Exercises PV 09/10

Wishnu Prasetya

April 16, 2012

CSP and Refinement Checking

1. Let's write the $M_{<2}$ protocol in CSP. Express the sender and the receiver as separate processes (because it seems convenient to do so). Take into account that you somehow has to express:

   (a) the fetching of a new data package by the sender.

   (b) the acceptance of a package by the receiver.

   (c) corrupted package.

   Answer: The process SENDER and RECEIVER are described below. The alphabet of each is just all the events that each process can produce.

$$
\begin{array}{lcl}
\mathsf{SENDER} & = & \mathsf{FETCH} \\
\mathsf{FETCH} & = & fetch \rightarrow (data_1 \sqcap data_{error}) \rightarrow \mathsf{SRDY} \\
\mathsf{SRDY} & = & (ack_1 \rightarrow \mathsf{FETCH}) \\
 & & \Box \\
 & & (ack_0 \rightarrow (data_1 \sqcap data_{error}) \rightarrow \mathsf{SRDY}) \\
 & & \Box \\
 & & (ack_{error} \rightarrow (data_0 \sqcap data_{error}) \rightarrow \mathsf{SRDY}) \\
\mathsf{RECEIVER} & = & \mathsf{RRDY} \\
\mathsf{RRDY} & = & (data_1 \rightarrow \mathsf{ACCEPT}) \\
 & & \Box \\
 & & (data_0 \rightarrow (ack_1 \sqcap ack_{error}) \rightarrow \mathsf{RRDY}) \\
 & & \Box \\
 & & (data_{error} \rightarrow (ack_0 \sqcap ack_{error}) \rightarrow \mathsf{RRDY}) \\
\mathsf{ACCEPT} & = & (ack_1 \sqcap ack_{error}) \rightarrow accept \rightarrow RRDY
\end{array}
$$

   The specification can be expressed as follows:

$$\mathsf{SPEC} \sqsubseteq (\mathsf{SENDER} \parallel \mathsf{RECEIVER})/Internals$$

   where:

$$\mathsf{SPEC} = fetch \rightarrow accept \rightarrow \mathsf{SPEC}$$

   and $Internals$ are all events except $fetch$ and $accept$. So:

$$Internals = (\alpha\mathsf{SENDER} \cup \alpha\mathsf{RECEIVER})/\{fetch, accept\}$$

2. Give a CSP process R which is equivalent with $P||Q$, where:

$$\begin{aligned} P &= x \to ((a \to P) \;\square\; (b \to P)) \\ Q &= y \to a \to Q \end{aligned}$$

with $\alpha P = \{x, a, b\}$ and $\alpha Q = \{y, a, b\}$. That is we want to have an expression directly in terms of the underlying subprocesses, that equivalently describes $P||Q$.

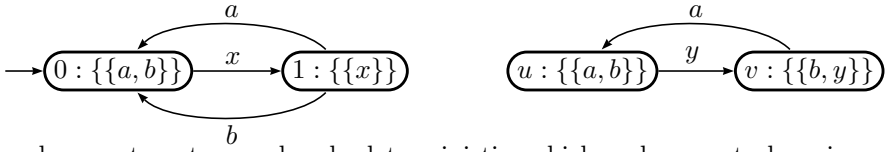
Answer:

One way to 'expand' $P||Q$ is by calculating it with calculation laws, if we have sufficient of them. [1] provides lots of laws, e.g. to calculate $(a \to P_1) \;||\; (b \to P_2)$. Unfortunately [1] does not provide enough laws for $||$; e.g. a law to calculate $(P_1 \square P_2)||P_3$ is missing.

We can also construct it by first constructing the automata that represent $P||Q$. First, we construct the automata for P and Q —see below.

If you choose to interpret CSP processes under its trace semantics, then the labeling with refusals are not necessary (trace semantic disregard refusals). If you interpret them under the failure semantic then of course you need the refusals information. In this I will do the latter.

Below, I only show the maximal refusals of each state. Note that if in a state s a process can refuse V (a set of events), it can also refuse any subset $V' \subseteq V$. E.g. in state 0, P thus can also refuse $\{a\}, \{b\}$, and .



The above automate are already deterministic, which makes our task easier. If they are not, you may want to convert them to deterministic ones first. Do be careful how you re-assign the refusals.

