Benedikt **Löwe**, Florian **Rudolph** (*eds.*),
Foundations of the Formal Sciences
Refereed Papers of a Research Colloquium,
Humboldt–Universität zu Berlin, May 7-9, 1999, p. 147–167

MODEL CHECKING TECHNIQUES FOR THE ANALYSIS OF REACTIVE SYSTEMS

STEPHAN MERZ

ABSTRACT. Model checking is nowadays being used routinely to design and debug reactive systems. This paper gives an overview on the theory and algorithms used for model checking, with a bias towards automata-theoretic approaches and linear-time temporal logic. We also describe elementary abstraction techniques that can be useful for large systems that cannot be directly handled by model checking.

1. THE TOPIC OF THIS SURVEY

Reactive systems such as process controllers, operating systems or communication networks maintain an ongoing interaction with their environment. Usually, such systems comprise several components that operate concurrently. This inherent concurrency makes reactive systems notoriously error-prone, due to race conditions or deadlocks that are difficult to find and reproduce by conventional testing. On the other hand, reactive systems find applications in fields where errors are expensive or may even endanger human lives. Therefore, industry is willing to apply (and pay for) formal methods to model and analyse hard- and software components for reactive systems. In particular, so called model checking technology [ClEmSi86, QueiSi81], which can automatically analyse finite-state models of reactive systems has found wide acceptance as a debugging tool.

Traditionally, from finite automata to Turing machines, computing devices are modelled as mapping inputs to outputs. Such models are not well-suited for reactive systems, which continuously receive input from their environment.Instead, reactive systems are better modelled as transition systems that communicate via messages or via shared variables. This survey paper explains the principles of the algorithmic analysis of finite-state models of reactive systems, which rely on results in logic and automata theory as well as on efficient data structures and algorithms. Model checking can be applied to models of rather large and industrially relevant systems [ClGrHi93]. Still, even a state space of 10^{120} states, which has sometimes been claimed to be manageable using symbolic model checking, is small when compared to present-day processors with a few dozen or even a hundred registers and several megabyte of cache memory. In these cases, smaller models have to

RECEIVED : August 9th, 1999

be constructed that can be handled automatically, and that allow the properties of interest to be established. Abstraction techniques help to support this state-space reduction by computer-assisted tools, although some user interaction is necessary to identify the abstraction function or the predicates of interest. Another important technique is the decomposition of systems into relatively independent subsystems whose properties can be established by model checking, and which are combined to yield properties of the entire system [McM97]. This, however, is out of the scope of this survey.

The plan of the paper is as follows: section 2 introduces transition systems, temporal logic, and ω-automata, and explores the relations between these concepts. It provides the theoretical background for the presentation of **PTL** model checking in section 3. The analysis of a cryptographic protocol gives an indication how the technique can be applied in practice. In section 4 we consider abstraction techniques, which enable large, and even infinite-state, systems to be reduced to tractable models. Our objective throughout is to give a coherent exposition of some central material, biased by personal prejudice and experience, and complemented by references to alternative approaches.

2. Temporal logic and automata

Reactive systems can be formally represented as transition systems, and their properties are conveniently expressed as temporal logic formulas. This section defines these concepts and presents some fundamental theory that is applied in the analysis of reactive systems.

2.1. **Transition systems.**

Definition 1. *A transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta, \alpha)$ is given by a set S of* states*, a non-empty subset $I \subseteq S$ of which are* initial*, a set $\mathcal{A}$ of* actions*, a total* transition relation *$\delta \subseteq S \times S$ (that is, for every $s \in S$ there is some $t \in S$ related by δ), and an* action labelling *$\alpha : \delta \to 2^{\mathcal{A}}$ that associates a non-empty set of actions with every pair of states $(s, t) \in \delta$.*

A run *of $\mathcal{T}$ is an infinite sequence $\rho = r_0 r_1 \ldots$ of states $r_i \in S$ such that $r_0 \in I$ and $(r_i, r_{i+1}) \in \delta$ holds for all $i \in \mathbb{N}$.*

Intuitively, a transition system defines the possible evolutions of a system, starting from an initial state. The literature is abundant with variations of the definition given above. For example, we have assumed that the transition relation is total to simplify some definitions below; this can be achieved by adding a "stuttering" transition that does not change the state. Only the stuttering transition is enabled in deadlocked states. Frequently, *fairness conditions* are associated with actions to ensure that actions are eventually executed provided they are often enough enabled. In this paper we mainly consider the linear-time temporal logic **PTL** where fairness assumptions can be expressed as formulas, hence there is no need to complicate the definition of a transition system.

In practice, reactive systems are not described as monolithic transition systems, but as composed of *processes* that interact with each other and the environment. The entire transition system can then be obtained as the product of simpler transition systems associated with the individual processes. This additional structure can be useful for optimizations such as the partial-order reduction methods described in section 3.2.

