available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cosrev

Survey

Linear Temporal Logic Symbolic Model Checking

Kristin Y. Rozier

NASA Ames Research Center, Moffett Field, CA 94035, USA

ARTICLE INFO

Article history:
Received 11 February 2010
Received in revised form
26 June 2010
Accepted 29 June 2010

Keywords:
Linear Temporal Logic (LTL)
Symbolic Model Checking (SMC)
Verification
Formal Methods

ABSTRACT

We are seeing an increased push in the use of formal verification techniques in safety-critical software and hardware in practice. Formal verification has been successfully used to verify systems such as air traffic control, airplane separation assurance, autopilot, CPU designs, life-support systems, medical equipment (such as devices which administer radiation), and many other systems which ensure human safety. This survey provides a perspective on the formal verification technique of linear temporal logic (LTL) symbolic model checking, from its history and evolution leading up to the state-of-the-art. We unify research from 1977 to 2009, providing a complete end-to-end analysis embracing a users' perspective by applying each step to a real-life aerospace example. We include an in-depth examination of the algorithms underlying the symbolic model-checking procedure, show proofs of important theorems, and point to directions of ongoing research. The primary focus is on model checking using LTL specifications, though other approaches are briefly discussed and compared to using LTL.

Published by Elsevier Inc.

1. Introduction

Verification of a software or hardware system involves checking whether the system in question behaves as it was designed to behave. *Design Validation* involves checking whether a system design satisfies the system requirements. (If it does not, it is desirable to find out early in the design process!) Both of these tasks, system verification and design validation, can be accomplished thoroughly and reliably using *formal methods*, such as *model checking*. Model checking is the formal process through which a desired behavioral property (the specification) is verified to hold for a given system (the model) via an exhaustive enumeration (either explicit or symbolic) of all of the reachable system states and the behaviors that cause the system to transition between them. If the specification is found not to hold in all system executions, a *counterexample* is produced, consisting of a trace of the model from a start state to an error state in which the specification is violated, providing a very helpful tool for debugging the system design.

The time-honored techniques of simulation and testing, both of which involve checking the system's behavior on a large set of expected inputs, also address similar questions and are extremely useful debugging tools in early stages of system design and verification. However, as a system is refined, the remaining bugs become fewer and more subtle and require more time to uncover. A major gap in the process of using simulation and/or testing for verification and validation is that there is no way to tell when these techniques are finished (i.e. when all of the bugs in the system have been found). In other words, testing and simulation can be used to demonstrate the presence of bugs but not the absence of bugs. There is not even an accurate way of estimating how many bugs remain. Another open question is that of coverage, of both the possible system inputs and the system state space. Quite simply, it has been proven

E-mail address: Kristin.Y.Rozier@nasa.gov.

1574-0137/$ - see front matter. Published by Elsevier Inc.
doi:10.1016/j.cosrev.2010.06.002

that testing and simulation cannot be used to guarantee an ultra-high level of reliability within any realistic period of time [1]. For some systems, this is an acceptable risk. For those systems, it is enough to reduce the bug level below a certain measurable tolerance, for example in terms of frequency in time. For *safety-critical* systems, or other systems, such as financial systems, where reliability is key because failure is potentially catastrophic, we require an absolute assurance that the system follows its specification via an examination of all possible behaviors, including those that are unexpected or unintended. This assurance is provided by model checking.

While there are a range of different techniques for formal verification, model checking is particularly well-suited for the automated verification of finite-state systems, both for software and for hardware. Once the system model and specification have been determined, the performance of the model checking step is often very fast, frequently completing within minutes. The counterexample returned in the case a bug is found provides necessary diagnostic feedback. Furthermore, iterative refinement and re-checking of the failed specification can provide a wealth of insight into the detected faulty system behavior. Model checking lends itself to integration into industrial design life-cycles as the learning curve is quite shallow and easily outweighed by the advantages of early fault detection. The required levels of user interaction and specialized expertise needed to effectively utilize a model checker are minimal compared to other methods of formal verification. Moreover, partial specifications can be checked, allowing verification steps to occur intermittently throughout system design. However, there is a trade-off between the high level of automation provided by model checking and the high level of expressiveness and control that may be required for verification in some cases. For this reason, certain systems benefit from the use of alternative verification techniques, such as theorem proving, which involves logically deducing the specification from the formal system description and a set of axioms and inference rules. Still, model checking's high level of automation makes it a preferable verification method where applicable since the performance time and quality of insight obtained from a negative result when using theorem proving for verification are highly dependent on the particular skill-set of the person providing the proof.

Formally, the technique of model checking checks that a system, starting at a start state, models a specification. Let M be a state-transition graph (i.e. an automaton) representing the system with set of states S and let $s \in S$ be the start state. Let φ be the specification in temporal logic. We check that $M,\ s \models \varphi$. In other words, we check that M satisfies ("models") φ. This technique of temporal logic model checking was developed independently by Clarke and Emerson [2] in the United States and Quielle and Sifakis [3] in France in 1981. Thus, 1981 is considered the birth year of model checking.

The primary focus of this paper is on model checking using Linear Temporal Logic (LTL) specifications. LTL was first introduced as a vehicle for reasoning about concurrent programs by Pnueli in 1977 [4]. LTL model checkers follow the automata-theoretic approach [5], in which the complemented LTL specification $\neg\varphi$ is translated to a Büchi automaton,[1] $A_{\neg\varphi}$, which is a finite automaton on infinite words that accepts exactly all computations that satisfy the formula $\neg\varphi$. $A_{\neg\varphi}$ is then composed with the model M under verification, forming $A_{M,\ \neg\varphi}$ [6]. Intuitively, any accepting path in $A_{M,\ \neg\varphi}$ represents a case where the system M allows a behavior that violates the specification φ. The model checker then searches for such a trace of the model that is accepted by the automaton $A_{M,\ \neg\varphi}$ via a nonemptiness check. If an accepting trace is found, it is returned as a counterexample. If no such trace exists (i.e. the language $\mathscr{L}(A_{M,\ \neg\varphi}) = \emptyset$), we have proven that $M,\ s \models \varphi$. This process is summarized in Table 1. The equivalent to the automata-theoretic approach for branching temporal logics utilizes automata on infinite trees and relies upon a reduction of satisfiability to the nonemptiness problem for these automata [7].

LTL model checkers can be classified as *explicit* or *symbolic*. Explicit model checkers, such as SPIN[2] [8] and SPOT[3] [9], construct the state-space of the model explicitly and create the automaton $A_{M,\ \neg\varphi}$ such that $\mathscr{L}(A_{M,\ \neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$, and $|A_{M,\ \neg\varphi}| = \mathcal{O}(|M| \cdot |A_{\neg\varphi}|)$, where vertical bars indicate the size of an automaton in terms of number of states and transitions. Next, the model checker searches for a trace falsifying the specification. This search equates to a nonemptiness check of the automaton $A_{M,\ \neg\varphi}$. The standard algorithm for this task is Tarjan's depth-first search algorithm for finding strongly connected components in the state-transition graph, which runs in time linear in the sum of the number of states and transitions. (In practice slightly more efficient algorithms are usually implemented [10–12].) However, constructing and searching the state space in this manner requires a considerable amount of space, even when utilizing optimization techniques such as on-the-fly state space construction [13–15]. Given that the size of the state space required for model checking is the largest challenge to its efficacy as a verification technique, utilizing techniques that conserve space is vital.

The *state explosion problem* is widely agreed to be the most formidable challenge facing the application of model checking to large and complex real-world systems. In short, the number of states required to model concurrent systems grows exponentially with the number of system components, constituting the main practical limitation of model checking. Sequential hardware circuits with n input variables and k registers require 2^{n+k} states to represent. Even simple systems, like an n-bit binary counter, can necessitate large state spaces (in this case, 2^n states). In general, a system with n variables over a domain of k possible values requires at least k^n states in the model and reasoning over real-valued variables, which have infinite possible values, results in a state-transition model with infinitely many states. Unfortunately, the state explosion problem is unavoidable in the worst case. However, a host of techniques have been developed over the last three decades that have successfully eased the problem for certain types of systems. For example, sophisticated data structures, clever algorithms for representing interleaving of concurrent components (called partial order reduction [16]), and the use of bisimulation equivalences [17] and compositional (also

[1] Büchi automata are formally defined in Section 3.4.

[2] http://spinroot.com/.

[3] http://spot.lip6.fr/.

Table 1 – Model checking definition.

Model checking	
Description	Implementation
1. Create a mathematical model of the system.	1. Define the system model M containing traces over the set *Prop* of propositions.
2. Encapsulate desired properties in a formal specification.	2. Let specification φ be a formula over the set *Prop*.
3. Check that the model satisfies the specification.	3. Check that $M \models \varphi$: • Translate the specification $\neg\varphi$ into a Büchi automaton $A_{\neg\varphi}$ and compose it with the system model M to form $A_{M,\neg\varphi}$. • Check $A_{M,\neg\varphi}$ for *nonemptiness*. That is, search for a trace that is accepted by $A_{M,\neg\varphi}$. – If such a trace exists, return it as a *counterexample*. – If no such trace exists, return *TRUE*.

called modular) verification [18] to reason about different levels of abstraction, all address the state explosion problem.

In order to mitigate the state explosion problem, symbolic model checkers, such as CadenceSMV [19], NuSMV [20], and VIS [21], represent the system model symbolically using sets of states and sets of transitions. They then analyze the state space symbolically using binary decision diagrams (BDDs) [22]. In contrast with explicit-state model checking, states in symbolic model checking, are represented *implicitly*, as a solution to a logical equation. This saves space in memory since syntactically small equations can represent comparatively large sets of states. All symbolic model checkers essentially use the symbolic translation for LTL specifications described in [23] and the analysis algorithm of [24], though some optimize further. The technique of using BDDs to reason about Boolean formulas representing the state space, thereby avoiding building the state graph explicitly, was invented by McMillan [25] and is considered to be one of the biggest breakthroughs in the history of model checking for its impact on the state explosion problem [26].[4]

We carefully consider each step in the end-to-end process of linear temporal logic symbolic model checking in order, giving the big picture before the lower-level details of each step. Therefore, the structure of this paper is as follows. Section 2 discusses strategies for symbolically modeling a system for verification and introduces our running example of verifying an automated air traffic control architecture. Next, we discuss specifying behavior properties in temporal logic and translating them into symbolic automata in Section 3. We demonstrate how to represent the combined system model and specification using BDDs in Section 4 and then show how to perform the nonemptiness check and produce counterexample traces in Section 5. To facilitate ease of understanding, formal definitions of each construct are given upon first use, rather than upon first mention. For example, while we mention LTL in the title of this paper, it is not defined until Section 3.1.1, where we first discuss the process of writing temporal logic specifications in detail. Finally, we conclude with a discussion in Section 6.

[4] Others independently published ideas similar to McMillan's symbolic model-checking algorithm at around the same time. See [27] for a high-level overview of these techniques.

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 paper. 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; the counterexample returned constitutes a viable path matching those criteria [28]. 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 that will be used throughout the rest of the paper to demonstrate the steps of LTL symbolic model checking as we introduce them.

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** → **model**: Have we modeled the system correctly?
  - **model** → **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 [29], 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 [30,16].)

In this paper, 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 [31]. 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)* [32] 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. They can prove the presence of errors in the model but cannot be used to prove the absence of errors, only the absence of errors reachable within k steps. 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 [33]. Hybrid automata model checkers like HyTech can verify some types of hybrid systems, that is, systems with a mix of discrete and continuous variables [34] 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 [35–40].

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) [41], displayed in Fig. 1.

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.

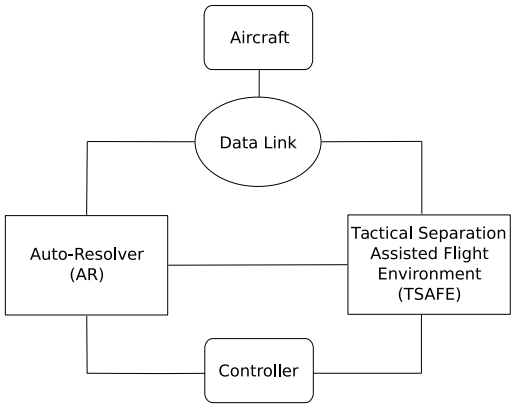


Fig. 1 – Architecture: Automated Airspace Concept for Air Traffic Control.

- 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.

LTL model checkers follow the automata-theoretic approach [5], where the relationship between programs and their specifications is considered as a relationship between languages [31]. 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 enumerates the alphabet Σ and Fig. 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 Fig. 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.

We illustrate a real system description for this protocol using NuSMV [20,42] since it is well-documented [43], open-source,[5] and frequently used in industry [44–49]. However, there are several tools available with similar capabilities including CadenceSMV (which has nearly identical syntax) [19], SAL-SMC [50], VIS [21], 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 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, as discussed in Section 3.5. 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 in Box I, 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. (See Box I.)

Though most systems are initially defined using assignments per the system requirements as shown in Box I, 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 Fig. 2 in Box II. Here, `INIT` describes the initial state and the `TRANS` statement lists all of the transitions in our model.

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

Table 2 – The system alphabet Σ: a set of Boolean-valued system variables.

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?

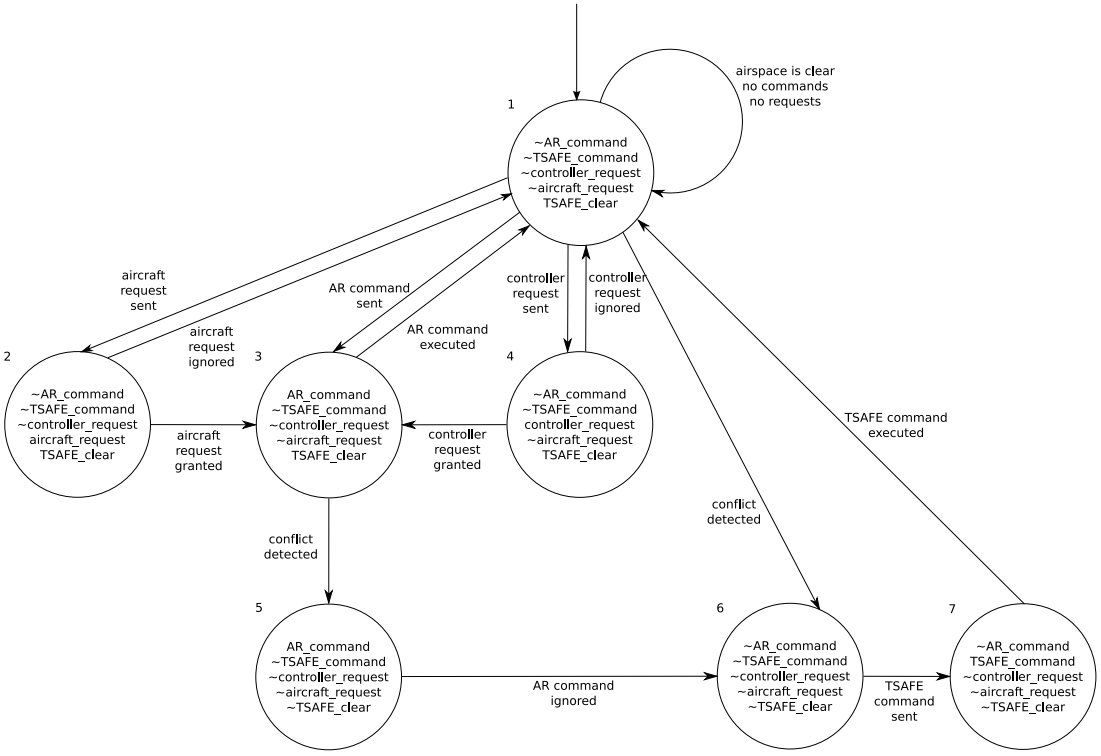


Fig. 2 – Automaton: Automated Airspace Concept for Air Traffic Control.

This model is equivalent to the one presented in Box I. Both are working NuSMV models that will be used as the basis for examples throughout this paper. (See Box II.)

3. Specifying the behavior property

Real-world systems are routinely developed from English-language specifications. Yet, once these systems are built, there is no way to verify that the resulting systems follow the English specifications. Consequently, there is also no way to tell whether following the natural language specifications is a desirable goal, i.e. whether the set of specifications is internally consistent, logically sound, or complete. For formal verification, English is quite simply too imprecise. Consider the statement by Groucho Marx, "I once shot an elephant in my pajamas. How he got in my pajamas I'll never know," or the double-meaning of "Students hate annoying professors." In order to check that a system models its specifications, it is necessary to have a formal, and very precise, notion of exactly what the specifications say. Therefore, we use logic for the specification language. Logic has the advantage of being able to express system specifications in a concise and unambiguous way that is mathematically rigorous and thereby enables automation of the verification process.

Propositional Logic:	
$\neg p$	not
$p \wedge q$	and
$p \vee q$	or
$p \rightarrow q$	implies

We formalize simple grammatical connections using the operators of *propositional logic*: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication). We reason about *atomic propositions*, such as the state variables used in the example of Section 2.2, which are Boolean-valued variables. We refer to the set of atomic propositions as *Prop*. A formula is either an atomic proposition or a sentence comprised of atomic propositions connected by logical operators. The truth value of a formula is determined by an assignment $\mathscr{A}$, which is a mapping of atomic propositions in the domain Σ to truth values: $\mathscr{A} : \Sigma \rightarrow \{0, 1\}$. Thus if there are n atomic propositions in the domain, there are 2^n possible assignments for a given formula. A formula φ is satisfied by an assignment $\mathscr{A}$ that

```
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
```

Box I.

```
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
```

Box II.

causes the overall formula to evaluate to *true*. We can think of our system model M as a set of assignments — each state represents a possible valuation of our state variables.

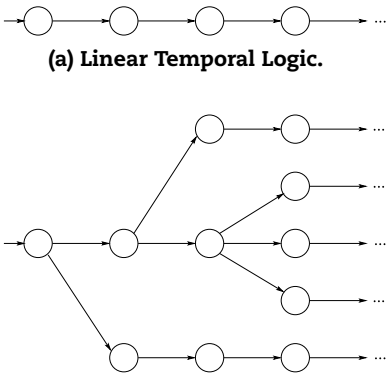
3.1. *Temporal logics*

Continuous systems necessarily involve a notion of time. Propositional logic is not expressive enough to describe such real systems. Yet, English descriptions are even less precise once we involve time. Consider, for example, the following English sentences:

- I said I would see you on Tuesday.
- "This is the worst disaster in California since I was elected." [California Governor Pat Brown]
- "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone." [Kansas State Legislature, early 1890's]

As a result, Amir Pnueli introduced the notion of using temporal logics, which were originally developed by philosophers for investigating how time is used in natural language arguments, to reason about concurrent systems [4]. (Burstall [51] and Kröger [52] independently proposed the use of weaker forms of temporal reasoning about computer programs at roughly the same time.) Temporal logics are modal logics geared toward the description of the temporal ordering of events. For most safety-critical systems, tying the system model to an explicit universal clock is overkill; it exacerbates the state explosion problem without adding value to the verification process. It is sufficient simply to guarantee that events happen in a certain partial (or, in some cases, total) order. Temporal logics were invented for this purpose. Because they describe the ordering of events in time without introducing time explicitly, temporal logics are particularly effective for describing concurrent systems [53]. Temporal logics conceptualize time in one of two ways. Linear temporal logics consider every moment in time as having a unique possible future. Essentially, they reason over a classical timeline. In branching temporal logics, each moment in time may split into several possible futures. In essence, these logics view the structure of time as a tree, rooted at the current time, with any number of branching paths from each node of the tree (Fig. 3).

Pnueli's historic 1977 paper [4] proposed the logic LTL. This logic extended propositional logic with the temporal operators $\mathcal{G}$ (for "globally") and $\mathcal{F}$ (for "in the future"), which we also call $\Box$ and $\Diamond$, respectively, and introduced the concept of *fairness*, which ensures an infinite-paths semantics. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator originally introduced by Kamp in 1968 [54] and the $\mathcal{X}$ ("next time") operator from the temporal logic $\mathcal{UB}$ (the *unified* system of *branching* time) [55]. In 1981, Clarke and Emerson extended $\mathcal{UB}$, thereby inventing the branching temporal logic they named Computational Tree Logic (CTL) and, with it, CTL model checking. Lichtenstein and Pnueli defined model checking for LTL in 1985 [56]. (For a more detailed history of the technical developments which lead from Prior's early observations on time and Church's logical specification of sequential circuits to LTL and automata-theoretic model checking, see [57].) The combination of LTL and CTL, called CTL*, was defined by Emerson and Halpern in 1986, though there are currently no industrial model checkers for this language.[6] Since both model checking in general, and symbolic model checking in particular, were originally defined for CTL, and since LTL model checking is frequently defined in the literature in relation to CTL model checking, we would be remiss not to mention the logic CTL here. Therefore, we follow the formal definition of LTL below with a brief description of the logics CTL and CTL* along with a short discussion of the merits and expressibility of LTL that motivate LTL model checking.



(a) Linear Temporal Logic.

(b) Branching Temporal Logic.

Fig. 3 – Two visions of the structure of time.

3.1.1. *LTL*

Linear Temporal Logic (LTL) reasons over linear traces through time. At each time instant, there is only one real future timeline that will occur. Traditionally, that timeline is defined as starting "now," in the current time step, and progressing infinitely into the future.

Linear Temporal Logic (LTL) formulas are composed of a finite set *Prop* of atomic propositions, the Boolean connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$, and the temporal connectives $\mathcal{U}$ (until), $\mathcal{R}$ (release), $\mathcal{X}$ (also called $\bigcirc$ for "next time"), $\Box$ (also called $\mathcal{G}$ for "globally") and $\Diamond$ (also called $\mathcal{F}$ for "in the future"). Intuitively, $\varphi\ \mathcal{U}\ \psi$ states that either ψ is true now or φ is true now and φ remains true until such a time when ψ holds. Dually, $\varphi\ \mathcal{R}\ \psi$, stated φ *releases* ψ, signifies that ψ must be true now and remain true until such a time when φ is true, thus releasing ψ. $\mathcal{X}\varphi$ means that φ is true in the next time step after the current one. Finally, $\Box\varphi$ commits φ to being true in every time step while $\Diamond\varphi$ designates that φ must either be true now or at some future time step. We define LTL formulas inductively:

Definition 1. For every $p \in Prop$, p is a formula. If φ and ψ are formulas, then so are:

$$\neg\varphi \qquad \varphi \wedge \psi \qquad \varphi \rightarrow \psi \qquad \varphi\ \mathcal{U}\ \psi \qquad \Box\varphi$$
$$\varphi \vee \psi \qquad \mathcal{X}\varphi \qquad \varphi\ \mathcal{R}\ \psi \qquad \Diamond\varphi$$

[6] There was one research prototype CTL* model checker called AltMC (**Alt**ernating Automata-based **M**odel **C**hecker). Visser and Barringer created AltMC in 1998 and explained how it could be integrated into the industrial model checker SPIN [58].

Furthermore, we define the closure of LTL formula φ, $cl(\varphi)$, as the set of all of the subformulas of φ and their negations (with redundancies, such as φ and $\neg\neg\varphi$, consolidated). LTL formulas describe the behavior of the variables in *Prop* over a linear series of time steps starting at time zero and extending infinitely into the future.[7] We satisfy such formulas over *computations*, which are functions that assign truth values to the elements of *Prop* at each time instant [60]. In essence, a computation path π satisfies a temporal formula φ if φ is true in the zeroth time step of π, π_0.

Definition 2. We interpret LTL formulas over computations of the form $\pi : \omega \to 2^{Prop}$, where ω is used in the standard way to denote the set of non-negative integers. We also use *iff* to abbreviate "if and only if." We define $\pi, i \vDash \varphi$ (computation π at time instant $i \in \omega$ satisfies, or "models," LTL formula φ) as follows:

- $\pi, i \vDash p$ for $p \in Prop$ iff $p \in \pi(i)$.
- $\pi, i \vDash \neg\varphi$ iff $\pi, i \nvDash \varphi$.
- $\pi, i \vDash \varphi \wedge \psi$ iff $\pi, i \vDash \varphi$ and $\pi, i \vDash \psi$.
- $\pi, i \vDash \varphi \vee \psi$ iff $\pi, i \vDash \varphi$ or $\pi, i \vDash \psi$.
- $\pi, i \vDash \mathcal{X}\varphi$ iff $\pi, i+1 \vDash \varphi$.
- $\pi, i \vDash \varphi\ \mathcal{U}\ \psi$ iff $\exists j \geq i$, such that $\pi, j \vDash \psi$ and $\forall k, i \leq k < j$, we have $\pi, k \vDash \varphi$.
- $\pi, i \vDash \varphi\ \mathcal{R}\ \psi$ iff $\forall j \geq i$, iff $\pi, j \nvDash \psi$, then $\exists k, i \leq k < j$, such that $\pi, k \vDash \varphi$.
- $\pi, i \vDash \Box\varphi$ iff $\forall j \geq i$, $\pi, j \vDash \varphi$.
- $\pi, i \vDash \Diamond\varphi$ iff $\exists j \geq i$, such that $\pi, j \vDash \varphi$.

We take $\vDash \varphi$ to be the set of computations that satisfy φ at time 0, i.e., $\{\pi : \pi, 0 \vDash \varphi\}$. We define the *prefix* of an infinite computation π to be the finite sequence starting from the zeroth time step, $\pi_0, \pi_1, \ldots, \pi_i$ for some $i \geq 0$.