It is by the way just a coincidence that every state above has just a single maximal refusal. In general it can be more.

The automaton for $N = P||Q$ can be constructucted as a form of 'product' of the two automata above. Think N as simulating interleaved execution of P and Q (but keep in mind that we have to synchronize over common events).

That states of N are pairs like $(0, u)$. Its initial state is (s_0, t_0) where this s_0 and t_0 are the intial states of P respectively Q. N can only move from state (s_1, t_1) to (s_2, t_2) via an event a, if either the following is true:

(a) a is P's own event (not an event it has to synchronize with Q), and a moves the automaton of P from s_1 to s_2. Q should remain on its place; so $t_2 = t_1$. This transition of N simulates an interleaved execution of P's a.

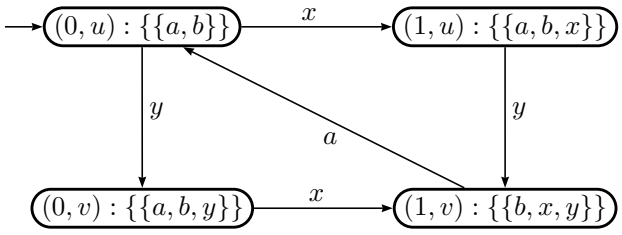
(b) a is Q's own event (not an event it has to synchronize with P), and a moves the automaton of Q from t_1 to t_2. P should remain in its place; so $s_2 = s_1$. This transition of N simulates an interleaved execution of Q's a.

(c) a is an event where P and Q have to synchronize, and a moves P from s_1 to s_2 and Q from t_1 to t_2. This transition of N simulates a synchronized execution of a.

The refusals of each combined state can be calculated according to the definition of refusals for ||. So:

$$refusals((s,t)) \;=\; \{X \cup Y \mid X{\in}refusals(s) \;\wedge\; Y{\in}refusals(y)\}$$

The resulting N is shown below:



which when you 'reverse engineer' you get:

$$\begin{array}{lcl} R_0 & = & (x \to R_1) \;\Box\; (y \to R_2) \\ R_1 & = & y \to R_3 \\ R_2 & = & x \to R_3 \\ R_3 & = & a \to R_0 \end{array}$$

with $\alpha R_i = \{x, y, a, b\}$.

3. Consider these processes:

$$P \quad = \quad (a \to STOP) \;\;\Box\;\; (a \to ((b \to P) \;\sqcap\; (a \to STOP)))$$

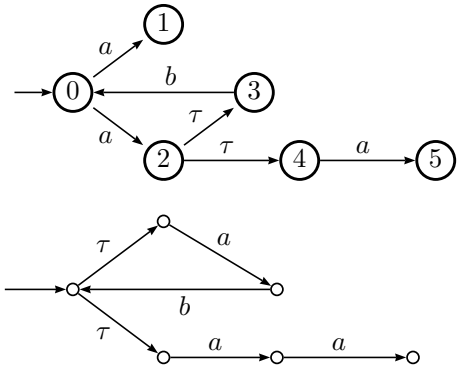
$$Q \quad = \quad (a \to b \to Q) \;\sqcap\; (a \to a \to STOP)$$

We want to check whether $P \sqsubseteq Q$ (or the other way around) under the trace semantic. How does the refinement checking procedure proceed?

Answer:

At the top level the procedure is as follows.

(a) Construct the automata M_P and M_Q representing P respectively Q. If they are non-deterministic, convert them to deterministic one. The conversion should be as such that it preserves the generated set of traces. For the above example, M_P and M_Q are shown below (in that order). I also name the states of M_P for later reference.



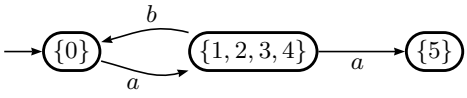
These automata are non-deterministic. We can turn them to deterministic ones. Let's do this first for M_P. The idea is to construct an automaton N_P that drives/simulates M_P. That is, N_P generates exactly the same traces as M_P. However, each state of N_P is actual a non-empty set of M_P's original states. This represents the set of possible states of M_P after doing the same trace in N_P. The initial state of N_P is S_0 consisting of M_P's initial state s_0, and all state of M_P that are reachable from s_0 with only τ-steps.