2.2. **Temporal logic.** A property of a reactive system can be defined semantically as a set of *behaviors* (infinite sequences of states), which represent those executions of the system that satisfy the property. It is convenient to express properties in a suitable logic. In this paper we will mainly consider linear-time temporal logic [Pn77, MaPn92, Krö87].

Assume given a denumerable set $\mathcal{V}$ of atomic propositions, which represent properties of individual states. For example, req_1 and $owns_1$ could be atomic propositions in a resource manager that indicate that at the present state there is a pending request for the resource from process 1, and that process 1 currently "owns" the resource.

Definition 2. *Formulas of the propositional linear-time temporal logic* **PTL** *are inductively defined as follows:*

- *Every atomic proposition $v \in \mathcal{V}$ is a formula.*
- *Boolean combinations of formulas are formulas.*
- *If φ and ψ are formulas then $\circ\varphi$ (read:* next φ*) and φ* **until** *ψ are formulas.*

PTL formulas are interpreted over ω-sequences of states, which interpret the atomic propositions in $\mathcal{V}$. Formally, a state is a function $s : \mathcal{V} \to \{tt, ff\}$ (where *tt* and *ff* are the Booleans true and false), and a behavior is a sequence $\sigma = s_0 s_1 \ldots$ of states. We let $\sigma_i = s_i$ denote the i-th state of behavior σ (counting from 0), and $\sigma|_i = s_i s_{i+1} \ldots$, the suffix of behavior σ with the first i states omitted.

Definition 3. *The relation $\sigma \models \varphi$ (read: φ* holds of *σ) is inductively defined as follows:*

- *$\sigma \models v$ (for $v \in \mathcal{V}$) iff $\sigma_0(v) = tt$.*
- *The semantics of boolean combinations is defined in the usual way.*
- *$\sigma \models \circ\varphi$ iff $\sigma|_1 \models \varphi$.*
- *$\sigma \models \varphi$* **until** *ψ iff for some $i \geq 0$, $\sigma|_i \models \psi$ and $\sigma|_j \models \varphi$ holds for $0 \leq j < i$.*

Useful **PTL** formulas can be defined as abbreviations: $\Diamond\varphi$ (*eventually* φ) is defined as **true until** φ and asserts that φ holds for some suffix. The dual formula $\Box\varphi$ (*always* φ), defined as $\neg\Diamond\neg\varphi$, stipulates that φ holds for every suffix. Finally, φ **unless** ψ is defined as (φ **until** ψ) $\vee \Box\varphi$; it requires that φ holds for as long as ψ does not hold, but does not require ψ to hold eventually.

PTL formulas can be used to assert properties about the accessibility of states and their relative ordering. As examples, consider the following properties of a resource manager for two processes:

$\Box\neg(owns_1 \wedge owns_2)$**:** It is never the case that both processes own the resource; in other words, the resource is exclusive. In general, formulas of the form $\Box P$ where P is non-temporal, express *invariants* of the system.

$\Box(req_1 \Rightarrow \Diamond owns_1)$**:** Every request by process 1 will eventually be honored. Note that in the presence of the invariance property above, this property is only implementable if process 2 cooperates by eventually releasing the resource whenever process 1 requests it. In general, formulas of the form $\Box(P \Rightarrow \Diamond Q)$ for non-temporal formulas P and Q express *response properties*.

$\Box\Diamond(req_1 \wedge \neg(owns_1 \vee owns_2)) \Rightarrow \Box\Diamond owns_1$**:** If it is infinitely often the case that process 1 has requested the resource and the resource is free then infinitely often process 1 owns the resource. This formula expresses (strong) fairness for granting the resource to process 1.

$\Box(req_1 \wedge req_2 \Rightarrow \neg owns_2$ **unless** $(owns_2$ **unless** $(\neg owns_2$ **unless** $owns_1)))$**:**
Whenever both processes compete for the resource, process 2 will be granted the resource at most once before it is granted to process 1. This property, known as "1-bounded overtaking", is an example for a *precedence property*.

PTL formulas can be interpreted over runs of transition systems once the interpretation of atomic formulas at system states has been fixed. We say that formula φ holds of transition system $\mathcal{T}$ (or that $\mathcal{T}$ is a model of φ) if $\sigma \models \varphi$ holds for all runs σ of $\mathcal{T}$.

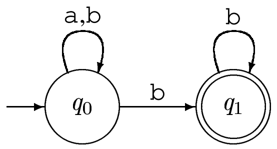

FIGURE 1. A Büchi automaton

2.3. ω**-Automata.** Not only can **PTL** formulas be interpreted over transition systems, but one can also construct an automaton (over ω-words) that characterizes any given **PTL** formula. We present some basic elements of the theory of ω-automata that was initiated by the works of Büchi [Bü62] and Muller [Mu63]. Much more material on this subject can be found in the excellent survey papers by Thomas [Th90, Th97].