We now restate the model-checking problem: program M satisfies ("models") formula φ iff every path π rooted at the initial state q of M satisfies φ, denoted $M,\ q \vDash \varphi$.

Examples of LTL properties:

- *Liveness: "Every request is followed by a grant"*
  $\Box(request \to \Diamond grant)$
- *Invariance: "At some point, p will hold forever"*
  $\Diamond\Box p$
- *"p oscillates every time step"*
  $\Box((p \wedge \mathcal{X}\neg p) \vee (\neg p \wedge \mathcal{X}p))$
- *Safety: "p never happens"*
  $\Box\neg p$
- *Fairness: "p happens infinitely often"*
  $(\Box\Diamond p) \to \varphi$
- *Mutual exclusion: "Two processes cannot enter their critical sections at the same time"*
  $\Box\neg(in\_CS_1 \wedge in\_CS_2)$
- *Partial correctness: "If p is true initially, then q will be true when the task is completed"*
  $p \to \Box(done \to q)$

[7] Though we will not discuss them here, some variations of LTL also consider time steps that have happened in the past. For example, we could add past-time versions of $\mathcal{X}$ and $\mathcal{U}$ called $\mathcal{Y}$ (also called $\ominus$ for "previous time" or "yesterday") and $\mathcal{S}$ (since), respectively. Past-time LTL was first introduced by Kamp in 1968 [54] but does not add expressive power to the future-time LTL defined here [59].

Equivalences. While we have presented the most common LTL syntax, operator equivalences allow us to reason about LTL using a reduced set of operators. In particular, the most common minimum set of LTL operators is $\neg$, $\vee$, $\mathcal{X}$, and $\mathcal{U}$. From propositional logic, we know that $(\varphi \to \psi)$ is equivalent to $(\neg\varphi \vee \psi)$ by definition and that $(\varphi \wedge \psi)$ is equivalent to $\neg(\neg\varphi \vee \neg\psi)$ by DeMorgan's law. We can define $(\Diamond\varphi)$ as $(true\ \mathcal{U}\varphi)$. Similarly, $(\Box\varphi) \equiv (false\ \mathcal{R}\ \varphi)$. The *expansion laws* state that $(\varphi\ \mathcal{U}\ \psi) = \psi \vee [\varphi \wedge \mathcal{X}(\varphi\ \mathcal{U}\ \psi)]$ and $(\varphi\ \mathcal{R}\ \psi) = \psi \wedge [\varphi \vee \mathcal{X}(\varphi\ \mathcal{R}\ \psi)]$. The operators $\Box$ and $\Diamond$ are logical duals as $(\Box\varphi)$ is equivalent to $(\neg\Diamond\neg\varphi)$ and $(\Diamond\varphi)$ is equivalent to $(\neg\Box\neg\varphi)$. Finally, $\mathcal{U}$ and $\mathcal{R}$ are also logical duals. Since this last relationship is not intuitive, we offer proof below. (Incidentally, $\mathcal{X}$ is the dual of itself: $(\neg\mathcal{X}\varphi) \equiv (\mathcal{X}\neg\varphi)$.)

Informally, $\varphi\ \mathcal{U}\ \psi$ signifies that either ψ is true now (in the current time step) or φ is true now and for every following time step until such a time when ψ is true. Note that this operator is also referred to as *strong until* because ψ *must* be true at some time. Conversely, *weak until*, sometimes included as the operator $\mathcal{W}$, is also satisfied if φ is true continuously but ψ is never true. Formally,

$\pi, i \vDash \varphi\ \mathcal{W}\psi$ iff either $\forall j \geq i,\ \pi, j \vDash \varphi$ or $\exists j \geq i$
such that $\pi, j \vDash \psi$ and $\forall k, i \leq k < j$, we have $\pi, k \vDash \varphi$.

$\mathcal{R}$ is the dual of $\mathcal{U}$, so $\varphi\ \mathcal{R}\ \psi = \neg(\neg\varphi\ \mathcal{U}\ \neg\psi)$. We say that "$\varphi$ releases ψ" because ψ must be true now and, in the future, $\neg\psi$ implies φ was previously true. Note that at that point in the past, both φ and ψ were true at the same time and that φ may never be true, as long as ψ is always true. We can define $\mathcal{R}$ in terms of the $\mathcal{U}$-operator using the following lemma, where iff is used in the standard way to abbreviate "if and only if."

Lemma 1. $\neg(\neg a\ \mathcal{U}\ \neg b) = a\ \mathcal{R}\ b$.

Proof. We use the formal semantic definitions of $\mathcal{U}$ and $\mathcal{R}$, given in [61].

$$
\begin{aligned}
\pi, i \vDash \xi\ \mathcal{U}\ \psi \quad &\text{iff} \quad \text{for some } j \geq i, \text{ we have } \pi, j \vDash \psi \\
&\quad \text{and for all } k,\ i \leq k < j, \text{ we have } \pi, \\
&\quad k \vDash \xi. \\
\pi, i \vDash \neg(\xi)\ \mathcal{U}\ \neg(\psi) \quad &\text{iff} \quad \text{for some } j \geq i, \text{ we have } \pi, j \vDash \neg(\psi) \\
&\quad \text{and for all } k,\ i \leq k < j, \text{ we have } \pi, \\
&\quad k \vDash \neg(\xi). \\
\pi, i \vDash \neg(\xi)\ \mathcal{U}\ \neg(\psi) \quad &\text{iff} \quad ((\exists\, j \geq i :\ \pi, j \vDash \neg(\psi)) \\
&\quad \wedge(\forall\, k,\ i \leq k < j :\ \pi, k \vDash \neg(\xi))). \\
\pi, i \vDash \neg(\neg(\xi)\ \mathcal{U}\ \neg(\psi)) \quad &\text{iff} \quad \neg((\exists\, j \geq i :\ \pi, j \vDash \neg(\psi)) \\
&\quad \wedge(\forall\, k,\ i \leq k < j :\ \pi, k \vDash \neg(\xi))). \\
&\text{iff} \quad (\neg(\exists\, j \geq i :\ \pi, j \vDash \neg(\psi)) \\
&\quad \vee\neg(\forall\, k,\ i \leq k < j :\ \pi, k \vDash \neg(\xi))). \\
&\text{iff} \quad ((\forall\, j \geq i :\ \pi, j \nvDash \neg(\psi)) \\
&\quad \vee(\exists\, k,\ i \leq k < j :\ \pi, k \nvDash \neg(\xi))). \\
&\text{iff} \quad (\forall\, j \geq i :\ \pi, j \vDash \psi \\
&\quad \vee\exists\, k,\ i \leq k < j :\ \pi, k \vDash \xi). \\
&\text{iff} \quad (\forall\, j \geq i :\ \pi, j \nvDash \psi \\
&\quad \to\exists\, k,\ i \leq k < j :\ \pi, k \vDash \xi). \\
&\text{iff} \quad \text{for all } j \geq i \text{ if } \pi, j \nvDash \psi, \\
&\quad \text{then for some } k,\ i \leq k < j \text{ we have} \\
&\quad \pi, k \vDash \xi. \\
&\text{iff} \quad \pi, i \vDash \xi\ \mathcal{R}\ \psi. \quad \square
\end{aligned}
$$

3.2. *Logical expressiveness of LTL*

In a paper on LTL symbolic model checking, inevitably the question arises: why LTL? What are the alternative specification logics? Why (and when) should we choose LTL? In order to put LTL in the proper context, we briefly, but formally, define the most viable alternatives and discuss when LTL is (and is not) an appropriate choice.

3.2.1. *CTL*

Computational Tree Logic (CTL) is a branching time logic that reasons over many possible traces through time. Historically, CTL was the first logic used in model checking [2] and it remains a popular specification logic. Unlike LTL, for which every time instance has exactly one immediate successor, in CTL a time instance has a finite, non-zero number of immediate successors. A branching timeline starts in the current time step, and may progress to any one of potentially many possible infinite futures. In addition to reasoning along a timeline, as we do for linear time logic, branching time temporal operators must also reason across the possible branches. Consequently, the temporal operators in CTL are all two-part operators with one part specifying, similarly to LTL, the action to occur along a future timeline and another part specifying whether this action takes place on at least one branch or on all branches.

Computational Tree Logic (CTL) formulas are composed of a finite set *Prop* of atomic propositions, the Boolean connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$, and indivisible quantifier pairings of the path quantifiers $\mathcal{A}$ (always, for all paths), and $\mathcal{E}$ (there exists a path), with the linear temporal connectives of LTL: $\mathcal{U}$ (until), $\mathcal{X}$ (also called $\bigcirc$ for "next time"), $\boxtimes$ (also called $\mathcal{G}$ for "globally") and $\Diamond$ (also called $\mathcal{F}$ for "in the future"). (See Fig. 4.) We define CTL formulas inductively:

Definition 3. For every $p \in Prop$, p is a formula. If φ and ψ are formulas, then so are:

$$\neg\varphi \quad \varphi\wedge\psi \quad \varphi\vee\psi \quad \varphi\rightarrow\psi \quad \mathcal{A}(\varphi\,\mathcal{U}\,\psi) \quad \mathcal{E}(\varphi\,\mathcal{U}\,\psi)$$
$$\mathcal{A}\mathcal{X}\varphi \quad \mathcal{E}\mathcal{X}\varphi \quad \mathcal{A}\Box\varphi \quad \mathcal{E}\Box\varphi \quad \mathcal{A}\Diamond\varphi \quad \mathcal{E}\Diamond\varphi$$

CTL formulas describe the behavior of the variables in *Prop* over a branching series of time steps starting at time zero and extending infinitely into the future. As for LTL, *computations* are functions that assign truth values to the elements of *Prop* at each time instant.

Definition 4. We interpret CTL formulas over sets of possible computations of the form $\pi : \omega \rightarrow 2^{Prop}$. Due to the branching nature of the logic, there may be many future paths possible from any one time step: $\pi_0, \pi_1, \pi_2, \ldots$ We will call the set of all of the possible paths from the current time step Π. We define $\Pi, i \vDash \varphi$ (set of possible computations $\pi_{0\ldots m} \in \Pi$ at time instant $i \in \omega$ satisfies CTL formula φ) as follows:

- $\Pi, i \vDash p$ for $p \in Prop$ if $p \in \Pi(i)$.
- $\Pi, i \vDash \neg\varphi$ if $\Pi, i \nvDash \varphi$.
- $\Pi, i \vDash \varphi \wedge \psi$ if $\Pi, i \vDash \varphi$ and $\Pi, i \vDash \psi$.
- $\Pi, i \vDash \varphi \vee \psi$ if $\Pi, i \vDash \varphi$ or $\Pi, i \vDash \psi$.
- $\Pi, i \vDash \mathcal{A}\mathcal{X}\varphi$ if $(\forall n)\pi_n, i+1 \vDash \varphi$.
- $\Pi, i \vDash \mathcal{E}\mathcal{X}\varphi$ if $(\exists n)\pi_n, i+1 \vDash \varphi$.

Linear Temporal Logic (LTL) formulas reason about linear timelines:

- a finite set *Prop* of atomic propositions
- Boolean connectives: $\neg$, $\wedge$, $\vee$, and $\rightarrow$
- temporal connectives:
  - $\mathcal{X}\varphi$ next time
  - $\varphi\,\mathcal{U}\,\psi$ until
  - $\varphi\,\mathcal{R}\,\psi$ release
  - $\Box\varphi$ also called $\mathcal{G}$ for "globally"
  - $\Diamond\varphi$ also called $\mathcal{F}$ for "in the future"

Computational Tree Logic (CTL) reasons about branching paths:

- temporal connectives are always proceeded by path quantifiers:
  - $\mathcal{A}$ for all paths
  - $\mathcal{E}$ exists a path

Fig. 4 – Syntax of LTL and CTL.

- $\Pi, i \vDash \mathcal{A}(\varphi\ \mathcal{U}\ \psi)$ if $(\forall n)(\exists j \geq i)$, such that $\pi_n, j \vDash \psi$ and $\forall k, i \leq k < j$, we have $\pi_n, k \vDash \varphi$.
- $\Pi, i \vDash \mathcal{E}(\varphi\ \mathcal{U}\ \psi)$ if $(\exists n)(\exists j \geq i)$, such that $\pi_n, j \vDash \psi$ and $\forall k, i \leq k < j$, we have $\pi_n, k \vDash \varphi$.
- $\Pi, i \vDash \mathcal{A}\Box\varphi$ if $(\forall n)(\forall j \geq i)$, $\pi_n, j \vDash \varphi$.
- $\Pi, i \vDash \mathcal{E}\Box\varphi$ if $(\exists n)(\forall j \geq i)$, $\pi_n, j \vDash \varphi$.
- $\Pi, i \vDash \mathcal{A}\Diamond\varphi$ if $(\forall n)(\exists j \geq i)$, such that $\pi_n, j \vDash \varphi$.
- $\Pi, i \vDash \mathcal{E}\Diamond\varphi$ if $(\exists n)(\exists j \geq i)$, such that $\pi_n, j \vDash \varphi$.

Equivalences. The path quantifiers $\mathcal{A}$ and $\mathcal{E}$, representing universal and existential quantification over the branching paths are duals and can be defined in terms of each other. Similarly, $\Box$ and $\Diamond$ signify universal and existential quantification over time steps along a single paths and are therefore also duals. (Again, the $\mathcal{U}$-operator is *strong until* because its second argument *must* be true at some time.) These realizations lead us to the following set of semantic equivalences:

- $(\mathcal{A}\Diamond\varphi) \equiv (\neg\mathcal{E}\Box(\neg\varphi)) \equiv (\mathcal{A}(true\ \mathcal{U}\ \varphi))$.
- $(\mathcal{A}\Box\varphi) \equiv (\neg\mathcal{E}\Diamond(\neg\varphi)) \equiv (\neg\mathcal{E}(true\ \mathcal{U}\ \neg\varphi))$.
- $(\mathcal{E}\Diamond\varphi) \equiv (\neg\mathcal{A}\Box(\neg\varphi)) \equiv (\mathcal{E}(true\ \mathcal{U}\ \varphi))$.
- $(\mathcal{E}\Box\varphi) \equiv (\neg\mathcal{A}\Diamond(\neg\varphi)) \equiv (\neg\mathcal{A}(true\ \mathcal{U}\ \neg\varphi))$.
- $(\mathcal{A}\mathcal{X}\varphi) \equiv (\neg\mathcal{E}\mathcal{X}(\neg\varphi))$.
- $(\mathcal{E}\mathcal{X}\varphi) \equiv (\neg\mathcal{A}\mathcal{X}(\neg\varphi))$.
- $(\mathcal{A}[\varphi\ \mathcal{U}\ \psi]) \equiv (\neg\mathcal{E}[(\neg\psi)\ \mathcal{U}(\neg\varphi \wedge \neg\psi)]) \wedge (\neg\mathcal{E}\Box(\neg\psi))$.

There is an automata-theoretic approach to branching time model checking, similar to the ideas described in this paper for LTL. While we do not translate branching temporal formulas into nondeterministic tree automata due to the double exponential blow-up inherent in this process, we can translate them into alternating tree automata in linear time [7].

3.2.2. *LTL vs CTL*

The logical expressiveness of LTL and CTL is incomparable [62]. (See Fig. 5.) Since CTL allows explicit existential quantification over paths, it is more expressive in some cases where we want to reason about the possibility of the existence of a specific path through the transition system model M, such as when M is best described as a computation tree. For example, there are no LTL equivalents of the CTL formulas $(\mathcal{E}\mathcal{X}\ p)$ and $(\mathcal{A}\Box\mathcal{E}\Diamond p)$ since LTL cannot express the possibility of p happening on some path (but not necessarily all

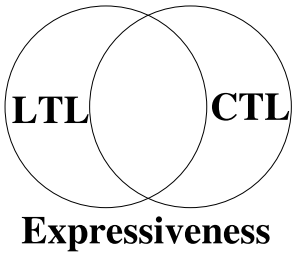


Fig. 5 – Venn Diagram: expressiveness of common temporal logics.

paths) next time, or in the future. LTL describes executions of the system, not the way in which they can be organized into a branching tree. Intuitively, it is difficult (or impossible) to express in LTL situations where distinct behaviors occur on distinct branches at the same time. Conversely, it is difficult (or impossible) to express in CTL some situations where the same behavior may occur on distinct branches at distinct times, a circumstance where the ability of LTL to describe individual paths is quite useful. Realistically, the former rarely happens and LTL turns out to be more expressive from a practical point of view than CTL [63].

Model checking time complexity. The model-checking algorithms for LTL and CTL are different. The major argument in favor of using a branching time logic, like CTL, instead of LTL for property specification is that the model-checking problem for CTL has lower computational complexity than for LTL. Specifically, let $|M|$ indicate the size of the system model in terms of state space and $|\varphi|$ indicate the size of the specification calculated as the total number of symbols: propositions, logical connectives, and temporal operators. Then the model-checking algorithm for CTL runs in time $\mathcal{O}(|M||\varphi|)$ [64] and the model-checking algorithm for LTL runs in time $|M| \cdot 2^{\mathcal{O}(|\varphi|)}$ [56]. Intuitively, this is because CTL is state-based (i.e. reasoning over states in time), and this set of states is easily converted into an automaton whereas the succinct path-based model of LTL (where many possible paths may pass through a single state) must be expanded. This difference in the computational complexity of translating the specification $\neg\varphi$ into the Büchi automaton $A_{\neg\varphi}$ is solely responsible for the difference in the model checking computational complexity for an LTL formula φ and for a CTL formula φ. For LTL, this logic-to-automaton translation step is clearly a bottleneck in the model-checking algorithm; the problem of checking Büchi automata[8] $A_{M,\neg\varphi}$ for nonemptiness is NLOGSPACE-complete [65] and decidable in linear time [66]. There is no hope of reconciling the time complexity of the general model-checking problem for LTL with that of CTL since the model-checking problem for LTL is PSPACE-complete [67]. The best algorithms for LTL model checking, which are exponential in the length of the formula but linear in the size of the model, were proposed by Lichtenstein and Pnueli [56] and Vardi and Wolper [5], the latter algorithm being the basis for most LTL model checking tools today.

However, the comparison of the specification logics LTL and CTL by time complexity is somewhat misleading and certainly not significant enough to be the sole deciding factor of which logic to use for specification. Though model checking can be done in time linear in the size of the specification for CTL [64,3] the requirement for double-operators means that CTL formulas tend to be longer and more complicated than LTL formulas. In fact, it is not entirely clear how much effect the exponential blow-up for LTL model checking has in practice given that the size of most LTL specifications is very small. Furthermore, if we examine specific, practical variants of the model-checking problem for CTL, we find that the complexity dominance of this logic does not hold [63]. Performing specification debugging via satisfiability checking as we describe in Section 3.3 is still PSPACE-complete for LTL [67] but is EXPTIME-complete for CTL [68,69]. For the practical model checking applications of compositional verification, verification of open or reactive systems (i.e. those systems that interact with an environment), verification of concurrent systems, and automata-theoretic verification, LTL-based algorithms either dominate those for CTL or perform similarly [70].

Since there are fast, efficient tools for LTL model checking, and it is questionable to what extent this theoretical difference affects the process of model checking in practice, we focus on the language that is more suitable for specification. From a system design point of view, it is more important to be able to write clear and correct specifications to input into the model checker.

Usually, we want to describe behavioral, rather than structural, properties of the model, making LTL the better choice for specification since such properties are easily expressed in LTL but may not be expressible in CTL. For example, we may want to say that p happens within the next two time steps, $(\mathcal{X}\, p \vee \mathcal{X}\mathcal{X}\, p)$, or that if p ever happens, q will happen too, $(\Diamond p \rightarrow \Diamond q)$, neither of which is expressible in CTL [71]. Similarly, we cannot state in CTL that if p happens in the future, q will happen in the next time step after that, which is simply $\Diamond(p \wedge \mathcal{X}\, q)$ in LTL [72]. Worst of all, it is not obvious that these useful properties should not be expressible in CTL. Indeed, a thorough comparison of the two logics concludes that CTL is unintuitive, hard to use, ill-suited for compositional reasoning, and fundamentally incompatible with semi-formal verification while LTL suffers from none of these inherent limitations and is better suited to model checking in general [63].

LTL is a fair language whereas CTL is not. That is to say that LTL can express properties of fairness, including both *strong fairness* and *weak fairness* whereas CTL cannot (though CTL model checkers generally allow users to specify fairness statements separately to account for this shortcoming). For example, the LTL formula $(\Box\Diamond p \rightarrow \Diamond q)$, which expresses the property that infinitely many p's implies eventually q, is the form of a common fairness statement: a continuous request will eventually be acknowledged. Yet this sentiment is not expressible in CTL. Since fairness properties occur frequently in the specifications we wish to verify about real-life reactive systems, this adds to the desirability of LTL as a specification language.

For another example, the common invariance property $\Diamond\Box p$, meaning "at some point, p will hold forever" cannot be expressed in CTL. It is difficult to see why this formula is not equivalent to the CTL formula $\mathcal{A}\Diamond\mathcal{A}\Box p$; after all,

[8] Büchi automata are formally defined in Section 3.4.

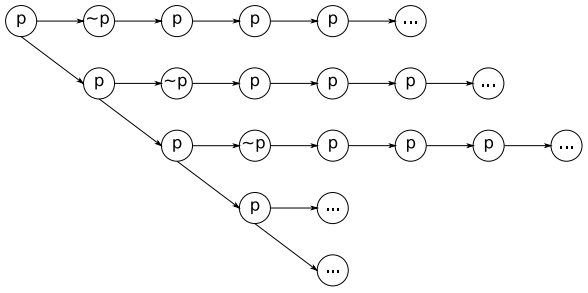


Fig. 6 – A situation where the LTL formula $\Diamond\Box p$ holds but the CTL formula $(\mathcal{A}\Diamond\mathcal{A}\Box p)$ does not.

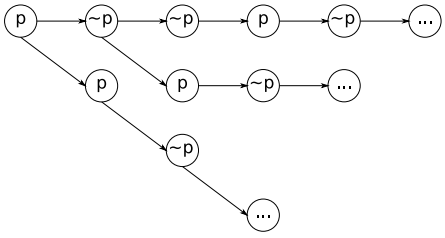


Fig. 7 – A situation where the three equivalent formulas LTL formula $\mathcal{X}\Diamond p$, LTL formula $\Diamond\mathcal{X}\ p$, and CTL formula $\mathcal{A}\mathcal{X}\mathcal{A}\Diamond p$ hold but the CTL formula $(\mathcal{A}\Diamond\mathcal{A}\mathcal{X}\ p)$, which is strictly stronger, does not.

we are basically claiming that on all paths, there's a point where p holds in all future states. (The standard semantic interpretation of LTL corresponds to the "for all paths" syntax of CTL. For this reason, we consider there to be an implicit $\mathcal{A}$ operator in front of all LTL formulas when we compare them to CTL formulas.) To illustrate the inequivalence of these two formulas, we show in Fig. 6 a timeline that satisfies $\Diamond\Box p$ but does not satisfy $\mathcal{A}\Diamond\mathcal{A}\Box p$. Note that $\mathcal{A}\Diamond\mathcal{A}\Box p$ does not hold along the vertical spine in particular — there is never a point where $\mathcal{A}\Box p$ holds along this path. There is always one child where p holds forever and one child where $\neg p$ holds.

Another frequently cited example of the unintuitiveness of CTL is the property that the CTL formula $\mathcal{A}\mathcal{X}\mathcal{A}\Diamond\ p$ is not equivalent to the formula $(\mathcal{A}\Diamond\mathcal{A}\mathcal{X}\ p)$. Again, the distinction is subtle. The former formula states that, as of the next time step, it is true that p will definitely hold at some point in the future or, in other words, p holds sometime in the strict future. This formula is equivalent to the LTL formulas $\mathcal{X}\Diamond p$ and $\Diamond\mathcal{X}\ p$. On the other hand, the meaning of $(\mathcal{A}\Diamond\mathcal{A}\mathcal{X}\ p)$ is the strictly stronger (and actually quite strange) assertion that on all paths, in the future, there is some point where p is true in the next time step on all of the branches from that point. Fig. 7 illustrates this subtlety. Note that in this timeline, p is true in the strict future along every path but there is not a point on every path where p is true in the next step on all branches.

These examples illustrate why LTL is frequently considered a more straightforward language, better suited to specification and more usable for verification engineers and system designers. LTL is the preferred logic of the two for general property specification and on-the-fly verification [73]. Verification engineers have found expressing specifications in LTL to be more intuitive, and less error-prone, than using CTL, particularly for verifying concurrent software architectures [74]. The vast majority of CTL specifications used in practice are equivalent to LTL specifications; it is rare that the added ability to reason over computation tree branches is needed and it frequently requires engineers to look at their designs in an unnatural way [63]. The expressiveness, simplicity, and usability of LTL, particularly for practical applications like open system or compositional verification, and specification debugging, make it a good choice for industrial verification.

3.2.3. *CTL**

Emerson and Halpern first invented the logic CTL* in 1983 [75]. This logic combines the syntaxes of the two logics LTL and CTL. It includes all of the logical operators in LTL and both path quantifiers of CTL but does not have the CTL restriction that temporal operators must appear in pairs. Both LTL and CTL are proper subsets of CTL*, as are all combinations of LTL and CTL formulas; CTL* is more expressive than both LTL and CTL combined. For example, the formulas $E(\Box\Diamond p)$ and $A(\Diamond\Box p) \vee A\Box(E\Diamond p)$ are both in CTL* but in neither LTL nor CTL.

