Homework 6: Symbolic Model Checking with NuXmv

Prof. Kristin Yvonne Rozier

1. The System Model Is an Automaton

The notions we've seen in previous homeworks of states, traces, and state spaces aren't specific to explicit-state model checking, SPIN, and Promela, but are useful in any formalization of concurrent and reactive systems. Note that also for probabilistic (also explicit-state) model checking, i.e. with PRISM, all of the types of models we can analyze are types of state-transition systems. Specifically in the case of PRISM these include Deterministic Time Markov Chains (DTMC), Markov Decision Processes (MDP), Continuous Time Markov Chains (CTMC), Probabilistic Automata (PA), and Probabilistic Timed Automata (PTA). Now we also look symbolic model checking and again, the system models can be framed in terms of automata.

Here we give a more precise definition of a concurrent system in terms states and traces. Concurrent systems can inherently be represented a *state-transition systems* or *automata*, which are really just fancy graphs. Our definition of a system's state space automaton is similar to many conventional automata definitions, but explicitly includes a way to relate the current state to the propositions that define our states and allow us to write temporal logic formulas to reason about system traces. Like any graph, we have a set of vertices, which we call states, and a set of edges, which we call transitions because they allow the system to transition between different system states. Our graph becomes an automaton when we add a few more touches that enable us to reason effectively about the system, such as it's *alphabet* or the set of system variables, and designations on certain states, such as the start state, which is the state our system is in when we turn it on.

Definition 1. *A* **Deterministic Finite Automaton** *(DFA) is a quintuple* $(Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite set of states. In the case of a state space automaton, a state is an assignment to all program variables (both local and global), including the program counter (line number) for each process.*
- Σ *is a finite alphabet. In this case, this is the set of Boolean system variables, Prop.*
- $\delta : Q \times \Sigma \rightarrow Q$ *is the transition relation. Equivalently, this is the edge relation of the state space.*
- $q_0 \in Q$ *is the initial state. Equivalently, this is the state the system is in when it is initially turned on.*
- $F \subseteq Q$ *is a set of final states. This is a way of designating a set of states that are of special interest so we can check if they are visited on a path traversing the automaton, such as the set of states that represent "something bad."*

A run of a finite automaton over a finite computation $\pi = \pi_0, \pi_1, \pi_2, \ldots \pi_n \in \Sigma$ *is accepting if it terminates in a state* $q_n \in F$.

It is the set *Prop* that will allow us to talk about our program variables in our logic formulas. The elements of *Prop* are simple propositions involving program variables. For example, `philosopher_2_has_fork` `(ctr>3)` `(in[].size() == max_buffer_size)` (that is, the buffer `in[]` is full). Each of these might be true in some states and false in other states. We can define, for example, a function P that gives a truth assignment to the propositions (a different truth assignment for each state).

Keeping in mind the structure of the system as an automaton is not only useful when modeling a system in Spin with Promela, where such an automaton can be explicitly constructed from the system description. It will also help us write models in the symbolic model checker NuXmv (formerly known as NuSMV). As we will see, the language of NuXmv for specifying a system directly mirrors the language used to define an automaton above.

2. Modeling the System

While it is sometimes possible to perform verification directly on a completed system, this approach is undesirable for two reasons. Firstly, it is significantly more efficient and cost-effective to perform verification as early as possible in the system design process, thereby avoiding the possible discovery of an error in the completed system that requires a redesign. Secondly, it is simpler and easier to reason about a model of the system than the system itself because the model includes only the relevant features of the larger system and because the model is easier to build and redesign as necessary.

Surveying the vast array of system modeling techniques is outside the scope of this class. Models can be extracted from automata, code, scripts or other higher-level specification language descriptions, sets of Boolean formulas, or other mathematical descriptions. They can be comprised of sets of models, such as a set of models of the whole system at different levels of detail, or a set of models of different independent subsystems plus a model of the communications protocol between subsystems, etc. Furthermore, there are many strategies for modeling to optimize clarity, provide generality for reusability, or minimize model checking time, either by reducing time or space complexity during the model checking step. Models can be designed not only to find bugs in a system design but also to solve problems from other domains. For example, model checking is sometimes used for path planning where the model and the specification property describe the environment to be traversed and the constraints on the path and the counterexample returned constitutes a viable path matching those criteria [10]. Here we simply list the necessary components of a system model and introduce an example model of a real-world automated air traffic control architecture.

Whatever the original form of the system model, we eventually translate it into some form of state-transition system that is simply a type of graph called an automaton or Kripke structure. As the name implies, this state-transition system minimally must contain some set of system states (the vertices of the graph) and transitions between states (the edges of the graph). The system is described by a set Σ of system variables, also called the system's alphabet. Here, we define a state as a set of assignments to the system variables. Therefore, each vertex in the graph is labeled by all of the system variables and their values in that state. Each unique variable assignment constitutes a unique state. Since not all variable assignments may be possible, the lower bound on the number of states in a system model is 1 and the upper bound is $2^{|\Sigma|}$, presuming Σ is a set of Boolean variables. We consider a system to transition from one state to another (i.e. traverse an edge from one vertex to another) when the values of one or more system variables change from the values they had in the originating state to the values they have in the destination state.

Given that system design and modeling is largely an art form, we simply list here the chief concerns to keep in mind when creating the system model:

- **Defining Σ:** What is the minimum set of system variables required to accurately describe the full set of behaviors of the system?
- **Level of abstraction:** Which details of the system are relevant to the verification process and how do we ensure we have included all of these? Which details serve chiefly to obscure the former? For example, when modeling an automated air traffic control architecture, we include in the model that the controller can request a specific trajectory. However, details of the graphical user interface (GUI) used to make this request are left out.
- **Validation:**
  - **system $\rightarrow$ model** Have we modeled the system correctly?
  - **model $\rightarrow$ system** How will we ensure the resulting hardware or software system created after verification matches the model we have verified?

2.1. Modeling Limitations

Not all systems can be modeled in such a way as to undergo formal verification via symbolic model checking. However the model is created, there is always the chance of creating more states than can be reasoned about in computer memory and having the model checking step halted by the state explosion problem. This problem can be mitigated during the modeling phase by being cognizant that, in the worst case, the model checker will have to explore the entire state space. Keeping the state space as small as possible by modeling only relevant aspects of the system, minimizing the set of state variables, utilizing sophisticated data structures and abstractions [12], and representing concurrent threads independently to take maximum advantage of partial order reductions performed by the model checker, are some strategies for achieving this goal. (Partial order reduction is a technique that recognizes cases where multiple different interleavings of concurrent processes in the system M have the exact same effect on the property φ, thus only one such sequence needs to be checked [42, 32].)

