Verification Condition Generator**, presented in class (slides 16-17 of Lecture 2). Explain briefly the idea behind the vc-function. How does it relate to the wlp-function, presented on slide 11? Why does it introduce an accumulator? How is this accumulator used? [2p]
   Solution: For while-free programs, the accumulator is in fact dummy (*true*), and the clauses for all non-loop commands just propagate it unchanged. The reason the accumulator is needed is that in the case of an annotated while-loop, the candidate loop invariant η is propagated "up-ward" as the weakest precondition, instead of the real one, and the proof obligations that have been accumulated so far need to be checked separately (every loop gives rise to two proof obligations, corresponding to conditions (1) and (3) of when η is a suitable loop invariant, see slide 14 of Lecture 2). Introducing an accumulator allows the vc-function to be defined by *structural induction* on the command C, like the wlp-function.
2. Consider the wp-function over the **Intermediate Language**, presented on slide 23, and the treatment of annotated while-loops presented on slide 26. Explain briefly the idea behind this treatment. Why does it work? How does it relate to the $\mathsf{Partial-while}$ rule of Hoare logic? [2p]
   Solution: Asserting η makes sure that η really holds at loop entry. The rest deals with the two branches of the loop, and are better understood by inserting "`havoc` *vars*; `assume` η;" into the two branches of the choice command. This would result in:

   `havoc` *vars*; `assume` η; `assume` B; c; `assert` η; `assume` *false* []
   `havoc` *vars*; `assume` η; `assume` $\neg B$

   The first branch checks whether η is indeed a loop invariant, exactly as the premise to the $\mathsf{Partial-while}$ rule of Hoare logic (recall the Hoare-triple Correctness Theorem from slide 22). The `havoc`-ing is used to make sure that the Hoare triple holds for *any* values of the variables (and not just the particular ones with which one might enter the loop body). "`assume` η; `assume` B;" is equivalent to "`assume` $\eta \wedge B$;", while "`assume` *false*" has the effect of making ψ immaterial when analysing this branch. Analogously, the second branch checks that $\eta \wedge \neg B$ entails ψ for any values of the variables.

3. Consider the **adequate set** of CTL connectives: 3p

$$\phi ::= \mathsf{false} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{EX}\,\phi \mid \mathsf{AF}\,\phi \mid \mathsf{E}\,(\phi\,\mathsf{U}\,\phi)$$

which we used in the Labelling Algorithm, and the re-write rules for the remaining connectives, presented on slide 8 of Lecture 6. Complete the rules with a rule for $\mathsf{A}\,(\phi\,\mathsf{U}\,\psi)$. Justify formally your new rule. (*Hint*: See book.)
Solution: We can add the following rule:

$$\mathsf{A}\,(\phi\,\mathsf{U}\,\psi) \equiv \mathsf{AF}\ \psi \wedge \neg\mathsf{E}\,(\neg\psi\,\mathsf{U}\,(\neg\phi \wedge \neg\psi))$$

It can be formally justified as follows:

$$\begin{aligned}
\mathsf{A}\,(\phi\,\mathsf{U}\,\psi) &\equiv \neg(\mathsf{E}\,(\neg\psi\,\mathsf{U}\,(\neg\phi \wedge \neg\psi)) \vee \mathsf{EG}\ \neg\psi) && \{\text{Book, page 216}\}\\
&\equiv \neg(\mathsf{E}\,(\neg\psi\,\mathsf{U}\,(\neg\phi \wedge \neg\psi)) \vee \neg\mathsf{AF}\ \psi) && \{\text{Re-writing } \mathsf{EG}\}\\
&\equiv \neg\mathsf{E}\,(\neg\psi\,\mathsf{U}\,(\neg\phi \wedge \neg\psi)) \wedge \mathsf{AF}\ \psi && \{\text{Re-writing } \vee\}\\
&\equiv \mathsf{AF}\ \psi \wedge \neg\mathsf{E}\,(\neg\psi\,\mathsf{U}\,(\neg\phi \wedge \neg\psi)) && \{\text{Commut. of } \wedge\}
\end{aligned}$$

3p 4. Apply the **CDCL algorithm** from the Lecture 4 slides to determine the (un)satisfiability of the following Boolean formula. Whenever the algorithm does not clearly specify what clause or literal to consider next, hence allowing for more than one choice, you are free to make your own choice (assumption). Each step of the algorithm should be properly explained in the final solution.