However, this expressive power comes at a great cost. The model-checking problem for CTL* is PSPACE-complete [67], which the same general complexity as for LTL, though the algorithm is considerably more complex to implement and there are currently no model checkers for this logic. Indeed, for practical model-checking problems such as compositional verification and verification of open or reactive systems (i.e. those systems that interact with an environment) CTL* is dominated by LTL [70]. Furthermore, the simple task of specification debugging via satisfiability checking is 2EXPTIME-complete for CTL* [76,77]. Simply translating a CTL* specification into an automaton for model checking involves a doubly-exponential blow-up [78]. So, despite the deceptive time complexity for the general model checking problem, adding branching to LTL is not free. In practice, should LTL prove to be too limited to express a desired property, CTL* is almost certainly sufficient [73]. However, the lack of industrial model-checking tools that accept CTL* specifications is a deterrent to the use of this logic.

3.2.4. *Industrial logics based on LTL*

In several cases, industrial companies have defined extensions of LTL, specialized for their verification needs. Usually these linear time logics add operators to make the expression of specific properties easier and to extend the specification language to full ω-regularity. The optimal position in the trade-off between logical expressiveness and model checking time complexity remains under discussion.

Definition 5. An **ω-regular expression** is an expression of the form $\bigcup_i \alpha_i(\beta_i)^\omega$ where i is non-zero and finite and α and β are regular expressions over the alphabet Σ.

We refer to the standard definition of a regular expression, comprised of the elements of the alphabet Σ, parentheses, and the operators $+$, $\cdot$, and $*$ for union, concatenation, and star-closure, respectively. An ω-regular expression adds the exponent ω that, in some sense, extends the $*$-exponent since a^* designates an arbitrary, possibly zero, finite number of repetitions of a while a^ω designates an infinite number of repetitions of a.

Definition 6. An ω-**regular language** is one that can be described by an ω-regular expression. Also, a language is ω-regular if and only if there exists a Büchi automaton that accepts it. (We discuss Büchi automata in detail in Section 3.4.) The family of ω-regular languages is closed under union, intersection, and complementation.

LTL can express a strict subset of ω-regular expressions; it can describe specifically the $*$-free ω-regular events. For example, LTL cannot express the sentiment that a particular event must occur exactly every n time steps of an infinite computation and that this event may or may not occur during any of the other time steps. Wolper first pointed this out and defined Extended Temporal Logic (ETL), which augmented LTL with operators corresponding to right-linear grammars, thus expanding the expressiveness to all properties that can be described by ω-regular expressions [79]. Vardi and Wolper followed this by proposing ETLs where the temporal operators are defined by finite ω-automata, which provide useful tools for hardware specification [65]. However, model checking with ETL involves a difficult complementation construction for Büchi automata [80]. Banieqbal and Barringer and, separately, Vardi created a linear μ-calculus by extending LTL with fixpoint operators, which allows for a more natural description of constructs like recursive procedures and modules in compositional verification [81, 82]. Sistla, Vardi, and Wolper's Quantified Propositional Temporal Logic (QPTL) avoids common user difficulties with fixpoint calculi and achieves ω-regularity instead by allowing quantification over propositional variables but at the cost of a nonelementary time complexity [83]. Emerson and Trefler proposed dealing with real-time correctness properties while avoiding the nonelementary time complexity by using Real Time Propositional LTL (RTPLTL), which adds time bounds to temporal operators referencing multiple independent clocks but can be checked in time exponential in the size of the regular expression [84].

Verification engineers in industry have also extended LTL to suit their specific needs. After successful verification efforts from 1995 to 1999 using symbolic model checking with specifications in FSL, a home-grown linear temporal logic specialized for hardware checking [85], Intel developed the formal specification language ForSpec [86]. The temporal logic underlying ForSpec, called FTL, extends past-time LTL to add explicit operators for manipulating multiple clocks and reset signals, expressions for reasoning about regular events, and time bounds on the temporal operators that allow users to specify time windows during which an operator is satisfied. The consequence of the increased expressivity of FTL is that checking satisfiability is EXPSPACE-complete. IBM developed their own "syntactic sugar" [87] in parallel from the early 1990s. IBM's Sugar extends LTL with Sugar Extended Regular Expressions (SEREs), the ability to explicitly reference multiple independent clocks, and limited Optional Branching Extensions (OBEs) [88]. Motorola's CBV and Verisity's Temporal ***e*** [89] are also linear-time logics that achieve full ω-regularity by adding regular expressions and clock operations. In 2003, the standardization committee Accellera, considering these four industrial specification languages, announced the industry standard languages SystemVerilog Assertion (SVA) language [90] and Property Specification Language (PSL), which is based chiefly on Sugar with heavy influence from ForSpec [91,92,57]. It is worth noting that restricted variants of LTL have also proved useful for specializing the logic without increasing computational complexity. For example, Allen Linear Temporal Logic (ALTL) [93] marries a restricted LTL, without $\mathcal{X}$-, $\mathcal{U}$-, or $\mathcal{R}$-operators, with Allen's temporal intervals [94], but ALTL satisfiability checking is only NP-complete.

In summary, the industrial languages described in this section are all based on LTL and use model checking methods similar to those described in this paper. Thus, insights into LTL model checking extend to all of these languages.

3.3. *Specification debugging*

Just as designing a bug-free system is difficult, causing us to employ model checking to find the bugs, writing correct specifications to check the system is difficult. Therefore, an important step in between writing specifications and performing the model checking step is specification debugging. As specifications are usually significantly smaller than the systems they describe, they are significantly easier to check. The goal of specification debugging is to answer as best as possible the question "do the specifications say what I meant?" Though it is impossible to answer this question absolutely, it is usually sufficient to run a series of tests on the set of specifications meant to flag situations where the specifications as written clearly cannot match their authors' intentions.

When the model does not satisfy the specification, model-checking tools accompany this negative answer with a counterexample, which points to an inconsistency between the system and the desired behaviors. It is often the case that there is an error in the system model or in the formal specification. Such errors may not be detected when the answer of the model-checking tool is positive: while a positive answer does guarantee that the model satisfies the specification, the answer to the real question, namely, whether the system has the intended behavior, may be different. We need to ask two questions:

1. If there is disagreement between the system model and the specification, which one has the error?
2. If there is agreement, is this caused by an error? (A positive model checking result does not mean there is no error. For example, the specification could have a bug!)

Specification testing can be performed by specification authors writing small system models that they believe should or should not satisfy each specification and then verifying this is the case via model checking. However, this process can be a time-consuming, tedious, and error-prone one.

The realization of this unfortunate situation has led to the development of several *sanity checks* for formal verification [95]. The goal of these checks is to detect errors in the system model or the properties. Sanity checks in industrial tools are typically simple, ad hoc, tests, such as checking for enabling conditions that are never enabled [96]. Standard specification testing can also be preformed more intelligently using *concept analysis*, which produces a hierarchical set of clusters of test traces grouped by similarities [97]. This technique allows the specification author to inspect a small number of clusters instead of a large

number of individual traces and use the similarities in the clusters to help determine whether the set of traces in each cluster points to specification error(s) or not.

Of course, it is extremely desirable to run automated tests on specifications. *Vacuity detection* provides a systematic approach for checking whether a subformula of the specification does not affect the satisfaction of the specification in the model. Intuitively, a specification is satisfied vacuously in a model if it is satisfied in some non-interesting way. For example, the Linear Temporal Logic (LTL) specification $\Box(req \rightarrow \Diamond grant)$ ("every request is eventually followed by a grant") is satisfied vacuously in a model with no requests. While vacuity checking cannot ensure that whenever a model satisfies a formula, the model is correct, it does identify certain positive results as vacuous, increasing the likelihood of capturing modeling and specification errors. Several papers on vacuity checking have been published over the last few years [98–105], and various industrial model-checking tools support vacuity checking [98–100].

At a minimum, it is necessary to employ *LTL satisfiability checking* [106]. If the specification is *valid*, that is, true in *all* models, then model checking this specification always results in a positive answer. Basically, the specification is irrelevant. Consider for example the specification $\Box(b_1 \rightarrow \Diamond b_2)$, where b_1 and b_2 are propositional formulas. If b_1 and b_2 are logically equivalent, then this specification is valid and is satisfied by all models. Clearly, if a formal property is valid, then this is certainly due to an error. Similarly, if a formal property is *unsatisfiable*, that is, true in *no* model, then this is also certainly due to an error. For example, if the specification is $\Box(b_1 \wedge \Diamond b_2)$ where b_1 and b_2 are contradictory, then the specification can never be true. Even if each individual property written by the specifier is satisfiable, their conjunction may very well be unsatisfiable. Recall that a logical formula φ is valid iff its negation $\neg\varphi$ is not satisfiable. Thus, as a necessary sanity check for debugging a specification, we must ensure that both the specification φ and its negation $\neg\varphi$ are satisfiable and that the conjunction of all specifications is satisfiable.

Fortunately, this check is easily performed using standard model-checking tools to check the specifications against a universal model before checking them against the system model. A basic observation underlying this conclusion is that LTL satisfiability checking can be reduced to model checking. Consider a formula φ over a set *Prop* of atomic propositions. If a model M is *universal*, that is, it contains all possible traces over *Prop*, then φ is satisfiable precisely when the model M does *not* satisfy $\neg\varphi$. Thus, it is easy to include satisfiability checking using LTL model-checking tools as a specification debugging step in the verification process.[9]

[9] In contrast, for branching-time logics such as CTL or CTL*, satisfiability checking is significantly harder than model checking. For LTL, both satisfiability checking and model checking are PSPACE-complete with respect to formula size [67]. On the other hand, with respect to formula size, model checking is NLOGSPACE-complete for CTL [107] and PSPACE-complete for CTL* [66], while satisfiability is EXPTIME-complete for CTL [69,68] and 2EXPTIME-complete for CTL* [77,76].

3.4. 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[10] [5]. This step is performed automatically by the model checker.

Definition 7. 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 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 $q \in Q$, $w \in \Sigma^\omega$, $\sigma \in \Sigma$:

$$\delta^\omega(q, \lambda) = q$$
$$\delta^\omega(q, w\sigma) = \delta(\delta^\omega(q, w), \sigma).$$

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 2. [11] *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 $\models \varphi$.*

[10] Büchi automata were introduced by J.R. Büchi in 1962 [108].

[11] The proof in this paper was inspired by the direct proof techniques employed in [56] (which proved a variant of Theorem 2 that reasons more directly over graphs instead of Büchi automata), [109], and [72]. Other proof strategies include the use of alternating automata [61,110]. 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) [5,65,111,112]. (Note that [65, 111,112] proved a variant of Theorem 2 for types of Extended Temporal Logic (ETL) that subsume LTL.)

Proof of Theorem 2. 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 the equivalences described in Section 3.1.1.

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 $(\xi\ \mathcal{U}\ \psi) = \xi \vee [\varphi \wedge \mathcal{X}(\xi\ \mathcal{U}\ \psi)]$ 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 with respect to the closure of φ is a maximal, consistent subset of $cl(\varphi)$. 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 elementary (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. $C_i \subseteq cl(\varphi)$
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 only constants (*true* or *false*), propositions, or $\mathcal{X}$-rooted subformulas. Then φ has 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. 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[12] 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.[13] 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 [114].

We prove each direction separately based upon these definitions.

If: $\pi \in \mathcal{L}_\omega(A_\varphi) \rightarrow \pi \models (\varphi)$

Presume $\pi \in \mathcal{L}_\omega(A_\varphi)$. 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, $(C_0 \cap Prop)(C_1 \cap Prop)(C_2 \cap Prop)\ldots \models \varphi$. We show by structural induction on the structure of $\psi \in cl(\varphi)$ that $\psi \in C_0 \leftrightarrow \pi \models (\psi)$.

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

Induction step: Presume the claim holds for subformulas $\xi, \eta \in cl(\varphi)$. 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, $\mathcal{X}\xi \in C_i \rightarrow \forall q_{i+1}, \sigma : \delta(q_i, \sigma) = q_{i+1}, \xi \in C_{i+1}$. 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.

[12] 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.

[13] 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 [113–119,9].

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 $\forall i \geq 0 : \xi, \mathcal{X}(\xi\ \mathcal{U}\ \eta) \in C_i$ and $\eta \notin C_i$ or $\exists i \geq 0 : \eta \in C_i$ and $\forall j : 0 \leq j \leq i, \xi, \mathcal{X}(\xi\ \mathcal{U}\ \eta) \in C_i$. 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: $\pi \models (\varphi) \rightarrow \pi \in \mathcal{L}_\omega(A_\varphi)$

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 $(\xi\ \mathcal{U}\ \psi) \in cl(\varphi)$, 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 $\mathcal{X}\psi \in C_i \rightarrow \psi \in C_{i+1}$, 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. □

This theorem reduces LTL satisfiability checking to automata-theoretic nonemptiness checking, as φ is satisfiable iff $models(\varphi) \neq \emptyset$ iff $\mathcal{L}_\omega(A_\varphi) \neq \emptyset$.[14]

3.5. *The two types of properties*

There are two fundamental properties we can prove about programs: *safety* and *liveness*. These categories were originally introduced by Leslie Lamport [120]. They are also referred to as *invariance* and *eventuality*, respectively [59]. Distinguishing between the two types of properties is helpful because different techniques can be used to prove each type [121,122]. For example, taking advantage of the structure of safety properties can be used to optimize assume-guarantee reasoning [122]. Showing a safety property holds involves an invariance argument while liveness properties require demonstration of well-foundedness. Here, we adapt the formal definitions of safety and liveness from Alpern and Schneider [123]. Kindler [124] provides a complete survey of historical (and sometimes conflicting) definitions.

Intuition 1. *A* ***Safety Property*** *expresses the sentiment that "something bad never happens."*

In general, safety properties take the form $\Box good$, where *good* is a temporal logic formula describing a good system behavior. In other words, the system always stays in some allowed region of the state space. Recall that LTL properties are interpreted over computations, which are infinite words over the alphabet $\Sigma = 2^{Prop}$ where *Prop* is the set of atomic propositions in the formula. Intuitively, "something bad" only needs to happen once in a computation for the property to be violated; what happens after that in the infinite computation does not affect the outcome of the model checking step. Therefore, if a safety property φ is violated, there must be some finite prefix $\alpha = \pi_0, \pi_1, \ldots, \pi_i$ of the computation π in which this violation occurs. Then, for any infinite computation β over the alphabet Σ, the concatenation of computations $\alpha \cdot \beta$ cannot satisfy φ.

Safety properties were first formally defined in [125]. For property φ, infinite computation $\pi = \pi_0, \pi_1, \ldots$, and alphabet $\Sigma = 2^{Prop}$, we define safety as follows:

$$(\forall \pi, \pi \in \Sigma^\omega : \pi \models \varphi \leftrightarrow (\forall i, 0 \leq i : (\exists \beta, \beta \in \Sigma^\omega : \pi_{0\ldots i} \cdot \beta \models \varphi))),$$

where $\cdot$ is the concatenation operator and Σ^ω denotes the set of infinite words (or ω-words) over Σ.

Restated, φ is a safety property satisfied by computation π if and only if for all lengths of finite prefixes of π, that finite prefix concatenated with some infinite computation models φ. This is the contrapositive of the statement that if $\pi \not\models \varphi$ then there is some finite prefix of π in which something bad happens that cannot be extended into an infinite computation that will satisfy φ.

We can also define a safety property in terms of language closure [123]. In Section 3.4 we translate the LTL formula into a Büchi automaton. The closure of a reduced Büchi automaton is obtained by making all states accepting states. (A Büchi automaton is reduced if there is a path to an accepting state from every state in the automaton; i.e. there are no dead end "trap" states.) A closure of an automaton representing a safety property must accept all prefixes of the language defined by that property.

Lemma 3. *A reduced Büchi automaton A_φ specifies a safety property φ if and only if $\mathcal{L}(A_\varphi) = \mathcal{L}(cl(A_\varphi))$.*[15]

Examples of safety properties include invariance, partial correctness, mutual exclusion, ordering (i.e. first-come-first-serve), deadlock freedom, and reachability. Safety properties are also called invariance properties because they describe

[14] For another version of this proof with several illustrative examples, see [72].

[15] Note that Sistla showed that the problem of determining whether an LTL formula, or the nondeterministic Büchi automaton representing it, express a safety property is PSPACE-complete [126].

predicates that do not change under a set of transformations. Proving partial correctness of a sequential program provides assurance that the program never halts with a wrong answer. Mutual exclusion prohibits the simultaneous use of a common resource by multiple processes (i.e. two processes cannot write to the same file at the same time). Ordering of events, such as first-come-first-served, forbids service out of the established order. Freedom from deadlock ensures that the system will never reach a standstill from which no progress can be made, such as when all processes are simultaneously locked out of, and waiting on, a common resource. In a sense, reachability is the dual of safety as it is an assurance that a certain program state can be reached (i.e. $\neg(\Box\neg good)$).

In the case of our air traffic control example, a safety property we want to check is that no conflict goes unaddressed. More specifically, whenever a conflict is detected, TSAFE immediately issues a command to resolve it. In LTL, we would express this safety property as $\Box(\neg TSAFE\_clear \rightarrow \mathcal{X}(TSAFE\_command))$. Another safety property we want to check is the absence of conflicting commands. That is, both TSAFE and the auto-resolver cannot issue commands to the aircraft at the same time. In LTL, this is $\Box(\neg(AR\_command \wedge TSAFE\_command))$.

Intuition 2. *A* ***Liveness Property*** *expresses the sentiment that "something good must eventually happen."*

Typically, liveness properties express $\Diamond good$, where *good* once again is a temporal logic formula describing a good system behavior. Whereas safety properties reason about reaching specific states, liveness properties reason about control flow between states. Liveness is loosely defined as a program's ability to execute in a timely manner.

Let Σ^{finite} be the set of all finite computations over Σ. For property φ, we define liveness as follows:

$$(\forall \alpha, \alpha \in \Sigma^{finite} : (\exists \beta, \beta \in \Sigma^{\omega} : \alpha \cdot \beta \models \varphi)),$$

where $\cdot$ is the concatenation operator.

The formal definition of liveness is basically defined oppositely from safety. In short, there is no finite prefix over Σ that cannot be extended into an infinite computation that satisfies φ. Intuitively, this corresponds to the notion that the "something good" that satisfies the property can still happen after any finite execution.

As with safety, we can also frame the definition of liveness in terms of language closure of reduced Büchi automata. If A_φ is a reduced Büchi automaton then a computation is not in $cl(A_\varphi)$ if and only if it attempts to traverse a transition that is not defined in the transition relation δ for A_φ.[16] Furthermore, since every finite prefix of liveness property φ can be extended into an infinite computation that satisfies φ, then the *closure* of A_φ actually accepts all infinite words over the alphabet Σ.

Lemma 4. *A reduced Büchi automaton A_φ specifies a liveness property φ if and only if $\mathcal{L}(cl(A_\varphi)) = \Sigma^{\omega}$.*

[16] Büchi automata and the transition relation δ are formally defined in Section 3.4.

Liveness properties are also called eventualities since they promise that some event will happen, eventually. In other words, the system will eventually progress through "useful" states, even if some stuttering is allowed. Termination is a liveness property; it asserts the program will eventually halt instead of hanging forever. Other examples of liveness properties include total correctness, accessibility/absence of starvation, fairness (also called justice), compassion (also called strong fairness), and livelock freedom. Total correctness extends partial correctness with termination; it states that not only must the program not terminate in a bad state, the program must terminate in a good state. Starvation occurs when a process is unable to make progress due to insufficient access to shared resources; it is assuaged by assuring regular accessibility to necessary resources. Fairness guarantees each process the chance to make progress while compassion extends fairness to include the existence of sets of cyclically recurring system states. A fair scheduler executes a process infinitely often while a compassionate scheduler ensures that a process that is enabled infinitely often is executed infinitely often. Freedom from livelock means that the system will never reach a *live* standstill, where the processes in question cannot make progress yet are not blocked but thrashing indefinitely. An intuitive example of livelock occurs when two people try to pass each other in the hallway yet repeatedly step to the same side of the hallway in an effort to let the other person pass. Both people are still walking but they are now oscillating side-to-side instead of making progress forward.

Common examples of liveness properties include: "every request will eventually be granted," "a message will eventually reach its destination," and "a process will eventually enter its critical section." In the case of our air traffic control example, a liveness property to check is that all conflicts are eventually resolved: $\Box(\neg TSAFE\_clear \rightarrow \Diamond TSAFE\_clear)$. We may also want to check that every request, either from the controller or from the aircraft, is addressed by the system, either granted or ignored. This adds two more liveness properties to our specification: $\Box(controller\_request \rightarrow \Diamond\neg controller\_request)$ and $\Box(aircraft\_request \rightarrow \Diamond\neg aircraft\_request)$.

Model checking of safety properties is simpler than model checking of liveness properties. Due to the temporal structure of liveness properties, they must be witnessed by an infinite counterexample in the shape of a lasso. That is, a counterexample with a finite prefix leading to a bad cycle that represents an unending run of the system in which the liveness property is never fulfilled. Safety properties, on the other hand, are always violated within the finite prefix, eliminating the need to compute the remainder of the counterexample corresponding to the loop at the end of the lasso. Intuitively, a safety property is violated the first instance the system enters some prohibited state. Though we provide the full model-checking algorithm for all possible LTL specifications in Section 5, a simpler check using symbolic reachability analysis can be used instead to check only safety properties [122].

With only one exception, safety and liveness properties are disjoint. The only property that is both a safety and a liveness property is the property *true*, which describes all possible

behaviors, or the words in the set $(2^{Prop})^{\omega}$.[17] The proof of this statement follows from the definitions of safety and liveness above, specifically $\mathscr{L}(true) = \mathscr{L}(cl(true)) = \Sigma^{\omega}$. Remarkably, while every LTL formula is not strictly either a safety or a liveness property, Alpern and Schneider [127] proved that every LTL formula is a combination of a safety and a liveness property.

3.6. *Specifying the property as an automaton*

Instead of using a temporal logic to describe a behavioral property, we can use an automaton directly. Many model checkers, like NuSMV and CadenceSMV, will accept specifications in either format and some, like Lucent's FormalCheck [128], deal exclusively with automata inputs. However, complementing a Büchi automaton is quite difficult (and likely involves the expensive step of determinization) whereas complementing an LTL formula can be done in constant time by simply preceeding the formula with a negation symbol. Currently, the best time complexity for complementation of Büchi automata is $2^{\mathcal{O}(n \log n)}$ [129] but this is still an open area of research [130–132]. Advantages of this method include the ability to directly and efficiently construct the automaton A_{φ} rather than relying on an LTL-to-Büchi translation, while disadvantages include the unintuitiveness of representing a behavior specification as an automaton.

3.7. *LTL → Symbolic GBA*

Given an LTL specification φ, the model checker creates a generalized Büchi automaton $A_{\neg\varphi}$, that recognizes precisely the executions that do not satisfy φ, and is destined for composition with the model under verification. The size of $A_{\neg\varphi}$ is $\mathcal{O}(2^{|\varphi|})$ in the worse case, which provides motivation for utilizing a succinct, symbolic representation instead of explicitly constructing the automaton. Symbolic model checkers, such as NuSMV [20], CadenceSMV [19], SAL-SMC [50], and VIS [21], represent and analyze the system model and specifications symbolically. In the symbolic representation of automata, states are viewed as truth assignments to Boolean state variables and the transition relation is defined as a conjunction of Boolean constraints on pairs of current and next states [22]. All symbolic model checkers use the LTL-to-symbolic automaton translation described in [23], some with minor optimizations. Essentially, these tools support LTL model checking via the symbolic translation of LTL to a fair transition system which can be checked for nonemptiness. Recall that fairness constraints specify sets of states that must occur infinitely often in any path. They are necessary to ensure that the subformula ψ holds in some time step for specifications of the form $\xi\,\mathcal{U}\,\psi$ and $\Diamond\psi$. We enumerate the steps of the standard LTL-to-symbolic automaton algorithm [23], in detail, below. Bolded steps are those that produce output that appears in the symbolic automaton.

Input: An LTL formula f.

