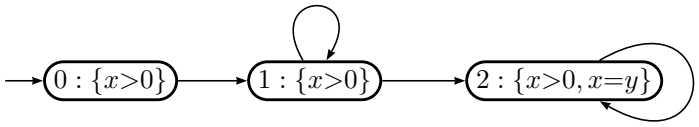
Exam Program Verification 2007/2008
UNNIK-211, 01-07-2008, 09:00-12:00

Lecturer: Wishnu Prasetya

July 18, 2008

1. [2.5 pt] Consider this Kripke structure K:

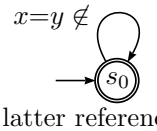


The set of states of this automaton is $\{0, 1, 2\}$, with 0 as the initial state. The set *Prop* of atomic observations is $\{x{>}0,\ x{=}y\}$. The labeling of every set with these observations are as given in the above picture.

We are going to do LTL model checking to verify whether the above 'program' satisfies the property: $\mathsf{true}\ \mathbf{U}\ (x{=}y)$.

(a) Draw a Buchi automaton B representing $\neg(\mathsf{true}\ \mathbf{U}\ (x{=}y))$.

Answer: well, since $\mathsf{true}\ \mathbf{U}\ q\ =\ \Diamond q$, the above is equivalent to $\Box\neg(x{=}y)$. A Buchi automaton accepting it is:



For latter reference we name the only state above a.

(b) Give the formal definition of the Buchi automaton you draw in (a). That is, describe it in terms of a tuple (Σ, Q, ρ, I, F); what are your Σ in this case, your Q, your ρ and so on.

Answer:

i. $\Sigma = \{\emptyset, \{x{>}0\}, \{x{=}y\}, \{x{>}0,\ x{=}y\}\}$ (set of labels)

ii. $Q = \{a\}$ (the set of states)

iii. ρ (the transition relation) is such that

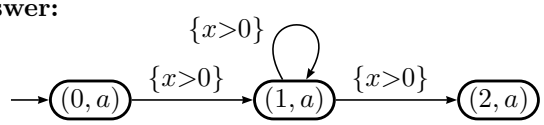
$$\rho\ (a, L) = \{0\} \quad \text{, if } x{=}y \notin L$$

for all other cases there is no transition; so $\rho\ (a, K) = \emptyset$.

iv. $F = \{a\}$ (the set of accepting states)

(c) Construct the automaton $K \cap B$.

Answer:



where all states are accepting.

Question: Explain why we need this automaton for model checking your property.

Answer: Because it allows us to search for an (infinite) run accepted by the above $K \cap B$ automaton (as a Buchi). Such a run corresponds to an execution that can be generated by K, and is accepted by B; thus our counter example witnessing that the property represented by the negation of that of B is not valid.

(d) So, according to your $K \cap B$, does your original program K satisfy the property $\mathsf{true}\ \mathbf{U}\ (x{=}y)$?

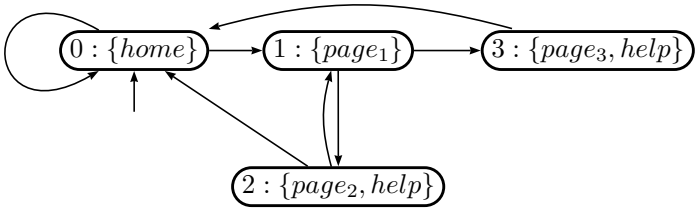
If yes, explain why. If no, give your counter example in terms of a path in $K \cap B$.

Answer: No. The counter example would be the infinite run $(0, a), (1, a), (1, a)...$ (keep cycling in $(1, a)$). This run is an accepting run in $K \cap B$.

(e) SPIN uses a nested DFS algorithm to find a counter example. Explain why a single DFS is not sufficient.

Answer: the top level DFS is needed to cover all accepting states of $K \cap B$; at each accepting state, a new DFS is started to check where it is cyclic. If it is, then we find an accepting run.

2. [2.5 pt] Consider this Kripke structure K:



The set of states of this automaton is $\{0, 1, 2, 3\}$, with 0 as the initial state. The set *Prop* of atomic observations is:

$$\{home, page_1, page_2, page_3, help\}$$

The labeling of every set with these observations are as given in the above picture.

(a) We want to express that when *home* is true, there *exists* a path leading to a state where $page_3$ is true. How to express this in CTL?

Answer: $home \rightarrow \mathbf{EF}page_3$

(b) Can we express the same thing in LTL? If yes, give the formula. If no, motivate why not.

Answer: no we can't. An LTL property must hold over all executions (which in LTL are sequences). So in LTL $K \models \varphi$ means that $\sigma \models \varphi$ hold for *all* sequence σ of execution that can be generated by K, starting from its initial state.

So in LTL we can't express a property that only holds for *some* execution.