$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$
$\wedge\ (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$

Solution: The CDCL algorithms takes as input a set of clauses, hence we translate the Boolean formula to the following list of clauses.

(a) $x_1 \vee x_2 \vee x_3$
(b) $\neg x_1 \vee x_2 \vee x_3$
(c) $x_1 \vee \neg x_2 \vee x_3$
(d) $\neg x_1 \vee \neg x_2 \vee x_3$
(e) $x_1 \vee x_2 \vee \neg x_3$
(f) $\neg x_1 \vee \neg x_2 \vee \neg x_3$
(g) $x_1 \vee \neg x_2 \vee \neg x_3$
(h) $\neg x_1 \vee x_2 \vee \neg x_3$

Step 1: We apply the CDCL algorithm with the above clauses as input. Since there are no unit clauses, and there are still clauses to satisfy, we can make a decision by assigning a truth value to an unassigned variable. We choose $x_1 = 0$, which does not make any clause to become unit. Hence, we make another decision by assigning $x_2 = 0$. As a result, the clause (a) becomes unit, and the algorithm applies unit propagation by setting $x_3 = 1$. Unit propagation leads to a conflict between clauses (a) and (e). As the path to the conflict contains two decisions, namely $x_1 = 0$ and $x_2 = 0$, we learn that these assignments can not both hold at the same time. Therefore, we learn the conflict clause:
(i) $x_1 \vee x_2$

Step 2: We apply another iteration of the CDCL algorithm with the extended clauses (a-i) as input. Again, since there are no unit clauses, and there are still clauses to satisfy, we need to make a decision. We choose $x_1 = 0$, which now forces the clause (i) to become a unit clause and set $x_2 = 1$. As a reslut, the clause (c) also becomes a unit clause and $x_3 = 1$, which is in conflict with the clause (g). As the path to the conflict contains one decision, namely $x_1 = 0$, we learn the conflict clause:
(j) x_1
Step 3: We now apply another iteration of the CDCL algorithm with the extended clauses (a-j) as input. Since there is a unit clause, i.e., x_1, we first apply unit propagation on literal x_1, and then make a decision by setting $x2 = 0$. As a reslut, the clause (b) also becomes a unit clause and $x_3 = 1$, which is in conflict with the clause (h). As the path to the conflict contains one decision, namely $x_2 = 0$, we learn the conflict clause:
(k) x_2
Step 4: We now apply a final iteration of the CDCL algorithm with the extended clauses (a-k) as input. Since there are two unit clauses, i.e., x_1 and x_2, we apply unit propagation on literals x_1 and x_2. As a reslut, the clause (d) also becomes a unit clause and $x_3 = 1$, which is in conflict with the clause (f). As the path to the conflict contains no decisions, we can conclude that the original formula is *unsatisfiable*.

---

2 Level C

For grade D you need to have passed level E and obtained 5 (out of 10) points from this section. For passing level C you need 8 points from this section.

1. Consider again the wp-function presented on slide 23 of Lecture 2. In particular, notice that (unlike the vc-function) the wp-function works *without* using an accumulator parameter. Use the same underlying idea to present a modified version of the vc-function (that is, also defined on the source language!) that is still defined by structural induction on C, but does *not* use an accumulator. Explain and justify your new definition. (*Hint*: Combine the translation to Intermediate Language with the wp-function to derive a defining clause for $\mathsf{vc}(\{\eta\}\ \texttt{while}\ B\ \{C\}, \psi)$. Show your derivation as justification.) Explain also how, even though without using an accumulator, your version achieves the same result as the original vc-function. 5p

   Solution: Since we are getting rid of the accumulator, for the non-loop commands we can reuse the defining clauses of the wlp-function, presented on slide 11:

$$\begin{aligned} vc(x = E, \psi) &\stackrel{\mathsf{def}}{=} \psi[E/x] \\ vc(C_1; C_2, \psi) &\stackrel{\mathsf{def}}{=} vc(C_1, vc(C_2, \psi)) \\ vc(\texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}, \psi) &\stackrel{\mathsf{def}}{=} (B \Rightarrow vc(C_1, \psi)) \wedge (\neg B \Rightarrow vc(C_2, \psi)) \end{aligned}$$

   For the annotated loop we can derive the defining clause:

$$vc(\{\eta\}\ \texttt{while}\ B\ \{C\}, \psi) \stackrel{\mathsf{def}}{=} \eta \wedge ((\eta \wedge B \Rightarrow vc(C, \eta)) \wedge (\eta \wedge \neg B \Rightarrow \psi))[vars'/vars]$$

   where $vars$ are the variables that are assigned to in C, and $vars'$ are fresh variables.