In this class, we presume the system model M to have a finite state space defined over discretely-valued variables. Indeed, the termination of this algorithm is guaranteed by the finite nature of the state space we are exploring [47]. Some systems inherently have an infinite state space and thus cannot be model checked using the techniques presented here; instead, these systems must be verified using specialized techniques for providing weaker assurances about larger state spaces or using alternative verification techniques, such as theorem proving. *Bounded model checking (BMC)* [2] can be used to reason about some models whose state spaces exceed the capacity of symbolic model checking. For a given k, BMC tools search for a counterexample of length k or shorter so they can prove the presence of errors in the model but cannot be used to prove the absence of errors.

Real-time systems with continuous-valued clocks that can be described in terms of linear constraints on clock values can be analyzed using timed automata symbolic model checkers such as UPPAAL [26]. Hybrid automata model checkers like HyTech can verify some types of hybrid systems, that is, systems with a mix of discrete and continuous variables [23] but the infinite nature of the state space means termination of the model-checking algorithm is not guaranteed. Increasing the effectiveness of techniques for model checking real-valued and infinite-state systems is currently a very active area of research [24, 6, 33, 25, 28, 37].

2.2. An Example: The Automated Airspace Concept for Air Traffic Control

Model checking is most frequently used for the formal verification of safety-critical systems, or systems whose failure could result in serious injury or death. Examples of safety-critical systems include air traffic control, airplane separation assurance algorithms, life support systems, essential communications systems, hazardous environment controls, medical equipment systems (such as those that administer radiation or assist in surgery), train signaling systems, and automotive control systems. Other types of systems where model checking is frequently applied include components of safety-critical systems, such as processor chip designs, financial systems that could cause catastrophic failure, missiles and other weapons systems, commercial products whose malfunctioning would trigger catastrophic financial costs for the manufacturer, and any other systems where the cost of a single failure outweighs the cost of formal verification.

We illustrate the symbolic model-checking algorithm for LTL on a small example that describes the architecture for the Automated Airspace Concept for air traffic control in NASA's Next Generation Air Transportation System (NGATS)[17], displayed in Figure 1.

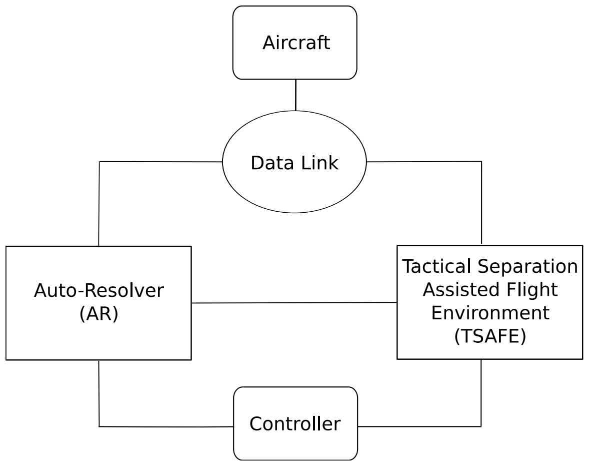

Figure 1: Architecture: Automated Airspace Concept for Air Traffic Control

The following rules govern the operation of this air traffic control architecture:

- The auto-resolver is enabled by default and sends flight commands to the aircraft.
- The aircraft may request a specific flight command.
- The controller may request a specific flight command.
- If a near-term conflict is detected (i.e. a loss of separation is projected to occur within an amount of time below the system's threshold), TSAFE (the Tactical Separation Assisted Flight Environment tool) automatically engages and sends a flight command to resolve the conflict. TSAFE commands always override all requests and commands issued by the auto-resolver.
- Requests (either from the aircraft or from the controller) may only be made when the airspace is clear of potential short-term conflicts. Note that in this model, we uphold the condition that neither of the request variables may be true at the same time as the `TSAFE_clear` variable is false by canceling any standing requests before acknowledging a short-term conflict.
- Only one request may be made at a time. If the aircraft and the controller both submit requests at the same time, only one is registered.

Variable	Description
`AR_command`	Has a command been issued by the auto-resolver?
`TSAFE_command`	Has a command been issued by TSAFE?
`controller_request`	Has a request been issued by the controller?
`aircraft_request`	Has a request been issued by the aircraft?
`TSAFE_clear`	Does TSAFE detect the airspace is clear of conflicts?

Table 1: The system alphabet Σ: a set of Boolean-valued system variables

LTL model checkers follow the automata-theoretic approach [45], where the relationship between programs and their specifications is considered as a relationship between languages [47]. We can model the air traffic control architecture as an automaton whose language consists of the set of all of the possible computations of the system. A computation is essentially an infinite sequence of system states, corresponding to the behavior of the system starting and then running indefinitely. This method allows us to reason about the system's behavior by asking questions about automata. The alphabet, Σ, of our automaton is a set of system variables in the form of Boolean-valued propositions that describe the current state of the program. Table 2.2 enumerates the alphabet Σ and Figure 2 shows explicitly the automaton representing this automated air traffic control architecture. For clarity, each state is labeled with both the variables that are true in that state and the negations of the variables that are false in that state. The transitions are labeled by actions that cause the system to change state. The start state, State 1, is designated by an incoming, unlabeled arrow not originating at any vertex.

The following is an example traversal of the automaton in Figure 2. We start in State 1. If the controller issues a request, we move to State 4 and then if that request is granted by the auto-resolver we move to State 3, where we see an auto-resolver command has been issued in response to the controller's request. The detection of a conflict brings us to State 5 where only one action is available: to ignore the auto-resolver command requested by the controller and move to State 6. TSAFE issues a command to resolve the conflict, bringing us to State 7, and the execution of that command returns us to State 1 and a conflict-free airspace.