Definition 4. *A* Büchi automaton $\mathcal{B} = (Q, I, \delta, F)$ *over an alphabet* Σ *is given by a finite set* Q *of* states*, a non-empty set* $I \subseteq Q$ *of* initial states*, a* transition relation $\delta \subseteq Q \times \Sigma \times Q$*, and a set* $F \subseteq Q$ *of* accepting states*.*

A run *of* $\mathcal{B}$ *over an* ω*-word* $w = w_0 w_1 \ldots \in \Sigma^\omega$ *is an infinite sequence* $\rho = r_0 r_1 \ldots$ *of states* $r_i \in Q$ *such that* $r_0 = q_0$ *and* $(r_i, w_i, r_{i+1}) \in \delta$ *holds for all* $i \in \mathbb{N}$*. The run* ρ *is* accepting *iff there exists some* $q \in F$ *such that* $r_i = q$ *holds for infinitely many* $i \in \mathbb{N}$*.*

The language $\mathcal{L}(\mathcal{B})$ *defined by* $\mathcal{B}$ *is the set of* ω*-words for which there exists some accepting run of* $\mathcal{B}$*. An* ω*-language* $L \subseteq \Sigma^\omega$ *is called* ω-regular *iff* $L = \mathcal{L}(\mathcal{B})$ *for some Büchi automaton* $\mathcal{B}$*.*

Thus, Büchi automata are a straightforward generalization of finite automata over finite words [HopUll79]: their structure is identical, but the acceptance condition changes. For our purposes it is important that the emptiness problem for Büchi automata is decidable.

Proposition 5. *For a Büchi automaton* $\mathcal{B} = (Q, I, \delta, F)$ *with* n *states, it is decidable in time* $O(n)$ *whether* $\mathcal{L}(\mathcal{B}) = \emptyset$*.*

Proof. Obviously, we have that $\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff there exist $q_0 \in I$, $q \in F$, $x \in \Sigma^*$, and $y \in \Sigma^+$ such that $q_0 \stackrel{x}{\Rightarrow} q$ and $q \stackrel{y}{\Rightarrow} q$ (where $q \stackrel{w}{\Rightarrow} q'$ means that there is a path in $\mathcal{B}$ from state q to q' labelled with w). The existence of such paths can efficiently be decided using Tarjan's algorithm which enumerates the strongly connected subcomponents of $\mathcal{B}$ that are reachable from states in I, and checking whether they contain some accepting state. QED

Many other results about finite automata operating on finite words carry over to Büchi automata, the notable exception being that deterministic Büchi automata are strictly weaker than nondeterministic ones. For example, there is no deterministic Büchi automaton equivalent to the automaton shown in figure 1 (with initial state q_0 and accepting state q_1), which accepts precisely those words in $\{\mathtt{a}, \mathtt{b}\}^\omega$ that contain only finitely many a's [Th90]. Therefore, the standard method of proving that the complement of a regular language is again regular, which first constructs a deterministic automaton that is easily complemented, does not apply to Büchi automata. Nevertheless, the class of ω-regular languages is closed under complementation (see [Th97, Th∞] for expositions of different proof strategies). The following result is due to Safra [Sa88], who gave an explicit construction of essentially optimal complexity.

Proposition 6. *For a Büchi automaton* $\mathcal{B}$ *over* Σ *with* n *states there is a Büchi automaton* $\overline{\mathcal{B}}$ *with* $2^{O(n \log n)}$ *states such that* $\mathcal{L}(\overline{\mathcal{B}}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$*.*

Generalized Büchi automata [VaWo94] define the acceptance condition via a (finite) set $\mathcal{F} = \{F_1, \ldots, F_n\}$ of sets of states, and require an accepting run to contain some state from each F_i infinitely often. Simple automata-theoretic manipulations allow to construct an equivalent Büchi automaton for a given generalized Büchi automaton, and the results obtained for Büchi automata carry over essentially unchanged to generalized Büchi automata. In particular, testing for nonemptiness requires checking that some strongly connected subcomponent reachable from some initial state contains some state from every F_i.

Other types of ω-automata have been defined, which are sometimes exponentially more succinct than Büchi automata. These include Muller, Rabin, and Streett automata, which have more elaborate acceptance conditions (such as requiring that every state in some set $R \subseteq Q$ be eventually followed by some state from $G \subseteq Q$). Deterministic Rabin and Streett automata are at the heart of Safra's complementation proof for Büchi automata.

2.4. **From PTL to Büchi automata.** Behaviors can be viewed as ω-words over the alphabet $2^{\mathcal{V}}$ (which is finite if $\mathcal{V}$ is chosen as a finite set, such as the set of atomic propositions that occur in a given **PTL** formula). From this perspective, **PTL** formulas and Büchi automata both describe ω-languages, and it is natural to compare the expressiveness of the two formalisms. For example, the Büchi automaton of figure 1 represents the same requirement as the **PTL** formula $\Diamond\Box\mathtt{b}$.