When N_P reach a state U, it means that it has driven M_P in such a way that the latter can be in any state in U. Now, N_P can only make a transition from state U to a non-empty state V with an event a iff

$$V \;=\; \{v \mid (\exists u \in U.\; u \stackrel{\tau^* a \tau^*}{\longrightarrow} v)\}$$

where $u \stackrel{\tau^* a \tau^*}{\longrightarrow} v$ means that v can be reached (in M_P) from u by doing a single event a, possibly mixed with any number of τ-steps. The important thing to note here is that an N_P constructed in this way preserve the set of traces of the original M_P.

Following this procedure we obtain this N_P:



It turns out applying the same procedure to M_Q gives us an automaton of the same shape. It differs from N_P only in the 'identifications' of the states (the sets labelling each state above). However, for the purpose of comparing the traces ses of both automata these identifications are irrelevant. So, in this respect we simply get the *same* deterministic automaton N_Q for M_Q.

(b) It happens in this example, that both N_P and N_Q are equal, so obviously the generate the same set of traces. Therefore, under the traces semantics P and Q are equivalent (which implies refinement in both ways).

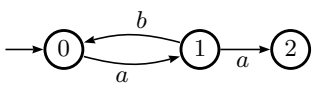
But in general, in case N_P and N_Q are distinct, we proceed by constructing an automaton representing $N_P \cap N_Q$. The states of this automaton are of the form (u, v) where u is a state from N_P and v is a state of N_Q. As we proceed in constructing $N_P \cap N_Q$, check at each new state (u, v) added to it whether $initials(u) \supseteq initials(v)$. If this ever fails to hold, the Q does not refine P. We'll do this in the next exercise.

Redo the question above to check $P \sqsubseteq R$ and $R \sqsubseteq P$ (in trace semantic), where P is as above, and R is defined as below:

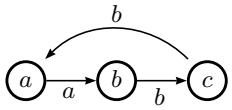
$$R \;=\; a \rightarrow (b \rightarrow (b \rightarrow R))$$

Answer:

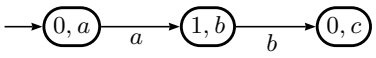
(a) As before first we construct deterministic automata representing P and R. That of P has been given above. I'll show it again below; I also name the states for later references. Let's call the automaton M_P:



That of R is straight forward to construct; let's call it M_R:



(b) We construct the automaton $M_P \cap M_Q$; which results in the automaton N below. You can think N as simultaneously driving M_P and M_Q. That states of N are pairs like (u, v). Its initial state is (s_0, t_0) where this s_0 and t_0 are the intial states of M_P respectively M_Q. When N is in a state (u, v) it means that it has driven M_P to the state u and M_Q to state v. N can only move from a state (u_1, v_1) to state (u_2, v_2) with an event a if this move is possible in both M_P and M_Q. That is, if a moves M_P from state u_1 to u_2, and a moves M_Q from state v_1 to v_2. Consequently, N can only produce traces can be produced by both M_P and M_Q. Our N is:



In traces semantics $P \sqsubseteq R$ iff $traces(P) \supseteq traces(R)$. According to our theory this is the case iff for any state (u, v) in N, $initials(u)$ (in M_P) subsumes ($\supseteq$) $initials(v)$ (in M_R).

This is unfortunately not the case here: $initials(0) = \{a\}$, whereas $initials(c) = \{b\}$. So for state $(0, c)$ of N:

$$initials(0) \not\supseteq initials(c)$$

which thus implies that the refiment $P \sqsubseteq R$ does *not* hold.

Furthermore, we can expect N to be constructed incrementally (the algorithm that constructs it does not suddenly produce the entire N). So we can stop the construction as soon as we find such a contradicting state (u, v) as described above.

The above answer the question of whether $P \sqsubseteq R$. I leave the other question, whether the reverse $R \sqsubseteq P$ holds, to you.

4. Prove the following under the trace semantic:

   (a) $P \;\Box\; STOP \;=\; P$

   (b) $P \;\sqcap\; STOP \;=\; P$

   Do they still hold under the failure semantic?