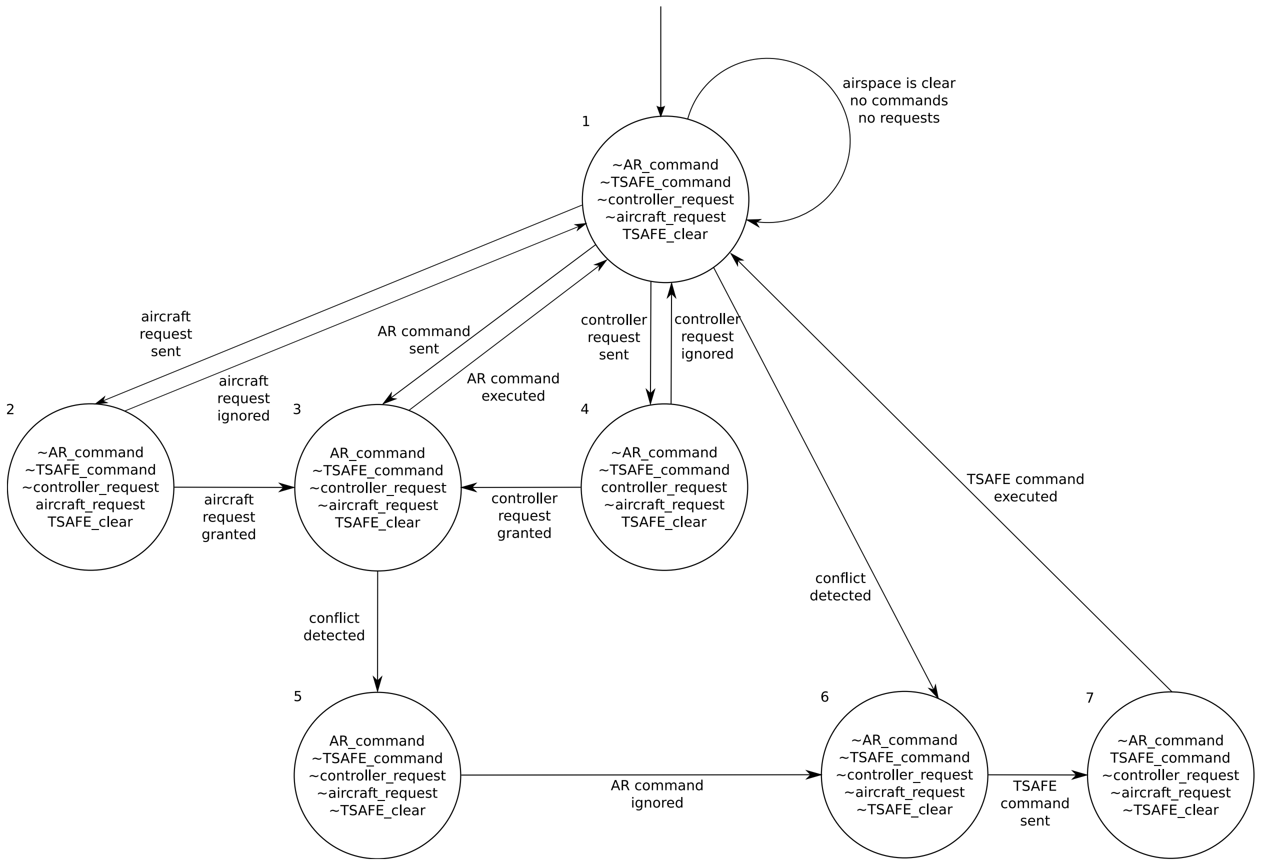

Figure 2: Automaton: Automated Airspace Concept for Air Traffic Control

We illustrate a real system description for this protocol using NuSMV [9, 8] since it is well-documented [34], open-source[1], and frequently used in industry [35, 22, 41, 7, 31, 30]. However, there are several tools available with similar capabilities including CadenceSMV (which has nearly identical syntax) [29], SAL-SMC [15], VIS [4], and others. The system description is fairly straightforward: after declaring our set of system variables, we simply list the conditions that determine how each of their values changes over time. We accomplish this using an `ASSIGN` statement describing the value of each state variable in the next state depending on the conditions in the current state. The function `next(var)` denotes the value of the variable `var` in the next state. If it is assigned using a `case` statement then the value of `var` is set to the value on the right hand side of colon whenever the condition on the left hand side of the colon is true. (In the case of multiple true conditions in a `case` statement, the path to follow is chosen nondeterministically.) The function `init()` is used to specify the initial values of each of the state variables. Since our example is short and rather simple, with no concurrent processes or subroutines, we place the entire specification in the `main` module, which is where execution of the model begins. In addition to using this system description for model

[1]The source code and documentation are available for download from: `http://nusmv.irst.itc.it/`.

checking, NuSMV also supports testing and simulation so one model can be used for the entire process of design, development, and formal verification of the system design.

Though our system model currently represents only the automated air traffic control architecture and not yet any of the specifications we wish to check, we think ahead and include a statement turning on *fairness*. Most of the time when we refer to fairness, we are referring to a type of liveness property that specifies that something must happen infinitely often. In that case, we are asking NuSMV to reason over all fair paths, or paths that start at an initial state on which all fairness constraints are true infinitely often. Enabling `FAIRNESS` in general, as we have done below, restricts the model checker to considering execution paths along which our main module is enabled for execution infinitely often. Intuitively, if there were multiple modules in our system model instead of just the one we need to represent our small example, enabling `FAIRNESS` for the set of modules would mean that each of them has the chance to be executed infinitely often over our prospective timeline. That is, every module has a fair shot at execution. In short, the use of `FAIRNESS` in our model guarantees that the model checker will return a representation of an infinite trace as a counterexample for any specifications we wish to check hold in this model.

```
MODULE main
  VAR
    -- Declare the set of system variables.
    AR_command : boolean;
    TSAFE_command : boolean;
    controller_request : boolean;
    aircraft_request : boolean;
    TSAFE_clear : boolean;

  ASSIGN

    -- Initially, the airspace is clear and there are no commands or requests.
    init(AR_command) := 0;
    init(TSAFE_command) := 0;
    init(controller_request) := 0;
    init(aircraft_request) := 0;
    init(TSAFE_clear) := 1;

    next(AR_command) :=
      case
        TSAFE_command : 0;     -- AR cannot issue commands when there is an outstanding TSAFE command
        -- if a conflict is detected, the current AR_command holds until TSAFE ignores & replaces it
        TSAFE_clear & AR_command & next(!TSAFE_clear & !TSAFE_command) : 1;
        TSAFE_clear : 0, 1;    -- AR_command is possible when no conflicts are detected
        !TSAFE_clear : 0;      -- no AR_command may be issued when TSAFE is in control
      esac;

    next(TSAFE_command) :=
      case
        -- Issue a TSAFE command when it's not clear and there are no other commands currently being executed.
        (!AR_command & !TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) : 1;
        -- default: no TSAFE command
        1 : 0;
      esac;

    next(controller_request) :=
      case
        -- If there are no conflicts or pending requests, the controller can make a request (or not).
        (!AR_command & !TSAFE_command & !controller_request & !aircraft_request & TSAFE_clear) : 0, 1;
        -- Controller requests are granted or denied in the next time step.
        controller_request : 0;
        -- default
        1 : 0;
      esac;

    next(aircraft_request) :=
      case
        -- If there are no conflicts or pending requests, the aircraft can make a request (or not).
        !AR_command & !TSAFE_command & !controller_request & !aircraft_request & TSAFE_clear : 0, 1;
        -- Aircraft requests are granted or denied in the next time step.
        aircraft_request : 0;
        -- default
        1 : 0;
      esac;

    next(TSAFE_clear) :=
      case
        -- Since requests are only allowed if the auto-resolver detects no long-term conflicts,
        -- in our model, a short term conflict only occurs when there are no requests.
        controller_request | next(controller_request) : 1;
        aircraft_request | next(aircraft_request): 1;
        TSAFE_command : 1;                 -- a TSAFE command resolves the conflict
        !TSAFE_clear & !TSAFE_command : 0; -- only a TSAFE command resolves the current conflict
        -- default: a conflict can be detected at any time
        1 : 0, 1;
      esac;

  FAIRNESS
    1
```

