CS142 Homework Set #6 Solutions

Timur Kuzhagaliyev

November 17, 2017

Problem 1

a) The statement is true. Proof:

$$
\begin{array}{ll}
 & P \textbf{ ensures } \neg P \text{ in } F \\
\equiv & \quad \{ \text{ Definition of ensures } \} \\
 & ((P \wedge \neg\neg P) \textbf{ next } (P \vee \neg P)) \wedge \textbf{transient } (P \wedge \neg\neg P) \text{ in } F \\
\equiv & \quad \{ \text{ Predicate calculus } \} \\
 & (P \textbf{ next } \mathit{true}) \wedge \textbf{transient } P \text{ in } F \\
\equiv & \quad \{ [P \textbf{ next } \mathit{true} \equiv \mathit{true}] \text{ as shown in previous homework sets, } \mathit{true} \text{ is the identity of } \wedge \ \} \\
 & \textbf{transient } P \text{ in } F \\
\equiv & \quad \{ \text{ Definition of transient } \} \\
 & (\exists\, a : a \in F : \{P\}\ a\ \{\neg P\}) \\
\Rightarrow & \quad \left\{ \begin{array}{l} \text{Actions of } F \,[]\, G \text{ are a union of actions in } F \text{ and } G. \text{ If there exists at least one action} \\ \text{that take us from } P \text{ to } \neg P \text{ in } F, \text{ the same action must also exist in } F \,[]\, G, \text{ for any } G. \end{array} \right\} \\
 & (\exists\, a : a \in F \,[]\, G : \{P\}\ a\ \{\neg P\}) \text{ for any } G \\
\equiv & \quad \{ \text{ Definition of transient } \} \\
 & \textbf{transient } P \text{ in } F \,[]\, G \text{ for any } G \\
\equiv & \quad \{ [P \textbf{ next } \mathit{true} \equiv \mathit{true}] \text{ as shown in previous homework sets, } \mathit{true} \text{ is the identity of } \wedge \ \} \\
 & (P \textbf{ next } \mathit{true}) \wedge \textbf{transient } P \text{ in } F \,[]\, G \text{ for any } G \\
\equiv & \quad \{ \text{ Predicate calculus and definition of ensures } \} \\
 & P \textbf{ ensures } \neg P \text{ in } F \,[]\, G \text{ for any } G
\end{array}
$$

b) The statement is false. We can produce a counter-example. Consider the following two programs:

Program	F
var	$x:$ integer
initially	$x = 0$
assign	$x := x + 1$

Program	G
var	$x:$ integer
initially	$x = 0$
assign	$x := 0$

Both F and G have a single action, let's denote these by f and g respectively. Note that $x = 0 \rightsquigarrow x = 2$ in F and $x = 0$ is stable in G. The union of the two programs, $F \,[]\, G$, will have both actions f and g. Now, consider the sequence of actions $(f, g)^\omega$, where ω stands for infinite repetition. This sequence satisfies weak fairness since every action gets called infinitely often, but $x = 2$ never holds because x gets reset to 0 after every increment. Therefore, $x = 0 \rightsquigarrow x = 2$ does not hold in $F \,[]\, G$.

c) This statement is also false, which can be shown using a counter-example. Consider the following two programs:

Program	F
var	$z:$ integer
	$p, q:$ booleans
initially	$z = 0$
$\wedge$	$p = true$
assign	$p \longrightarrow z := z + 1$
$[]$	$q := false$

Program	G
var	$z:$ integer
	$p, q:$ booleans
initially	$z = 0$
$\wedge$	$q = true$
assign	$q \longrightarrow z := z + 1$
$[]$	$p := false$

I'll refer to actions of F as f_1, f_2 and actions of G as g_1, g_2. Note that $z = 0 \rightsquigarrow z = 2$ holds in F and G on their own, but does not hold in $F \,[]\, G$. Action f_2 deactivates the guard on g_1 and action g_2 deactivates the guard on f_1. Consider the sequence of actions $(f_2, g_2, f_1, f_2)^\omega$. This sequence satisfies weak fairness since every action gets called infinitely often, but z never gets incremented and just stays 0 forever. Therefore, $z = 0 \rightsquigarrow z = 2$ does not hold in $F \,[]\, G$.

Problem 2

a) The idea behind superposition is to extend an existing program, while preserving all of its original actions and not modifying any of its original variables. To count how many times the actions of GCD were executed, we can define a program $SuperGCD$ by superposition from GCD.

I will assume that the action was "executed" when it has been picked and its guard was activated.

Program	$SuperGCD$
var	$x, y, count$: integers
	$claim$: boolean
initially	$count = 0$
$\wedge$	$claim = false$
transform	each statement s in GCD to
s $\|\|$	$count := count + 1$
add	
	$x = y \longrightarrow claim := true$

In my **transform** section, I assume that actions used to augment actions of GCD inherit the same guard. If augmented actions can have 2 different guards, one could add the $x \neq y$ guard to the right hand side in the **transform** section, which would achieve the same logic.

Clearly, for $claim$ to become $true$, its guard must be activated. This implies $x = y$, so no other action can be executed by definition of GCD and our augmented actions. Therefore we consider $x = y \wedge claim$ to be the fixed point of $SuperGCD$.

b) To prove the correctness of $SuperGCD$, we need to prove that it satisfies the safety and progress properties. Let C be the actual action execution count and $terminated$ be boolean which holds iff GCD has terminated. I'll define 2 invariants that guarantee safety: **invariant**$(count \leq C)$ and **invariant**$(claim \Rightarrow terminated)$.