   Answer:

   $traces(P \;\Box\; STOP)$
   $=$
   $traces(P) \cup traces(STOP)$
   $=$
   $traces(P) \cup \{\langle\rangle\}$
   $=$ // $\langle\rangle$ is in the traces of any proces; thus also in P
   $traces(P)$

The proof for $\sqcap$ is the same, because as with $\Box$, $traces(P \sqcap Q) = traces(P) \cup traces(Q)$.

(b) does not hold under the failures semantic, as shown in this counter example. Suppose the alphabet is $\{a\}$ and P is just $a \rightarrow STOP$.

$$
\begin{array}{l}
refusals(P \sqcap STOP) \\
= \\
refusals(P) \cup refusals(STOP) \\
= \\
\{\emptyset\} \cup \{\emptyset, \{a\}\} \\
= \\
\{\emptyset, \{a\}\}
\end{array}
$$

which implies that $(\langle\rangle, \{a\})$ is a failure of $P \sqcap STOP$. But it is not a failure of P. So they are not equal.

(a) still holds in the failures semantic. I'll leave the work out to you.

5. What is a 'failure' in CSP?

   Answer:

   A *failure* is a pair (s, X) where s is a trace and X is a refusal. When we say (s, X) is a failure of a process P it means that P can produce the trace s, after which it may refuse to do the choice of events as offered (by the environment) in X.

   Describe the failure sets of the following processes. Assume $\{a, b\}$ as the alphabet.

   (a) $(b \rightarrow STOP) \Box STOP$

   Answer:

   i. $(\langle\rangle, \emptyset)$, $(\langle\rangle, \{a\})$
   ii. $(\langle b\rangle, X)$, for all $X \subseteq \{a, b\}$

   (b) $(b \rightarrow STOP) \sqcap STOP$

   Answer:

   i. $(\langle\rangle, X)$, for all $X \subseteq \{a, b\}$
   ii. $(\langle b\rangle, X)$, for all $X \subseteq \{a, b\}$

   (c) $P = a \rightarrow ((b \rightarrow P) \Box STOP)$

   Answer:

   i. $(\langle\rangle, \emptyset)$, $(\langle\rangle, \{b\})$
   ii. $(\langle a\rangle, \emptyset)$, $(\langle a\rangle, \{a\})$
   iii. $(\langle ab\rangle, X)$, for all $X \subseteq \{a, b\}$

   (d) $Q = a \rightarrow ((b \rightarrow Q) \sqcap STOP)$

   Answer:

   i. $(\langle\rangle, \emptyset)$, $(\langle\rangle, \{b\})$
   ii. $(\langle a\rangle, X)$, for all $X \subseteq \{a, b\}$
   iii. $(\langle ab\rangle, X)$, for all $X \subseteq \{a, b\}$

   So, does P refines Q under the failures semantic? (or perhaps the other way around, or perhaps neither?)

   Answer: $P \not\sqsubseteq Q$, but $Q \sqsubseteq P$.

6. Redo exercise No. 3, but now using the failures semantic. That is, refinement is now defined in terms of failures, and you're asked to perform the refinement checking to check whether $P \sqsubseteq Q$ and whether $P \sqsubseteq R$.

Answer:

Tthat of $P \sqsubseteq R$ is easier to answer. Recall that failures-based refinement implies traces-based refinement. That is:

> *If $P \sqsubseteq R$ under the failures semantics then it also holds under the traces semantics.*

So, the contra-position of this also holds:

> *If $P \not\sqsubseteq R$ under the traces semantics then it also doesn't hold under the failures semantics.*

Since we have shown previosly that $P \not\sqsubseteq R$ under the traces semantics, the conclusion for the failures semantics is then obvious.

So, I'll also show the refinement checking for $P \sqsubseteq Q$. Recall that this relation holds under the traces semantics. We'll see if it also holds under the more discriminative failures semantics.

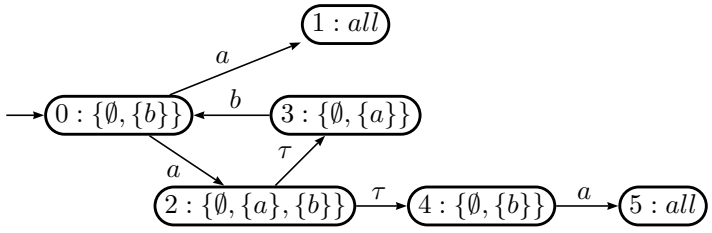
At the top level the refinement checking is now a bit different:

(a) Construct the automata M_P and M_Q representing P respectively Q. These automata should retain the non-deterministic choices that were possible in the original P and Q.