This achieves the same as the original vc-function, because it renames the variables that are assigned to in C to fresh ones. With this, we can propagate the accumulated precondition up-ward as a conjunct to η, instead of propagating it separately via an accumulator. Here is a derivation that justifies the last clause (in `typewriter` font):

```
vc({eta} while B {C}, psi)

= {Partial translation to intermediate language (we do not
   translate C to c), where vars are the variables that are
   assigned to in C}

wp(assert eta; havoc vars; assume eta;
     (assume B; C; assert eta; assume false [] assume !B), psi)

= {Composition}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C; assert eta; assume false [] assume !B, psi))

= {Choice}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C; assert eta; assume false, psi) /\ wp(assume !B, psi))

= {Assume}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C; assert eta; assume false, psi) /\ (!B ==> psi))

= {Composition}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C; assert eta, wp(assume false, psi)) /\ (!B ==> psi))

= {Assume}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C; assert eta, false ==> psi) /\ (!B ==> psi))

= {Optimization: false ==> psi is equivalent to true}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C; assert eta, true) /\ (!B ==> psi))

= {Composition}
```

```
wp(assert eta; havoc vars; assume eta,
     wp(assume B; C, wp(assert eta, true)) /\ (!B ==> psi))

= {Assert}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C, eta /\ true) /\ (!B ==> psi))

= {Optimization: eta /\ true is equivalent to eta}

wp(assert eta; havoc vars; assume eta,
     wp(assume B; C, eta) /\ (!B ==> psi))

= {Composition}

wp(assert eta; havoc vars; assume eta,
     wp(assume B, wp(C, eta)) /\ (!B ==> psi))

= {Assume}

wp(assert eta; havoc vars; assume eta, (B ==> wp(C, eta)) /\ (!B ==> psi))

= {Composition}

wp(assert eta; havoc vars, wp(assume eta, (B ==> wp(C, eta)) /\ (!B ==> psi)))

= {Assume}

wp(assert eta; havoc vars, eta ==> (B ==> wp(C, eta)) /\ (!B ==> psi))

= {Optimization: eta ==> (B ==> wp(C, eta)) /\ (!B ==> psi) is
   equivalent to (eta /\ B ==> wp(C, eta)) /\ (eta /\ !B ==> psi)}

wp(assert eta; havoc vars, (eta /\ B ==> wp(C, eta)) /\ (eta /\ !B ==> psi))

= {Composition}

wp(assert eta, wp(havoc vars, (eta /\ B ==> wp(C, eta)) /\ (eta /\ !B ==> psi)))

= {Havoc}

wp(assert eta, ((eta /\ B ==> wp(C, eta)) /\ (eta /\ !B ==> psi))[vars'/vars])

= {Assert}

eta /\ ((eta /\ B ==> wp(C, eta)) /\ (eta /\ !B ==> psi))[vars'/vars]
```

2. In the course, we gave a "local" **Semantics of CTL** (slide 24 of Lecture 5), in the sense that it was given as a satisfaction relation $\mathcal{M}, s \models \phi$ over individual states s. This presentation makes it less suitable for a formal justification of the **Labelling Algorithm** (slides 9-10 of Lecture 6).

3p (a) Define an alternative, *global semantics* of CTL, by means of a *denotation* $\|\phi\|^{\mathcal{M}}$ consisting of all states $s \in S$ that satisfy ϕ. That is, define $\| \phi \|^{\mathcal{M}}$ by *structural induction*, where it suffices (for brevity) to consider just the adequate set from Problem E3 above. You are encouraged to introduce suitable state transformers in order to make the formal definition more elegant.