We now show that for every **PTL** formula φ built from the finite set $\mathcal{V}$ of atomic propositions there is a (generalized) Büchi automaton $\mathcal{A}_\varphi$ over $2^{\mathcal{V}}$ that accepts precisely those behaviors for which φ holds. Define the *Fischer-Ladner closure* $\mathcal{C}(\varphi)$ of φ as the set of subformulas of φ and their negations, identifying $\neg\neg\psi$ and ψ. The states of $\mathcal{A}_\varphi$ are sets of formulas in $\mathcal{C}(\varphi)$ with the intuition that in every accepting run of $\mathcal{A}_\varphi$ that passes through some state q, the formulas $\psi \in q$ will be satisfied in the suffix of the behavior. Formally, the states of $\mathcal{A}_\varphi$ are maximal subsets of $\mathcal{C}(\varphi)$ that satisfy the following *healthiness conditions*:

- $\psi \in q$ iff $\neg\psi \notin q$, for all $\psi \in \mathcal{C}(\varphi)$.
- $\psi_1 \vee \psi_2 \in q$ iff $\psi_1 \in q$ or $\psi_2 \in q$, whenever $\psi_1 \vee \psi_2 \in \mathcal{C}(\varphi)$.
- Conditions for other boolean connectives are similar.
- If ψ_1 **until** $\psi_2 \in q$ then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\neg(\psi_1$ **until** $\psi_2) \in q$ then $\psi_2 \notin q$.

The initial states of $\mathcal{A}_\varphi$ are precisely those states that contain φ.
The transition relation δ of $\mathcal{A}_\varphi$ is defined such that $(q, a, q') \in \delta$ iff

1. $a = q \cap \mathcal{V}$ is the set of atomic formulas that occur in q (and must therefore hold immediately in every suffix that satisfies the formulas from q).
2. q' contains ψ (resp., $\neg\psi$) whenever q contains $\circ\psi$ (resp., $\neg\circ\psi$).
3. If ψ_1 **until** $\psi_2 \in q$ and $\psi_2 \notin q$ then ψ_1 **until** $\psi_2 \in q'$.
4. If $\neg(\psi_1$ **until** $\psi_2) \in q$ and $\psi_1 \in q$ then $\neg(\psi_1$ **until** $\psi_2) \in q'$.

The healthiness and next-state conditions for the **until** operator are motivated by the "recursion law"

$$\psi_1 \textbf{ until } \psi_2 \quad \Leftrightarrow \quad \psi_2 \vee (\psi_1 \wedge \circ(\psi_1 \textbf{ until } \psi_2))$$

It remains to define the acceptance condition of $\mathcal{A}_\varphi$ which ensure that ψ_2 will eventually be satisfied whenever some state promises satisfaction of ψ_1 **until** ψ_2. (The healthiness conditions and next-state relation alone only ensure that ψ_1 will be satisfied as long as ψ_2 is not.) Let ψ_1^1 **until** $\psi_2^1, \ldots, \psi_1^k$ **until** ψ_2^k be all subformulas of this form that occur in

$\mathcal{C}(\varphi)$. The acceptance condition $\mathcal{F}$ is defined as $\{F_1, \ldots, F_k\}$ where F_i is the set of all states of $\mathcal{A}_\varphi$ that either do not contain ψ_1^i **until** ψ_2^i or contain the formula ψ_2^i.

This construction implies the existence of a characteristic Büchi automaton for any given **PTL** formula. The following proposition is due to [LiPnZu85, VaWo94].

Proposition 7. *For every* **PTL** *formula φ of length n there exists a Büchi automaton with $2^{O(n)}$ states that accepts precisely the behaviors of which φ holds.*

Combining propositions 5 and 7, it follows that the satisfiability problem for **PTL** is solvable in exponential time. Although this result is asymptotically optimal (more precisely, the satisfiability problem for **PTL** is PSPACE-complete [Sis85], the construction outlined above inevitably yields a Büchi automaton of exponential size. Constructions that often avoid this exponential blow-up have been investigated [GePeVaWo95, DanGiuVa99] and are being used in practical implementations.

As an aside, Büchi automata or ω-regular languages are actually more expressive than **PTL** formulas. To attain the same level of expressiveness (which, by Büchi's theorem, is that of monadic second order logic of linear orders interpreted over the natural numbers), **PTL** can be augmented by so-called "automaton operators" [Wo83], by fixed-point definitions [St92] or by quantification over propositional variables. However, correctness properties for practical applications can usually be formulated in **PTL**, and few tools [Kl98] support the more expressive variants.

2.5. **Variations on the theme.** *Branching-time temporal logics* [BeHaPn83, Em80, Em90] are interpreted over an entire transition system (or, equivalently, over the tree of paths obtained by unwinding the transition system) rather than over its runs. They combine temporal connectives such as $\circ$ and **until** with quantifiers over the set of paths that start at the current state in the underlying transition system. In particular, one can express *possibility properties* such as "every request by process 1 may eventually be honored". One popular branching-time temporal logic is the Computation Tree Logic **CTL**, whose formulas are inductively defined as follows:

- Every atomic proposition $v \in \mathcal{V}$ is a formula.
- Boolean combinations of formulas are formulas.
- If φ and ψ are formulas then $\exists\circ\varphi$, $\exists\Box\varphi$, and $\exists(\varphi$ **until** $\psi)$ are formulas.

