A synchronous reactive component consists of a finite set I of typed input variables, a finite set O of typed output variables, a finite set S of typed state variables, an initialization *Init* defining the initial values of the state variables, and a reaction description React defining the set of reactions of the component. An execution of a synchronous reactive component begins in the state specified by its initialization. Then, for a finite number of rounds, a reaction of the component that leads from the current state to some state of the component is executed. If the input and output variables of the component are not of event type, each reaction reads in all input variables and displays to all output variables.

The construct RailRoadSystem2 is a composite synchronous reactive component composed of three synchronous reactive components: TrainW, TrainE, and Controller2 (where TrainW and TrainE are both instances of the Train component). RailRoadSystem2 simulates a railroad system consisting of two trains each with their own separate tracks, and a shared bridge with signals on each end.

A Train component has a single input variable signal of the enumeration type {green, red}. This represents the signal on a particular train’s incoming side of the bridge. Each train component has a single state variable mode of the enumeration type {away, wait, bridge}. This represents whether a train is away from the shared bridge, waiting to be let on to the bridge by its signal, or on the bridge. The initialization of the component specifies the initial value of mode to be away. Since the set of state variables of the Train component is not empty, the Train component is not a combinational component. Finally, a Train component has a single output variable out of the event enumeration type event({arrive, leave}). As it is of event type, the output variable out may not be outputted in each round of the synchronous reactive component. However, as the input variable of Train is not of type event, the component is not event-triggered. The Train component is a finite-state component, however, as the type of each of its input, output, and state variables is finite.

The reaction description of Train is split into two tasks. A task of a synchronous reactive component has a read-set consisting of input, state, output, or local variables, a write-set consisting of output, state, or local variables, and an update description that assigns new values to the elements of its write-set based on the current values of the elements of its read-set. The task A1 has a read-set consisting of the state variable mode and a write-set consisting of the output variable out. The task is nondeterministic, as when the current mode of the train is away or bridge, multiple updates are possible. The possible updates are as follows. In any mode, a possible update is to not do anything (in this case, the output variable out will be absent for this round). Also, when the state of the component is away, the update can assign the value arrive to out. Finally, when the state of the component is bridge, the update can assign the value leave to out. The task is input-enabled, as an update is possible for any value of the state variable mode. The task A2 has a read-set consisting of mode, out, and the input variable signal. It has a write-set consisting of mode. Since out belongs to the read-set of A2 and the write-set of A1, A1 precedes A2 in the precedence relation of the tasks. The task A2 is also nondeterministic. If the current state of the component is away, the update will set the state to wait if the output had been set in A1 to be arrive. Otherwise, the state will remain away. If the current state of the component is wait, the update will set the state to bridge if the value of signal is green. Otherwise, the state will remain wait. If the current state of the component is bridge, the update will set the state to be away if out had been set in A1 to be leave. Otherwise, the state will remain bridge. This task is also input-enabled.

Meanwhile, the Controller2 component has two event type input variables outW and outE, both of type event({arrive, leave}). It has four state variables – west and east of enumeration type {green, red}, and nearW and nearE of Boolean type. The initialization of Controller2 sets west and east to red, and nearW and nearE to 0(false). Finally, the component has two output variables signalW and signalE, both of enumeration type {green, red}. As with Train, the Controller2 component is finite-state but not combinational. However, the component is event-triggered, as all of its input variables are of event type, all of its output variables are latched (as in every reaction signalW is set to the current value of west, and same for signalE and east), and in any state of the component, if neither outW nor outE are present, the state variables will remain the same for any possible reaction.

The reaction description of Controller2 is split into 3 tasks. The task A1 has a read-set of the state variable west and a write-set of the output variable signalW , and simply sets signalW to the current value of mode. The task is deterministic and input-enabled. The task A2 has a set-set of east and a write-set of signalE, and likewise sets signalE to the current value of east. This task is also deterministic and input-enabled. Finally, the task A3 has a read-set of the state variables west and east, the input event variables outW and outE, and the state variables nearW and nearE. It has a write-set of east, west, nearW,and nearE. The update description of this task is as follows. If outE is present and has a value of arrive, then nearE is set to 1. If outE is present and has a value of leave, then nearE is set to 0. The same operations are performed for outW and nearW. Then, if nearE is currently 0 then east if set to red. Otherwise, if west is currently red then east is set to green. Then, if nearW is currently 0 then west is set to red. Otherwise, if east is currently red then west is set to green. This task is also deterministic and input-enabled. The tasks A1 and A2 do not depend on the input variables, and so they come before A3 in the precedence relation of Controller2.