1. Negate f and **declare the set AP_f of atomic propositions of f.**
2. Build *el_list*, the list of elementary formulas in f:
   - $el(p) = \{p\}$ if $p \in AP$.
   - $el(\neg g) = el(g)$.
   - $el(g \vee h) = el(g) \cup el(h)$.
   - $el(g \wedge h) = el(g) \cup el(h)$.
   - $el(\mathcal{X}g) = \{\mathcal{X}g\} \cup el(g)$.
   - $el(g\ \mathcal{U}\ h) = \{\mathcal{X}(g\ \mathcal{U}\ h)\} \cup el(g) \cup el(h)$.
   - $el(g\ \mathcal{R}\ h) = \{\mathcal{X}(g\ \mathcal{R}\ h)\} \cup el(g) \cup el(h)$.
   - $el(\Box g) = \{X(\Box g)\} \cup el(g)$.
   - $el(\Diamond g) = \{X(\Diamond g)\} \cup el(g)$.
3. **Declare a new variable $EL_{\mathcal{X}_g}$ for each formula $\mathcal{X}g$ in the list** *el_list*.
4. Define the function $sat(g)$, which associates each elementary subformula g of f with each state that satisfies g:
   - $sat(g) = \{q | g \in q\}$ where $g \in el(f)$, $q \in Q$.
   - $sat(\neg g) = \{q | q \notin sat(g)\}$.
   - $sat(g \vee h) = sat(g) \cup sat(h)$.
   - $sat(g \wedge h) = sat(g) \cap sat(h)$.
   - $sat(g\ \mathcal{U}\ h) = sat(h) \cup (sat(g) \cap sat(\mathcal{X}(g\ \mathcal{U}\ h)))$.
   - $sat(g\ \mathcal{R}\ h) = sat(h) \cap (sat(g) \cup sat(\mathcal{X}(g\ \mathcal{R}\ h)))$.
   - $sat(\Box g) = sat(g) \cap sat(\mathcal{X}(\Box g))$.
   - $sat(\Diamond g) = sat(g) \cup sat(\mathcal{X}(\Diamond g))$.
5. **Add fairness constraints to the SMV input model:**

   $\{sat(\neg(g\ \mathcal{U}\ h) \vee h) | g\ \mathcal{U}\ h \text{ occurs in } f\}$.

   $\{sat(\neg(\Diamond\ g) \vee g) | \Diamond\ g \text{ occurs in } f\}$.

   Note that we do not add a fairness constraint for subformulas of the form $g\ \mathcal{R}\ h$. While $\mathcal{R}$ is the dual of $\mathcal{U}$, it is acceptable if h is true infinitely often and g is never true.
6. **Construct the characteristic function S_h of $sat(h)$. For each subformula h in $\neg f$:**

$S_h = p$	if p is an atomic proposition.
$S_h = EL_h$	if h is elementary formula $\mathcal{X}g$ in *el_list*.
$S_h = !S_g$	if $h = \neg g$
$S_h = S_{g1} \vert S_{g2}$	if $h = g_1 \vee g_2$
$S_h = S_{g1} \& S_{g2}$	if $h = g_1 \wedge g_2$
$S_h = S_{g2} \vert (S_{g1} \& S_{\mathcal{X}_{(g1\ \mathcal{U}\ g2)}})$	if $h = g_1\ \mathcal{U}\ g_2$
$S_h = S_{g2} \& (S_{g1} \vert S_{\mathcal{X}_{(g1\ \mathcal{R}\ g2)}})$	if $h = g_1\ \mathcal{R}\ g_2$
$S_h = S_g \& S_{\mathcal{X}_{(\Box g)}}$	if $h = \Box g$
$S_h = S_g \vert S_{\mathcal{X}_{(\Diamond g)}}$	if $h = \Diamond g$

7. **For each subformula rooted at an $\mathcal{X}$, define a `TRANS` statement of the form:**
   - `TRANS (S_X_g = next(S_g))`
8. Print the SMV program. (Since we type code in ASCII text, the operators $\Box$ and $\Diamond$ are represented in SMV syntax by their alphabetic character equivalents, $\mathcal{G}$ and $\mathcal{F}$, respectively.) (See Box III.)

Proof of the correctness of this LTL-to-symbolic-automaton construction is given in [23].

[17] Note ω was originally defined by Cantor to be the lowest transfinite ordinal number so $(2^{Prop})^{\omega}$ designates the set of all infinite words over the alphabet $\Sigma = 2^{Prop}$.

```
MODULE main

VAR /*declare a boolean variable for each atomic proposition in f*/

a: boolean;
b: boolean;

/*and declare a new variable EL_X_g for each formula (X g) in el_list*/
EL_X_g: boolean;

DEFINE  /*for each S_h in the characteristic function, put a line here*/
  S_a := a;
  S_b := b;
  S_g1 := ...
  S_g2 := ...

TRANS   /*for each (X g) in el_list, generate a line here*/
  ( S_X_g1 = next(S_g1) ) &
  ( S_X_g2 = next(S_g2) ) &
           ...
  ( S_X_gn = next(S_gn) )

/*for each (g U h) and (F g) in the parse tree, generate lines like this:*/
FAIRNESS   ! S_gUh | S_h
FAIRNESS   ! S_Fg | S_g

/*end with a SPEC statement*/
SPEC       !(S_f & EG true)
```

Box III.

3.8. *Air traffic control example: specifications*

Specifications are the temporal logic formulas we supply to a model checker to describe the desirable behaviors the system we are checking must have. It is usually helpful to keep in mind the goal of specifications while writing them. In general, the purpose of the system model to the model checker is to answer the question "What does the system do?" The purpose of the set of specifications is to answer the question "What *should* the system do?" We are basically saying to the model checker, "Here's how the system works; please verify that it has the list of emergent behaviors described by our specification set." To that end, we design specifications such that each specification describes a particular desirable behavior and the entire set of specifications describes a coherent pattern of behavior.

We have discussed earlier in this section issues relating to the set of specifications, such as the necessity of checking that each specification, and the set of specifications as a whole, are satisfiable and checking that each specification, and each part of each specification, contributes to the behavioral description (vacuity checking). Another concern when authoring a set of specifications is that of *coverage*, or ensuring that behaviors across the entire system state space are described by the set of specifications. There are several metrics to determine coverage of the model by the set of specifications but that is a large and active area of research beyond the scope of this paper. For this example, we simply check that we have described different types of behaviors, specifying both safety and liveness conditions, and adequately utilized the system variables.

We have already written some specifications for our automated air traffic control architecture in Section 3.5. Here they are again, as a set:

- *Liveness: "Every conflict is addressed"*
  $\Box(\neg TSAFE\_clear \rightarrow \Diamond TSAFE\_command)$
  Or, we can choose to specify a stricter version of this property:
- *Safety: "Every conflict is addressed immediately (i.e in one time step)"*
  $\Box(\neg TSAFE\_clear \rightarrow \mathcal{X}(TSAFE\_command))$
- *Safety: "The system will never issue conflicting commands"*
  $\Box(\neg(AR\_command \wedge TSAFE\_command))$
- *Liveness: "All conflicts are eventually resolved"*
  $\Box(\neg TSAFE\_clear \rightarrow \Diamond TSAFE\_clear)$
- *Liveness: "All controller requests are eventually addressed"*
  $\Box(controller\_request \rightarrow \Diamond\neg controller\_request)$
- *Liveness: "All aircraft requests are eventually addressed"*
  $\Box(aircraft\_request \rightarrow \Diamond\neg aircraft\_request)$

We again use the model checker NuSMV to illustrate the next step in the symbolic model-checking process. We add our specifications to the end of the file in which we have specified the system model, each prefaced with the label LTLSPEC. (See Box IV.)

We will take the first specification in our set, $\Box(\neg TSAFE\_clear \rightarrow \Diamond TSAFE\_command)$, as an example of the processing of specifications under the hood in a symbolic model checker. Following the steps enumerated in Section 3.7, NuSMV constructs internally a symbolic automaton that looks something like the one given in Box V.

4. Representing the combined system and property using BDDs

The basis for symbolic model checking is the realization that both the system model and the specification property can be represented symbolically, using Boolean equations,

```
LTLSPEC (G ((! TSAFE_clear) -> (F TSAFE_command)))
LTLSPEC (G ((! TSAFE_clear) -> (X TSAFE_command)))
LTLSPEC (G (! AR_command & TSAFE_command))
LTLSPEC (G ((! TSAFE_clear) -> (F TSAFE_clear)))
LTLSPEC (G (controller_request -> (F (! controller_request))))
LTLSPEC (G (aircraft_request -> (F (! aircraft_request))))
```

Box IV.

```
MODULE main

  --formula: (G ((TSAFE_clear ) | (F (TSAFE_command ))))
  VAR
    -- declare a boolean variable for each atomic proposition in f
  TSAFE_clear : boolean;
  TSAFE_command : boolean;

  VAR
    -- and declare a new variable EL_X_g for each formula (X g) in el_list
    -- generated by a primary operator X, U, R, G, or F
  EL_X_G__TSAFE_clear_OR_F_TSAFE_command : boolean;
  EL_X_F_TSAFE_command : boolean;

  DEFINE
    -- for each S_h in the characteristic function, put a line here
  S_G__TSAFE_clear_OR_F_TSAFE_command := ((TSAFE_clear ) | (S_F_TSAFE_command))
    & EL_X_G__TSAFE_clear_OR_F_TSAFE_command;
  S_F_TSAFE_command := TSAFE_command | EL_X_F_TSAFE_command;

  TRANS
    -- for each (X g) in el_list, generate a line here
  ( EL_X_G__TSAFE_clear_OR_F_TSAFE_command = (next(S_G__TSAFE_clear_OR_F_TSAFE_command) )) &
  ( EL_X_F_TSAFE_command = (next(S_F_TSAFE_command) ))

  JUSTICE   (!S_F_TSAFE_command | TSAFE_command)

  SPEC       !(S_G__TSAFE_clear_OR_F_TSAFE_command & EG TRUE)
```

Box V.

rather than explicitly, using automata. Furthermore, they can be manipulated efficiently using operations over Binary Decision Diagrams (BDDs). Symbolic representation and manipulation of the intermediate products of model checking leads to much more succinct structures needing to be stored in computer memory, thereby directly combating the state explosion problem. Certainly there are pathological constructions of systems that do not benefit substantially from this space reduction, but symbolic model checking increases the scalability of model checking for a great many common systems, especially those with regular structure such as hardware circuits [53,27].

In this section, we will demonstrate how the automaton $A_{M,\neg\varphi}$ can easily grow to a size that cannot be practically represented in memory and present the most popular alternative representation, which uses BDDs to mitigate this problem. The technique of *symbolic model checking* was introduced by McMillan [25] in 1992 specifically as a response to the state explosion problem. The power of symbolic model checking derives from using equations describing sets of states and sets of transitions to implicitly define the state space, thereby using significantly less memory than explicit representations that may enumerate the entire state space. Most symbolic model checkers utilize BDDs to accomplish this task, though other approaches are possible [133].

Boolean equations of the sort used to describe the transition systems plied by symbolic model checking are commonly represented in a variety of ways, such as truth tables and Binary Decision Trees (BDT), which are exponential in the size of the formulas they represent.

Definition 8. A **Binary Decision Tree (BDT)** is a rooted, directed, acyclic graph (DAG) with vertices labeled by the n variables of the corresponding Boolean formula. Each variable node has exactly two children representing the two possible assignments to that variable, 0 and 1, labeled *low* and *high*, respectively. (Note that, except for the root, each node has exactly one parent.) A path through the BDT represents a valuation of the represented function given an assignment of values to the n variables; it starts at the root, follows the outgoing edge matching the assignment to the variable in the current vertex, and terminates in a vertex labeled either 0 or 1, corresponding to the value of the formula for that variable valuation. The BDT representing a formula over n variables has $2^n - 1$ variable vertices plus 2^n terminal vertices in $\{0, 1\}$, for a total size of $2^{n+1} - 1$ nodes.

A BDD is a refinement of a BDT. While in the worst case, a BDD is only roughly half the size[18] of its equivalent BDT, usually there is more significant improvement gained by the following construction.

[18] Eliminating duplicate terminal nodes from a BDT of size $2^{n+1} - 1$ with 2^n terminal vertices leaves a BDD of size $2^n + 1$ nodes.

Definition 9. A **Binary Decision Diagram (BDD)** is a rooted, directed, acyclic graph (DAG) with internal vertices labeled by some sufficient subset of the n variables of the corresponding Boolean formula and exactly two terminal vertices, labeled 0 and 1. Each variable node has exactly two children representing the two possible assignments to that variable, 0 and 1, labeled *low* and *high*, respectively. (Note that, except for the root, each node may have any number of parents.) A path through the BDD represents a valuation of the represented formula given an assignment of values to the n variables; it starts at the root, follows the outgoing edge matching the assignment to the variable in the current vertex, and terminates in the vertex corresponding to the value of the formula for that variable valuation.

A BDD is considered *ordered* if it follows a given total order of variables from root to leaves. Note that not all variables may appear in the tree or along any path in the tree; a variable only appears along a path if its value influences the value of the formula, given the assignments to previous variables along the path. If we are reasoning over multiple ordered BDDs (OBDDs), they are presumed to all follow the same variable ordering. A BDD is *reduced* if it contains no vertex v such that $low(v) = high(v)$ and no pair of vertices v_1 and v_2 such that the subgraphs rooted by v_1 and v_2 are isomorphic. That is, if there is a one-to-one function f such that for any vertex v'_1 in the subgraph rooted at v_1, for some vertex v'_2 in the subgraph rooted at v_2, $f(v'_1) = v'_2$ implies that either v'_1 and v'_2 are both terminal vertices with the same value, or v'_1 and v'_2 are both non-terminal vertices with the same variable label and the property that $f(low(v'_1)) = low(v'_2)$ and $f(high(v'_1)) = high(v'_2)$, then either v_1 or v_2 will be eliminated. Note that each path realizes a unique set of variable assignments; each variable valuation corresponds to exactly one path. Furthermore, every vertex in the graph lies along at least one path; no part of the graph is unreachable.

In practice, any non-reduced BDD can be easily reduced by applying two reduction rules. The *redundancy elimination rule* removes any non-terminal node v_1 where $low(v_1) = high(v_1) = v_2$ and connects all incoming edges of v_1 directly to v_2 instead. The *isomorphism elimination rule* removes vertex v_1 wherever there exists a vertex v_2 such that $var(v_1) = var(v_2)$, $low(v_1) = low(v_2)$, and $high(v_1) = high(v_2)$ and redirects all incoming edges of v_1 directly to v_2. (We use $var(v)$ to denote the variable labeling vertex v.) Note that v_1 and v_2 may be either terminal or non-terminal vertices; this rule eliminates duplicate terminals as well as isomorphic subgraphs. A non-reduced BDD can be reduced in a single pass of the tree by careful application of these two rules in a bottom-up fashion. An example of reducing a Binary Decision Tree into a BDD is given in Fig. 9. All three subfigures represent the formula $x_1 \vee x_2 \vee x_3$ with varying degrees of succinctness as each of the two reduction rules is applied.

When BDDs are both reduced and ordered (also called ROBDDs), they have many highly desirable properties. In this paper, we consider all BDDs to be both reduced and ordered; the algorithms used for symbolic model checking, and for efficient BDD manipulation in general, all maintain these properties. Reduced ordered BDDs are closed under the class of symbolic operations, as outlined in Section 4.3. These include APPLY and SATISFY-ONE, which form the basis for the symbolic model-checking algorithm.

Using BDDs to reason about Boolean formulas representing the state space offers many advantages over explicit automata representations:

- *Complexity (both time and space):* BDDs provide reasonably small representations for a large class of the most interesting, and most commonly-encountered, Boolean functions. For example, all symmetric functions, including the popular even and odd parity functions, are easily represented by BDDs where the number of vertices grows at most with the square of the number of arguments to the function. Moreover, since our BDDs are reduced and ordered, they have the property of minimality. This means if B is a BDD representing a function f, then for every BDD B' that also represents f and has the same variable ordering as B, $size(B) \leq size(B')$. Even better, we can frequently avoid enumerating the entire state space. Specifically, we can express tasks in several problem domains entirely in terms of graph algorithms over BDDs, describe automata in terms of sets of states and transitions, or use other techniques along those lines to express only relevant portions of the state space. Since all of the algorithms used for symbolic analysis of BDDs have complexities polynomial in the sizes of the input BDDs, the total computation remains tractable as long as the sizes of the input BDDs are reasonable. The sum of these time-and-space advantages allows us to use BDDs for solving problems that may scale beyond the limits of more traditional techniques such as case analysis or combinatorial search.
- *Canonicity:* There is a unique, canonical BDD representation for every Boolean function, given a total variable ordering. This property yields very simple algorithms for the most common BDD operations. Testing BDDs for equivalence just involves testing whether their graphs match exactly. Testing for satisfiability reduces to comparing the BDD graph to that of the constant function 0. Similarly, a function is a tautology iff its BDD is the constant function 1. Testing whether a function is independent of a variable x involves a simple check of whether its BDD contains any vertices labeled by x. Note that this is an important advantage over explicit automata representations since there is no canonical automaton representation for any class of automata.
- *Efficiency:* Besides the benefits that directly follow from canonicity, BDD representation enables efficient algorithms for many basic operations, including intersection, complementation, comparison, subtraction, projection, and testing for implication. The time complexity of any single operation is bounded by the product of the graph sizes of the functions being operated on or, for single-BDD operations like complementation, proportional to the size of the single function graph. In other words, the performance of directly manipulating BDDs in a sequence of operations degrades slowly, if at all.
- *Simplicity:* The BDD data structure and the set of algorithms for manipulating BDDs are conceptually and implementationally simple.
- *Generality:* BDDs can be used for reasoning about all kinds of finite systems, basically any problem that can be stated in terms of Boolean functions. They are not tied to any particular family of automata; they are just as useful for LTL symbolic model checking as they are for the extensions of LTL described in Section 3.2.4, for example.

The notion of using Binary Decision Diagrams to reason about Boolean functions was first introduced by Lee in 1959 [134] and later popularized by Akers who evinced the utility of BDDs for reasoning about large digital systems [135]. Fortune, Hopcroft, and Schmidt [136] restricted the ordering of the decision variables in the vertices of the graph, creating the canonical form of Ordered Binary Decision Diagrams (OBDDs) (though they called them *B*-schemes), noting the significance of variable ordering on graph size, and showing the ease of testing for functional equivalence. Bryant, who coined the term OBDDs, ameliorated ordered BDDs to symbolic analysis by demonstrating efficient algorithms for combining two functions with a binary operation and composing two functions [137,138].

4.1. *Combining the system and property representations*

We directly construct the product, $A_{M,\neg\varphi}$, of the system model automaton M and the automaton representing the complemented specification $A_{\neg\varphi}$. This construction logically follows from the realization that the product of these two automata constitutes the intersection of the languages $\mathscr{L}(M)$ and $\mathscr{L}(A_{\neg\varphi})$ [139]. Recall from Section 3.2.4 that Büchi automata are automata that accept exactly the class of ω-regular languages. Furthermore, the class of ω-regular languages corresponds exactly to the languages that can be described by ω-regular expressions and is closed under the operations of union, intersection, and complementation [140]. Therefore, we can construct a Büchi automaton $A_{M,\neg\varphi}$ such that $\mathscr{L}(A_{M,\neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$ since $\mathscr{L}(A_{M,\neg\varphi})$ is an ω-regular language.

Intuitively, $A_{M,\neg\varphi}$ simulates running the two multiplicand automata in parallel, also called the *synchronous parallel composition* of M and $A_{\neg\varphi}$. Executing these two automata simultaneously allows us to use $A_{\neg\varphi}$ to continuously monitor the behavior of M and judge whether M satisfies the desired property φ at all times.[19]

The states of $A_{M,\neg\varphi}$ are pairs of states, one each from M and $A_{\neg\varphi}$, plus a track label $t \in \{0, 1\}$. (For simplicity, we call a state in $M q_M$ and a state in $A_{\neg\varphi} q_\varphi$.) A run of $A_{M,\neg\varphi}$ starts in the pair of the two start states in track 0: $(q_{0M}, q_{0_\varphi}, 0)$. Upon reading a character $\sigma \in \Sigma$, $A_{M,\neg\varphi}$ transitions from the current state, $q = (q_M, q_\varphi, t)$, to a next state, $q' = (q'_M, q'_\varphi, t)$, wherever M transitions from the M-component of q, q_M, to the M-component of q', q'_M, upon reading σ and $A_{\neg\varphi}$ also transitions from its associated component, q_φ, to q'_φ upon reading σ. The value of t remains the same unless either $t = 0$ and $q_M \in F_M$ or $t = 1$ and $q_\varphi \in F_\varphi$, in which case the t-bit flips. In essence, the two tracks ensure $A_{M,\neg\varphi}$ starts in track 0, passes through a final state in M, switches to track 1, passes through a final state in $A_{\neg\varphi}$, switches back to track 0, and repeats the pattern. We assign the set of final states in $A_{M,\neg\varphi}$ to essentially match those of M since the path of any run only returns to track 0 if it has previously passed through an accepting state of M in track 0 and an accepting state of $A_{\neg\varphi}$ in track 1. Since the accepting condition for a Büchi automaton necessitates visiting the set of final states infinitely often, visiting infinitely often a state whose M-component is in F_M while in track 0 suffices. This construction is necessarily more complex than the straight *Cartesian product* [139] of the two automata, $M \times A_{\neg\varphi}$, since we must allow for runs that pass infinitely often through final states in both M and $A_{\neg\varphi}$, but not necessarily at the same time. (For an illustrative example, see [110].) Note that since q_{0_φ} is defined as the state(s) labeled by $\neg\varphi$, the initial state(s) our product automaton, $A_{M,\neg\varphi}$ must also be labeled by $\neg\varphi$.

We presume that the alphabets of both automata are the same, a reasonable assumption considering that the formula $\neg\varphi$ describes a behavior of the system M. However, our construction is unchanged by defining the alphabet Σ of $A_{M,\neg\varphi}$ to be the union of the alphabets of M and $A_{\neg\varphi}$ if those alphabets differ.

Formally, we define $A_{M,\neg\varphi}$ as follows.

Definition 10. Let $M = (Q_M, \Sigma, \delta_M, q_{0M}, F_M)$ and $A_{\neg\varphi} = (Q_\varphi, \Sigma, \delta_\varphi, q_{0_\varphi}, F_\varphi)$. The intersection automaton $A_{M,\neg\varphi}$ with the property that $\mathscr{L}(A_{M,\neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$ is the automaton defined by the quintuple $A_{M,\neg\varphi} = (\hat{Q}, \Sigma, \hat{\delta}, (q_{0M}, q_{0_\varphi}, 0), \hat{F})$ where:

- State set $\hat{Q} = Q_M \times Q_\varphi \times \{0, 1\}$ consists of triples $(q_{iM}, q_{j_\varphi}, t)$.
- Σ is the shared alphabet.
- Transition relation $\hat{\delta}$ is defined such that $A_{M,\neg\varphi}$ is in state $(q_{iM}, q_{j_\varphi}, t)$ whenever M is in state q_{iM} and $A_{\neg\varphi}$ is in state q_{j_φ}. For any $\sigma \in \Sigma$,

$$\hat{\delta}((q_{iM}, q_{j_\varphi}, t), \sigma) = (q_{kM}, q_{l_\varphi}, t),$$

whenever

$$\delta_M(q_{iM}, \sigma) = q_{kM}$$

and

$$\delta_\varphi(q_{j_\varphi}, \sigma) = q_{l_\varphi}$$

unless

$$t = 0 \quad \text{and} \quad q_{iM} \in F_M,$$

or

$$t = 1 \quad \text{and} \quad q_{j_\varphi} \in F_\varphi$$

in which case

$$\hat{\delta}((q_{iM}, q_{j_\varphi}, t), \sigma) = (q_{kM}, q_{l_\varphi}, \bar{t}).$$

- The initial state is $(q_{0M}, q_{0_\varphi}, 0)$.
- The set of final states $\hat{F}$ is the set of all states $(q_{iM}, q_{j_\varphi}, 0)$ such that $q_{iM} \in F_M$ and $q_{j_\varphi} \in Q_\varphi$.

Theorem 5. *Büchi automata are closed under complementation.*

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a Büchi automaton that accepts the language L. The construction of the Büchi automaton $\overline{A}$ that accepts the complement language $\overline{L}$ is rather involved. It does not suffice to simply complement the accepting set F (i.e. let $A' = (Q, \Sigma, \delta, q_0, Q - F)$) because a run not passing through any state in F infinitely often is different from a run passing through some state in $Q - F$ infinitely often. For example, an accepting run of A on the word w might pass through both states in F and in $Q - F$ infinitely often, so both A and A' would accept w. Thus, $\mathscr{L}(A') \neq \Sigma^\omega - \mathscr{L}(A)$. The complex construction proving Theorem 5 is given in [108], which yields a doubly exponential construction for $\overline{A}$ and

[19] Systems can also be composed asynchronously, where δ is defined such that it is possible for either M or φ to progress while the other does not transition and the values of any system variable specific to the non-transitioning system are preserved. However, this type of composition does not further the model-checking algorithm presented here.

in [83], which reduces the construction to singly exponential with a quadratic exponent. Specifically, for Büchi automaton A, over the alphabet Σ, there is a Büchi automaton $\overline{A}$ with $2^{\mathcal{O}(|Q|\log|Q|)}$ states such that $\mathscr{L}(\overline{A}) = \Sigma^\omega - \mathscr{L}(A)$ [129]. Michel proved that this is an optimal bound [141].

As an important aside, since Büchi automata are not closed under determinization and nondeterministic Büchi automata are more expressive than deterministic Büchi automata, the complement automaton $\overline{A}$ produced by these algorithms for some deterministic Büchi automaton A may be a nondeterministic Büchi automaton that cannot be determinized. Efficient methods for constructing and reasoning about complemented Büchi automata remain an area of interest. For example, we can check A for nonuniversality (i.e. $\mathscr{L}(A) \neq \Sigma^\omega$) in polynomial space by constructing $\overline{A}$ on the fly and checking for nonemptiness [110].

