ASSET Exercises

Antti Valmari
Tampere University of Technology

2017-10-10

Instructions and General Remarks

Some of the exercise descriptions are **incomplete on purpose**. The idea is that the student develops a solution by trial and error, and also reports on at least some erroneous attempts.

A component may **directly communicate** with another in the model only if a direct communication link exists also in the modelled real system. For instance, in the bank example of the course material, all communication between the ATM and the bank is via the channels. Reliable communication would be easy to obtain in the model if the ATM could communicate directly with the bank, but this is not allowed in the model, because that cannot be implemented in the real world, and the whole point of the problem is to find and demonstrate correct a protocol that can be implemented in the real world.

Communication may take place via shared variables that model, for instance, flags or queues. In that case, one transition accesses both the variables of one communicating component and the shared variable, and another transition accesses the shared variable and the variables of the other communicating component. If the value to be communicated is complicated enough, more than one shared variable may be used in the same communication. Typically the former transition tests that the former communicating component is in the right local state and perhaps also that the shared variable is empty, and if yes, then copies a value from a local variable of the former component to the shared variable. Typically the latter transition tests the local state of the latter communicating component and that the shared variable is not empty, and if yes, then perhaps also copies the value of the shared variable to a local variable.

Communication may also be synchronous, that is, there is no shared variable. Instead, the same transition accesses variables of both communicating components. At a high level of abstraction, it is also possible that more than two components participate the same transition.

If the description of the problem does not say which kind of communication to use, then one should be used that most naturally models the system. Sometimes, especially at a high level of abstraction, both kinds of communication are natural. Then usually synchronous communication is better, because it yields a smaller state space.

The size of the state sapce usually depends heavily on the **atomicity level**, that is, how many successive transitions are used to model a sequence of operations. For instance, Peterson's algorithm breaks if its statements $wants_1 := \mathsf{true}$ and $turn := 2$ are swapped. However, if they are modelled by a single transition that performs both of them, then the error is not detected. Therefore, it is important to use an appropriate level of atomicity.

Again, the right level should be chosen according to what is appropriate for the exercise. For instance, if the exercise is about designing a protocol that facilitates reliable communication over unreliable connections, then the model should be rather detailed, whereas if the exercise is about voting a leader in a distributed system, then it is typically appropriate to model communication simply by writing the communicated value directly to a local variable of the receiving component. In the latter case, if also the loss of messages must be modelled, it can be modelled with an alternative transition that behaves otherwise similarly but does not write the communicated value.

If a transition only accesses variables of a single component, then it is always legal to merge it with the previous transition, **to reduce the size of the state space**. The size can also often be reduced by adding statements that assign 0 to a variable when its value becomes irrelevant, for instance, when it is certain that the next operation on the variable is reading a value sent by another component. If a system is symmetric, it suffices to model one of the symmetric situations. For instance, to verify that each client can get service, it may suffice to choose one of them and verify that it can get service, and then conclude that by symmetry, also the others can. Using the symmetry method of ASSET is not required. However, the student is welcome to try if they wants; it may yield extra points.

It is not easy to write sufficient **checking functions**. With `check_state` one can specify a condition that every reachable state must satisfy. Every halted state must satisfy the condition specified with `check_deadlock`. They are usually relatively easy to write and understand. Unfortunately, they do not reveal errors where the system keeps on running without getting its job done.

With `is_must_progress` one may specify that the system must not be able to run forever in a loop where it never goes via a state that satisfies a given condition. For instance, one may specify that every loop must contain a state where Client 1 either is receiving service or has chosen to never again request for service. Unfortunately, this requirement is often surprisingly difficult or even impossible to satisfy. It may require modelling so-called fairness assumptions, which are a complicated topic and not supported by ASSET. Instead, ASSET features `is_may_progress` which represents a strictly weaker requirement. It allows a loop of the above kind, but only if there is a path out from the loop to a state where the condition is satisfied. If you are unsure whether to use one, the other, both, or none, then use `is_may_progress`.

The modeller must distinguish operations that a component must do if it can from operations that the component may choose to not do even if it can. Most operations are of the former kind. A typical example of the latter is request for service $\overset{\searrow}{\circ}\xrightarrow{\mathsf{r}}\ldots$. The typical approach in mainstream verification is either to use fairness assumptions or (accidentally or intentionally) ignore the issue. With ASSET, an appropriate way to model the latter is to add a transition to a terminal state that acts as an alternative to requesting the service, and to split the transition that models requesting to a transition that loses the possibility of choosing the termination branch followed by a transition that makes the request $\circ\xleftarrow{\tau}\overset{\searrow}{\circ}\xrightarrow{\tau}\circ\xrightarrow{\mathsf{r}}\ldots$. If it is certain that other components and shared variables cannot disable the transition that models requesting, then the splitting is not necessary $\circ\xleftarrow{\tau}\overset{\searrow}{\circ}\xrightarrow{\mathsf{r}}\ldots$. The purpose of splitting is to not lose errors where a client is prevented from requesting service.