Though most systems are initially defined using assignments per the system requirements as shown above, should the system already be in automaton form, we can specify it by directly describing the system states and how we transition between them. In other words, instead of our previous imperative encoding, using assignment statements, we can equivalently employ a declarative schema using logical constraints. For clarity, we illustrate a direct specification of the automaton in Figure 2 below. Here, `INIT` describes the initial state and the `TRANS` statement lists all of the transitions in our model. This model is equivalent to the one presented above. Both are working NuXmv/NuSMV models.

```
MODULE main
  VAR
    -- Declare the set of system variables.
    AR_command : boolean;
    TSAFE_command : boolean;
    controller_request : boolean;
    aircraft_request : boolean;
    TSAFE_clear : boolean;

  INIT
    -- Initially, the airspace is clear and there are no commands or requests.
    AR_command = 0 &
    TSAFE_command = 0 &
    controller_request = 0 &
    aircraft_request = 0 &
    TSAFE_clear = 1;
  TRANS

-- State 1 ->
    ((!AR_command & !TSAFE_command & !controller_request & !aircraft_request & TSAFE_clear) ->
     -- State 1: no conflicts detected or requests made
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 2: aircraft request sent OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(aircraft_request)) & (next(TSAFE_clear))) |
     -- State 3: AR command sent OR
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 4: controller request sent OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 6: conflict detected
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 2 ->
    ((!AR_command & !TSAFE_command & !controller_request & aircraft_request & TSAFE_clear) ->
     -- State 1: aircraft request ignored OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 3: aircraft request granted
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear)))
     ) &

-- State 3 ->
    ((AR_command & !TSAFE_command & !controller_request & !aircraft_request & TSAFE_clear) ->
     -- State 1: AR command executed OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 5: conflict detected
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 4 ->
    ((!AR_command & !TSAFE_command & controller_request & !aircraft_request & TSAFE_clear) ->
     -- State 1: controller request ignored OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 3: controller request granted
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear)))
     ) &

-- State 5 ->
    ((AR_command & !TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) ->
     -- State 6: AR command ignored
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &
```

```
-- State 6 ->
    ((!AR_command & !TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) ->
     -- State 7: TSAFE command sent
     ((next(!AR_command)) & (next(TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 7 ->
    ((!AR_command & TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) ->
     -- State 1: TSAFE command executed
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear)))
     )

  FAIRNESS
    1

MODULE main
  VAR
    -- Declare the set of system variables.
    AR_command : boolean;
    TSAFE_command : boolean;
    controller_request : boolean;
    aircraft_request : boolean;
    TSAFE_clear : boolean;

  INIT
    -- Initially, the airspace is clear and there are no commands or requests.
    AR_command = 0 &
    TSAFE_command = 0 &
    controller_request = 0 &
    aircraft_request = 0 &
    TSAFE_clear = 1;

  TRANS

-- State 1 ->
    ((!AR_command & !TSAFE_command & !controller_request & !aircraft_request & TSAFE_clear) ->
     -- State 1: no conflicts detected or requests made
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 2: aircraft request sent OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(aircraft_request)) & (next(TSAFE_clear))) |
     -- State 3: AR command sent OR
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 4: controller request sent OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 6: conflict detected
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 2 ->
    ((!AR_command & !TSAFE_command & !controller_request & aircraft_request & TSAFE_clear) ->
     -- State 1: aircraft request ignored OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 3: aircraft request granted
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear)))
     ) &

-- State 3 ->
    ((AR_command & !TSAFE_command & !controller_request & !aircraft_request & TSAFE_clear) ->
     -- State 1: AR command executed OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 5: conflict detected
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 4 ->
    ((!AR_command & !TSAFE_command & controller_request & !aircraft_request & TSAFE_clear) ->
     -- State 1: controller request ignored OR
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear))) |
     -- State 3: controller request granted
     ((next(AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear)))
     ) &

-- State 5 ->
    ((AR_command & !TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) ->
     -- State 6: AR command ignored
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 6 ->
    ((!AR_command & !TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) ->
     -- State 7: TSAFE command sent
     ((next(!AR_command)) & (next(TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(!TSAFE_clear)))
     ) &

-- State 7 ->
    ((!AR_command & TSAFE_command & !controller_request & !aircraft_request & !TSAFE_clear) ->
     -- State 1: TSAFE command executed
     ((next(!AR_command)) & (next(!TSAFE_command)) & (next(!controller_request))
      & (next(!aircraft_request)) & (next(TSAFE_clear)))
     )

  FAIRNESS
    1
```

3. A First LTL Model Checking Exercise In NuXmv

Now consider the problem of model checking a familiar scenario in NuXmv from scratch. We visit the scenario of the Farmer, the Wolf, the Goat, and the Cabbage. These four participants are on the left bank of a river traversing their farm and must successfully cross to the right bank. There are several rules that apply to this farm:

- The only way to traverse the river in in the one rowboat, which has exactly two seats.
- Only the farmer can row. Therefore, for the boat to move, the farmer must occupy one of the two seats to row the boat.
- Each of the Wolf, Goat, and Cabbage occupy one seat on the boat. Thus, a maximum of one of them may traverse the river with the Farmer at any one time.
- The Wolf has a taste for goat meat! If the Wolf is ever left alone with the Goat, the Wolf will eat the Goat. The Wolf will always behave himself and refrain from eating the Goat in the presence of oversight from either the Farmer or Cabbage however. There is not an option for the Goat to increase his self-defense training at this time so the only way to ensure the Goat is left alive is to never leave Goat and Wolf alone together.
- The Goat has a taste for cabbage! If the Goat is ever left alone with the Cabbage, the Goat will eat the Cabbage. The Goat will always behave himself and refrain from eating the Cabbage in the presence of oversight from either the Farmer or Wolf however. Though he is a crafty vegetable, the Cabbage is still no match for the Goat so the only way to ensure the Cabbage is left alive is to never leave Goat and Cabbage alone together.

Though it requires a bit of backtracking, there is certainly a way for all four farm participants to cross the river and arrive safely, without anyone being eaten along the way!

Note that this is also an example of the overlap between the fields of model checking and planning. One sure way to verify there exists some path through a system that has certain characteristics is to assert that this is not the case and use model checking to either confirm that this is the case or provide a counterexample exemplifying such a path.

Exercise 1 [50 points]
Write a NuXmv/NuSMV model that captures the rules of this farm, including processes for the Farmer, Wolf, Goat, and Cabbage.
Write an LTL specification (or re-use the one you wrote for a previous homework) that states that they cannot all cross the river successfully, i.e. with everyone in one piece on the other side. To be clear: this means that your "safety property" is that everyone can cross successfully so your negated specification (representing a "bad" system behavior or something that should *never* happen) will represent that all parties cannot cross successfully. **The result of model checking your LTL specification against your NuXmv model should be a counterexample trace demonstrating a correct crossing of all four participants.** Note that NuXmv does not always find the *shortest* counterexample; that is a whole other area of consideration. So, it is likely that the counterexample your model checking run produces may have some inefficient transitions, such as the farmer and goat going on a few joyrides back and forth across the river without making progress toward crossing everyone.