Theorem 6. *$\mathscr{L}(A_{M,\neg\varphi})$ is an ω-regular language and $A_{M,\neg\varphi}$ is a Büchi automaton.*

Proof of Theorem 6. Recall that $L_M = \mathscr{L}(M)$ is an ω-regular language since it is the language accepted by a Büchi automaton and that $L_\varphi = \mathscr{L}(A_{\neg\varphi})$ is also an ω-regular language since $\neg\varphi$ is an LTL formula, which describes a ($*$-free) ω-regular expression. We know that $\mathscr{L}(A_{M,\neg\varphi}) = L_M \cap L_\varphi$ is an ω-regular language, and, therefore, that $A_{M,\neg\varphi}$ is a Büchi automaton because ω-regular languages are closed under intersection. First, we offer proof of closure under union; closure under intersection follows from closure under union and complementation.

By definition of ω-regular languages, there exist ω-regular expressions r_M and r_φ that describe the languages L_M and L_φ, respectively. That is, $L_M = \mathscr{L}(r_M)$ and $L_\varphi = \mathscr{L}(r_\varphi)$. By definition of ω-regular expressions, $r_M + r_\varphi$ is an ω-regular expression denoting the language $L_M \cup L_\varphi$, which demonstrates closure under union.

Closure under intersection now follows from DeMorgan's Law:

$$L_M \cap L_\varphi = \overline{\overline{L_M} \cup \overline{L_\varphi}}.$$

By closure under complementation, $\overline{L_M}$ and $\overline{L_\varphi}$ are both ω-regular languages. By closure under union, $\overline{L_M} \cup \overline{L_\varphi}$ is an ω-regular language. Again by closure under complementation,

$$\overline{\overline{L_M} \cup \overline{L_\varphi}} = L_M \cap L_\varphi$$

is an ω-regular language. By definition of ω-regular languages, it follows that $A_{M,\neg\varphi}$ is a Büchi automaton. □

Theorem 7. *Let $L_M = \mathscr{L}(M)$ and $L_\varphi = \mathscr{L}(A_{\neg\varphi})$. Given an infinite word $w = w_0, w_1, w_2, \ldots \in \Sigma$, $w \in L_M \cap L_\varphi$ if and only if it is accepted by $A_{M,\neg\varphi}$.*

Proof of Theorem 7.

If-direction: $w \in L_M \cap L_\varphi \rightarrow w$ is accepted by $A_{M,\neg\varphi}$.

An infinite word w is in the language $L_M \cap L_\varphi$ if $w \in L_M$ and $w \in L_\varphi$. By definition, $w \in L_M$ iff w represents an accepting execution of the Büchi automaton M. Symmetrically, $w \in L_\varphi$ iff w is accepted by the automaton $A_{\neg\varphi}$ corresponding to the LTL formula $\neg\varphi$. Then we know that $\delta^\omega_M(q_{0M}, w)$ transitions through some final state $q_{iM} \in F_M$ infinitely often and that $\delta^\omega_\varphi(q_{0_\varphi}, w)$ similarly transitions through some accepting state $q_{j_\varphi} \in F_\varphi$ infinitely often. By definition of $\hat{\delta}$, $A_{M,\neg\varphi}$ is in state (q_{iM}, q_{j_φ}) exclusively when M is in state q_{iM} and $A_{\neg\varphi}$ is in state q_{j_φ} upon reading the same word. That is, for any w,

$$\hat{\delta}^\omega((q_{iM}, q_{j_\varphi}), w) = (q_{kM}, q_{l_\varphi}),$$

whenever

$$\delta^\omega_M(q_{iM}, w) = q_{kM}$$

and

$$\delta^\omega_\varphi(q_{j_\varphi}, w) = q_{l_\varphi}.$$

Since whenever $q_{iM} \in F_M$ and $q_{j_\varphi} \in F_\varphi$, $(q_{iM}, q_{j_\varphi}) \in \hat{F}$, w is accepted by $A_{M,\neg\varphi}$.

Only-if direction: w is accepted by $A_{M,\neg\varphi} \rightarrow w \in L_M \cap L_\varphi$.

By definition, an infinite word w is accepted by $A_{M,\neg\varphi}$ if the run over w in $A_{M,\neg\varphi}$ visits at least one state in F infinitely often. That is, $\hat{\delta}^\omega((q_{0M}, q_{0_\varphi}), w)$ transitions infinitely often through some state $(q_{iM}, q_{j_\varphi}) \in \hat{F}$. By construction, for any w, $\hat{\delta}^\omega((q_{0M}, q_{0_\varphi}), w) = (\delta^\omega_M(q_{0M}, w), \delta^\omega_\varphi(q_{0_\varphi}, w))$. By definition of $\hat{F}$, (q_{iM}, q_{j_φ}) iff $q_{iM} \in F_M$ and $q_{j_\varphi} \in F_\varphi$. Therefore, $\delta^\omega_M(q_{0M}, w)$ transitions through $q_{iM} \in F_M$ infinitely often and $w \in L_M$. Symmetrically, $\delta^\omega_\varphi(q_{0_\varphi}, w)$ transitions through $q_{j_\varphi} \in F_\varphi$ infinitely often and $w \in L_\varphi$. Ergo, $w \in L_M \cap L_\varphi$. □

The product automaton $A_{M,\neg\varphi}$ has size $\mathcal{O}(|M| \times |A_{\neg\varphi}|)$ in the worst case. (Though the product can be much smaller; in the best case, it has size 0 [8].) Due to this size blow-up, representing the automaton as succinctly as possible becomes vital to palliate the state explosion problem and ensure $A_{M,\neg\varphi}$ can be stored in computer memory. It is easy to see that even small differences in the sizes of both M and $A_{\neg\varphi}$ can have a significant impact on our ability to store, and to reason about, $A_{M,\neg\varphi}$. We must choose a representation of $A_{M,\neg\varphi}$ that minimizes the storage memory requirement while maximizing the efficiency of the nonemptiness check. We accomplish this task by representing $A_{M,\neg\varphi}$ using BDDs.

4.2. *Representing automata using BDDs*

Recall that $A_{M,\neg\varphi}$ is a Büchi Automaton with finite set of states Q, defined by assignments to the variables in the alphabet Σ, and transition relation $\delta : Q \times \Sigma \rightarrow Q$. We consider state $q_i \in Q$ as a tuple $\langle q_i, \sigma_0, \sigma_1, \ldots\rangle$ containing the name of the state and the assignments to each of the system variables, in order, in that state. In this paper, we presume that the set of system variables consists of Boolean-valued atomic propositions, which can each be represented by a single bit. However, this representation is easily extended to other data types by utilizing more bits to represent each system variable. For example, an integer variable with a range of $[0, 255]$ would be represented using 8 bits. (Unbounded variables cannot be encoded in this fashion as they cause the representation of $A_{M,\neg\varphi}$ to have an infinite number of states and thereby to be immune to the model-checking algorithm presented here. Infinite state model checking requires the use of alternative techniques, e.g. [142]). Choosing a Boolean encoding for an automaton is a bit of an art; there are several encoding variations to choose from. When reasoning about a single automaton, for instance, there may be an advantage to encoding the state numbers in the representation along with the values of the variables in each state. In our air

traffic control example, we have 7 states and thus need 4 bits to represent this range of state labels. The Boolean tuple representation of State 6, where all 5 system variables are false, would be ⟨011000000⟩. (The first 4 bits are the binary representation of the label 6 and the last 5 bits give the 5 false values of the system variables.) In practice, the state label is optional and can be included or not in the Boolean tuple for optimization purposes. This sort of encoding with state labels is not advantageous when reasoning about combinations of automata, as in symbolic model checking, as the state labels become meaningless. Also, we would rather avoid this kind of explicit enumeration of the states in the BDD representation since this does not offer a space-saving advantage over explicit-state model checking.

We represent symbolic automata using BDDs by encoding the transition relations directly. Basically, a transition is a pair of states: an origin state and a terminal state. A simple way to represent a transition from state q_i to state q_{i+1} is via the tuple $\langle q_i, q_{i_1} \rangle$, which is encoded using twice the number of bits as a single state. For instance, the transition from State 6 to State 7 in our air traffic control example automaton, presuming the variable ordering {*AR_command*, *TSAFE_command*, *controller_request*, *aircraft_request*, *TSAFE_clear*}, would be ⟨0000001000⟩. (The first 5 bits give the 5 false values of the system variables in State 6; the next 5 bits represent the values of the system variables in State 7, with the value of *TSAFE_command* changed to *True*.)

Our BDD is constructed using $2|\Sigma|$ variables: two copies of each variable represent its value in the originating state and in the target state. The variables $\sigma_1, \sigma_2, \ldots, \sigma_n$ represent the values of the atomic propositions in the current state and the primed variables $\sigma'_1, \sigma'_2, \ldots, \sigma'_n$ represent the values of the atomic propositions in the next state. Finding an optimal BDD variable ordering is NP-complete and, for any constant $c > 1$, it is NP-hard to compute a variable ordering to make a BDD at most c times larger than optimal [169]. Though finding an optimal variable ordering is a tall order, a good rule of thumb is to group the most closely-related variables together in the ordering and some order involving interleaving the current and next time variants of the variable set together (as in $\sigma_1, \sigma'_1, \sigma_2, \sigma'_2, \ldots, \sigma_n, \sigma'_n$) has been shown to have some nice advantages for model checking [72].

Alternative encodings include breaking a larger BDD representation into combinations of smaller BDDs to take advantage of natural logical separations and minimize the size of intermediate constructions [138]. Variations on the basic BDD structure can also be helpful. For example, shared BDDs, or BDDs with multiple root nodes, one for each function represented, save memory by overlapping BDD representations and adding a pair of hash tables for bookkeeping [143]. Oppositely, BDDs with expanded sets of terminal nodes (i.e. terminals other than 0 and 1), such as Multi-Terminal BDDs, offer unique advantages, especially for extensions of model checking that deal with uncertainty [144,145]. Examining memory-saving refinements on the basic BDD structure that sustain or enhance the efficient manipulation of automata by reasoning over sets of states and sets of transitions is an ongoing area of research.

4.2.1. *The impact of variable ordering*

The first decision we make when forming a BDD is the variable ordering. This is also the most important decision since the size of the graph representing any particular function may be highly sensitive to the chosen variable ordering. Many common functions, such as integer addition, have BDD representations that are linear for the best variable orderings and exponential for the worst. Unfortunately, computing an optimal ordering is an NP-complete problem. Even if we take the time to compute the optimal ordering, not all functions have reasonably-sized representations. For some systems, even a reasonably-sized BDD representation is too large to reason about practically, and for others, the BDD representation will grow exponentially with the size of the input function regardless of variable ordering [137]. Therefore, the best course we can take is to utilize a set of comparatively simple heuristics to choose an adequate ordering, avoiding orderings that would cause the BDD to grow exponentially whenever possible. A classic example of a function with both linear and exponential BDD representations, depending on variable ordering, is displayed in Fig. 10.

4.3. *BDD operations*

We introduce the algorithms underlying the symbolic model-checking procedure, as defined by Bryant [137] and reviewed by Andersen [146]. The full model-checking algorithm presented in Section 5 builds upon these basic operations.

If φ is purely propositional (i.e. contains no temporal operators) then we can easily construct a (reduced) BDD directly from $\neg\varphi$ using a straightforward algorithm like BUILD, which simply loops through the variable ordering, recursively constructing each variable's low and high subtrees[20] [146]. (See Box VI.) Otherwise, we translate $\neg\varphi$ into a symbolic automaton and then encode the result, $A_{\neg\varphi}$ as a BDD, as described in Section 4.2.

Once we have built BDDs representing each of M and $A_{\neg\varphi}$, we need to compute their product, as described in Section 4.1. Let Σ_{BDD} be the set of BDD variables encoding Σ. The application of all binary operators over BDDs, including conjunction, disjunction, union, intersection, complementation (via xor with 1), and testing for implication, is implemented in a streamlined fashion utilizing a singular universal function. (See Box VII.) APPLY takes as input a binary operator and BDDs representing two functions f_1 and f_2, both over the alphabet $\Sigma = (\sigma_1, \ldots, \sigma_n)$ and produces a reduced graph representing the function $f_1 < op > f_2$ defined as:

$$[f_1 < op > f_2](\sigma_1, \ldots, \sigma_n) = f_1(\sigma_1, \ldots, \sigma_n) < op > f_2(\sigma_1, \ldots, \sigma_n).$$

APPLY begins by comparing the root vertices of the two BDDs and proceeds downward. There are four cases. If vertices v_1 and v_2 are both terminal vertices, the result is simply the terminal vertex representing the result of $value(v_1) < op > value(v_2)$. If the variable labels of v_1 and v_2 are the same, we create a vertex in the result BDD with that variable label and operate on the pair of low subtrees, $low(v_1)$ and $low(v_2)$, and the pair of high subtrees, $high(v_1)$

[20] Unfortunately, this simple algorithm requires exponential (2^n) time.

```
algorithm BUILD(¬φ : propositional logic formula) : BDD // Build a reduced BDD for ¬φ

i : integer  // i indexes the variable order
n : integer  // n is the number of Boolean variables in Σ_BDD

//Recursive routine to implement BUILD
function BUILD-STEP(¬φ, i): vertex
begin
  if (i > n) then //if there are no variables left to add
    if (¬φ == false) then return 0
    else return 1
  //Recursive calls to build sub-trees
  else v0 = BUILD-STEP(¬φ[σ_i = 0], i + 1) //compute the low subtree
       v1 = BUILD-STEP(¬φ[σ_i = 1], i + 1) //compute the high subtree
       //new_vertex creates a new vertex only if v0 and v1 are different
       //  and the σ-labeled node with them as children does not already exist
       new_vertex(σ_i, v0, v1) //make/connect the new node
  return
end

return (BDD = BUILD-STEP(¬φ, 0))
```

Box VI.

```
algorithm APPLY(op : operator; v1, v2 : vertex): vertex // Evaluate v1 op v2

u : vertex  // u is the root vertex of the BDD representing v1 op v2
init(T)     // T is a hash table;
            // T(i, j) is either ∅ or the earlier computed result of Apply-step(i, j)

//Recursive routine to implement APPLY
function APPLY-STEP(v1, v2 : vertex): vertex
begin
  if (T(v1, v2) ≠ ∅) then return T(v1, v2)  //this vertex pair has already been evaluated
  else if (v1 ∈ {0,1} and v2 ∈ {0,1}) then //both v1 and v2 are terminal vertices
    u = v1 op v2                           //evaluate the simple Boolean expression (This is the base case.)
  else if (var(v1) == var(v2)) then        //if v1 and v2 are rooted at the same variable
                                           //progress down both trees to the next variable in the ordering
    u = new_vertex(var(v1), APPLY-STEP(low(v1), low(v2)), APPLY-STEP(high(v1), high(v2)))
  else if (var(v1) < var(v2)) then         //if v1 is first in the variable ordering/nearer to root
                                           //compare the children of v1 to v2
    u = new_vertex(var(v1), APPLY-STEP(low(v1), v2), APPLY-STEP(high(v1), v2))
  else if (var(v1) > var(v2)) then         //if v2 is first in the variable ordering/nearer to root
                                           //compare v2 to the children of v2
    u = new_vertex(var(v2), APPLY-STEP(v1, low(v2)), APPLY-STEP(v1, high(v2)))

  T(v1, v2) = u                            //store the result in T
  return u
end

u = APPLY-STEP(v1, v2)
return u
```

Box VII.

and *high*(v_2), to create the low and high subtrees of our new vertex, respectively. Otherwise, one of the variables, *var*(v_1) or *var*(v_2) is before the other in the total variable ordering. Note that this later vertex may or may not be a terminal vertex; either way, the algorithm is the same. If v_1 is the earlier vertex, then the function represented by the subtree with root v_2 is independent of the variable *var*(v_1). (If this were not true, we would encounter the variable in both graphs.) In this case, we create a vertex in the result BDD labeled *var*(v_1) and recur on the pair of subtrees {*low*(v_1), v_2} to construct this new node's low subtree and on the pair of subtrees {*high*(v_1), v_2} to construct this new node's high subtree. The last case, where *var*(v_2) occurs before *var*(v_1) in the total variable ordering, is symmetric. We presume `new_vertex()` is some function that creates a new vertex iff no isometric vertex exists, thus maintaining the reduced nature of the BDD under construction. Bryant [137] and Wegener and Sieling [147] describe implementation optimizations that yield a time complexity for this algorithm of $\mathcal{O}(|G_1||G_2|)$ where $|G_1|$ and $|G_2|$ are the sizes of the two graphs being operated upon.

Recall that finding a counterexample trace equates to finding a word in the language $\mathscr{L}(A_{M,\neg\varphi})$, which is essentially a satisfying assignment to the variables in Σ through time. The model checker uses a BDD-based fixpoint algorithm to find a *fair path* in the model-automaton product [24]. The basic algorithm underlying this process is Bryant's SATISFY-ONE [137]. (See Box VII.) This algorithm is a simple recognition

```
algorithm SATISFY-ONE(v: vertex, x: var_array) : Boolean
  //Find any satisfying assignment of the BDD rooted at v
  //  Return the counterexample in x

  if (v.value == 0) then return false //unsatisfiable
  if (v.value == 1) then return true  //satisfiable

  i = v.level //i is the level of vertex v, also the place in the variable order
  x[i] = 0    //guess 0 for this vertex
  if SATISFY-ONE(v.low, x) then return true //satisfiable
  x[i] = 1    //backtrack if x[i] = 0 results in false and guess 1 instead
  return SATISFY-ONE(v.high, x) //we know x[i] = 1 will be true
```

Box VIII.

of the BDD principle that every non-terminal vertex has the terminal vertex labeled 1 as a descendant. Thus, a classic depth-first search with backtracking upon visiting the 0 terminal is guaranteed to find a satisfying path from the root to the terminal 1 in linear time[21]. Starting from the root, SATISFY-ONE arbitrarily guesses the low branch of each non-terminal vertex first, and stores a satisfying assignment (aka counterexample trace) of length $n = |\Sigma_{\text{BDD}}|$, in an array as it traverses the graph. It returns false if the function is unsatisfiable, and true otherwise. (See Box VIII.)

Note that the reason we check for the existence of a single satisfying set and not for the whole set $S_{M,\neg\varphi}$ of satisfying sets is because enumerating the entire set requires time proportional to the length of the counterexamples times the number of elements in $S_{M,\neg\varphi}$. (For a propositional formula with $n = |\Sigma_{\text{BDD}}|$, the time complexity is $\mathcal{O}(n \cdot |S_{M,\neg\varphi}|)$.) Considering the counterexample traces may be quite lengthy (i.e. n may be large), and there may be many of them, this is a highly inefficient check, unlikely to be performed in any reasonable timeframe. Furthermore, the utility of performing such a check is questionable: a single bug may generate any number of counterexamples. For verification purposes, where counterexamples may trace through many time steps, it is much more efficient to use the model checker to find a bug, then fix that bug before searching for additional counterexamples.

By changing the domain of the model-checking problem to reasoning over BDD operations from explicit manipulations of automata, we increase the size of the system we can reason about. Combining concepts from Boolean algebra and graph theory allows us to achieve such a high level of algorithmic efficiency that the performance of symbolic model checking using these techniques is limited mostly by the sizes of the BDDs involved. Table 3 compares the two methods for each step in the model-checking problem.

4.4. *Air traffic control example: a BDD*

We now revisit our example of the automated airspace concept for air traffic control. Using the methods presented in Section 4.2 for encoding automata as BDDs, and ordering the variables as we presented them, we arrive at the BDD representation of the automaton introduced in Section 2.2, displayed in Fig. 11.

[21] One of the reasons maintaining a reduced BDD is important is that this algorithm can require exponential time if not [137].

The variable ordering used in Fig. 11 is {AR_command, AR_command', TSAFE_command, TSAFE_command', controller_request, controller_request', aircraft_request, aircraft_request', TSAFE_clear, TSAFE_clear'}. Note that this variable ordering was chosen for our example to be easy to read, not to be optimal. Nevertheless, the binary decision tree for this ten-variable model would have $2^{n+1} - 1 = 2^{10+1} - 1 = 2047$ total nodes while the BDD we picture here has only 47. Note also that this BDD is easily constructed from either of the NuSMV models we presented in Section 2.2, Boxes I and II, since it simply encodes the changes in variables that are possible in a single time step.

Returning to our example liveness specification, we want to check that whenever a near-term conflict is detected (i.e. whenever TSAFE_clear is false) that the conflict is addressed (i.e. a correcting command from TSAFE is issued). In order to do that, we check the LTL formula $\Box(\neg \mathit{TSAFE\_clear} \rightarrow \Diamond \mathit{TSAFE\_command})$.

In Section 3.8, Box V, we saw the symbolic automaton created by the model checker for this formula. In Fig. 12, we present the BDD representation of that symbolic automaton, which encodes the current and next-time versions of the four variables and the transition relation in that representation. Note in particular how small this collection of Boolean variables and pointers is in comparison to what we would need to store to explicitly enumerate all states of the automaton. This is an example of the concept of reasoning about states and transitions more succinctly, as sets of states and sets of transitions. We need only a few of the system variables for this representation, though the paths through this BDD encode sets of states where the values of the system variables not present in the BDD vary. We use the variable ordering of Fig. 11 with the current and next-time versions of the specification's elementary variables EL_X_G__TSAFE_clear_OR_F_TSAFE_command and EL_X_F_TSAFE_command tacked onto the end, which will allow us to combine these BDDs for the model checking step.

5. Checking for counterexamples

We construct the automaton $A_{M,\neg\varphi}$ as the composition of the system automaton M, and the automaton for the complimented LTL specification $A_{\neg\varphi}$. The model checking question, "does $M, q_0 \models \varphi$," then reduces to asking "is the language $\mathcal{L}(A_{M,\neg\varphi})$ empty?" The follow-up to this question is "if not, what word(s) does $A_{M,\neg\varphi}$ accept?" Any such word represents a computation of M that violates φ, constituting a

Table 3 – Table comparing explicit and symbolic model-checking algorithms. Recall that $el(\neg\varphi)$ is the set of elementary formulas of φ as defined in Section 3.7. We presume the ROBDDs for M and $\neg\varphi$ are created using the appropriate variants of the BUILD algorithm [146]. The ROBDD for $A_{M,\neg\varphi}$ is created by an extension of the algorithm APPLY($\wedge$, $ROBDD_M$, $ROBDD_{\neg\varphi}$) [146], implementing the dynamic programming optimizations that result in lower time-complexities. Finally, the algorithm used for finding a counterexample given an ROBDD is based on ANYSAT [146]. Note that, for both explicit and symbolic model checking, multiple steps below are performed at once, using on-the-fly techniques, which increases the efficiency of the process and may avoid constructing all of $A_{M,\neg\varphi}$. We separate out the steps here for simplicity only.

Operation	Explicit method	Time complexity	Symbolic method	Time complexity
Translate M from a higher-level language	Construct a Büchi automaton	Depends on M	Construct an ROBDD	$\mathcal{O}(2^{\lvert\Sigma\times\Sigma\rvert})$
Translate φ from LTL to $A_{\neg\varphi}$	Construct a Büchi automaton	$2^{\mathcal{O}(\lvert\varphi\rvert)}$	Construct an ROBDD	$\mathcal{O}(2^{\lvert el(\neg\varphi)\rvert})$
Create $A_{M,\neg\varphi}$	Construct automaton for $\mathcal{L}(M)\cap\mathcal{L}(A_{\neg\varphi})$	$\mathcal{O}(\lvert M\rvert\times\lvert A_{\neg\varphi}\rvert)$	Compute $(ROBDD_M \wedge ROBDD_{\neg\varphi})$	$\mathcal{O}(\lvert ROBDD_M\rvert\lvert ROBDD_{\neg\varphi}\rvert) = \mathcal{O}(2^{\lvert\Sigma\times\Sigma\rvert}2^{\lvert el(\neg\varphi)\rvert})$
Check $A_{M,\neg\varphi}$ for nonemptiness	Search for an accepting path via iterative depth-first search of the strongly connected components	$\mathcal{O}(\lvert A_{M,\neg\varphi}\rvert)$	Compute the fixpoint of the combined BDD with the spec $\mathcal{E}\Box\, true$	$\mathcal{O}(\lvert BDD representation\rvert)$[a]
Construct a counterexample	Compute an accepting cycle through the SCC graph	$\mathcal{O}(\lvert trace\rvert)$	Find a path leading to the terminal node 1 by a depth-first traversal of $ROBDD_{M,\neg\varphi}$	Linear in the length of the counterexample found: $\mathcal{O}(\lvert trace\rvert)$

[a]Here, the time complexity depends upon the size of the symbolic (BDD) representation in terms of the distances between states in the automaton graph, the number and arrangement of the strongly connected components in this graph, and the number of fairness conditions asserted.

counterexample to the check $M, q_0 \models \varphi$ and providing a valuable debugging tool for locating the bug. Thus, in practice, the model-checking step is performed in two parts: an emptiness check on the language $\mathcal{L}(A_{M,\neg\varphi})$ and the construction of the (preferably shortest) counterexample witness if $\mathcal{L}(A_{M,\neg\varphi}) \neq \emptyset$.