CTL formulas are interpreted at a state s of an infinite tree $\mathcal{T}$, as follows:

- $\mathcal{T}, s \models v$ (for $v \in \mathcal{V}$) iff $s(v) = tt$.
- The semantics of boolean combinations is defined in the usual way.
- $\mathcal{T}, s \models \exists\circ\varphi$ iff $\mathcal{T}, t \models \varphi$ for some son t of s in $\mathcal{T}$.
- $\mathcal{T}, s \models \exists\Box\varphi$ iff for some path $s = s_0, s_1, \ldots$ in $\mathcal{T}$, $\mathcal{T}, s_i \models \varphi$ holds for all $i \geq 0$.
- $\mathcal{T}, s \models \exists(\varphi$ **until** $\psi)$ iff there exist a path $s = s_0, s_1, \ldots$ in $\mathcal{T}$ and some $i \geq 0$ such that $\mathcal{T}, s_i \models \psi$ and $\mathcal{T}, s_j \models \varphi$ holds for all $0 \leq j < i$.

Derived operators include $\exists\Diamond\varphi \equiv \exists($**true until** $\varphi)$, $\forall\circ\varphi \equiv \neg\exists\circ\neg\varphi$, and $\forall\Box\varphi \equiv \neg\exists\Diamond\neg\varphi$. For example, the formula $\forall\Box\neg(owns_1 \wedge owns_2)$ expresses the mutual exclusion property of our resource manager, whereas $\forall\Box(req_1 \Rightarrow \exists\Diamond\, owns_1)$ asserts that whenever process 1 requests the resource it is at least possible that the request will eventually be honored, although there may also be executions that do not honor the request.

It turns out that the expressiveness of **CTL** and **PTL** is incomparable [La83]: whereas **PTL** is clearly incapable of expressing possibility properties, fairness properties cannot be expressed in **CTL** because path quantifiers and temporal operators strictly alternate. The logic **CTL*** lifts this restriction and (strictly) subsumes both **PTL** and **CTL**. In turn, **CTL*** is contained in Kozen's version of the μ-calculus [Ko83].

The choice of linear versus branching time logics has been at the center of a heated debate some 20 years ago. Today, the choice is mainly guided by which properties are relevant for the given problem and what tools are available.

Alternating-time temporal logics [AlHenKu97] refine the quantifiers of branching time logics by considering the different processes (or "agents") that constitute a reactive system. It is then possible to assert, for example, that the manager can ensure mutual exclusion, or that it is possible for the manager and process 2 to cooperate to ensure eventual access to the resource for process 1.

Automata-theoretic characterizations of branching-time logic are based on tree automata [Th90, Th97], which again define a notion of regular tree languages. More recently, alternating automata have attracted considerable interest [MuSaSch88, KuVa97, Th∞]; they combine the "existential", nondeterministic branching of Büchi automata with its dual, "universal" branching: given a state and an input symbol, the automaton chooses a set of states that will be active simultaneously. Therefore, runs of alternating automata, even over ω-words, are infinite trees or dags (directed acyclic graphs). Alternating automata allow a rather uniform automata-theoretic presentation of decision procedures for linear-time, branching-time, and alternating-time logics [Va95].

3. Model checking **PTL** specifications

Building on the foundations of section 2, we now present the basic model checking algorithm for **PTL** specifications over finite-state systems, discuss a class of optimization techniques, and as a concrete application present an analysis of a cryptographic protocol.

3.1. **Basic PTL model checking algorithm.** The model checking problem requires to decide, for a given transition system $\mathcal{T}$ and **PTL** formula φ, whether φ holds of $\mathcal{T}$ or not. Equivalently, we may ask whether there exists a run of $\mathcal{T}$ that does not satisfy φ. In the automata-theoretic framework presented in section 2, we consider the product automaton of $\mathcal{T}$ and $\mathcal{A}_{\neg\varphi}$, and decide whether it has an accepting run. For the construction of the product, $\mathcal{T}$ is considered as a Büchi automaton with trivial acceptance condition (remember that we had assumed $\mathcal{T}$ not to contain fairness conditions).