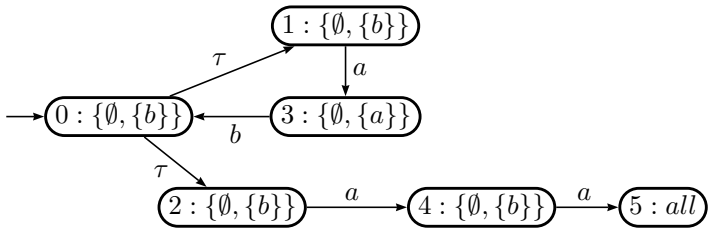
These automata have been shown before, but now we additionally label the states with the set of refusals that correspond to those states. Again, below I will only write the maximal refusals of each state.

That is if u is a state of, say, M_P, we label u with the set of offers (within M_P's alphabet) that M_P may refuse when it is in that state u, or in any other state u' which can be reached from u by doing only τ-steps. We will also name the states.

This is M_P:

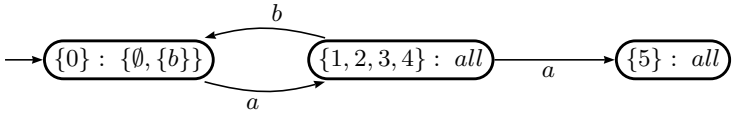


And M_Q:

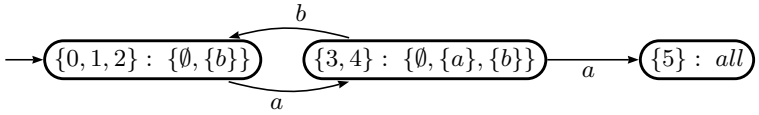


(b) As before, now we collapse both automata to deterministic versions. However we will keep the refusals information. More presicely, as we construct the deterministic version N_P from the above M_P; recall that each state of N_P is a set of M_P's original states.

When U is a state of N_P the set of possible refusals that correspond to this state is thus the union of all refulsals of all states in U in the original M_P. For example, if $U = \{p, q\}$ then $refusals(U) = refusals(p) \cup refusals(q)$. So, we obtain the following N_P:

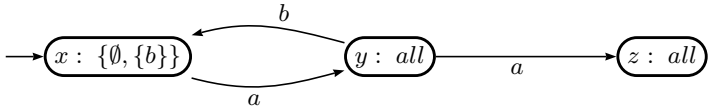


And this N_Q:

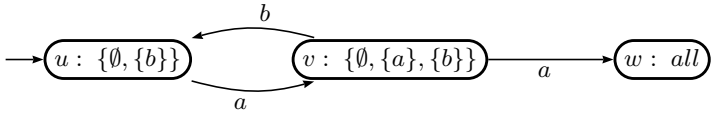


N_P and N_Q have the exactly the same shape; so the generate the same set of traces. However, notice that they don't have the same refusals.

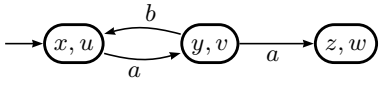
Let's first just remove the sets like $\{0, 1, 2\}$ that label the aboves states; they're irrelevant now. We'll also give the states new names for later reference. So, N_P and N_Q are now as below (in that order):



And this N_Q:



Constructing $N_P \cap N_Q$ gives, not surprisingly, the same automaton:



Also, not surprisingly, for any state (s, t) in the above $N_P \cap N_Q$ we have that $initials(s)$ in N_P subsumes $initials(t)$ in N_Q. Actually we have in this case that both initials are the same. These imply $P \sqsubseteq Q$ and $Q \sqsubseteq P$ under the traces semantics.

For the relations to also hold under the failures semantics, then it should also be the case that $refusals(s)$ in N_P subsumes $refusals(t)$ in N_Q. This is the case! So we conclude $P \sqsubseteq Q$ under the failures semantics.

However, we see that $refusals(v)$ does *not* subsume $refusals(y)$, whereas (y, v) is a state in $N_P \cap N_Q$. This implies that the reverse relation $Q \sqsubseteq P$ does *not* hold!

References

[1] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 2004.