The earliest complete symbolic model-checking algorithm was designed in 1992 by Burch, Clarke, McMillan, Dill, and Hwang to take advantage of regularity in the state graph of the system and utilize the intuitive complexity of the state space rather than the explicit size of the space [22]. We present an updated algorithm here, better suited to symbolic LTL model checking specifically [148], which takes full advantage of our automata-theoretic approach, allowing very naturally for extensions to the basic algorithm [5]. An issue with using the original symbolic model checking algorithm, and later variants thereof, for LTL model checking is that they operated via a translation to the type of fair transition system also used for CTL model checking with fairness constraints [23], which itself involved a conversion to the mu-calculus [22,149]. This method is indirect, complex, and does not account for the full expressibility of LTL such as *full fairness*, also called *strong fairness* or *compassion*. Recall from Section 3.5 that compassion extends justice to include the existence of sets of cyclically recurring system states. Compassion can be particularly useful when specifying special types of coordination, such as semaphores or synchronous communication protocols. In Section 3.5 we showed that compassion requirements can be included via specifications. We can also include compassion natively in the system model.

The classical automata-theoretic approach to model checking using justice corresponds to reasoning over generalized Büchi automata whereas adding compassion extends this to reasoning over Streett automata [150]. Streett automata are essentially the same as Büchi automata but with more general acceptance conditions, enabling the direct encoding of stronger notions of fairness [112]. Instead of specifying F, the set of final states that must be visited infinitely often in Büchi acceptance, Streett acceptance takes the form of a set of pairs of sets of states, (L_i, U_i) such that if a run visits a state in L_i infinitely often, it must also visit a state in U_i infinitely often [112]. The pairs of sets of states in Streett acceptance conditions correspond nicely to the definition of *generalized fairness* [151]. Of course, since Street automata are expressively equivalent to Büchi automata and there is a straightforward conversion between the two [112], adding compassion at the algorithmic level does not substantially change our nonemptiness check [5].

Recall from Fig. 8 that a counterexample witness takes the shape of a lasso: a finite prefix starting from the initial state, q_0, and leading to an accepting cycle of the automaton $A_{M,\neg\varphi}$. For LTL model checking, fairness requirements are captured by the automata acceptance conditions, so each justice requirement and compassion requirement constitutes an additional acceptance condition. The challenge, then, is to find a reachable *fair cycle*, or a cycle that passes through at least one state satisfying each acceptance condition. Such a cycle is often referred to as a 'bad cycle' because it is a cycle on which all acceptance conditions are satisfied yet our specification, φ, is violated, which is certainly an undesirable outcome considering our goal is to prove that the system M never violates φ. We return as the counterexample, the list of states comprising the finite stem from q_0 to some state in this fair cycle, followed by a listing of the states visited in the cycle. See Section 5.4 for a counterexample returned from NuSMV for our air traffic control example.

We want to find the shortest counterexample because that facilitates debugging and, in some cases, returning a shorter

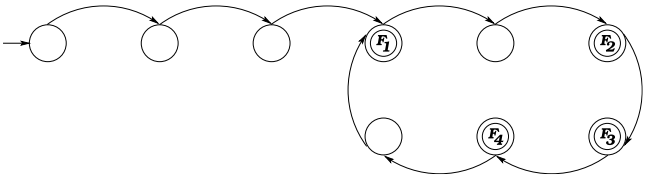
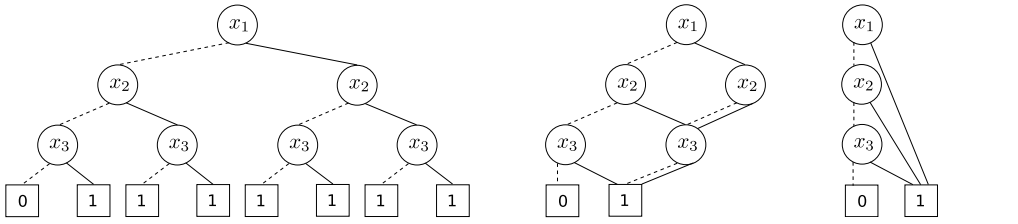
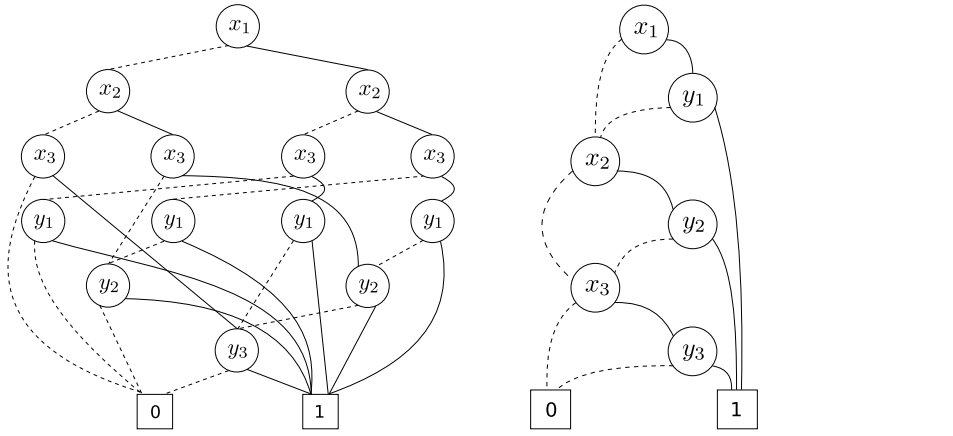


Fig. 8 – A counterexample trace takes the form of an accepting lasso. Again, the start state is designated by an incoming, unlabeled arrow not originating at any vertex. Here, $F_1 \ldots F_4$ represent final states satisfying four acceptance conditions.



(a) Binary Decision Tree. **(b) Isomorphism elimination rule.** **(c) Redundancy elimination rule.**

Fig. 9 – Conversion of a Binary Decision Tree for $x_1 \vee x_2 \vee x_3$ into a Binary Decision Diagram.



(a) Variable order $= \{x_1, x_2, x_3, y_1, y_2, y_3\}$. **(b) Variable order $= \{x_1, y_1, x_2, y_2, x_3, y_3\}$.**

Fig. 10 – BDDs for the function $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$.

counterexample can save time and memory. However, this step in the model-checking process needs to be fast and finding the shortest counterexample is NP-complete [152], so we depend instead on reliable heuristics to efficiently find a (hopefully) short counterexample quickly.[22] Furthermore, we want to construct this counterexample directly from our symbolic automaton, $A_{M,\neg\varphi}$ for maximum efficiency. For this reason, there has been a concentration on finding efficient symbolic cycle-detection algorithms. The first such effort, by Emerson and Lei [24], produced an algorithm that operates in quadratic time due to the presence of a doubly-nested fixpoint operator, but is still a standard to which all later algorithms are compared. All of these algorithms perform some iterative computation over the reachable states in the automaton $A_{M,\neg\varphi}$ to find paths to fair cycles. In order to accomplish this, it is necessary for us to treat this automaton as a graph and employ some classic techniques from graph theory.

5.1. *Automata as graphs*

Formally, a *predicate* over the set of states Q is any subset $P \subseteq Q$. A *binary relation* over Q is any set of pairs $R \subseteq Q \times Q$ such that, for predicates P_1 and P_2:

$$P_1 \times P_2 = \{(q_1, q_2) \in Q^2 | q_1 \in P_1, q_2 \in P_2\}.$$

Recall that a Büchi automaton can be viewed as a special case of a directed graph $G = (V, E)$ where V is the finite set of vertices, or the states of the automaton such that $V = Q$,

[22] For an algorithm that always returns the shortest counterexample, see [153].

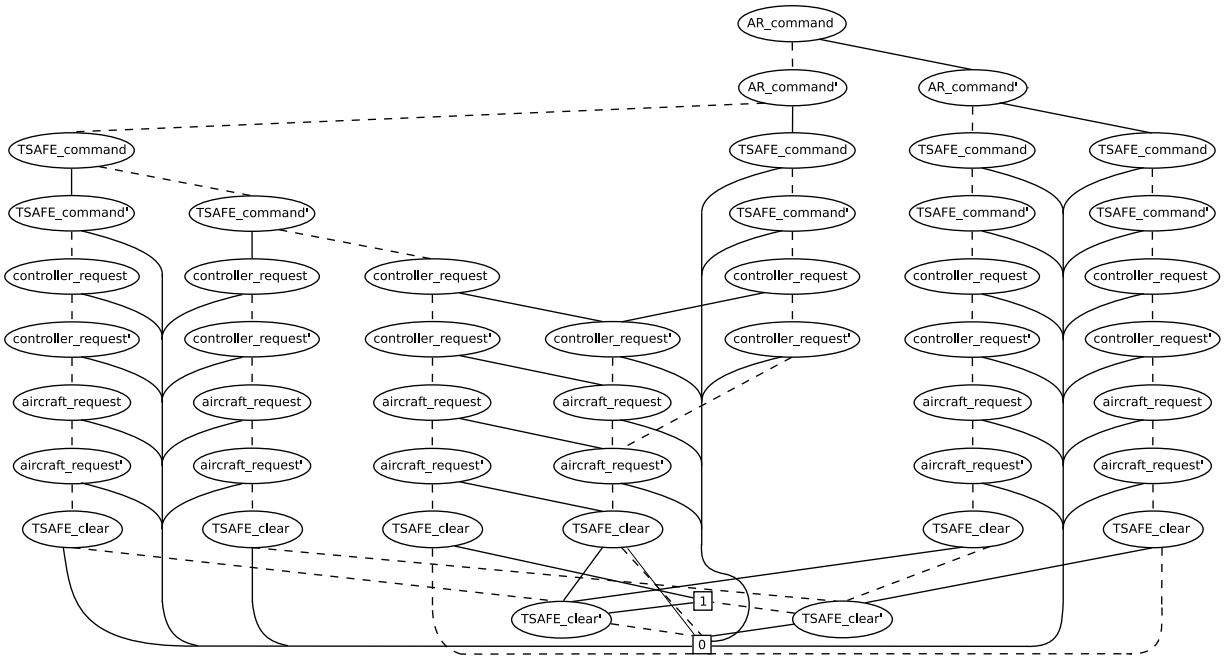


Fig. 11 – BDD representing the automaton in Fig. 1.

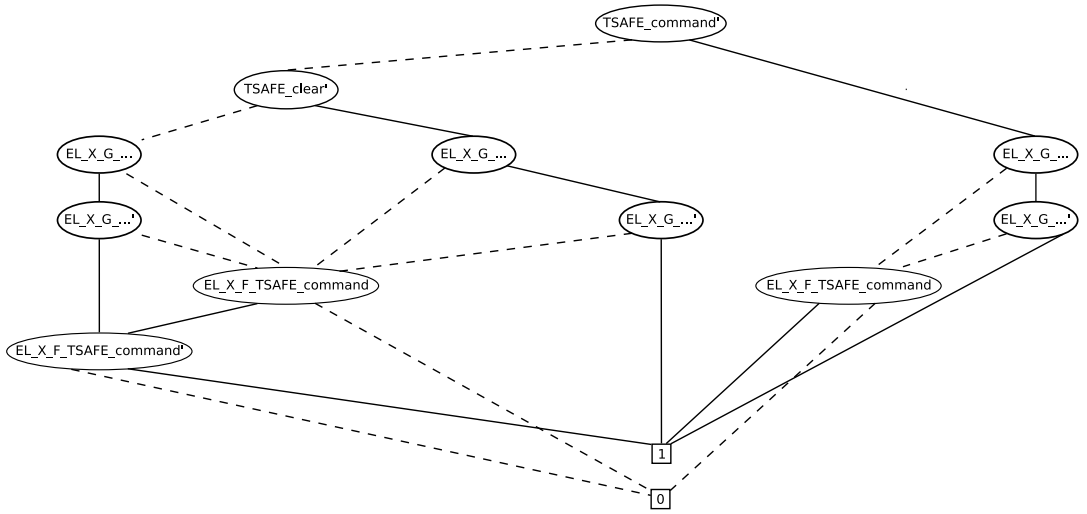


Fig. 12 – BDD representing the symbolic automaton in Section 3.8, Box V, for the formula $\Box(\neg TSAFE\_clear \rightarrow \Diamond TSAFE\_command)$. (The variable `EL_X_G__TSAFE_clear_OR_F_TSAFE_command` is abbreviated `EL_X_G_. . .`.)

and $E \subseteq V \times V$ is the set of edges, or the binary relation encapsulating the transitions of the automaton such that $(q, q') \in E$ whenever $\delta(q, \sigma) = q'$ for two states q and q' and any alphabet character $\sigma \in \Sigma$. A path through the graph that corresponds to a computation of the automaton, from some state q_j to state q_k is a sequence $\langle q_j, q_{j+1}, q_{j+2}, \ldots q_k\rangle$ of vertices such that $(q_{i-1}, q_i) \in E$ for $i = j+1, j+2, \ldots, k$. Along such a path, we call vertex q_{i-1} a *predecessor* of state q_i and vertex q_{i+1} a *successor* of q_i. Note that a path must contain at least one edge but the presence of self-loops means that a state can be its own predecessor or successor. If there is a path from q_j to q_k, we say that q_k is *reachable* from q_j or that $q_j \rightsquigarrow q_k$. Let R represent the binary relation corresponding to the transition relation δ of $A_{M,\neg\varphi}$. We can extend this relationship for a predicate P and the relation R using pre- and post-composition as follows:

$$R \circ P = \{q \in Q | (q, q') \in R \text{ for some } q' \in P\},$$

$$P \circ R = \{q \in Q | (\hat{q}, q) \in R \text{ for } \hat{q} \in P\}.$$

In other words, $R \circ P$ is the set of all predecessors of states in the set P. Conversely, $P \circ R$ is the set of all successors of states in the set P, or the set of all states reachable in a single transition from P in the graph of $A_{M,\neg\varphi}$, which we call $G_{A_{M,\neg\varphi}}$. We can employ the $*$-operator to define $R^* \circ P$, the set of all states that can reach a P-state within a finite number (0 or more) steps. Similarly, we designate the set of all states

reachable in a finite number of steps from a P-state as $P \circ R^*$. Expanded, we have:

$$R^* \circ P = P \cup R \circ P \cup R \circ (R \circ P) \cup R \circ (R \circ (R \circ P)) \cup \cdots,$$

$$P \circ R^* = P \cup P \circ R \cup (P \circ R) \circ R \cup ((P \circ R) \circ R) \circ R \cup \cdots.$$

Since $A_{M,\neg\varphi}$ is finite, it is clear that both $R^* \circ P$ and $P \circ R^*$ converge in a finite number of steps. Because $P \circ R^*$ consists of the set $\{q' \in Q | q \rightsquigarrow q'\}$, it is also called the *forward set* of state q. In the same vein, $R^* \circ P$ comprises the *backward set* of state q or $\{\hat{q} \in Q | \hat{q} \rightsquigarrow q\}$. We define the diameter d of a graph to be the length of the longest minimal path between any two states $(q_j, q_k) \in G$ such that $q_j \rightsquigarrow q_k$. In other words, for all pairs of connected vertices $(q_j, q_k) \in G$, setting the distance from q_j to q_k to the smallest number of states on any path between them, d is the longest such distance in G.

A *connected component*, is a maximal connected subgraph such that two vertices are in the same connected component if and only if there is some path between them. Meanwhile, a *strongly connected component* (SCC) is a maximal subgraph such that every vertex in the component is reachable from every other vertex in the component. That is, for any strongly connected component S, for all vertices q_j, q_k such that $q_j \in S$ and $q_k \in S$, $q_j \rightsquigarrow q_k$ and $q_k \rightsquigarrow q_j$. For example, the graph of our automated air traffic control architecture in Fig. 2 consists of one SCC since every state in that graph has a path to every other state in the graph. An individual vertex, not connected to itself, comprises a *singular* or *trivial* SCC. A path $\langle q_j, q_{j+1}, q_{j+2}, \ldots q_k \rangle$ forms a cycle if $q_j = q_k$ and the path contains at least one edge. Thus, a strongly connected component is essentially a set of vertices such that every pair of vertices in the set is contained in some cycle. Algorithms for finding SCCs in symbolic graphs take advantage of the following property:

Theorem 8. *The intersection of a node's forward and backward sets is either empty or a strongly connected component (SCC).*

Proof of Theorem 8. Let q be a vertex in directed graph G. Let $F(q)$ represent the forward set of q and $B(q)$ represent the backward set of q. First, presume q is in an SCC S. By definition of an SCC, all nodes are reachable from, and can reach, all other nodes in the SCC. Therefore, all of the nodes in the SCC containing q would have to be in both q's forward and backward sets, or $S \subseteq F(q) \cap B(q)$. Furthermore, no additional nodes not in the SCC containing q can be in the set $F(q) \cap B(q)$. Let q' be a node in $F(q) \cap B(q)$. Then $q \rightsquigarrow q'$ and $q' \rightsquigarrow q$. Since every node $\hat{q} \in S$ has a path to and can be reached from q, then it must be the case that q' and $\hat{q}$ are strongly connected by some path through q. In other words, $q' \rightsquigarrow q \rightsquigarrow \hat{q}$ and $\hat{q} \rightsquigarrow q \rightsquigarrow q'$. Therefore, we know that $F(q) \cap B(q) \subseteq S$. Together, we have $S \subseteq F(q) \cap B(q) \wedge F(q) \cap B(q) \subseteq S \rightarrow F(q) \cap B(q) = S$. It logically follows that the node q is not in a non-trivial SCC iff $F(q) \cap B(q) = \emptyset$. □

A *terminal SCC* is an SCC with no edges leading out to nodes outside the SCC. Symmetrically, an *initial SCC* is one with no edges leading in from nodes outside the SCC. In Theorem 8 above, we take the intersection of node q's forward and backward sets because a node q in an SCC can still have a transition to another node outside that SCC, so if the node q is in an SCC that is not terminal, there may be many more nodes in its forward set that are reachable from that SCC. Symmetrically, a node q in an SCC can still have a transition in from another node outside that SCC so if the node q is in an SCC that is not initial, there may be many more nodes in its backward set that can reach that SCC. All of the algorithms for symbolic cycle detection utilize some combination of computing forward sets (successors) while looking for initial SCCs, backward sets (predecessors) while looking for terminal SCCs, or some intermingling of both strategies.

Recall that the acceptance condition for a generalized Büchi automaton requires cycling through a state in each set in F (the set of sets of states where each acceptance condition holds) infinitely often. Recall also that an accepting run of a generalized Büchi automaton resembles a lasso with a path from the start state to an accepting cycle. Thus, finding an accepting run of a generalized Büchi automaton comes down to finding a path to a strongly connected component in the graph of $A_{M,\neg\varphi}$, $G_{A_{M,\neg\varphi}}$, that contains at least one state satisfying each acceptance condition.

Strong Fairness. In LTL model checking, we may place these conditions on acceptance: that the counterexample found must be just and that it must be compassionate. For each justice requirement J in the set of justice requirements $\mathcal{J}$ over $A_{M,\neg\varphi}$, we define a set of J-states (states that satisfy J). We call a subgraph *just* if it contains a J-state for every justice requirement $J \in \mathcal{J}$. Similarly, for each compassion requirement over $A_{M,\neg\varphi}$, we define a pair of sets $(y, z) \in \mathcal{C}$ that constitute that requirement where $\mathcal{C}$ is the set of all such pairs. We call a subgraph *compassionate* if, for every compassion requirement $(y, z) \in \mathcal{C}$ the subgraph either contains a z-state or else does not contain any y-state. Combining these, we have that a subgraph is *fair* if it is a non-singular strongly connected component that is both just and compassionate, in that it intersects each accepting set of states in both $\mathcal{J}$ and $\mathcal{C}$. Therefore, a counterexample lasso must be reachable from q_0 and cycle infinitely often through at least one state satisfying each acceptance condition, which involves visiting a minimum set of states satisfying any justice or compassion requirements we have asserted over $A_{M,\neg\varphi}$.

Theorem 9 ([154]). *An infinite initialized path π is a computation of $A_{M,\neg\varphi}$ iff the set S of states that appear infinitely often in π comprises a fair SCC of the graph representing $A_{M,\neg\varphi}$, $G_{A_{M,\neg\varphi}}$.*

Proof of Theorem 9.

If-direction: π is a computation of $A_{M,\neg\varphi} \rightarrow S$ is a fair SCC in $G_{A_{M,\neg\varphi}}$.

Presume $A_{M,\neg\varphi}$ has a computation $\pi = \pi_0, \pi_1, \ldots$. By definition, π carves out an infinite path, starting in state q_0 that passes infinitely often through at least one state satisfying each acceptance condition (justice or compassion requirement) of $A_{M,\neg\varphi}$. Since $A_{M,\neg\varphi}$ is finite, π takes the shape of a lasso with a finite prefix of states starting from q_0 leading to a cycle containing all of the states π visits infinitely often. We define S to be the set of all states along that cycle. Then, S must be fair since π is an accepting run of $A_{M,\neg\varphi}$. Furthermore, S must be strongly connected since for all pairs of states $s, s' \in S$, both s and s' appear infinitely often in π, so $s \rightsquigarrow s'$ and $s' \rightsquigarrow s$.

Only-if direction: S is a fair SCC in $G_{A_{M,\neg\varphi}} \rightarrow \pi$ is a computation of $A_{M,\neg\varphi}$.

Presume S is a fair SCC in $G_{A_{M,\neg\varphi}}$. Consider some path π, originating in q_0 such that the set of states visited by π infinitely often is defined to be S. We can construct such a path by connecting every pair of states $s, s' \in S$ such that $(s, s') \in E$, which is equivalent to saying $\delta(s, \sigma) = s'$ for some character $\sigma \in \Sigma$. That is, we traverse every edge between states in S. In this way, we form the cycle such that there are infinitely many positions j such that $\pi_j = s$ and $\pi_{j+1} = s'$. By definition of a fair SCC, we know S intersects each accepting set of states in $G_{A_{M,\neg\varphi}}$. Therefore, by definition, π is a computation of $A_{M,\neg\varphi}$. □

5.2. *Symbolic methods for graph traversal*

The standard algorithm for finding strongly connected components in a directed graph is depth-first search. Indeed, this is the algorithm used in explicit-state model checking for finding and returning counterexamples. However, in symbolic model checking we are not using an explicit automaton graph; our graph is instead encapsulated succinctly using BDDs. We have encoded the graph in terms of sets of states and sets of transitions, altering the problem of traversing the graph. Due to the nature of this encoding, depth-first approaches, while optimal for examining individual states, are not suitable for symbolic cycle detection. Rather, we employ breadth-first, set-based cycle-detection algorithms better suited to searching over the characteristic function, S_h, as defined in Section 3.7.

Our goal is to locate a fair subgraph of $G_{A_{M,\neg\varphi}}$ because, as we know from Theorem 9, this will allow us to construct a counterexample, which is a computation of $A_{M,\neg\varphi}$. This search is an iterative process. Essentially, we will compute some part of the transitive closure of the transition relation of $A_{M,\neg\varphi}$, i.e. $R^* \circ P$ or $P \circ R^*$, preferably without incurring the computational expense of computing the entire transitive closure. To accomplish this, symbolic algorithms take advantage of meta-strategies for dealing with the set-based encoding of graphs such as intelligently partitioning $G_{A_{M,\neg\varphi}}$ and reasoning over the SCC quotient graph of $G_{A_{M,\neg\varphi}}$. The SCC *quotient graph* is a directed, acyclic graph $G_{quotient} = (V_{quotient}, E_{quotient})$ such that the set of vertices $V_{quotient}$ is the set of SCCs in $G_{A_{M,\neg\varphi}}$. Again, let R represent the binary relation corresponding to the transition relation δ of $A_{M,\neg\varphi}$. For two vertices $V_1, V_2 \in V_{quotient}$, there is an edge $(V_1, V_2) \in E_{quotient}$ iff $V_1 \neq V_2$ (so there are no self-loops) and $\exists(v_1 \in V_1, v_2 \in V_2) : (v_1, v_2) \in R$. Restated, there is an edge in $G_{quotient}$ whenever there is an edge in $G_{A_{M,\neg\varphi}}$ from a state in one SCC to a state in a different SCC. Note that it is easy to prove that $G_{quotient}$ is acyclic since if there were a cycle in this graph, then all of the states in all of the SCCs on such a cycle would be reachable from each other, which contradicts their status as separate SCCs. $G_{quotient}$ defines a partial order over the SCCs in $G_{A_{M,\neg\varphi}}$ such that $v_1 \leq v_2$ for any states $v_1 \in V_1, v_2 \in V_2$ iff $v_1 \rightsquigarrow v_2$. Using $G_{quotient}$, we can identify the initial SCCs of $G_{A_{M,\neg\varphi}}$ as the sources of the graph and the terminal SCCs as the sinks. It is easier to reason about sets of predecessors and successors and partition the graph $G_{A_{M,\neg\varphi}}$ to limit our search. Symbolic algorithms utilize these types of tools to reduce the fair subgraph search problem to a smaller set of nodes than the entire graph $G_{A_{M,\neg\varphi}}$.

Whereas the time required for the depth-first exploration of the graph of $A_{M,\neg\varphi}$ performed in explicit-state model-checking is directly dependent on the size of $A_{M,\neg\varphi}$, this is not the case for symbolic model checking algorithms. After all, symbolic algorithms derive their efficiency by representing compact characteristic functions instead of large numbers of individual states. In practice, the number of states in $A_{M,\neg\varphi}$ is not nearly as accurate a predictor of the time required for symbolic model checking as the graph diameter d, the length of the longest path in the SCC quotient graph, the number of justice and compassion requirements, the number of reachable SCCs (and how many of those are trivial), and the total number and size of the set of SCCs [155]. Basically, the number of states is frequently very large but some shapes of $A_{M,\neg\varphi}$ are much easier to model check than others. This is an advantage of utilizing symbolic algorithms since many very large graphs can be reduced to very succinct symbolic representations. The trade-off between the size of the representation of $A_{M,\neg\varphi}$ and the search complexity is necessary for practical verification. As the number of states in $A_{M,\neg\varphi}$ grows, considering every state individually quickly becomes infeasible so symbolic examination of the state space, while more difficult, is necessary to achieve scalability. Generally, the length of the counterexample found also depends heavily on the structure of the graph and the size of the symbolic representation.