Formally, assume given a finite transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta_{\mathcal{T}}, \alpha)$ and a **PTL** formula φ, represented by the corresponding Büchi automaton $\mathcal{A}_{\neg\varphi} = (Q, J, \delta_{\mathcal{A}}, F)$, and assume that the states of the transition system define the interpretation of the atomic formulas $\mathcal{V}$ of φ. The model checking algorithm operates on pairs (s, q) of states where $s \in S$ is a state of $\mathcal{T}$ and $q \in Q$ is a state of $\mathcal{A}_{\neg\varphi}$. Call a pair (s_0, q_0) *initial* whenever $s_0 \in I$ and $q_0 \in J$ are initial states of $\mathcal{T}$ and $\mathcal{A}_{\neg\varphi}$, and call (s', q') a *successor* of (s, q) if both $(s, s') \in \delta_{\mathcal{T}}$ and $(q, s(\mathcal{V}), q') \in \delta_{\mathcal{A}}$ hold (where $s(\mathcal{V})$ denotes the interpretation of the atomic formulas in state s), that is, if both components are possible successor states in $\mathcal{T}$ and $\mathcal{A}_{\neg\varphi}$, respectively. Finally, call (s, q) *accepting* if $q \in F$. The model checking algorithm schematically presented in figure 2, essentially taken from [CourVaWoYa92], explores all reachable states and simultaneously checks for *acceptance cycles*. The algorithm uses a stack (resulting in a depth-first search) that contains those pairs whose successors still need to be generated and a set of pairs that have already been encountered. The main body of the algorithm is given by the procedure `dfs`, which can operate in one of two modes, as indicated by the boolean flag `search_cycle`. In the basic state exploration mode (where `search_cycle` is false), reachable pairs are generated until some accepting pair is encountered. At this point, the search switches to cycle search mode and tries to find a path back to the accepting pair. As shown in [CourVaWoYa92], the algorithm is

```
dfs(boolean search_cycle) {
   p = top(stack);
   foreach (q in successors(p)) {
      if (search_cycle and (q == seed))
         report acceptance cycle and exit;
      if ((q, search_cycle) not in seen) {
         enter (q, search_cycle) into seen;
         push q onto stack;
         dfs(search_cycle);
         if (not search_cycle and (q is accepting)) {
            seed = q; dfs(true);
   }  }  }
   pop(stack);
}
// initialization
seen = emptyset(); stack = emptystack();
foreach initial pair p {
   push p onto stack;
   enter (q, false) into seen;
}
dfs(false)
```

FIGURE 2. Basic **PTL** model checking algorithm.

guaranteed to find some acceptance cycle if one exists, though perhaps not all cycles will be generated (even if the search were continued instead of exiting).

This so-called "on-the-fly" algorithm interleaves the construction of the state space and the search for cycles and stops as soon as some cycle has been found. It therefore avoids having to construct (and store) the entire product automaton. When a cycle is found, a behavior can be reconstructed from the contents of the stack, which is both a run of $\mathcal{T}$ and satisfies $\neg\varphi$, providing a concrete counterexample of why the property does not hold of the transition system.

The algorithm of figure 2 has complexity linear in the product of the sizes of $\mathcal{T}$ and of $\mathcal{A}_{\neg\varphi}$ (by proposition 7, the latter can be exponential in the length of φ). In practice, the exponential factor in the size of the formula is not really problematic, because formulas used as correctness assertions tend to be short. The limiting factor is the size of the model. In practice, the model is not supplied explicitly, but is described in programming language-like notations that may contain state variables, communication channels, parallel processes etc (see also section 3.3). The models generated from such descriptions can be of size exponential in the size of the description, and therefore the algorithm is in fact exponential in both of its inputs. Current technology limits the applicability of model checking to models with a few million reachable states. The following section describes techniques that try to overcome this limit, while section 4 describes how bigger, and even infinite-state systems, can be reduced to systems of manageable size.

3.2. **Partial-order reduction techniques.** Models of reactive systems often contain symmetry that can be exploited to optimize the model checking procedure. One class of optimizations that has been particularly successful in the case of asynchronous systems relies on the fact that systems are composed of individual processes that operate largely independently except for occasional synchronization. The "global" transition system representing the entire system will represent all interleavings of the actions of the individual processes, but for most properties the relative order of independent actions is irrelevant.

Formally, for every action $a \in \mathcal{A}$ of a given transition system $\mathcal{T}$ let $pre(a)$ denote the set of states for which there is some successor state t (i.e., $(s,t) \in \delta$) such that $a \in \alpha(s,t)$, and for every state $s \in pre(a)$ denote by $post(a,s)$ the set of successor states that can be reached by action a. Call two actions $a, b \in \mathcal{A}$ *independent* if all of the following conditions hold:

- For all $s \in pre(a)$ and $t \in post(a,s)$, $t \in pre(b)$ iff $s \in pre(b)$.
- For all $s \in pre(b)$ and $t \in post(b,s)$, $t \in pre(a)$ iff $s \in pre(a)$.
- For all $s \in pre(a) \cap pre(b)$, $\displaystyle\bigcup_{t \in post(a,s)} post(b,t) = \bigcup_{t \in post(b,s)} post(a,s)$.

In other words, independent actions can neither enable nor disable each other, and their execution commutes in that the same states are reachable when the actions are executed in either order. Define *trace equivalence* on finite sequences of actions as the smallest equivalence relation that contains all sequences that differ by the exchange of adjacent independent actions. The definition of independence ensures that from any given state, trace equivalent action sequences lead to the same set of result states; hence for the analysis of reachability it suffices to consider one representative of equivalent traces.