(Though, $K \not\models \neg\Diamond page_3$ does mean that there exists therefore an execution satisfying $\Diamond page_3$; though $\not\models$ is strictly speaking not an LTL, but it is something at the meta level with respect to LTL)

(c) We will do CTL model checking to check if the automaton K above satisfies the property $\mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]$. The model checking algorithm proceeds by iteratively labeling the states of K with formulas. Fill in the following tabel to reflect the first 4 iterations of your model checking procedure:

Answer: the entries in red in the table below.

Iteration	State			
	0	1	2	3
0	$\{home\}$	$\{page_1\}$	$\{page_2, help\}$	$\{page_3, help\}$
1	$\{home\}$	$\{page_1\}$	$\{page_2,\ help,\ \mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]\}$	$\{page_3, help,\ \mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]\}$
2	$\{home\}$	$\{page_1,\ \mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]\}$	as above	as above
3	$\{home\}$	$\{page_1,\ \mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]\}$	as above	as above

(d) When should we terminate the iterations in the above model checking procedure?

Answer: after the labelling does not change anymore. So in the above example after the 4^{th} iteration.

(e) Does K satisfy $\mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]$, according to your model checking? Explain.

Answer: no. In CTL $K \models \varphi$ iff *phi* holds on K's initial state. After the labelling above, the initial state 0 is not labelled by the formula ($\mathbf{A}[\mathsf{true}\ \mathbf{U}\ help]$), implying it does not hold there.

3. [2.5 pt] Consider the two CSP processes given below. The alphabet of both is $\{a, b\}$.

$$P_1 = (b \rightarrow STOP) \ \square \ (a \rightarrow a \rightarrow P_1)$$

$$P_2 = (b \rightarrow STOP) \ \sqcap \ (a \rightarrow a \rightarrow P_2)$$

(a) Describe the kind of traces that can be generated by P_1 and P_2. So, how are these processes ralated to each other in terms of trace-based refinement?

Answer:

Both P_1 and P_2 generate traces with even number of a's, followed by a single b, or any prefix thereof. So, this set of traces:

$$\{a^{2k}b \mid k{\geq}0\} \ \cup \ \{a^k \mid k{\geq}0\}$$

Since the generate the same set of traces, then they are equivalent in terms of trace-based refinement (though not necessarily so in terms of the more powerful failure-based refinement).

(b) Give all failures of P_1 respectively P_2 whose traces are of *length 2 or less.*

Answer: for P_1:

trace length	failures
0	$(\langle\rangle, \emptyset)$
1	$(\langle a\rangle, \emptyset),\ (\langle a\rangle, \{b\}),\ (\langle b\rangle, \emptyset),\ (\langle b\rangle, \{a\}),\ (\langle b\rangle, \{b\}),\ (\langle b\rangle, \{a, b\})$
2	$(\langle\rangle, \emptyset)$

For P_2:

trace length	failures
0	$(\langle\rangle, \emptyset),\ (\langle\rangle, \{a\}),\ (\langle\rangle, \{b\})$
1	$(\langle a\rangle, \emptyset),\ (\langle a\rangle, \{b\}),\ (\langle b\rangle, \emptyset),\ (\langle b\rangle, \{a\}),\ (\langle b\rangle, \{b\}),\ (\langle b\rangle, \{a, b\})$
2	$(\langle\rangle, \emptyset),\ (\langle\rangle, \{a\}),\ (\langle\rangle, \{b\})$

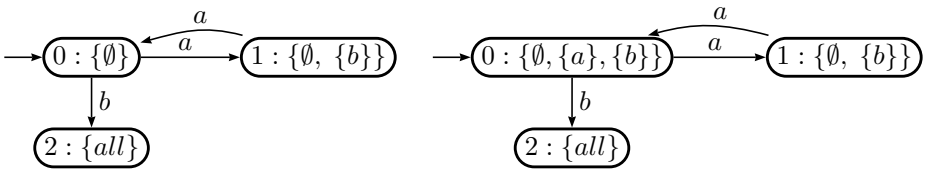
(c) Does $P_1 \sqsubseteq P_2$ hold under the failures semantic? How about $P_2 \sqsubseteq P_1$?

Answer: $P_1 \sqsubseteq P_2$ does *not* hold, because as you can see in the table above there are some failures of P_2, e.g. $(\langle a\rangle, \{b\})$ which are not included in P_1.

However, the reverse $P_2 \sqsubseteq P_1$ does hold.

(d) Draw the automata M_1 and M_2 representing P_1 and respectively P_2. Label each state of these automata with its refusals.

Answer: left is M_1 and right is M_2.

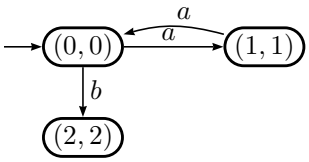


(e) Construct the automaton $M_1 \cap M_2$, and explain how we can check $P_1 \sqsubseteq P_2$ using $M_1 \cap M_2$. So, does $P_1 \sqsubseteq P_2$ hold according to your refinement checking procedure?