Exercise 2 [20 points]
Describe what steps you took to *validate* your model, recalling that validation is the check that you have modeled what you intended. How do you know that you have modeled the state space of the system (e.g., how the system works) rather than just the counterexample you expect to see? Possible techniques for validation include applying organized code-review techniques to your model, model checking validation properties such as that there exists a path where the goat gets eaten in your model, other forms of code inspection, calculating requirements coverage metrics, applying algorithms for *reachability* of states, and other checks that you may think of or read in the literature. Be creative! **Describe at least two completed validation checks for full credit; validation checks not covered in class are fair game and are encouraged.**

While symbolic model checking was originally designed to have a scalability advantage over explicit-state model checking, in practice it has several other advantages, including opening up many more options for different algorithms to do the back-end search for a counterexample trace inside the model checker. Different back-end options offer different performance for the verification process, which becomes especially important when reasoning about large and complex systems. One of the advantages of symbolic model checking is that it is easy to try several such options; hence the recommended verification process in symbolic model checking is currently different than that for explicit-state model checking. Here is a good symbolic model checking verification process:

1. Compute or make an educated guess as to a reasonable value k, a good maximum length for a counterexample (as described in Section 2.1). Note that if one can prove, given the size and shape of the model, that no counterexample longer than k can exist then this is the ideal k. In practice, that is usually too difficult and a good estimate or educated guess for k is used instead. Some tools, including nuXmv, have commands built-in to help compute a k, but the algorithms used are often as expensive as model checking itself!
2. Perform bounded model checking using k as an input. The reason for doing this step is that if there is a counterexample (which there often is early in the design and modeling stages), BMC is likely to find that counterexample in less time than unbounded model checking. Recall that if BMC gives a counterexample, it is a real one, but if there is no counterexample it is not a proof that no counterexample exists. However, if you have chosen k wisely, a proof can be constructed. (An example of a real-life verification effort following these steps is published in [48]; note that all models, commands, and results from that study are available online `https://ti.arc.nasa.gov/m/profile/kyrozier/AAC/AAC.html`.)
3. Perform BDD-based model checking. Recall that BDD-based algorithms for symbolic model checking are both *sound* and *complete*: if there is a counterexample of any length it will be found and if no counterexample is found, than none exists.
4. Perform IC3 (also known as PDR)-based model checking. This is a new back-end search algorithm invented by Aaron Bradley in 2011 [3] and using Satisfiability Modulo Theories (SMT) solvers under the hood. By itself, IC3/PDR is incomplete like BMC, and can only check properties with counterexamples below a certain length. However, Niklas Sörensson created an algorithm to extend IC3/PDR called *k-liveness* [11], which, on a very high level, performs the search in the other direction, thus making a complete back-end algorithm, equivalent to the BDD-based back-end algorithm. Using command-line flags or environmental settings for nuXmv, users can easily switch between model-checking back-ends. In practice, this means, if one back-end is slow, or does not terminate in a reasonable about of time, another one might.

The next exercise steps through this verification process. The process is well-established, but for each model-checking tool implementing it may be different. **Note that this exercise is largely directed at learning to read the manual and experiment with a new tool** and plan appropriate time for that.

Exercise 3 [40 points]

1. [10 points] NuSMV has the ability to compute the reachable states from a model, if the model is sufficiently small. One can configure it with the command `./configure CFLAGS=-O3` and run it with the flags `-dynamic -df -int`. Then the `compute_reachable` command returns a Finite State Machine diameter that can be used for k. Compute (or make an educated guess) for k using your choice of either this technique or a different one that you find in the literature. Show your command, why you chose it, and the value you chose for k.
2. [10 points] Perform BMC verification using your k, your model, your LTL formula, and the appropriate command for nuXmv/NuSMV. Show your command. Did you get the desired counterexample using this technique?
3. [10 points] Perform BDD-based symbolic model checking using nuXmv/NuSMV. Show your command and the tool's output.
4. [10 points] Perform IC3-based symbolic model checking using nuXmv/NuSMV. Note that there are commands for both plain IC3 and IC3 with k-liveness. Experiment. Show your command and the tool's output.

4. Translating the Property into a Symbolic Automaton

In LTL model checking, we check LTL formulas representing desired behaviors against a formal model of the system designed to exhibit these behaviors. To accomplish this task, we employ the automata-theoretic approach, in which the LTL formulas must be translated into Büchi automata[2] [45]. This step is performed automatically by the model checker.

Definition 2. *A* **Büchi Automaton** *(BA) is a quintuple* $(Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite set of states.*
- Σ *is a finite alphabet.*
- $\delta : Q \times \Sigma \times Q$ *is a transition relation.*
- $q_0 \in Q$ *is the initial state.*
- $F \subseteq Q$ *is a set of final states.*

A run of a Büchi automaton over an infinite word $w = w_0, w_1, w_2, \ldots \in \Sigma$ *is a sequence of states* $q_0, q_1, q_2, \ldots \in Q$ *such that* $\forall i \geq 0$*,* $\delta(q_i, w_i) = q_{i+1}$*. An infinite word* w *is accepted by the automaton if the run over* w *visits at least one state in* F *infinitely often. We denote the set of infinite words accepted by an automaton* A *by* $\mathscr{L}_\omega(A)$*. A* Generalized Büchi Automaton *(GBA) is one for which* F *is a set of acceptance sets such*

[2]Büchi automata were introduced by J.R. Büchi in 1962 [5].

that an infinite word w is accepted by the automaton if the run over w visits at least one state in each set in F infinitely often.

We also define the extended transition relation $\delta^\omega : Q \times \Sigma^\omega \times Q$, which takes a string, rather than a single character, as the second argument and yields the result of reading in that string. Let λ represent the empty string. Then we can define δ^ω recursively. For all $q \in Q$, $w \in \Sigma^\omega$, $\sigma \in \Sigma$:

$$\begin{aligned}\delta^\omega(q, \lambda) &= q \\ \delta^\omega(q, w\sigma) &= \delta(\delta^\omega(q, w), \sigma).\end{aligned}$$

A computation satisfying LTL formula φ is an infinite word over the alphabet $\Sigma = 2^{Prop}$, which is accepted by the Büchi automaton A_φ corresponding to φ. For this reason, A_φ is also called a *tester* for φ; it characterizes all computations that satisfy φ. The set of computations of φ comprise the language $\mathscr{L}_\omega(A_\varphi)$. Every LTL formula has an equivalent Büchi automaton. The next theorem relates the expressive power of LTL to that of Büchi automata.