Partial-order reduction algorithms [Val90, GodWo94, HoPe94, PenGeKui∞] differ in how this idea is extended to full **PTL** model checking and how it is implemented as a practical algorithm with low overhead. First, the semantic definition of independence given above is replaced by a sufficient syntactic criterion appropriate for the particular modelling language. For example, in a language based on shared variables, two actions of different processes are certainly independent if they do not update the same variable. For message passing systems, send and receive operations concerning the same channel are independent at those states where the channel is neither empty nor full. Second, simple criteria need to be found that ensure that at least one, and ideally exactly one, representative from every class of trace equivalent action sequences is considered. Finally, for model checking the formula φ being analysed must be taken into account: call an action *visible* if it changes any variable that occurs in φ. Holzmann and Peled [HoPe94] define an action to be *safe* if it is not visible and if it is independent (w.r.t. the syntactic approximation of independence) of all actions of different processes. In the depth-first search algorithm shown in figure 2 it is then enough to consider the successor states for some process all of whose actions are safe, in effect delaying the actions of the other processes. However, the delayed actions must be considered when some state is reached that has already been encountered before. This rather simple heuristic often leads to substantial savings and, in contrast to more elaborate algorithms, carries almost no overhead because the set of safe actions can be determined statically.

3.3. **Analysis of the Needham-Schroeder public-key protocol.** As a concrete application of **PTL** model checking we analyze a public-key cryptographic protocol suggested by Needham and Schroeder using the model checker SPIN [Ho91], which implements the techniques presented in the previous sections.

3.3.1. *Description of the protocol.* The objective of the protocol, symbolically represented in figure 3, is to allow two agents A(lice) and B(ob) to agree on a shared secret, represented by a pair (N_A, N_B) of random numbers.[1] The agents are assumed to have previously

[1] Agreement on a shared secret is a fundamental cryptographic problem: such a secret may for example be used to generate a session key for the encryption of further messages sent between the agents. Numbers such as

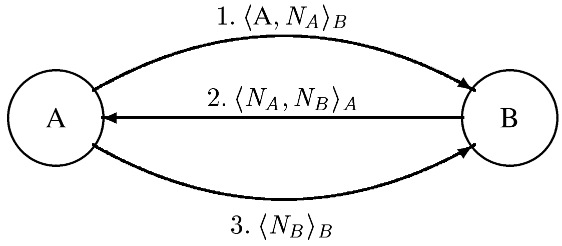

FIGURE 3. Needham-Schroeder public-key protocol.

generated a pair of public and corresponding private keys and to know each other's public key. Integrity of these keys ensures that a message M encrypted with, say, A's public key (denoted as $\langle M \rangle_A$) can only be decrypted by agent A. The protocol requires A and B to exchange three messages:

1. Alice initiates the protocol and sends the message $\langle \mathrm{A}, N_A \rangle_B$ to Bob. The first part of the message tells Bob the identity of the agent who wants to establish the secret. The second part contains the number chosen by his partner.
2. Bob generates a random number N_B and responds with the message $\langle N_A, N_B \rangle_A$. Since that message contains Alice's nonce N_A from the first message, which only Bob could decrypt, Alice concludes that the message must originate with Bob and accepts the pair (N_A, N_B) as the shared secret.
3. Finally, Alice responds with the message $\langle N_B \rangle_A$. Following the same line of reasoning as for message 2, upon receipt of that message Bob is convinced that it originates with Alice and also accepts (N_A, N_B) as the secret shared between the two agents.

The protocol is intended to be used over an unsafe medium. More precisely, an attacker may (besides acting like any honest agent) intercept messages intended for other agents and perhaps replay them later. However, because we are only interested in attacks on the protocol rather than on the encryption algorithm such as RSA, we assume that even an attacker can only decrypt messages encrypted with his own public key. The protocol contains a flaw, and the reader is invited to find it before continuing. The error was discovered, using model checking technology, some 17 years after the protocol was first published [Lo96].

3.3.2. *A* SPIN *model.* In order to analyse the protocol via model checking we must construct a finite-state model of the protocol. Specifically, we make the following simplifying assumptions:

1. There are only three agents present in the network, namely A, B, and I(ntruder).
2. The honest agents A and B are only capable of participating in one run of the protocol each where A acts as the initiator (sending messages 1 and 3) and B as the responder (sending message 2). They block upon receipt of an unexpected message.
3. Agent I is capable of temporarily storing exactly one intercepted message.

In any analysis of real-world systems, it is important to clearly identify the assumptions that underly the formal model of the system. In our example, the assumptions imply that complex protocol errors caused by the interference of several runs of the protocol by the

---

N_A and N_B are usually called "nonces" in the literature on cryptographic protocols, to indicate that they should be used only once by any honest agent participating in the protocol.

same agent cannot be detected. In general, model checking should be regarded as a debugging rather than a verification technique. (The formal verification of cryptographic protocols has, for example, been studied by Paulson [Pau99].)