The system RailRoadSystem2 is composed of a single instance of Controller2 and 2 instances of Train, labeled TrainW and TrainE. For TrainW, the input variable signal is renamed to signalW, the output variable out is renamed to outW, and the state variable mode is renamed to modeW. Similarly, for TrainE signal is renamed to signalE, out is renamed to outE, and mode is renamed to modeE. After renaming, there are no name conflicts between the state variables of the 3 components. Furthermore, the sets of output variables of the components are disjoint. The precedence relation of RailRoadSystem2 contains the precedence relations of its constituent components – there is an edge from task A1 of TrainW to A2 of TrainW, from A1 of TrainE to A2 of TrainE, from A1 of Controller2 to A3 of Controller2, and from A2 of Controller2 to A3 of Controller2. Furthermore, since outW belongs to the write-set of task A1 of TrainW and the read-set of task A3 of Controller2, there is a new edge from A1 of TrainW to A3 of Controller2. Similarly, there is a new edge from A1 of TrainE to A3 of Controller2. Also, since signalW is in the write-set of task A1 of Controller2 and the read-set of task A2 of TrainW, there is a new edge from A1 of Controler2 to A2 of TrainW. Similarly, there is a new edge from A2 of Controller2 to A2 of TrainE. The precedence relation of the composite remains acyclic, and so the 3 constituent components are compatible. The composite system RailRoadSystem2 has state variables modeW, modeE, west, east, nearW, and nearE, as described previously. It has output variables signalW, signalE, outW, and outE. It has no input variables, and there is no output hiding.

A safety requirement for a system demands that the system does not perform undesirable behavior (i.e. nothing bad ever happens). A safety requirement essentially divides the states of a system into “safe” and “unsafe”, and demands that no execution of the system starting from an initial state ends up in an unsafe state. A violation of a safety requirement can be demonstrated by a finite execution in which the undesirable behavior occurs (i.e. an unsafe state is entered). Once way to verify safety requirements is the use of a safety monitor. A safety monitor for a synchronous reactive component C is another synchronous reactive component M such that the set of input variables of M is a subset of C’s input and output variables, the set of output variables of M is disjoint from C’s input and output variables, and the reaction description of M is given as an extended state machine with a subset of those states declared as accepting. Then, C satisfies a particular safety property if for any execution of the composite system C || M, the state of M is never an accepting state. For RailRoadSystem2, one potential “safety” property is that if the west train arrives at the bridge, then the east train cannot leave the bridge twice before the signal for the west train becomes green. A safety monitor for this safety property is a synchronous reactive component M with input variables outW, outE, and signalW, and no output variables. The monitor M would have 4 states, with a transition from state 0 to state 1 if outW is arrive, from state 1 to state 2 if outE is leave, and from 2 to 3 if outE is leave. There are also transitions from state 1 to 0 and from state 2 to 0 if signalW is green, as well as self-looping else transitions for states 0, 1, and 2. The only accepting state is state 3. If the safety requirement described previously does not hold, then there will be an execution of RailRoadSystem2 || M such that M enters state 3. However, not all safety requirements can be expressed as safety monitors.

Instead, safety requirements are usually expressed as properties of transition systems. A transition system T consists of a finite set of typed state variables, and initialization defining the set of initial states, and a transition description defining the set of transitions between states. Given any synchronous reactive component (such as RailRoadSystem2), it is trivial to obtain a corresponding transition system from the set of state variables, the initialization, and the reaction description (the transition system’s set of transitions contains all pairs of states (s,t) such that s->t is a reaction for some input i and some output o. A property of a transition system is simply a Boolean value expression over the state variables of the system. A property P of a transition system T is an invariant of T if every reachable state of T satisfies P (P is true). A state of T is reachable if there is a finite sequence of transitions leading to it from an initial state of T. For the transition system corresponding to RailRoadSystem2, a potential safety property is P = ¬(modeW = bridge ^ modeE = bridge). This property is an invariant of the system, as there is no possible execution that leads to a state where both modeW and modeE are bridge in the same round.

Some invariants of a particular transition system T are known as inductive invariants. A property P of a T is an inductive invariant if every initial state of T satisfies P, and if a state s satisfies P (regardless of whether or not s is reachable) and (s,t) is a transition of T, then t also satisfies P. Not all invariants are inductive. However, for any non-inductive invariant P, it is possible to find another property Q such that Q is an inductive invariant and Q implies P (Q is known as the inductive strengthening of P). Therefore, to establish that a property P is an invariant of a transition system T, it is sufficient to find a property Q such that Q is an inductive invariant of T and Q implies P. Unfortunately, in general the verification of invariance is undecidable, and there is no ideal algorithm to determine whether or not a particular property of a particular transition system is an invariant or not. However, if all of the state variables of a transition system are of finite or countably infinite type, then there are algorithms to determine invariance of a particular property of the system (although the algorithms are not guaranteed to terminate).