It may be necessary to add extra state variables to the model, to check a property. For instance, to check that a communication protocol delivers what was originally sent, it may be necessary to store the sent value in a separate variable, against which the received value is checked. As another example, there may be a variable whose value tells whether there has been a request which has not yet been served. One can often think of these as modelling one or more extra tester components. The tester components may test and read all variables, even if it would violate the rules introduced above.

The checking functions must cover all correctness properties that are usually required of systems of the kind in the exercise and that can be expressed with the ASSET features mentioned above. However, if something is obvious, then it is not necessary to ensure that it is checked by the checking functions. For instance, it is not necessary to verify that the ATM of the bank system does not give money without a preceding request, because it is obvious from the structure of the ATM state machine that this holds. In general, **the use of common sense is allowed.**

Assignment to `err_msg` should only be used to catch errors in the C++ representation of the model, not to test the intended model. For instance, if the intention is that the local state may only assume the values from 0 to 5 but the compiler complains that the values 6 and 7 are not dealt with, then they can be dealt with by assigning a message to `err_msg`.

Reporting

You must report your result in a meeting of the course. It is likely that you will have to write more than one model before you find a model that works correctly. In addition to reporting your final model, choose the most interesting erroneous model and also report it: what was the error, how did you find it, and how did you fix it.

Problems

1. Model a mutual exclusion system that consists of two clients and a server. The system must guarantee eventual access in the may-sense, but need not guarantee it in the must-sense. The server communicates with each client via *two-step handshake*. That is, for each client $i \in \{1,2\}$, there are two boolean variables, r_i for "request" and g_i for "grant". The client requests for access to the critical section by setting r_i to True. The server grants access by setting g_i to True.

A transition can access at most one shared variable. It can either test its value or change the value, but cannot do both. (This problem is motivated by the design of logic circuits. There this restriction is often relevant.)

2. Model the horse jump game. There is an $mx \times my$ board. A horse must jump from square to square such that it visits each square precisely once. It need not get back to where it started. A jump consists of moving one square to left or right and two squares up or down, or two squares to left or right and one square up or down.

Use error checking features of ASSET to recognize a solution and print the corresponding sequence of jumps. Run the experiments with $mx = my = 5$, but write the model so that the experiments may be repeated with different small values of mx and my.

Can the problem be solved such that the horse does get back to where it started?

What if one corner is left out of the board? What if the centre square is left out?

What is the maximum number of squares on a 5 times 5 board that the horse can visit, if it starts on the lower left corner and only can move in three directions: one step up, two steps right, or one left and two down?

3. Model a childrens' game where each player is originally a mouse. The transitions of the game consist of any two players of the same kind meeting. When two mouses meet, one of them remains a mouse and the other becomes a dog. When two dogs meet, one of them becomes a mouse and the other becomes a monkey. When two monkeys meet, one of them becomes a mouse and the other becomes a human. Humans do not participate meetings. Based on results with different numbers of players, state a hypothesis on the possible end situations of the game. Can you confirm the hypothesis by reasoning?

4. Model a system consisting of a source, a target, and one or more communication paths. The idea is that the system mimics different paths in the Internet. If the source sends A before B and they travel via different paths, then it is possible that B reaches target before A. Even so, target must deliver messages to the local user in the correct order. Model each path from source to target with a component that can hold at most one message. You may also need paths from target to source, for instance, for acknowledgements. The path via which an acknowledgement travels is not determined by the path which was used in the opposite direction. Paths are reliable. For instance, they do not lose, break, or duplicate messages. Verify with ASSET that the system as a whole provides reliable communication.

5. Design a token recovery mechanism for a token-ring system. The system consists of n processes in a cycle. That is, process i receives messages only from process $(i-1) \bmod n$ and sends messages only to process $(i+1) \bmod n$. Any message may be lost, the sending process does not notice that it was lost, and the receiving process does not notice that it was ever sent. The processes circulate a message called *token*. Processes 1, 2, ..., $n-1$ are identical. Process 0 is a *master* whose task is to detect the loss of the token and re-generate the token. To perform its task, it may send additional types of messages to the ring. There must never be more than one token in the system.

6. Modify the alternating bit protocol such that instead of retransmitting the message in case of timeout, the sender informs the sending client of failure. If the acknowledgement arrives before timeout, then the sender informs the sending client of success. Demonstrate that this protocol may lose a later message but declare success. Fix this problem.