There are many variations of this algorithm that have been created to provide better time-complexities, better performance times, and shorter counterexamples.[23] Optimizing fair cycle detection for symbolic model checking remains an active area of research [156,153,157–159,155,160–162].

There are two general classes of symbolic cycle-detection algorithms:

- *SCC-hull algorithms:* Most symbolic cycle-detection algorithms [24,156,144,157,148,161] sequester an *SCC-hull*, or a set of states that contains all fair SCCs, without specifically enumerating the SCCs within. This set computation is not tight; if there are fair SCCs in the graph, the SCC-hull algorithms may return extra states besides those in the fair SCCs.[24] Basically, these algorithms maintain an *approximation set*, or a conservative overapproximation of the SCCs. The approximation set is iteratively refined by *pruning*, or locating and removing states that cannot lead to a bad cycle utilizing the property that any state on a bad cycle must have a successor and a predecessor that are both on the same cycle, in the same SCC, and, therefore, in the approximation set. In the case that there are no fair SCCs, these algorithms return the empty set. These algorithms locate the SCC-hull using a fixpoint algorithm that either computes the set of all states with a path to a fair SCC [24], from a fair SCC [156], or some combination of both [157] since both of these sets include the states contained in a fair SCC, if one exists. Counterexamples are

[23] Note that the problem of finding the shortest counterexample is NP-complete [152].

[24] We can compile a tighter set of fair SCCs by directly computing the exact transitive closure of the transition relation rather than an overapproximation but this method is comparatively inefficient for reasoning over BDDs [155].

generated by searching for a path from the initial state q_0 to a state in a fair cycle then iterating through the set of acceptance conditions using breadth-first-search to find the shortest path to a state in the next fair set until the cycle is completed. Depending on the specific SCC-hull algorithm, varying amounts of additional work may be required to isolate the fair SCC to be returned in a counterexample.

- *Symbolic SCC-enumeration algorithms:* Alternatively, SCC-enumeration algorithms [153,158–160] enumerate the SCCs of the state graph while taking advantage of the symbolic representation to avoid explicitly manipulating the states. Rather than extracting the (terminal or initial) SCCs from the set of all SCCs in a hull, these algorithms aim to compute reachability sets (either forward or backward) for fewer nodes by isolating SCCs sequentially. This is accomplished, for example, by recursively partitioning $G_{A_{M,\neg\varphi}}$ and iteratively applying reachability analysis to the subgraphs. The motivation is that many systems have only a limited number of fair SCCs so these algorithms can achieve a better worst-case complexity than SCC-hull algorithms [160], but in practice, their performance has been inferior [155]. Like with SCC-hull algorithms, these algorithms utilize some combination of forward and backward search, possibly in an interleaved manner. Pruning helps to reduce the number of trivial SCCs examined. After each SCC is identified, it is checked for fairness. The algorithm terminates as soon as the first fair SCC is located and may end up either enumerating all SCCs or paying additional computation cost for early elimination of unfair SCCs in the case that no fair SCC exists. Consequently, these algorithms tend to perform worse than SCC-hull algorithms when there are no fair SCCs or a large number of unfair SCCs present in $A_{M,\neg\varphi}$ [155]. Counterexamples are constructed similarly to, but possibly more efficiently than, SCC-hull algorithms since it is theoretically easier to locate a counterexample given that the enumerated fair SCC in question is tight. The length of the counterexample depends on which fair SCC is enumerated first, which is determined by the starting state of the algorithm, so optimizations include heuristics for choosing seed states in short counterexample traces.

5.3. SCC-hull algorithm for compassionate model checking

Here we present the full algorithm for symbolic LTL model checking, including for the sake of completeness, support for compassion at the algorithmic level, as opposed to adding it to the specification set or allowing only weaker forms of fairness. All symbolic LTL model checkers support justice but not all support compassion. For example, at the time of this writing, NuSMV supports compassion [43] but CadenceSMV does not [163]. Currently, compassion requirements are rarely used in industrial practice. The algorithm discussed in this section was first published in 1998 by Kesten, Pnueli, and Raviv [148] and is exactly the under-the-hood implementation in NuSMV today [43]. It is presented in a straightforward manner using standard set theory since set operations on the languages accepted by finite automata translate transparently to logical operations on Boolean functions, as we discussed in Section 4.

We call an automaton *feasible* if it has at least one computation. The corollary to Theorem 9 is that $A_{M,\neg\varphi}$ is feasible iff $G_{A_{M,\neg\varphi}}$ contains a fair subgraph (a non-singular strongly connected component that is both just and compassionate). Therefore, we first check for the presence of a fair subgraph and determine the feasibility of our combined automaton. The model-checking problem reduces to $M \models \varphi$ iff $A_{M,\neg\varphi}$ is not feasible. If $A_{M,\neg\varphi}$ is not feasible, then our verification is complete; we can conclude that $M \models \varphi$. If $A_{M,\neg\varphi}$ is feasible, we will need to construct a (preferably short) counterexample.

Indeed, the algorithm for checking whether an automaton is feasible is essentially a specialization of the standard iterative algorithm for finding strongly connected components in a graph. The original cycle-detection algorithms for explicit-state model checking recursively mapped strongly connected subgraphs, computing closures under both successors and predecessors [56,66]. Later it was proved that the requirement of bi-directional closure is too strong [148]. The algorithm, FEASIBLE(), presented below, requires only closure under successor states while looking for initial components of the graph [148].[25] Proofs of soundness, completeness, and termination of this algorithm are provided in [148,154].

Let $\|\psi\|$ denote the predicate consisting of all states that satisfy ψ, for some formula ψ, and $\|\delta\|$ denote the relation consisting of all state pairs $\langle q, q'\rangle$ such that $\delta(q,\sigma) = q'$ for any alphabet character $\sigma \in \Sigma$. Then we have the subroutine given in Box IX.

The algorithm FEASIBLE() takes as input a synchronous parallel composition of the system and specification, which in our case is $A_{M,\neg\varphi}$. Recall that in this construction, $\neg\varphi$ is essentially serving as a tester of M, continuously monitoring the behavior of the system and making sure the system satisfies the desired property. Our subroutine starts by computing the set of all successors of the initial state, q_0 in the predicate *new*. Because any run of $A_{M,\neg\varphi}$ will satisfy the initial condition and obey the transition relation but not necessarily satisfy the justice and compassion requirements, we need to check for this characteristic additionally. We loop through each justice and compassion requirement, subtracting states from *new* that are not reachable in a finite number of steps from states satisfying that requirement. Then we remove from *new* all states that do not also have a predecessor in *new* since such states cannot be part of a strongly connected component. We repeat this process until we reach a fixpoint; looping through every fairness requirement, checking for predecessors, and reducing *new* accordingly does not change the size of *new*. At this point, if *new* is empty, we return the emptyset and conclude that $A_{M,\neg\varphi}$ is not feasible and, therefore, $M \models \varphi$. If *new* is not empty, then it contains a fair strongly connected component of $G_{A_{M,\neg\varphi}}$ from which we must extract a counterexample witness.

To aid in our quest to efficiently find a short counterexample, if possible, we require a subroutine to find the shortest path between two states in $G_{A_{M,\neg\varphi}}$. Let λ represent the empty list. We use the $*$-operator to denote list *fusion* such that if

[25] An equivalent algorithm would be to check for closure under predecessor states while looking for terminal components in the state-transition graph.

```
algorithm FEASIBLE(A_M, ¬φ): predicate // Check feasibility of A_M, ¬φ
new, old : predicate            //a predicate is a subset of the set of states Q
R : relation                    //a relation is a set of pairs of states

old = ∅                         //initialize old to the empty set
R = ||δ||                       //R is the set of all state pairs in the transition relation of A_M, ¬φ
new = ||q0|| ∘ R*                //new is the set of all states reachable from a start
                                   state in A_M, ¬φ in a finite number of steps
while (new ≠ old) do
begin
  old = new
  for each J ∈ 𝒥 do              //for each justice requirement J in the set 𝒥
    new = (new ∩ ||J||) ∘ R*     //new is reduced to the set of new states reachable
                                   in a finite number of transitions from a state
                                   satisfying justice requirement J
  for each (y, z) ∈ 𝒞 do         //for each compassion requirement (y, z) in the set 𝒞
  begin
    new = (new − ||y||) ∪ [(new ∩ ||z||) ∘ R*] //subtract from new all states where y holds and add
                                                 to new all states  reachable in a finite number of
                                                 steps from a z-state in new
    R = R ∩ (Q × new)            //subtract from R all transitions that don't lead into a state in new
  end
  while (new ≠ new ∩(new ∘ R)) do //while new is not comprised of the set of all
                                    successors of new states
      new = new ∩ (new ∘ R)      //reduce new to states that have predecessors in new
end
return (new)                     //here, new should contain only fair strongly connected components
```

Box IX.

```
// Compute shortest path from source to destination
function PATH(source, destination : predicate; R: relation) : list

start, f : predicate          //a predicate is a subset of the set of states Q
L        : list               //a list is an ordered sequence of states
s        : state
start = source
L = λ                          //L is now the empty list
while (start ∩ destination == ∅) do
begin
  f = R ∘ destination          //start set f is the set of all predecessors of destination states
  while (start ∩ f == ∅) do //while we haven't found a path back to start
    f = R ∘ f                  //add all of the predecessors of the states currently in the set f
  s = choose(start ∩ f)        //pick one of the shortest path start states
  L = L * (s)                  //add this state to the end of the state list
  start = s ∘ R                //add all successors of s to the set start
end
return L * (choose(start ∩ destination)) //push the last state onto the end of the list and return the path
```

Box X.

$L_1 = (\ell_1, \ell_2, \ldots \ell_i)$ and $L_2 = (\ell_i, \ell_{i+1}, \ldots)$ are standard list data structures such that the last element of L_1 is the same as the first element of L_2, then $L_1 * L_2 = (\ell_1, \ell_2, \ldots \ell_i, \ell_{i+1}, \ldots)$ represents their concatenation, overlapping this shared element, or simply concatenating the lists if there is no overlap. We define the function *choose*(P) to consistently choose one element of a predicate P. Then we have the subroutine given in Box X.

We are now ready to present the full counterexample extraction algorithm. For completeness, we show the step of checking $A_{M,\neg\varphi}$ for feasibility in context. We define the function *last*(L_1) to return the last element in the list L_1 in the algorithm given in Box XI.

After checking $A_{M,\neg\varphi}$ for feasibility, we exit if $M \models \varphi$. If not, we know that the set returned by `feasible()`, which we save in *final*, contains all of the states in fair SCCs that are reachable from q_0. We look at the set of successors of states in the set *final*. We pick a state from this set and iterate, replacing our chosen state s with one of its predecessors until the set of predecessors of s is a subset of the set of successors of s. Basically, we are attempting to move to an initial SCC in the quotient graph of $G_{A_{M,\neg\varphi}}$. Termination of this step is guaranteed by the finite nature of $G_{A_{M,\neg\varphi}}$ and the structure of the quotient graph. Next, we compute the precise SCC containing s, restrict our attention to transitions between states in this SCC, and utilize our shortest path algorithm to find the prefix of our counterexample, or the path from q_0 to our fair SCC. Then we construct the fair cycle through our SCC that satisfies all fairness requirements. In order to do this, we loop through each justice requirement, adding to our cycle a path to a state that satisfies that requirement if there is not one in the cycle already. We do the same for each compassion requirement. Finally, we complete the cycle with the shortest path back to the state we started the cycle from and we are ready to return our counterexample.

```
algorithm WITNESS(A_{M, ¬φ}) : [list, list] // Extract a witness for A_{M, ¬φ}
final          : predicate
R              : relation
prefix, period : list
s              : state
final = FEASIBLE(A_{M, ¬φ})                  //the model checking step
if (final == ∅) then return (λ, λ)           //emptyset indicates that M, q_0 ⊨ φ
R = ||δ|| ∩ (final × Q)                      //R is the set of all transitions out of a state in a fair SCC in G_{A_{M,¬φ}}
s = choose (final)                           //pick one state from the set final
while (R* ∘ {s} − {s} ∘ R* ≠ ∅) do //while the states from which we can reach s
                                     are not all states that can be reached from s
  s = choose(R* ∘ {s} − {s} ∘ R*)            //replace s with a predecessor that is not also a successor
final = R* ∘ {s} ∩ {s} ∘ R*                  //compute the SCC containing s, i.e. the maximum set of states t
                                       such that (t ⇝ s) ∧ (s ⇝ t)
R = R ∩ (final × final)                      //R now contains only transitions within the SCC final
prefix = PATH(||q_0||, final, ||δ||)         //prefix is now a shortest path from an initial state to a state in final
period = (last(prefix))                      //period starts in the final state of prefix
for each J ∈ 𝒥 do                            //add in the justice requirements
  if (list − to − set(period) ∩ ||J|| == ∅) then //if there are no states in period satisfying justice requirement J
    period = period * PATH({last(period)}, final ∩ ||J||, R)  //add to the end of period
                                         a shortest path to a state satisfying J
for each (y, z) ∈ 𝒞 do                       //add in the compassion requirements
  if ((list − to − set(period) ∩ ||z|| == ∅) ∧ (final ∩ ||y|| ≠ ∅)) then
    //if this compassion requirement isn't satisfied
    period = period * PATH({last(period)}, final ∩ ||z||, R) //add a z state into period
period = period * PATH({last(period)}, {last(prefix)}, R)
//finish period with a path to the end of prefix to form a complete loop
return(prefix, period)
```

Box XI.

5.4. *Air traffic control example: model checking*

In this final installment of our automated air traffic control architecture verification example, we demonstrate the complete model-checking process for this real-world system from the users' perspective. We have shown in detail how the model checker negates the input specification, φ, symbolically compiles $\neg\varphi$ into $A_{\neg\varphi}$, and conjoins $A_{\neg\varphi}$ with the symbolic automaton system model M, forming $A_{M,\neg\varphi}$. If the automaton $A_{M,\neg\varphi}$ is not empty, then the model checker finds a fair path that satisfies the formula $\neg\varphi$ and returns a counterexample, allowing the user to correct the error in the system design. While knowledge of how each of these steps is accomplished internally to the model checker is beneficial, in practice the model-checking process from the user's perspective is simple. Familiarity with the model checker under the hood is useful chiefly to help the user make smarter choices when interacting with the model checker, improving verification performance and user experience. For example, some intelligent choices made in the process of modeling M or utilizing counterexamples to narrow in on a bug can help skirt the state explosion problem and make debugging easier and more efficient.

We continue to demonstrate model checking using the tool NuSMV. The input file `AAC.smv` contains one of the NuSMV models from Section 2.2 concatenated with the specifications from Section 3.8. As with any model checker, we could supply our own variable order, designate which of the model checker's available heuristics we wish to use, or let the model checker choose for us. Most model checkers incorporate sufficiently optimized variable heuristics by default so, in practice, we usually choose to let the tool automatically find an appropriate ordering. In Box XII, we show the output from NuSMV generated by checking our example liveness specification that whenever a near-term conflict is detected (i.e. whenever `TSAFE_clear` is false) that the conflict is addressed (i.e. a correcting command from TSAFE is issued), which we wrote in LTL as `LTLSPEC (G (controller_request ->(F (! controller_request))))`. In this case, the specification holds, and we can conclude that $M \models \varphi$.

Now let us try another specification: we want to ensure the human in the loop can always override the software algorithms. In other words, we check that the corrective trajectory requests from the controller are always granted. Since, in our simple model, controller requests are fulfilled by transferring the request into a corresponding command by the auto-resolver, in LTL we write this as $\Box$(*controller_request* $\rightarrow$ $\Diamond$(*AR_command*&$\neg$*controller_request*)). The result of checking that specification using NuSMV is given in Box XIII.

Our specification has failed! The counterexample returned guides us along a path through our system where the specification does not hold. Notice the path starts in our system model's initial state, which NuSMV names State 1.1, where all of the variables but `TSAFE_clear` are false. In the next state along our path, the only variable that changes value is `controller_request`, which becomes true. (This corresponds to a transition to State 4 in the automaton in Fig. 2.) Next, we transition to a state where `controller_request` is 0 and all of the other variables' values are the same as they were in the last state. (We are now in State 1 in Fig. 2.) Finally, we transition again but none of the values of the variables change. (This represents the transition from State 1 to itself where the airspace is clear and there are no commands or requests.) Notice the occurrence of "– `Loop starts here`" just before the description of NuSMV State 1.3 designates that we loop from State 1 to State 1 forever, thus completing the description of our infinite accepting run.

```
> NuSMV AAC.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22 13:50:47 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification  G (!TSAFE_clear = 1 ->  F TSAFE_command = 1)  is true
>
```

Box XII.

```
> NuSMV AAC_2.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22 13:50:47 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification  G (controller_request = 1 ->  F (!controller_request = 1 & AR_command = 1))
   is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  AR_command = 0
  TSAFE_command = 0
  controller_request = 0
  aircraft_request = 0
  TSAFE_clear = 1
-> Input: 1.2 <-
-> State: 1.2 <-
  controller_request = 1
-> Input: 1.3 <-
-- Loop starts here
-> State: 1.3 <-
  controller_request = 0
-> Input: 1.4 <-
-> State: 1.4 <-
>
```

Box XIII.

6. Discussion

While the complete algorithm presented here serves as the basis for LTL symbolic model checking performed in industry, in practice, extensions to this algorithm are much more commonly used. We present the foundation upon which more specialized and scalable variations are built. Industrial applications frequently necessitate the use of extensions that provide additional scalability such as compositional verification [74], in which subunits of the system M and the interactions between these subunits are model checked separately, each in the manner presented here. Since the entire automated airspace concept for air traffic control is extremely large, the full-scale version of our running example involves verifying components such as TSAFE and the auto-resolver as separate subunits, then verifying their interactions, i.e. via the high-level architecture we have shown. Also, bounded model checking [133] depends upon a propositional SAT solver to replace the BDD operations of classical symbolic model checking and bounds the length of any possible counterexample to be less than some constant k. While this variation no longer provides the guarantee that no counterexample of any length exists, it vastly increases the size of systems we are able to verify, providing better guarantees of termination in a reasonable timeframe. Allowances for adding operators to LTL, as described in Section 3.2.4, are also implemented through adjustments to the algorithm for translating standard LTL.

Our running example of verifying an automated air traffic control architecture serves both to illustrate the concepts discussed throughout this paper and also as a stand-alone introduction to the application of LTL symbolic model checking to an, admittedly very simple, real-world system. That said, several full-scale air traffic control systems in a similar vein have been successfully verified using model checking techniques stemming from those we discussed here. For example, symbolic model checking in SMV was used to verify the Traffic Alert and Collision Avoidance

System (TCAS II), an air traffic guidance system required on-board large commercial aircraft [164]. Properties checked included the absence of undesirable non-determinism, mutual exclusion, termination, absence of references to undefined parameters, and elimination of inconsistencies in the protocol specification. SMV also enabled verification of the A-7E aircraft software requirements to ensure internal aircraft modes were consistently enabled so that procedures for monitoring, for example, the aircraft's windspeed, velocity, and alignment, accurately contributed to the aircraft's calculated relative position [165]. The Small Aircraft Transportation System (SATS), which is an approach to air traffic management for non-towered non-radar airports in the United States, was model checked to verify the absence of deadlocks, that aircraft separation is maintained, and that the system is robust in the face of unexpected rare events such as equipment failures or aircraft deviating off-course [166]. An early specification for the TSAFE component of our air traffic control example was also model checked to verify the absence of synchronization faults, employing compositional verification techniques [167].

LTL symbolic model checking is ideal for the verification of *reactive systems*, or those systems that operate within a dynamic environment such as concurrent programs, embedded and process control programs, and operating systems. LTL allows us to naturally and organically specify reactive systems in terms of their ongoing behavior. By shifting the focus of the state explosion problem from the size of the state space (as in explicit-state model checking) to the size of the BDD representation, symbolic model checking can be used to verify much larger systems. Though the time complexity bottleneck of the LTL symbolic model checking algorithm is arguably the translation step from φ to $A_{\neg\varphi}$, there are numerous facets of the algorithm worth optimizing, not all of which were mentioned above. Given the promise of this method of formal verification and the real-world verification successes so far, optimization of virtually every segment of this algorithm is a possible subject for future research.

Arguably the most pressing challenge in model checking today is scalability. We must make model-checking tools more efficient, in terms of the size of the models they can reason about and the time and space they require, in order to scale our verification ability to handle real-world safety-critical systems. Currently, LTL symbolic model checking is best used for systems whose state sets have short and easily manipulated descriptions [168]. However, we have previously mentioned that there are some functions with unavoidably exponential symbolic representations.[26] In the future, we expect even more specialized variants of this algorithm to branch out from the common framework we have presented here in order to provide a greater array of options for practical LTL symbolic model checking of real-world systems, including those with characteristics we are currently unable to accommodate.

In the coming years, we expect to see more refined techniques for adding uncertainty in the model such as inaccurate sensor data or models of humans in the loop, abstractions for representing real-valued and infinite-range variables, algorithms that include probabilistic randomization, other extensions that allow probabilistic reasoning similar to rare-event simulation, and more tools for efficient model checking for extensions of LTL such as the industrial logics overviewed in Section 3.2.4. Furthermore, increasing the level of automation in the application of more scalable techniques will be necessary in order to complete large-scale verification efforts in reasonable time-frames. For example, increased automation of the process of partitioning our entire example automated air traffic control architecture for compositional verification would significantly contribute to the success of such a large task.

[26] See [137] for a proof that a BDD representation of a function describing the outputs of an integer multiplier grows exponentially regardless of the variable ordering.

Acknowledgements

Thanks to Moshe Y. Vardi, Eric W. D. Rozier, and Misty D. Davies for insightful comments on earlier drafts of this paper. Thanks also to Heinz Erzberger for enlightening conversations on the automated airspace concept for air traffic control, which made a wonderful running example.

REFERENCES

[1] R.W. Butler, G.B. Finelli, The infeasibility of quantifying the reliability of life-critical real-time software, IEEE TSE 19 (1) (1993) 3–12.
[2] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Proc. Workshop on Logic of Programs, in: LNCS, vol. 131, Springer, 1981, pp. 52–71.
[3] J.P. Queille, J. Sifakis, Specification and verification of concurrent systems in Cesar, in: Proc. 5th Int'l Symp. on Programming, in: LNCS, vol. 137, Springer, 1982, pp. 337–351.
[4] A. Pnueli, The temporal logic of programs, in: FOCS, Proc. 18th IEEE Symp., 1977, pp. 46–57.
[5] M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, in: Proc. 1st LICS, Cambridge, 1986, pp. 332–344.
[6] M.Y. Vardi, Automata-theoretic model checking revisited, in: VMCAI, Proc. 7th Int'l Conf., in: LNCS, vol. 4349, Springer, 2007, pp. 137–150.
[7] O. Bernholtz, M.Y. Vardi, P. Wolper, An automata-theoretic approach to branching-time model checking, in: D.L. Dill (Ed.), CAV, Proc. 6th Int'l Conf., in: LNCS, vol. 818, Springer, Berlin, 1994, pp. 142–155.
[8] G.J. Holzmann, The model checker SPIN, in: Formal Methods in Software Practice, IEEE TSE 23 (5) (1997) 279–295 (special issue).
[9] A. Duret-Lutz, D. Poitrenaud, SPOT: an extensible model checking library using transition-based generalized Büchi automata, in: MASCOTS, Proc. 12th Int'l Workshop, IEEE, 2004, pp. 76–83.
[10] C. Courcoubetis, M.Y. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithms for the verification of temporal properties, Form. Methods Syst. Des. 1 (1992) 275–288.
[11] G.J. Holzmann, D. Peled, M. Yannakakis, On nested depth-first search, in: Proc. 2nd SPIN Workshop, American Math. Soc., 1996, pp. 23–32.
[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, McGraw-Hill Higher Education, 2001.
[13] J.-C. Fernandez, L. Mounier, C. Jard, T. Jeron, On-the-fly verification of finite transition systems, Form. Methods Syst. Des. 1 (1992) 251–273.