Solution: This problem is related to Problem 7 of Book, page 248.
Let $\mathcal{M} = (S, \rightarrow, L)$ be a Kripke structure. Define:

$$\begin{array}{rlrl} \| \mathsf{false} \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} \emptyset & \| p \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} \{s \in S \mid p \in L(s)\} \\ \| \neg\phi \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} S - \| \phi \|^{\mathcal{M}} & \| \phi_1 \wedge \phi_2 \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} \| \phi_1 \|^{\mathcal{M}} \cap \| \phi_2 \|^{\mathcal{M}} \\ \| \mathsf{EX}\, \phi \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} pre_{\exists} (\| \phi \|^{\mathcal{M}}) & \| \mathsf{AF}\, \phi \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} pre^{*}_{\forall} (\| \phi \|^{\mathcal{M}}) \\ & & \| \mathsf{E}\, (\phi\, \mathsf{U}\, \psi) \|^{\mathcal{M}} & \stackrel{\mathsf{def}}{=} unt_{\exists} (\| \phi \|^{\mathcal{M}}, \| \psi \|^{\mathcal{M}}) \end{array}$$

where we introduce the state transformers (X and Y range over subsets of S):

$$\begin{array}{rl} pre_{\exists} (X) & \stackrel{\mathsf{def}}{=} \{s \in S \mid \exists s' \in X.\ s \rightarrow s'\} \\ pre^{*}_{\forall} (X) & \stackrel{\mathsf{def}}{=} \{s \in S \mid \forall \pi.\ (\pi(0) = s \Rightarrow \exists i \geq 0.\ \pi(i) \in X)\} \\ unt_{\exists} (X, Y) & \stackrel{\mathsf{def}}{=} \{s \in S \mid \exists \pi.\ (\pi(0) = s \wedge \exists i \geq 0.\ (\pi(i) \in Y\ \wedge\ \forall j < i.\ \pi(j) \in X))\} \end{array}$$

2p (b) Use your global semantics of CTL to formally justify the Labelling Algorithm.
Solution: The algorithm essentially works by structural induction on ϕ, computing $\|\phi\|^{\mathcal{M}}$. The first 5 cases directly follow the corresponding defining clauses of $\| \phi \|^{\mathcal{M}}$.
Case $\mathsf{AF}\, \phi$ uses the unfolding equivalence:

$$\mathsf{AF}\, \phi \equiv \phi \vee \mathsf{AX}\, \mathsf{AF}\, \phi$$

corresponding to the semantic equality:

$$pre^{*}_{\forall} (X) = X \cup pre_{\forall} (pre^{*}_{\forall} (X))$$

expressed with the help of the state transformer:

$$pre_{\forall} (X) \stackrel{\mathsf{def}}{=} \{s \in S \mid \forall s' \in X.\ s \rightarrow s'\}$$

Finally, case $\mathsf{E}\, (\phi\, \mathsf{U}\, \psi)$ uses the unfolding equivalence:

$$\mathsf{E}\, (\phi\, \mathsf{U}\, \psi) \equiv \psi \vee (\phi \wedge \mathsf{EX}\, \mathsf{E}\, (\phi\, \mathsf{U}\, \psi))$$

corresponding to the semantic equality:

$$unt_{\exists} (X, Y) = Y \cup (X \cap pre_{\exists} (unt_{\exists} (X, Y)))$$

3 Level A

For grade B you need to have passed level C and obtained 5 (out of 10) points from this section. For grade A you need 8 points from this section.

1. Consider again your global **Semantics of CTL** from Problem C2a. Show that your semantics is consistent with the local one given in class (slide 24 of Lecture 5). That is, prove by *structural induction* that $s \in \| \phi \|^{\mathcal{M}}$ if and only if $\mathcal{M}, s \models \phi$. State explicitly the induction hypotheses in each inductive case, and indicate where you use them. Show the proofs of (at least) the three cases p, $\phi_1 \wedge \phi_2$ and $\mathsf{EX}\,\phi$. 4p

   Solution:
   Case p. We have:

$$\begin{aligned} s \in \| p \|^{\mathcal{M}} &\Leftrightarrow s \in \{s \in S \mid p \in L(s)\} && \{\text{Def. } \| \phi \|^{\mathcal{M}}\} \\ &\Leftrightarrow p \in L(s) && \{\text{Set theory}\} \\ &\Leftrightarrow \mathcal{M}, s \models p && \{\text{Def. } \mathcal{M}, s \models \phi\} \end{aligned}$$