We can use the definitions of actions in $SuperGCD$ to prove **invariant**$(claim \Rightarrow terminated)$. Initially, $claim$ is false, so the property holds. As stated before, $claim$ only becomes $true$ when the guard on the relevant command is activated, what implies $x = y$. To show that this condition only holds when GCD has terminated, we can use the definition of GCD. We know that actions of GCD are only applied when $x \neq y$. When $x = y$, all guards in GCD are deactivated and no assignments occur, hence $termianted$ holds (see *aside* for details). Therefore **invariant**$(claim \Rightarrow terminated)$ holds.

Aside: One could argue that GCD could reach fixed point when one of the variables is zero and the other is greater than zero (e.g. $x = 10, y = 0$), hence $x \neq y$. This cannot occur since both x and y start with values strictly greater than zero, and the smaller number gets subtracted from the bigger number until equality is reached. Before equality is reached, subtracting the smaller value from the bigger one cannot generate a 0.

We can also show that **invariant**$(count \leq C)$ holds. Initially, $count$ is zero and since C is bounded below

by zero, the property holds. The actions of $SuperGCD$ that increment *count* were created by augmenting actions of GCD. Additionally, these actions inherit the same guards. This implies that the only time that *count* is incremented when an action of GCD is actually executed, hence *count* cannot exceed the true count C. Therefore **invariant**$(count \leq C)$.

Now that we have proved safety, we can use *count* as our metric. Using the invariant, we've shown that the metric is bounded above by C. We also know that the metric cannot decrease, since any action changing *count* increments it by 1. The metric is also guaranteed to increase eventually: we know that until $x = y$ holds, one of the two actions in GCD can be picked to increment *count* by one. Moreover, the count will increment only until it is equal to C, since when $count = C$, underlying GCD would have finished executing actions (by definition of C), so $x = y$ will hold and guards on both actions of GCD would be deactivated.

Above I have (implicitly) shown that $terminated \rightsquigarrow claim$ and that when GCD terminates, *count* is equal to the actual count. Therefore the program $SuperGCD$ is correct.

Problem 3

a) I would use definition of priority #3, as it is the only one that guarantees safety (graph is acyclic) and progress (everyone gets their turn eventually). The first proposed definition guarantees safety since the "greater than" relation is transitive and the agent ID is used to break ties, but it does not guarantee progress since the process with highest time can just keep using the resources repeatedly. The second definition does not guarantee safety since it has no way of breaking ties and the behaviour in case of a tie is undefined.

b) We can prove safety by showing that "priority graph is acyclic" property is invariant with our definition of priority.

Initially, all agents are thinking. In this case, the [agent's current clock value, agent id] pair is used to determine priority. Not that this relationship is transitive. Let's denote this pair as p: $(p_t < p_u \wedge p_u < p_v) \Rightarrow p_t < p_v$. The transitivity of this relationship, and the fact that is not symmetric $(p_t < p_u \not\equiv p_u < p_t)$ guarantees that the graph is initially acyclic.

Now we can show that the "priority graph is acyclic" property is stable. The same idea of transitivity applies here too: for any action in the system, the agent will adjust the first value in the tuple (the clock value). If the agent has just begun waiting or is inside CS, it's pair will show the clock time corresponding to the moment when it begun waiting (since an agent has to go through waiting before entering CS). If the agent has left CS, it will set the clock value in its pair to its current time. In both cases, the pair still contains the unique agent ID, meaning that the pairs can still be sorted, even in case of ties. The same fact that the relationship is transitive and not symmetric as in paragraph above applies here, guaranteeing that the graph is acyclic.

Therefore the "priority graph is acyclic" property is invariant, and hence the safety property is satisfied. The same idea can be expressed by showing that with out definition of priority, the graph is topologically sorted and hence must be acyclic.

c) In our case, lower numbers mean higher priority. To prove that the progress property is satisfied, we need show that every hungry agent gets to eat eventually. Let's define a metric M as the number of agents

that have higher priority than the current (waiting) agent.

M is clearly bounded below by 0 (and above by the total number of agents minus one). It can be shown that M is guaranteed to not increase. This is the consequence of the fact that the clocks always tick forward - if some other agent has a clock time higher than the current agent (and hence does not contribute to M), there is no action it can perform to get the clock time to be lower than ours and increase M (remember that current agent is waiting).

M is also guaranteed to decrease eventually if it's not equal to 0. If M is equal to 0, the current agent is at the top of the (topologically sorted) graph, so it can execute, and the progress property is satisfied. If the agent is not at the top of the graph, it means that there are $k = M$ agents above it. For all of the agents that are not waiting, the time will continue increasing and it will eventually exceed the time at which the current process begun waiting, decreasing M to some $k^* \leq k$. Now, we know that k^* processes above the current process are waiting. Additionally, one of them must be at the top of the graph. This process will enter and exit CS, meaning that its clock in the pair will now be its current clock reading, which will also eventually overtake current process' time, decreasing M to $k^* - 1$. We can repeat this logic for all of the other processes above the current process until M reaches 0 and the current process can enter CS.

Therefore the progress property is also satisfied.