Theorem 1. [3] *Given an LTL formula φ, we can construct a nondeterministic Büchi automaton $A_\varphi = \langle Q, \Sigma, \delta, q_0, F \rangle$ such that $|Q|$ is in $2^{O(|\varphi|)}$, $\Sigma = 2^{Prop}$, and $\mathscr{L}_\omega(A_\varphi)$ is exactly $\models \varphi$.*

Proof of Theorem 1. For simplicity, presume φ is in negation normal form ($\neg$ appears only preceeding propositions) and the temporal operators $\Box$ and $\Diamond$ have been eliminated via logical equivalences.

Recall that the closure of LTL formula φ, $cl(\varphi)$ is the set of all of the subformulas of φ and their negations (with redundancies such as φ and $\neg\neg\varphi$ consolidated). Specifically,

- $\varphi \in cl(\varphi)$.
- $\neg\psi \in cl(\varphi) \rightarrow \psi \in cl(\varphi)$.
- $\psi \in cl(\varphi) \rightarrow \neg\psi \in cl(\varphi)$.
- $\xi \wedge \psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.
- $\xi \vee \psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.
- $\mathcal{X}\ \psi \in cl(\varphi) \rightarrow \psi \in cl(\varphi)$.
- $\xi\ \mathcal{U}\ \psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.
- $\xi\ \mathcal{R}\ \psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.

Recall that the *expansion laws* state that $(\xi\ \mathcal{U}\ \psi) = \psi \vee [\xi \wedge \mathcal{X}(\xi\ \mathcal{U}\ \psi)]$ and $(\xi\ \mathcal{R}\ \psi) = \psi \wedge [\xi \vee \mathcal{X}(\xi\ \mathcal{R}\ \psi)]$; we can use these laws to construct an elementary set of formulas. An *elementary set* of formulas is one comprised exclusively of elementary formulas. A *cover* C of a set of formulas Ψ is a set of sets $C = C_0, C_1, \ldots$ such that $\bigwedge_{\psi \in \Psi} \psi \leftrightarrow \bigvee_{C_i \in C} \bigwedge_{\gamma \in C_i} \gamma$. In other words, any computation satisfying the conjunction of the formulas in the set Ψ also satisfies the conjunction of the formulas in at least one of the cover sets $C_i \in C$. An *elementary cover* is a cover comprised exclusively of maximal, consistent sets of formulas.

We define each elementary cover set $C_i \in C$ for the cover C of formula φ to have the following properties:

1. $C_i \subseteq cl(\varphi)$

[3]The proof included here was inspired by the direct proof techniques employed in [27] (which proved a variant of Theorem 1 that reasons more directly over graphs instead of Büchi automata), [13], and [1]. Other proof strategies include the use of alternating automata [43, 44]. Alternatively, one can consider the Büchi automaton A_φ to be a combination of two automata, a local automaton that reasons about consecutive sequences of states and an eventuality automaton that reasons about eventuality formulas ($\mathcal{U}-$ or $\mathcal{F}$-subformulas) [45, 46, 39, 36]. (Note that [46, 39, 36] proved a variant of Theorem 1 for types of Extended Temporal Logic (ETL) that subsume LTL.)

2. C_i is logically consistent (i.e. does not contain any logical contradictions). Specifically, for all subformulas $\xi, \psi \in cl(\varphi)$,
   - $\psi \in C_i \leftrightarrow \neg\psi \notin C_i$
   - $\xi \wedge \psi \in C_i \leftrightarrow \xi, \psi \in C_i$.
   - $\xi \vee \psi \in C_i \leftrightarrow (\xi \in C_i)$ or $(\psi \in C_i)$.
3. C_i is temporally consistent (i.e. logically consistent with respect to temporal operators). We use $\Rightarrow$ to denote subformulas that are *syntactically implied* by a set C_i.
   - $(\xi\ \mathcal{U}\ \psi \in cl(\varphi)) \rightarrow [(\psi \in C_i) \Rightarrow (\xi\ \mathcal{U}\ \psi \in C_i)]$.
   - $[(\xi\ \mathcal{U}\ \psi \in C_i) \wedge (\psi \notin C_i)] \rightarrow (\xi \in C_i)$.
   - $(\xi\ \mathcal{R}\ \psi \in cl(\varphi)) \rightarrow [(\psi \in C_i) \Rightarrow (\xi\ \mathcal{R}\ \psi \in C_i)]$.
4. C_i is maximal. That is, for every subformula $\psi \in cl(\varphi)$, either $\psi \in C_i$ or $\neg\psi \in C_i$.

We can obtain an elementary cover of φ by applying the expansion laws to φ until φ is a propositional formula in terms of elementary subformulas. That is, φ is a combination of only elementary formulas, with no $\mathcal{U}$- or $\mathcal{R}$-formulas occurring at the top level. We convert this expanded formula into disjunctive normal form (DNF) to obtain the cover. Each disjunct is an elementary set and the set of disjuncts is the elementary cover of φ. We round out each set C_i with the formulas in $cl(\varphi)$ that are syntactically implied by C_i according to the rules listed above. We define a state in our automaton A_φ for each cover set C_i in the cover of φ. Since each cover set is elementary, it consists of only atomic propositions and $\mathcal{X}$-subformulas. We label each such state with the subformulas in the elementary set to which it corresponds. (Note that the states in the cover of φ will be labeled by φ.) We will use the $\mathcal{X}$-subformulas of each state to define the transition relation.

Define q_0 as the state or set of states in the cover of φ (i.e. those labeled by φ). If there is more than one such state and we do not want to define q_0 to be a set of initial states, we can create a singular start state with outgoing λ-transitions[4] to all such states. That is, for φ-labeled states $s_0, s_1, \ldots$, we define $\delta(q_0, \lambda) = s_0; \delta(q_0, \lambda) = s_1; \ldots$.

Initially, we set $Q = q_0$, which is the cover of φ. For each state $q \in Q$, we apply the expansion laws and compute the successors of q as a cover of $\{\psi : \mathcal{X}\psi \in q\}$. For each computed successor state q' of q, we add a transition in δ, creating a new state $q' \in Q$ if necessary. That is, $\forall q', \sigma : \delta(q, \sigma) = q', \mathcal{X}\psi \in q \rightarrow \psi \in q'$. We iterate in this fashion until all $\mathcal{X}$-subformulas of all of the states in Q have been covered. The result is a *closed set of elementary covers*, such that there is an elementary cover in the set for each $\mathcal{X}$-subformula of each disjunct of each cover in the set.[5] Constructed in this way, Q is at most the set of all elementary sets of formulas in the closure of φ ($2^{cl(\varphi)}$) and is thus bounded by $2^{O(|\varphi|)}$.