[14] J. Geldenhuys, H. Hansen, Larger automata and less work for LTL model checking, in: 13th Int'l SPIN, in: LNCS, vol. 3925, Springer, 2006, pp. 53–70.
[15] X. Thirioux, Simple and efficient translation from LTL formulas to Büchi automata, Electron. Notes Theor. Comput. Sci. 66 (2) (2002) 145–159.
[16] D. Peled, All from one, one for all: on model checking using representatives, in: CAV, Proc. 5th Conf., in: LNCS, Springer, Elounda, 1993.
[17] R. Milner, An algebraic definition of simulation between programs, in: Proc. 2nd Int'l Joint Conf. on Artificial Intelligence, British Computer Society, 1971, pp. 481–489.
[18] C.B. Jones, Specification and design of (parallel) programs, in: R.E.A. Mason (Ed.), Information Processing 83: Proc. IFIP 9th World Congress, IFIP, North-Holland, 1983, pp. 321–332.
[19] K. McMillan, The SMV language. Technical report, Cadence Berkeley Lab, 1999.
[20] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A new symbolic model checker, STTT Int. J. 2 (4) (2000) 410–425.
[21] 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, T. Villa, VIS: a system for verification and synthesis, in: CAV, Proc. 8th Int'l Conf, in: LNCS, vol. 1102, Springer, 1996, pp. 428–432.
[22] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inform. Comput. 98 (2) (1992) 142–170.
[23] E.M. Clarke, O. Grumberg, K. Hamaguchi, Another look at LTL model checking, Form. Methods Syst. Des. 10 (1) (1997) 47–71.
[24] E.A. Emerson, C.L. Lei, Efficient model checking in fragments of the propositional μ-calculus, in: LICS, 1st Symp., Cambridge, Jun 1986, pp. 267–278.
[25] K.L. McMillan, Symbolic model checking: an approach to the state explosion problem, Ph.D. thesis, CMU, Pittsburgh, PA, USA, 1992.
[26] E.M. Clarke, The birth of model checking, in: 25 Years of Model Checking, 2008, pp. 1–26. doi:10.1007/978-3-540-69850-0_1.
[27] J.R. Burch, E.M. Clarke, D.E. Long, K.L. Mcmillan, D.L. Dill, Symbolic model checking for sequential circuit verification, IEEE TCAD 13 (1993) 401–424.
[28] A. Cimatti, F. Giunchiglia, E. Giunchiglia, P. Traverso, Planning via model checking: A decision procedure for AR, in: ECP, pp. 130–142, 1997.
[29] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, in: POPL, Proc. 19th ACM Symp., Albuquerque, New Mexico, Jan 1992, pp. 343–354.
[30] A. Valmari, A stubborn attack on state explosion, in: CAV, Proc. 2nd Conf., in: LNCS, vol. 531, Springer, Rutgers, 1990, pp. 156–165.
[31] P. Wolper, M.Y. Vardi, A.P. Sistla, Reasoning about infinite computation paths, in: FOCS, Proc. 24th IEEE Symp., Tucson, 1983, pp. 185–194.
[32] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: TACAS, in: LNCS, vol. 1579, Springer, 1999.
[33] K.G. Larsen, P. Petterson, W. Yi, UPPAAL: status & developments, in: CAV, Proc. 9th Int'l Conf., in: LNCS, vol. 1254, Springer, 1997, pp. 456–459.
[34] T.A. Henzinger, P-H. Ho, H. Wong-Toi, HYTECH: a model checker for hybrid systems, STTT 1 (1–2) (1997) 110–122.
[35] O. Kupferman, M.Y. Vardi, An automata-theoretic approach to reasoning about infinite-state systems, in: CAV, Proc. 12th Int'l Conf., in: LNCS, vol. 1855, Springer, 2000, pp. 36–52.
[36] M. Chechik, B. Devereux, A. Gurfinkel, Model-checking infinite state-space systems with fine-grained abstractions using SPIN, in: Proc. 8th SPIN, in: LNCS, vol. 2057, Springer, 2001, pp. 16–36.
[37] L. Pike, Real-time system verification by k-induction, Technical Report TM-2005-213751, NASA Langley Research Center, 2005.
[38] L. Lamport, Real-time model checking is really simple, in: CHARME, 2005, pp. 162–175.
[39] G. Lindstrom, P.C. Mehlitz, W. Visser, Model checking real time java using java pathfinder, in: ATVA, vol. 3707, Springer, 2005, pp. 444–456.
[40] V. Schuppan, A. Biere, Liveness checking as safety checking for infinite state spaces, Electron. Notes Theor. Comput. Sci. 149 (1) (2006) 79–96.
[41] H. Erzberger, The automated airspace concept, in: 4th USA/Europe Air Traffic Management R&D Seminar, Santa Fe, New Mexico, USA, Dec. 2001.
[42] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: CAV, Proc. 14th Int'l Conf., in: LNCS, vol. 2404, Springer, 2002, pp. 359–364.
[43] A. Cimatti, R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev, NuSMV 2.4 user manual. Technical report, CMU and ITC-irst, 2005.
[44] F. Raimondi, A. Lomuscio, M.J. Sergot, Towards model checking interpreted systems, in: Proc. of 2nd NASA Workshop on Formal Approaches to Agent-Based Systems, FAABS02, in: LNAI, vol. 2699, Springer, 2002, pp. 115–125.
[45] M. Gribaudo, A. Horvath, A. Bobbio, E. Tronci, E. Ciancamerla, M. Minichino, Model-checking based on fluid petri nets for the temperature control system of the icaro co-generative plant. Technical report, Proc. SAFECOMP, vol. 2434, LNCS, 2002.
[46] A.C. Tribble, S.P. Miller, Software safety analysis of a flight management system vertical navigation function - a status report, in: DASC, 22nd, vol. 1, Oct. 2003, pp. 1.B.1–1.1–9, vol.1.
[47] Y. Choi, M. Heimdahl, Model checking software requirement specifications using domain reduction abstraction, in: ASE, Proc. 18th IEEE Int'l Conf., Oct. 2003, pp. 314–317.
[48] S.P. Miller, A.C. Tribble, M.W. Whalen, M. Per, E. Heimdahl, Proving the shalls, STTT 8 (4–5) (2006) 303–319.
[49] S. Miller, Will this be formal?, in: Theorem Proving in Higher Order Logics, vol. 5170, Springer, 2008, pp. 6–11.
[50] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, A. Tiwari, SAL 2, in: R. Alur, D. Peled (Eds.), CAV, in: LNCS, vol. 3114, Springer, Boston, MA, 2004, pp. 496–500.
[51] R.M. Burstall, Program proving as hand simulation with a little induction, in: Information Processing 74, North-Holland, Stockholm, Sweden, 1974, pp. 308–312. Int'l Fed. for Information Processing.
[52] F Kroger, LAR: a logic of algorithmic reasoning, Acta Inform. 8 (1977) 243–266.
[53] E.M. Clarke, O. Grumberg, D. Long, Verification tools for finite-state concurrent systems, in: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Decade of Concurrency – Reflections and Perspectives (Proc. of REX School), in: LNCS, vol. 803, Springer, 1993, pp. 124–175.
[54] J.A.W. Kamp, Tense Logic and the Theory of Order, Ph.D. Thesis, UCLA, 1968.
[55] M. Ben-Ari, Z. Manna, A. Pnueli, The temporal logic of branching time, in: POPL, Proc. of 8th ACM SIGPLAN-SIGACT Symp., ACM, New York, NY, USA, 1981, pp. 164–176.
[56] O. Lichtenstein, A. Pnueli, Checking that finite state concurrent programs satisfy their linear specification, in: POPL, Proc. 12th ACM Symp., New Orleans, Jan 1985, pp. 97–107.
[57] M.Y. Vardi, From Church and Prior to PSL, in: Proc. of Workshop on 25 Years of Model Checking, 2006.

[58] W. Visser, H. Barringer, Practical CTL* model checking: Should SPIN be extended? STTT 2 (4) (2000) 350–365.
[59] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the temporal analysis of fairness, POPL, Proc. 7th ACM Symp., 1980, pp. 163–173.
[60] E.A. Emerson, Temporal and modal logic, in: J. Van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, MIT Press, 1990, pp. 997–1072 (chapter 16).
[61] M.Y. Vardi, Automata-theoretic techniques for temporal reasoning, in: Frank Wolter, Patrick Blackburn, Johan van Benthem (Eds.), Handbook of Modal Logic, Elsevier, 2006.
[62] L. Lamport, "sometimes" is sometimes "not never" — on the temporal logic of programs, in: POPL. Proc. 7th ACM Symp., Jan 1980, pp. 174–185.
[63] M.Y. Vardi, Branching vs. linear time: Final showdown, in: TACAS, Proc. 7th Int'l Conf., in: LNCS, vol. 2031, Springer, 2001, pp. 1–22.
[64] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM TOPLAS 8 (2) (1986) 244–263.
[65] M.Y. Vardi, P. Wolper, Reasoning about infinite computations, Inform. and Comput. 115 (1) (1994) 1–37.
[66] E.A. Emerson, C.-L. Lei, Modalities for model checking: branching time logic strikes back, in: POPL, 20th ACM Symp., New Orleans, Jan 1985, pp. 84–96.
[67] A.P. Sistla, E.M. Clarke, The complexity of propositional linear temporal logic, J. ACM 32 (1985) 733–749.
[68] M.J. Fischer, R.E. Ladner, Propositional dynamic logic of regular programs, J. Comput. Syst. Sci. 18 (1979) 194–211.
[69] E.A. Emerson, J.Y. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, J. Comput. Syst. Sci. 30 (1985) 1–24.
[70] M.Y. Vardi, Linear vs. branching time: a complexity-theoretic perspective, in: LICS, Proc. 13th IEEE Symp., 1998, pp. 394–405.
[71] M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge U. Press, New York, NY, USA, 2004.
[72] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.
[73] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, Systems and Software Verification: Model-Checking Techniques and Tools, Springer, 2001.
[74] D. Giannakopoulou, Model Checking for Concurrent Software Architectures. Ph.D. Thesis, Imperial College of Science, Technology, and Medicine, U. of London, Jan 1999.
[75] E.A. Emerson, J.Y. Halpern, Sometimes and not never revisited: on branching versus linear time, J. ACM 33 (1) (1986) 151–178.
[76] M.Y. Vardi, L. Stockmeyer, Improved upper and lower bounds for modal logics of programs, in: STOC, Proc. 17 ACM, 1985, pp. 240–251.
[77] E.A. Emerson, C. Jutla, The complexity of tree automata and logics of programs, in: FOCS, 29th IEEE Symp., White Plains, Oct 1988, pp. 328–337.
[78] E.A. Emerson, A.P. Sistla, Deciding branching time logic, in: STOC, Proc. 16th ACM, Washington, Apr 1984, pp. 14–24.
[79] P. Wolper, Temporal logic can be more expressive, Inf. Control 56 (1–2) (1983) 72–99.
[80] J.-M. Couvreur, Un point de vue symbolique sur la logique temporelle linéaire, in: Pub. du LaCIM, in: Actes du Colloque LaCIM 2000, Univ. du Québec à Montréal, Aug 2000, vol. 27, pp. 131–140.
[81] B. Banieqbal, H. Barringer, Temporal logic with fixed points, in: B. Banieqbal, H. Barringer, A. Pnueli (Eds.), Temporal Logic in Specification, in: LNCS, vol. 398, Springer, 1987, pp. 62–74.
[82] M.Y. Vardi, A temporal fixpoint calculus, in: POPL, Proc. 15th ACM Symp., San Diego, 1988, pp. 250–259.
[83] A.P. Sistla, M.Y. Vardi, P. Wolper, The complementation problem for Büchi automata with applications to temporal logic, Theoret. Comput. Sci. 49 (1987) 217–237.
[84] E.A. Emerson, R.J. Trefler, Generalized quantitative temporal reasoning: An automata theoretic approach, in: TAPSOFT, Proc., in: LNCS, vol. 1214, Springer, 1997, pp. 189–200.
[85] L. Fix, Fifteen years of formal property verification at intel. This Volume, 2007.
[86] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, Y. Zbar, The ForSpec temporal logic: A new temporal property-specification logic, in: TACAS, Proc. 8th Int'l, in: LNCS, vol. 2280, Springer, 2002, pp. 296–311.
[87] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, Y. Rodeh, The temporal logic Sugar, in: CAV, Proc. 13th Int'l Conf., Paris, France, July 2001, in: LNCS, vol. 2102, Springer, 2001, pp. 363–367.
[88] C. Eisner, D. Fisman, Sugar 2.0 proposal presented to the accellera formal verification technical committee, 2002.
[89] M.J. Morley, Semantics of temporal ***e***, in: T.F. Melham, F.G. Moller (Eds.), BANFF Higher Order Workshop. U. of Glasgow, Dept. of Computing Sci. Tech. Report, 1999.
[90] S. Vijayaraghavan, M. Ramanathan, A practical guide for SystemVerilog assertions, 2005.
[91] J. Havlicek, Y. Wolfsthal, PSL and SVA: Two standard assertion languages addressing complimentary engineering needs, in: Proc. Design Verification Conf., Feb. 2005.
[92] M.Y. Vardi, From monadic logic to PSL, in: Arnon Avron, Nachum Dershowitz, Alexander Rabinovich (Eds.), Pillars of Comp. Sci., in: LNCS, vol. 4800, Springer, 2008, pp. 656–681.
[93] G. Rosu, S. Bensalem, Allen Linear (interval) Temporal Logic — translation to LTL and monitor synthesis, in: CAV, 2006, pp. 263–277.
[94] J.F. Allen, Towards a general theory of action and time, Artif. Intell. 23 (2) (1984) 123–154.
[95] O. Kupferman, Sanity checks in formal verification, in: CONCUR, Proc. 17th Int'l Conf., in: LNCS, vol. 4137, Springer, 2006, pp. 37–51.
[96] R.P. Kurshan, FormalCheck User's Manual, Cadence Design, Inc., 1998.
[97] G. Ammons, D. Mandelin, R. Bodik, J.R. Larus, Debugging temporal specifications with concept analysis, PLDI, Proc. ACM Conf. (2003) 182–195.
[98] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, M.Y. Vardi, Enhanced vacuity detection for linear temporal logic, in: CAV, Proc 15th Int'l Conf, Springer, 2003.
[99] I. Beer, S. Ben-David, C. Eisner, Y. Rodeh, Efficient detection of vacuity in ACTL formulas, Form. Methods Syst. Des. 18 (2) (2001) 141–162.
[100] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, M.Y. Vardi, Regular vacuity, in: CHARME, in: LNCS, vol. 3725, Springer, 2005, pp. 191–206.
[101] A. Gurfinkel, M. Chechik, How vacuous is vacuous, in: TACAS, 10th Int'l Conf, in: LNCS, vol. 2988, Springer, 2004, pp. 451–466.
[102] A. Gurfinkel, M. Chechik, Extending extended vacuity, in: FMCAD, 5th Int'l Conf, in: LNCS, vol. 3312, Springer, 2004, pp. 306–321.
[103] O. Kupferman, M.Y. Vardi, Vacuity detection in temporal model checking, STTT 4 (2) (2003) 224–233.
[104] K.S. Namjoshi, An efficiently checkable, proof-based formulation of vacuity in model checking, in: 16th CAV, in: LNCS, vol. 3114, Springer, 2004, pp. 57–69.
[105] M. Purandare, F. Somenzi, Vacuum cleaning CTL formulae, in: CAV, Proc. 14th Conf., in: LNCS, Springer, 2002, pp. 485–499.
[106] K.Y. Rozier, M.Y. Vardi, LTL satisfiability checking, in: Proc. 14th SPIN, in: LNCS, vol. 4595, Springer, 2007, pp. 149–167.

[107] O. Kupferman, M.Y. Vardi, P. Wolper, An automata-theoretic approach to branching-time model checking, J. ACM 47 (2) (2000) 312–360.
[108] 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, Stanford U. Press, Stanford, 1962, pp. 1–12.
[109] J.-M. Couvreur, On-the-fly verification of linear temporal logic, in Proc. FM, 1999, pp. 253–271.
[110] M.Y. Vardi, An automata-theoretic approach to linear temporal logic, in: F. Moller, G. Birtwistle (Eds.), Logics for Concurrency: Structure versus Automata, in: LNCS, vol. 1043, Springer, Berlin, 1996, pp. 238–266.
[111] A.P. Sistla, M.Y. Vardi, P. Wolper, The complementation problem for Büchi automata with applications to temporal logic, in: ICALP, Proc. 10th, in: LNCS, vol. 194, Springer, Nafplion, 1985, pp. 465–474.
[112] S. Safra, M.Y. Vardi, On ω-automata and temporal logic, in: STOC, Proc. 21st ACM, Seattle, May 1989, pp. 127–137.
[113] R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: P. Dembiski, M. Sredniawa (Eds.), PSTV, Chapman & Hall, 1995.
[114] N. Daniele, F. Guinchiglia, M.Y. Vardi, Improved automata generation for linear temporal logic, in: CAV, Proc. 11th Int'l Conf, in: LNCS, vol. 1633, Springer, 1999, pp. 249–260.
[115] F. Somenzi, R. Bloem, Efficient Büchi automata from LTL formulae, in: CAV, Proc. 12th Int'l Conf, in: LNCS, vol. 1855, Springer, 2000, pp. 248–263.
[116] P. Gastin, D. Oddoux, Fast LTL to Büchi automata translation, in: CAV, Proc. 13th Int'l Conf, in: LNCS, vol. 2102, Springer, 2001, pp. 53–65.
[117] D. Giannakopoulou, 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.
[118] R. Sebastiani, S. Tonetta, More deterministic vs. smaller" Büchi automata for efficient LTL model checking, in: CHARME, Springer, 2003, pp. 126–140.
[119] C. Fritz, Concepts of automata construction from LTL, in: LPAR, Proc. 12th Int'l Conf., in: LNCS, vol. 3835, Springer, 2005, pp. 728–742.
[120] L. Lamport, Proving the correctness of multiprocess programs, IEEE TSE 3 (2) (1977) 125–143.
[121] S. Owicki, L. Lamport, Proving liveness properties of concurrent programs, ACM TOPLAS 4 (3) (1982) 455–495.
[122] O. Kupferman, M.Y. Vardi, Model checking of safety properties, in: CAV, Proc. 11th Int'l Conf., in: LNCS, vol. 1633, Springer, 1999, pp. 172–183.
[123] B. Alpern, F.B. Schneider, Recognizing safety and liveness, Distrib. Comput. 2 (1987) 117–126.
[124] E. Kindler, Safety and liveness properties: a survey, Bull. EATCS 53 (1994) 268–272.
[125] M.W. Alford, J.P. Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery, F.B. Schneider, Dist. Systems: Methods and Tools for Specification. An Advanced Course, Springer, NY, USA, 1985.
[126] A.P. Sistla, Safety, liveness, and fairness in temporal logic, Form. Asp. Comput. 6 (1994) 495–511.
[127] B. Alpern, F.B. Schneider, Defining liveness, Inform. Process. Lett. 21 (1985) 181–185.
[128] R.P. Kurshan, CAV of Coordinating Processes, Princeton Univ. Press, 1994.
[129] S. Safra, On the complexity of ω-automata, in: FOCS, Proc. 29th IEEE Symp., White Plains, 1988, pp. 319–327.
[130] O. Kupferman, M.Y. Vardi, Weak alternating automata are not that weak, in: ISTCS, Proc. 5th, IEEE Press, 1997, pp. 147–158.
[131] L. Doyen, J.-F. Raskin, Improved algorithms for the automata-based approach to model-checking, in: TACAS, in: LNCS, vol. 4424, Springer, 2007, pp. 451–465.
[132] S. Fogarty, M.Y. Vardi, Büchi complementation and size-change termination, in: TACAS, Proc. of 15th Int'l Conf., Springer, Berlin, Heidelberg, 2009, pp. 16–30.
[133] A. Biere, A. Cimatti, E. Clarke, O. Strichman, Y. Zhu, Bounded model checking, Adv. Comput. 58 (2003).
[134] C. Lee, Representation of switching circuits by binary decision programs, Bell Syst. Tech. J. 38 (1959) 985–999.
[135] S.B. Akers, Binary decision diagrams, IEEE TC C-27 (6) (1978) 509–516.
[136] S. Fortune, J.E. Hopcroft, E.M. Schmidt, The complexity of equivalence and containment for free single variable program schemes, in: ICALP, Proc. 5th, Springer, London, UK, 1978, pp. 227–240.
[137] R.E. Bryant, Graph-based algorithms for Boolean-function manipulation, IEEE TC C-35 (8) (1986) 677–691.
[138] R.E. Bryant, Symbolic Boolean manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24 (3) (1992) 293–318.
[139] P. Linz, An Introduction to Formal Languages and Automata, 2nd ed., D. C. Heath and Company, Lexington, MA, USA, 1996 (Chapter 1–4).
[140] P Gribomont, P Wolper, in: A. Thayse (Ed.), Temporal logic, in from modal logic to deductive databases, 1989.
[141] M. Michel, Complementation is more difficult with automata on infinite words, in: CNET, Paris, 1988.
[142] J. Esparza, Decidability of model checking for infinite-state concurrent systems, Acta Inform. 34 (1997) 85–107.
[143] S. Minato, N. Ishiura, S. Yajima, Shared binary decision diagram with attributed edges for efficient Boolean function manipulation, in: DAC, Proc. 27th ACM/IEEE Conf., 1990, pp. 52–57.
[144] G.D. Hachtel, E. Macii, A. Pardo, F. Somenzi, Markovian analysis of large finite state machines, IEEE Trans. Comput. Aided. Des. 15 (1996) 1479–1493.
[145] M. Chechik, S. Easterbrook, B. Devereux, Model checking with multi-valued temporal logics, in: ISMVL, 2000, pp. 187–192.
[146] Henrik Reif Andersen, An Introduction to Binary Decision Diagrams, Internet, Sept 1996.
[147] D. Sieling, I. Wegener, Reduction of OBDDs in linear time, Inform. Process. Lett. 48 (3) (1993) 139–144.
[148] Y. Kesten, A. Pnueli, L. Raviv, Algorithmic verification of linear temporal logic specifications, in: ICALP, Proc. 25th, in: LNCS, vol. 1443, Springer, 1998, pp. 1–16.
[149] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999 (Chapter 1–6, 8–9).
[150] R.S. Streett, Propositional dynamic logic of looping and converse, Inf. Control 54 (1982) 121–141.
[151] N. Francez, D. Kozen, Generalized fair termination, in: POPL, 1984, pp. 46–53.
[152] E.M. Clarke, O. Grumberg, K.L. McMillan, X. Zhao, Efficient generation of counterexamples and witnesses in symbolic model checking, in: DAC, Proc. 32nd, IEEE, 1995, pp. 427–432.
[153] R. Hojati, R.K. Brayton, R.P. Kurshan, BDD-based debugging of design using language containment and fair CTL, in: CAV, Springer-Verlag, London, UK, 1993, pp. 41–58.
[154] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Safety, Springer, New York, 1995.
[155] K. Ravi, R. Bloem, F. Somenzi, A comparative study of symbolic algorithms for the computation of fair cycles, in: FMCAD, Proc. Intl. Conf., in: LNCS, vol. 1954, Springer, 2000, pp. 143–160.
[156] R. Hojati, H. Touati, R. Kurshan, R. Brayton, Efficient ω-regular language containment, in: CAV, Proc. 4th Int'l Conf., in: LNCS, vol. 663, Springer, 1992.
[157] R.H. Hardin, R.P. Kurshan, S.K. Shukla, M.Y. Vardi, A new heuristic for bad cycle detection using BDDs, in: CAV, Proc. 9th Int'l Conf., in: LNCS, vol. 1254, Springer, 1997, pp. 268–278.

[158] A. Xie, P. Beerel, Implicit enumeration of strongly connected components, in: ICCAD, ACM, IEEE Press, San Jose, CA, USA, 1999, pp. 37–40.
[159] A. Xie, P.A. Beerel, Implicit enumeration of strongly connected components and an application to formal verification, IEEE Trans. Comput. Aided. Des. 19 (10) (2000) 1225–1230.
[160] R. Bloem, H.N. Gabow, F. Somenzi, An algorithm for strongly connected component analysis in $n \log n$ symbolic steps, in: FMCAD, in: LNCS, vol. 1954, Springer, 2000, pp. 37–54.
[161] K. Fisler, R. Fraer, G. Kamhi, M.Y. Vardi, Z. Yang, Is there a best symbolic cycle-detection algorithm? in: TACAS, 7th Int'l Conf., in: LNCS, vol. 2031, Springer, 2001, pp. 420–434.
[162] S. Ben-David, J. Pound, R.J. Trefler, D. Tsarkov, G.E. Weddell, Fair cycle detection using description logic reasoning, in: Description Logics, 2009.
[163] Cadence. SMV. http://www.cadence.com/company/cadence_labs_research.html.
[164] R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, J.D. Reese, Model checking large software specifications, IEEE TSE 24 (1996) 156–166.
[165] T. Sreemani, J.M. Atlee, Feasibility of model checking software requirements: a case study, in: COMPASS, IEEE, 1996, pp. 77–88.
[166] C. Muñoz, V. Carreño, G. Dowek, Formal analysis of the operational concept for the small aircraft transportation system, in: Rigorous Engineering of Fault-Tolerant Systems, in: LNCS, vol. 4157, 2006, pp. 306–325.
[167] A. Betin Can, T. Bultan, M. Lindvall, B. Lux, S. Topp, Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers, Autom. Softw. Eng. 14 (2) (2007) 129–178.
[168] P. Wolper, An introduction to model checking, in: Proc. of Software Quality Week, SQW'95, San Francisco, CA, 1995.
[169] D. Sieling, The nonapproximability of OBDD minimization, Inform. and Comput. 172 (1998) 103–138.

Kristin Y. Rozier is a Research Computer Scientist at NASA Ames Research Center. Kristin has been a member of the Robust Software Engineering (RSE) team since September, 2008, a member of the NASA Formal Methods (NFM) research group since November, 2003, and a NASA employee since May, 2001. She holds a B.S. and an M.S. from the College of William and Mary and is a Ph.D. candidate at Rice University under the advisement of Moshe Y. Vardi. Kristin is a member of PBK, ACM, IEEE, SWE, Systers, and AIAA, where she serves on the Intelligent Systems Technical Committee.