The enumerative search algorithm works with individual states. For each initial state of a transition system T, it performs depth-first search (DFS) starting from that state and attempts to find a state that does not satisfy the desired property P (i.e. satisfies ¬P). If the number of reachable states of the transition system is finite, then the algorithm is guaranteed to terminate, with the number of calls to DFS bounded by the number of reachable states. If the algorithm terminates and returns 0, then the property ¬P is not satisfied by any reachable state of T, and P must be an invariant of T. If the algorithm terminates and returns a sequence of states, then that sequence is a witness execution demonstrating that ¬P is satisfied by some reachable state of T, and therefore P is not an invariant of T. The symbolic search algorithm, on the other hand, works with sets of states (known as regions). It begins with the region of initial states of a transition system T, and performs breadth-first search (BFS) starting from that region (using the operations on regions described in Alur’s Principles of Cyber-Physical Systems) and attempts to find a region containing a state that does not satisfy the desired property P (i.e. satisfies ¬P). As with enumerative search, the algorithm is not guaranteed to terminate, but if it does then the returned value correctly indicates whether or not P is an invariant of T. For the symbolic search algorithm, it is beneficial to represent properties (which are Boolean expressions) as reduced ordered binary decision diagrams (ROBDD’s), as they allow for recomputations of certain operations to be avoided, among other benefits. The process for building an ROBDD for a Boolean expression is described in detail in Alur’s Principles of Cyber-Physical Systems.

An asynchronous process is similar to a synchronous reactive component in that it has typed inputs and outputs (known as channels) and maintains an internal state using a set of internal variables (for which there is an initialization defining te set of initial states). However, an asynchronous process has no concept of a round. Instead, it has a set of input tasks for each input channel, each described by a guard condition over the state variables and an update defining a set of input actions that update the state of the process, and a set of output tasks for each output channel, each also described by a guard condition over the state variables and an update defining a set of output actions that update the state of the process. An asynchronous process also has a set of internal tasks, each described in the same way as above. At any point in time, only a single input, output, or internal action that is currently enabled (its guard condition is true) is performed, and which action is chosen to be performed is arbitrary. As with synchronous reactive components, asynchronous components can be composed with each other. The process for doing so is described in Alur’s Principles of Cyber-Physical Systems. Likewise, the safety requirements for an asynchronous process can be verified through the use of a safety monitor, or by obtaining a transition system from the process and then establishing that a desired safety property is an invariant (e. g. by performing enumerative or symbolic search). An asynchronous process is non-blocking if for every input channel and for every state some task in the set of input tasks associated with that channel is enabled in that state.

Peterson’s Mutual Exclusion Protocol involves two asynchronous processes that share the 3 atomic registers turn (enumeration type {1,2}), flag1 (Boolean), and flag2 (Boolean). Flag1 and flag2 are initially set to 0, while turn in uninitialized. The asynchronous process P1 has input and output channels for the 3 shared registers, and one state variable of enumeration type {idle, try1, try2, try3, crit} that is initialized to idle. In the idle mode, it has an internal task that does nothing and sets the mode to remain in idle. In the idle mode, there is also an output task that sets flag1 to 1 and updates the mode to try1. In the try1 mode, there is a single task – an output task that sets turn to 1 and updates the mode to try2. In the try2 mode, there is an input task with guard flag2 = 0 that updates the mode to crit. There is also an input task with guard flag2 = 1 that updates the mode to try3. In the try3 mode, there is an input task with guard turn = 2 that updates the mode to crit. There is also an input task with guard turn = 1 that updates the mode to try2. Finally, in the crit mode there is an internal task that does nothing and sets the mode to remain in crit. There is also an output task that sets flag1 to 0 and updates the mode to idle. The asynchronous process P2 is almost identical to P1, except that the output task in mode try1 sets turn to 2, the input tasks in mode try2 have guard conditions flag1 = 0 and flag1 = 1, the input tasks in mode try2 have guard conditions turn = 1 and turn = 2, and the output task in mode crit sets flag2 to 0. Both P1 and P2 are non-blocking – if one process is in crit mode and the other process wants to set its mode to crit, that process will continuously alternate between try2 and try3 until the other process no longer is in crit mode. This system implements mutual exclusion: the processes P1 and P2 cannot both have mode = crit at the same time. The system also avoids deadlock (when all asynchronous processes in a system have no tasks enabled in their current states). Without loss of generality, if P1 is the only process that wants to set its mode to crit, then flag2 will be zero when it is in the mode try2 and the output task that updates its mode to crit will be enabled. If both P1 and P2 are in the mode try2 and are trying to set the mode to crit, then turn will either be 1 or 2 (depending on which process entered mode try2 first), and one of the processes will have its output task that updates mode to crit enabled.