Finally, we define the acceptance conditions, ensuring that each $\mathcal{U}$-subformula is eventually fulfilled and not simply promised forever. We do this by creating for every subformula of φ of the form $\xi\ \mathcal{U}\ \psi$ a set $F_{\xi\ \mathcal{U}\ \psi} \in F$ containing all states labeled by ψ, which fulfill the $\mathcal{U}$-subformula, and all states not labeled by $(\xi\ \mathcal{U}\ \psi)$, which do not promise it. This is the basic construction by Daniele, Giunchiglia, and Vardi [14].

We prove each direction separately based upon these definitions.

If: $\pi \in L_\omega(A_\varphi) \rightarrow \pi \models (\varphi)$

[4]We refer to λ as the empty string. The presence of a λ-transition in a nondeterministic automaton indicates that the automaton may traverse that transition at any time, without progressing any further along the computation path.

[5]Note that any LTL formula has infinitely many elementary covers and which one is chosen for this construction in practice can significantly affect the performance of the model-checking problem. Thus, many researchers study optimizing this construction [20, 14, 40, 19, 21, 38, 16, 18].

Presume $\pi \in L_\omega(A_\varphi)$. Then there is an accepting run in A_φ that we will label $\rho = q_0, q_1, \ldots$ which, by definition of Q, corresponds to the sets $C_0, C_1, \ldots$. By the definition of q_0 as the cover of φ, we know that φ holds in this state. Stated another way, the atomic propositions that are true in q_0 are true in the first time step of a satisfying computation of φ; $\pi_0 \in C_0$. It remains to show that the transition relation and acceptance conditions imply that $\pi \models (\varphi)$, or, in other words, $(C_0 \cap \mathit{Prop})(C_1 \cap \mathit{Prop})(C_2 \cap \mathit{Prop}) \ldots \models \varphi$. We show by structural induction on the structure of $\psi \in \mathit{cl}(\varphi)$ that $\psi \in C_0 \leftrightarrow \pi \models (\psi)$.

Base case: $|\psi| = 1$. Then $\psi \in \mathit{Prop}$. The claim follows from the definition of the labeling of states in Q, i.e. that in our construction, $C_i = \{\psi \in \mathit{cl}(\varphi) | C_i \cap \mathit{Prop})(C_{i+1} \cap \mathit{Prop})(C_{i+2} \cap \mathit{Prop}) \ldots \models \psi$.

Induction step: Presume the claim holds for subformulas $\xi, \eta \in \mathit{cl}(\varphi)$. Then it holds for:

1. $\psi = \neg\xi$. The claim follows from the definition of the labeling of states in Q.
2. $\psi = \mathcal{X}\xi$. From the definition of δ, $\mathcal{X}\xi \in C_i \to \forall q_{i+1}, \sigma : \delta(q_i, \sigma) = q_{i+1}, \xi \in C_{i+1}$. Therefore, $\psi \in C_i \leftrightarrow \pi \models (\psi)$.
3. $\psi = \xi \wedge \eta$. The claim follows from the definition of the labeling of states in Q.
4. $\psi = \xi \vee \eta$. The claim follows from 1, 3, and DeMorgan's law.
5. $\psi = \xi\ \mathcal{U}\ \eta$. If $\pi \models \psi$ then there is some $i \geq 0$ such that $\pi_i, \pi_{i+1}, \ldots \models \eta$ and $\forall j : 0 \leq j < i$, $\pi_j, \pi_{j+1}, \ldots \models \xi$. From our induction hypothesis, we have that $\eta \in C_i$ and $\xi \in C_j$. By induction we have that $\xi\ \mathcal{U}\ \eta \in C_i, C_{i-1}, \ldots, C_0$. From our construction, if $(\xi\ \mathcal{U}\ \eta) \in C_0$ then either $\forall i \geq 0 : \xi, \mathcal{X}(\xi\ \mathcal{U}\ \eta) \in C_0$ and $\eta \notin C_0$ or $\exists i \geq 0 : \eta \in C_i$ and $\forall j : 0 \leq j \leq i, \xi, \mathcal{X}(\xi\ \mathcal{U}\ \eta) \in C_0$. Since we know that ρ, the run of π in A_φ, is accepting, by the definition of F, only the latter case is possible. Therefore, $\psi \in C_0 \leftrightarrow \pi \models (\psi)$.
6. $\psi = \xi\mathcal{R}\eta$. The claim follows from 1, 5, and the definition of $\mathcal{R}$.

Only if: $\pi \models (\varphi) \to \pi \in L_\omega(A_\varphi)$

Since q_0 is defined by the cover of φ, there must be a state $q \in q_0$ such that $\pi_0 \models q$. In general, we want to show that $\forall i : i \geq 0, \pi_i \models q_i$, so we show how to choose π_{i+1}. We know there is a set $C_{i+1} \in \delta(C_i, \pi_i)$ such that $\pi_i, \pi_{i+1}, \ldots \models C_i, C_{i+1}, \ldots$ since for all i:

- The atomic propositions that are true in π_0 are true in the state q_i of our run: $\pi_i \in C_i$.
- For every formula of the form $\mathcal{X}\psi$ in C_i, we know that $\pi_i, \pi_{i+1}, \ldots \models \mathcal{X}\psi$, which means $\pi_{i+1}, \pi_{i+2}, \ldots \models \psi$, so $\psi \in C_{i+1}$.
- For every formula of the form $\xi\ \mathcal{U}\ \psi$ in C_i, we know that $\pi_i, \pi_{i+1}, \ldots \models \xi\ \mathcal{U}\ \psi$, which means either $\pi_i, \pi_{i+1}, \ldots \models \psi$, so $\psi \in C_i$, or $\pi_i, \pi_{i+1}, \ldots \models \xi \wedge \mathcal{X}(\xi\ \mathcal{U}\ \psi)$, so $\xi \in C_i$ and $(\xi\ \mathcal{U}\ \psi) \in C_{i+1}$.

Furthermore, we know from the definition of F that for all subformulas of the form $(\xi\ \mathcal{U}\ \psi) \in \mathit{cl}(\varphi)$, there is a set $F_{\xi\ \mathcal{U}\ \psi} \in F$ containing all $C_i : (\psi \in C_i) \vee ((\xi\ \mathcal{U}\ \psi) \notin C_i)$. We can choose an accepting run as follows. Let U be the set of $\mathcal{U}$-subformulas not fulfilled in the current state, C_i. Then $\forall \xi, \psi : ((\xi\ \mathcal{U}\ \psi) \in C_i) \wedge \psi \notin C_i, (\mathcal{X}(\xi\ \mathcal{U}\ \psi) \in C_i)$. For each such formula in U in succession, we choose the shortest path from the current state to a state that contains ψ, thus fulfilling the claim. We know there must exist such a path for each $\mathcal{U}$-subformula by the construction of δ such that $\mathcal{X}\psi \in C_i \to \psi \in C_{i+1}$, and by the definition of F. Note that if some state $C_i \notin F_{\xi\ \mathcal{U}\ \psi}$, then $((\xi\ \mathcal{U}\ \psi) \in C_i)$ and $(\psi \notin C_i)$ and, also, $\pi_i, \pi_{i+1}, \ldots \models (\xi\ \mathcal{U}\ \psi)$ but $\pi_i, \pi_{i+1}, \ldots \not\models \psi$. This creates a contradiction since we know that $\pi \models (\varphi)$, by definition of Büchi acceptance, necessitates that π passes through a state in each set $F_{\xi\ \mathcal{U}\ \psi} \in F$ infinitely often. $\square$

This theorem reduces LTL satisfiability checking to automata-theoretic nonemptiness checking, as φ is satisfiable iff $models(\varphi) \neq \emptyset$ iff $L_\omega(A_\varphi) \neq \emptyset$.[6]

References

[1] C. Baier and J-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[2] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*. Springer, 1999.