Answer: the intersection automaton $M_1 \cap M_2$ is shown below. We do not label the states with refusals, and such labelling has no purpose in the context of what we want to use $M_1 \cap M_2$ for (which is to check $P_1 \sqsubseteq P_2$).

The states of $M_1 \cap M_2$ have the form (i, j) with i corresponds to the state of M_1, and j that of M_2.

Without the refusal labelling, it is not surprising that we obtain exactly the same automaton as M_1 and M_1; we have ealier concluded that their traces are the same.



To check refinement, we have to check for every state (i, j), in the above automaton that:

i. $initials(i)$ (the set first actions possible after state i in M_1) subsumes $initials(j)$ (the set first actions possible after state i in M_1).
This is to check trace inclusion.

ii. $refusals(i)$ (the set refusals in state i in M_1) subsumes $refusals(j)$.
This is to check refusal inclusion.

We have to check this for every state in $M_1 \cap M_2$; we can e.g. traverse the its states in depth-first order.

As we do so, when checking the state $(0, 0)$ we'll find out that $refusals(0)$ in M_1 does *not* subsume $refusals(0)$ in M_2. So the refinement $P_1 sqsubseteq P_2$ does not hold.

4. [2.5 pt] Consider the following concepts:

> "Abstractly, a program P can be seen as a function that maps an initial state to a set of possible end-states. If P does not terminate when executed on an initial state s, we will express this by mapping s to an empty state. That is, in our abstract representation $P\ s = \emptyset$.
>
> A program P always terminates if for *all* (intial) state s, $P\ s$ is not empty. It follows that if P and Q are two programs that always terminate, so does $P; Q$."

We want to express those concepts in HOL (to eventually prove the claimed theorem, though we will not do so here).

(a) Give a HOL type that will be sufficient to represent the above abstract concept of "program", then give HOL definitions that capture the concepts "P always terminate" and "$P; Q$".

Answer: since the text above does not say anything about the structure of a state, we'll let it generic. We'll represent it with a type variable `'state`.

We'll choose to represent "a set of states" by a predicate, encoded by a function of type `'state->bool`. This gives the following representation programs. That is, we represent a program by a function of this type:

```
'state -> 'state -> bool
```

(b) Write a formula capturing the theorem "if P and Q are two programs that always terminate, so does $P; Q$"

Answer: Let's first define the concept 'always terminate':

```
Define ‘alwaysTerminate P = (!s. (?t. P s t))‘
```

`P` here is a 'program'; so a function of the type given in (a). The definition above says that for all starting state s, there is some state t which is related to s. So, $P\ s$ could not be empty, which means according to our abstract description above that P terminates.

We also need to model the ";" composition in HOL; this is quite easy. For convenience I'll assume that the name `THEN` has been declared as an infix.

```
Define ‘P THEN Q = (\s u. (?t. P s t /\ Q t u))‘
```

Now the requested situation can be captured by:

```
alwaysTerminate P /\ alwaysTerminate Q  ==>  alwaysTerminate (P THEN Q)
```

(c) In HOL a goal has the following type:

$$\texttt{type}\ goal\ =\ (term\ list) * term$$

and a tactic has the following type:

$$\texttt{type}\ tactic\ =\ goal \rightarrow ((goal\ list)\ *\ (thm\ list \rightarrow thm))$$

So, when given a goal v, a tactic tac will produce a pair (z, f). Explain the roles of z and f and their relations to v.

Answer: f is the so-called *proof function*. When a tactic manage to reduce a goal g to subgoals, each of which is either an axiom or can be proven directly form the primitive inference rule, we're basically done. But in HOL we still need to convert the goal g to a theorem. A tactic cannot do this on its own, because in HOL the only way to produce a theorem is by calling its primitive inference rules (or by composing them).

So this is where the proof function is for. It calls the primitive rules and compose them to convert g to a theorem. The input is the theorems that correspond to the proven subgoals of g.

(d) Let's now apply the above understanding. Write the combinator `TRY` that will behave as follows. Given a tactic *tac*, `TRY tac` will apply *tac* on the given goal. If it succeeds, then we are done. However if *tac* fails on the goal (that is, if it throws an exeception), then we do nothing with the original goal (and throws no exception).

You can write `TRY` in ML, Haskell, or even in some pseudo imperative language.

Answer: here is in ML:

```
fun TRY tac goal = tac goal handle _ => ALL_TAC goal
```

So, it applies `tac` to `goal`. If it throws an exception, then we'll just apply $\texttt{ALL}_\texttt{T}\texttt{AC}$, which is just a skipping/identity tactic (it does not do anything).