   Case $\phi_1 \wedge \phi_2$. Assume $s \in \| \phi_1 \|^{\mathcal{M}}$ if and only if $\mathcal{M}, s \models \phi_1$, and $s \in \| \phi_2 \|^{\mathcal{M}}$ if and only if $\mathcal{M}, s \models \phi_2$, for all $s \in S$ (induction hypotheses). We have:

$$\begin{aligned} s \in \| \phi_1 \wedge \phi_2 \|^{\mathcal{M}} &\Leftrightarrow s \in \| \phi_1 \|^{\mathcal{M}} \cap \| \phi_2 \|^{\mathcal{M}} && \{\text{Def. } \| \phi \|^{\mathcal{M}}\} \\ &\Leftrightarrow s \in \| \phi_1 \|^{\mathcal{M}} \text{ and } s \in \| \phi_2 \|^{\mathcal{M}} && \{\text{Set theory}\} \\ &\Leftrightarrow \mathcal{M}, s \models \phi_1 \text{ and } \mathcal{M}, s \models \phi_2 && \{\text{Ind. hyp.}\} \\ &\Leftrightarrow \mathcal{M}, s \models \phi_1 \wedge \phi_2 && \{\text{Def. } \mathcal{M}, s \models \phi\} \end{aligned}$$

   Case $\mathsf{EX}\,\phi$. Assume $s \in \| \phi \|^{\mathcal{M}}$ if and only if $\mathcal{M}, s \models \phi$, for all $s \in S$ (induction hypothesis). We have:

$$\begin{aligned} s \in \| \mathsf{EX}\,\phi \|^{\mathcal{M}} &\Leftrightarrow s \in pre_{\exists}(\| \phi \|^{\mathcal{M}}) && \{\text{Def. } \| \phi \|^{\mathcal{M}}\} \\ &\Leftrightarrow \exists s' \in \| \phi \|^{\mathcal{M}}.\ s \to s' && \{\text{Def. } pre_{\exists}\} \\ &\Leftrightarrow \exists s'.\ (s \to s' \wedge s' \in \| \phi \|^{\mathcal{M}}) && \{\text{Logic}\} \\ &\Leftrightarrow \exists s'.\ (s \to s' \wedge \mathcal{M}, s' \models \phi) && \{\text{Ind. hyp.}\} \\ &\Leftrightarrow \mathcal{M}, s \models \mathsf{EX}\,\phi && \{\text{Def. } \mathcal{M}, s \models \phi\} \end{aligned}$$

2. Consider the following program snippet and show via **Predicate Abstraction** that location `BOOM` is not reachable.

```
x = num;
y = num + 1;
if (x == y) {
   //BOOM
}
```

   (a) Create an abstract *Boolean program* using the set of predicates $P = \{x == num\}$, and explain why location `BOOM` is reachable in the abstract program. Explain why this is not a real counterexample for the original program. 3p

Solution: The predicate $x == num$ yields the following Boolean program.

```
1. bool b0;
2. b0 = true;
3. b0 = b0;
4. if ( * ) {
   //BOOM
}
```

The resulting Boolean program is obtained as follows:

1. We declare a variable $b0$ corresponding to predicate $x == num$.
2. The predicate captures the assignment $x = num$ precisely, since, after the execution of the assignment statement, the predicate is true.
3. The assignment $y = num + 1$ is represented either as $b0 = b0$ or as $skip$, since the predicate cannot capture any relation on variable y, and neither x nor num is modified by the assignment statement.
4. The truth value of the conditional statement is unknown, since the predicate does not capture any relation of variable y.

Although location `BOOM` is reachable in the Boolean program, this is a spurious counterexample since variables x and y will never be equal in the original program.

3p (b) Extend the set of predicates P with additional predicates that allow to prove that location `BOOM` is not reachable.

Solution: We extend the predicate set by adding the predicate $x == y$, hence yielding the following Boolean program.

```
1. bool b0, b1;
2. b0 = true; b1=*;
3. b0 = b0; b1 = b0 ? false : *;
4. if ( b1 ) {
   //BOOM
}
```

The resulting Boolean program is obtained as follows:

1. We declare the variables $b0$ and $b1$ corresponding to predicates $x == num$ and $x == y$, respectively.
2. As before, predicate $b0$ captures the assignment $x = num$ precisely, while predicate $b1$ can have any value since the value of variable x is modified by the assignment statement.
3. As before, assignment $y = num + 1$ is represented either as $b0 = b0$ or as $skip$, since the predicate cannot capture any relation on variable y, and neither x nor num is modified by the assignment statement. On the other hand, predicate $b1$ is $false$ if predicate $b0$ is $true$, since $x == num$ and $y == num + 1$ imply $x \neq y$. Otherwise, if $b0$ is $false$, nothing can be concluded about the value of predicate $b1$.
4. The truth value of the conditional statement is precisely captured by predicate $b0$.

As a result, location `BOOM` is never reachable in the Boolean program, which in turn implies that it is unreachable in the original program.