[3] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[4] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, T. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *CAV, Proc. 8th Int'l Conf*, volume 1102 of *LNCS*, pages 428–432. Springer, 1996.

[5] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int'l Congress on Logic, Method, and Philosophy of Sci. 1960*, pages 1–12, Stanford, 1962. Stanford U. Press.

[6] M. Chechik, B. Devereux, and A. Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using SPIN. In *Proc. 8th SPIN*, volume 2057 of *LNCS*, pages 16–36. Springer, 2001.

[7] Y. Choi and M. Heimdahl. Model checking software requirement specifications using domain reduction abstraction. In *ASE, Proc. 18th IEEE Int'l Conf.*, pages 314–317, Oct. 2003.

[8] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV, Proc. 14th Int'l Conf.*, LNCS 2404, pages 359–364. Springer, 2002.

[9] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *STTT Int'l J.*, 2(4):410–425, 2000.

[10] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *ECP*, pages 130–142, 1997.

[11] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 52–59. IEEE, 2012.

[12] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *POPL, Proc. 19th ACM Symp.*, pages 343–354, Albuquerque, New Mexico, Jan 1992.

[13] J-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proc. FM*, pages 253–271, 1999.

[14] N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *CAV, Proc. 11th Int'l Conf*, volume 1633 of *LNCS*, pages 249–260. Springer, 1999.

[6] For another version of this proof with several illustrative examples, see [1].

[15] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 496–500, Boston, MA, Jul 2004. Springer.

[16] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. In *MASCOTS, Proc. 12th Int'l Workshop*, pages 76–83. IEEE, 2004.

[17] H. Erzberger. The automated airspace concept. In *4th USA/Europe Air Traffic Management R&D Seminar*, Santa Fe, New Mexico, USA, Dec. 2001.

[18] C. Fritz. Concepts of automata construction from LTL. In *LPAR, Proc. 12th Int'l Conf.*, LNCS 3835, pages 728–742. Springer, 2005.

[19] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV, Proc. 13th Int'l Conf*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.

[20] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *PSTV*. Chapman & Hall, Aug 1995.

[21] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE, Proc of 22 IFIP Int'l Conf*, Nov 2002.

[22] M. Gribaudo, A. Horvath, A. Bobbio, E. Tronci, E. Ciancamerla, and M. Minichino. Model-checking based on fluid petri nets for the temperature control system of the icaro co-generative plant. Technical report, Proc. SAFECOMP, volume 2434, LNCS, 2002.

[23] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *STTT J.*, 1(1–2):110–122, 1997.

[24] O. Kupferman and M.Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *CAV, Proc. 12th Int'l Conf.*, volume 1855 of *LNCS*, pages 36–52. Springer, 2000.

[25] L. Lamport. Real-time model checking is really simple. In *CHARME*, pages 162–175, 2005.

[26] K. G. Larsen, P. Petterson, and W. Yi. UPPAAL: Status & developments. In *CAV, Proc. 9th Int'l Conf.*, volume 1254 of *LNCS*, pages 456–459. Springer, 1997.

[27] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL, Proc. 12th ACM Symp.*, pages 97–107, New Orleans, Jan 1985.

[28] G. Lindstrom, P. C. Mehlitz, and W. Visser. Model checking real time java using java pathfinder. In *ATVA*, volume 3707, pages 444–456. Springer, 2005.

[29] K. McMillan. The SMV language. Technical report, Cadence Berkeley Lab, 1999.

[30] S. Miller. Will this be formal? In *Theorem Proving in Higher Order Logics*, volume 5170, pages 6–11. Springer, Oct. 2008.

[31] S. P. Miller, A. C. Tribble, M. W. Whalen, M. Per, and E. Heimdahl. Proving the shalls. *STTT*, 8(4-5):303–319, 2006.

[32] D. Peled. All from one, one for all: on model checking using representatives. In *CAV, Proc. 5th Conf.*, Elounda, Jun 1993. LNCS, Springer.

[33] L. Pike. Real-time system verification by k-induction. Technical Report TM-2005-213751, NASA Langley Research Center, May 2005.

[34] A. Cimatti R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. NuSMV 2.4 user manual. Technical report, CMU and ITC-irst, 2005.

[35] F. Raimondi, A. Lomuscio, and M. J. Sergot. Towards model checking interpreted systems. In *In Proc. of 2nd NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS02), volume 2699 of LNAI*, pages 115–125. Springer, 2002.

[36] S. Safra and M.Y. Vardi. On ω-automata and temporal logic. In *STOC, Proc. 21st ACM*, pages 127–137, Seattle, May 1989.

[37] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes Theoretical Comp. Sci.*, 149(1):79–96, 2006.

[38] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *CHARME*, pages 126–140. Springer, 2003.

[39] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In *ICALP, Proc. 10th*, volume 194, pages 465–474, Nafplion, Jul 1985. LNCS, Springer.

[40] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV, Proc. 12th Int'l Conf*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.

[41] A.C. Tribble and S.P. Miller. Software safety analysis of a flight management system vertical navigation function - a status report. In *DASC, 22nd*, volume 1, pages 1.B.1–1.1–9 vol.1, Oct. 2003.

[42] A. Valmari. A stubborn attack on state explosion. In *CAV, Proc. 2nd Conf.*, volume 531 of *LNCS*, pages 156–165, Rutgers, Jun 1990. Springer.

[43] M. Y. Vardi. Automata-theoretic techniques for temporal reasoning. In Frank Wolter Patrick Blackburn, Johan van Benthem, editor, *Handbook of Modal Logic*. Elsevier, Nov 2006.

[44] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer, Berlin, 1996.

[45] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st LICS*, pages 332–344, Cambridge, Jun 1986.

[46] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, Nov 1994.

[47] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *FOCS, Proc. 24th IEEE Symp.*, pages 185–194, Tucson, 1983.

[48] Yang Zhao and Kristin Yvonne Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming Journal*, 96(3):337–353, December 2014.