For asynchronous process, as the choice of which enabled task to execute in a particular state is arbitrary, a concept of fairness must be developed. If weak fairness is assumed for a particular task A of an asynchronous process P, then for any infinite execution of P, if A is enabled at a particular step of the execution, then at some later step of the execution A is either executed or not enabled. This means that if a task is enabled, then eventually it is either taken or disabled. If strong fairness is assumed for a task A of an asynchronous process P, then for any infinite execution of P, if for infinitely many indices j A is enabled at step j, then for infinitely many indices L A is executed at step L. This means that if a task is repeatedly enabled, then it is repeatedly executed (it is either continuously disabled or repeatedly executed). Strong fairness implies weak fairness, but not vice versa.

For the asynchronous processes in Peterson’s Mutual Exclusion Protocol system, there are no fairness assumptions for the internal tasks that maintain state idle and state crit. There is also no fairness assumption for the output task that updates the mode from idle to try1 and sets flag1/flag2, as there is no reason to assume that either process will repeatedly want to enter the critical section. On the other hand, weak fairness is assumed for the output task that updates the mode from try1 to try2 and sets turn, as otherwise a process can be stuck in mode try1 forever. Similarly, weak fairness is assumed for all of the output tasks out of modes try2 and try3. Furthermore, weak fairness is assumed for the output task that updates the mode from crit to idle and sets flag1/flag2, as otherwise a process can remain in the critical section forever. In all cases, strong fairness is not necessary, as a task will never be disabled after it has been enabled while the process remains in the same state.

A liveness requirement for a system demands that a certain desired behavior is performed (i.e. something good eventually happens). Liveness requirements are specified using temporal logic. One type of temporal logic is Linear Temporal Logic (LTL). The formulas of LTL given a set V of typed variables are defined inductively as follows: Any Boolean expression over V is an LTL formula. If P is an LTL formula, then so are ¬P, ○P, ◊P, and □P, where is the ¬ standard negation logical operator, ○ is the next operator, ◊ is the eventually operator, and □ is the always operator. Finally, if P and Q are LTL formulae, then so are P ^ Q, P V Q, P → Q, and P ***U*** Q where ^ is the standard conjunction logical operator, V is the disjunction logical operator, → is the implication logical operator, and ***U*** is the until operator. Given any infinite sequence p of valuations over V (a trace over V) and a position j, (p, j) satisfies an atomic LTL formula P (i.e. Boolean expression) if the valuation at position j satisfies P. Furthermore, given an LTL formula P, (p,j) satisfies ¬P if the valuation at the jth position does not satisfy P, ○P if the (j+1)th valuation satisfies P (the next position satisfies P), ◊P if for some k >= j, the kth valuation satisfies P (P is eventually satisfied), and □P if for all k >= j, the kth valuation satisfies P (P is always satisfied henceforth). Finally, if P and Q are LTL formulas, then (p,j) satisfies P ^ Q if the jth valuation satisfies both P and Q, P V Q if the jth valuation satisfies either P or Q, P → Q if the jth valuation either doesn’t satisfy P or satisfies Q, and P ***U*** Q if there is some position k >= such that the kth valuation satisfies Q, and for all i such that i >= j but i < k, the ith valuation satisfies P (eventually Q is satisfied, and all valuations before that point satisfy P). An LTL formula is said to be valid if all possible traces satisfy it.

In the case of RailRoadSystem2, one potential liveness requirement is to demand that the west train not wait forever to be let onto the bridge. The LTL formula

□◊(modeW = bridge) attempts to specify this by asserting that the west train repeatedly enters the bridge. However, this formula is not valid, as the west train may not ever approach the bridge. Another attempt at an LTL formula could be

□((modeW = wait) → ◊(signalW = green)). However, this formula is also not valid as the east train could remain on the bridge forever while the west train is waiting to be let on. A valid LTL formula that specifies the desired liveness requirement would be □◊(modeE != bridge) → □((modeW = wait) → ◊(signalW = green)).

For any synchronous reactive component or asynchronous process C, the set V of its input, output, and state variables/channels, and an LTL formula P over V that specifies a desired liveness requirement of C, it is possible to construct a nondeterministic finite automaton M known as a generalized Buchi automaton with a finite set of states, a finite set of initial states, and a finite set of accepting states (subset of the set of states) such that if there is an infinite execution of the system C||M where M repeatedly enters an accepting state, then P is not valid for C. Similarly, one can obtain the corresponding transition system T from C, search for a reachable state that violates P, and then search for a cycle of states of T that contains that particular state. This can be done working with individual states in nested DFS (akin to enumerative search) or working with regions of states in nested BFS (akin to symbolic search).