With these assumptions, it is straightforward to write a model for the honest agents A and B in SPIN's modelling language PROMELA ("protocol meta-language"), which has been designed for the analysis of communication protocols and contains primitives for the sending and receipt of messages over channels.

```
mtype = {msg1, msg2, msg3, agentA, agentB, agentI,
         nonceA, nonceB, nonceI, keyA, keyB, keyI};
typedef Crypt { mtype key, info1, info2; }
chan network = [0] of {mtype, /* msg# */
                       mtype, /* receiver */
                       Crypt};

active proctype A() {
  mtype  partner, pkey, pnonce;
  Crypt  data;

  if /* choose a partner for this run */
  :: partner = agentB; pkey = keyB;
  :: partner = agentI; pkey = keyI;
  fi;
  network!msg1(partner, {pkey, agentA, nonceA});

  network?msg2(agentA, data);
  do
  :: (data.key == keyA) && (data.info1 == nonceA)
     -> break;
  od;
  pnonce = data.info2;

  network!msg3(partner, {pkey, pnonce, 0});
  partnerA = partner;
}
```

FIGURE 4. PROMELA code for agent A.

Figure 4 contains an almost verbatim excerpt from the actual PROMELA code for agent A. The `mtype` declaration defines a number of symbolic constants, including the names, keys, and nonces of the three agents. A `typedef` declaration is used to represent blocks of encrypted data as triples that consist of the key and two data fields. (Of course, no real encryption takes place in the model!) Finally, a message in transit is represented as a tuple consisting of the message number, the intended receiver, and a block of encrypted data. All communication is modelled as synchronous, indicated by buffer length 0, although this is unimportant for this protocol. Agent A first chooses a partner (either B or I) for the protocol run, and looks up the corresponding public key. She then sends the first message as prescribed by the protocol and waits for somebody to send her a message of the second type. If that message's encrypted part is unreadable or does not contain her own nonce she blocks (modelled as an infinite loop), otherwise she extracts the partner's nonce and responds with message 3. (The assignment to the global variable `partnerA` indicates

```
active proctype I() {
  bool  knows_nonceA, knows_nonceB;
  mtype msg, type_intercepted;
  Crypt data, intercepted;

  do
  :: network?msg (_, data)
     -> if /* Perhaps store the message */
        :: intercepted = data; type_intercepted = msg;
        :: skip;
        fi;
        if /* Try to decrypt the message */
        :: (data.key == keyI) ->
           if
           :: data.info1 == nonceA
              -> knows_nonceA = true;
           :: data.info2 == nonceA
              -> knows_nonceA = true;
           :: else -> skip;
           fi;
           if
           :: data.info1 == nonceB
              -> knows_nonceB = true;
           :: data.info2 == nonceB
              -> knows_nonceB = true;
           :: else -> skip;
           fi;
        :: else -> skip;
        fi;
  :: if /* Replay or send msg1 to B */
     :: (type_intercepted == msg1)
        -> network!msg1(agentB, intercepted);
     :: data.key = keyB;
        if
        :: data.info1 = agentA;
        :: data.info1 = agentI;
        fi;
        if
        :: knows_nonceA -> data.info2 = nonceA;
        :: knows_nonceB -> data.info2 = nonceB;
        :: data.info2 = nonceI;
        fi;
        network!msg1(agentB, data);
     fi;
  :: ... /* Replay or send msg2 to A. */
  :: ... /* Replay or send msg3 to B. */
  od;
}
```

FIGURE 5. PROMELA code for agent I.

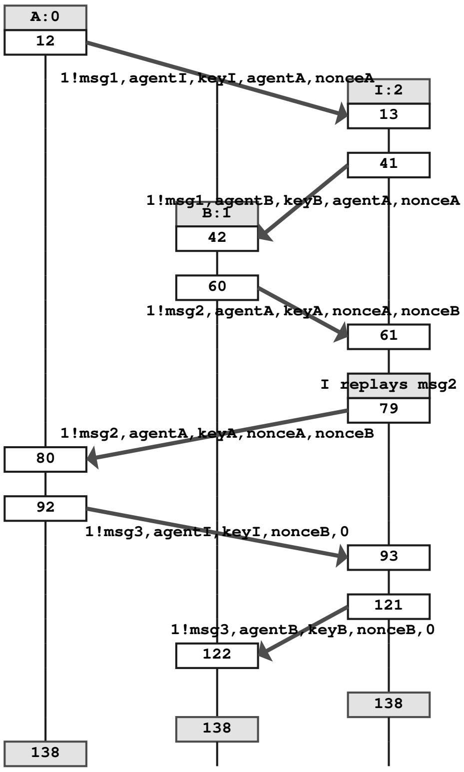

FIGURE 6. Message sequence chart visualizing the attack.

successful termination of A's run of the protocol and is used in the correctness assertion.) The code for agent B is similar.

Unlike the code for the honest agents A and B, the code for I, part of which is shown in figure 5, does not prescribe a fixed protocol—we are trying to use SPIN to find the attack. Rather, it describes the actions that I can perform at any given moment. The overall structure is that of an infinite loop that offers a nondeterministic choice between either receiving a message from the network or sending one of the three possible messages. The first alternative models reception of a message from the network (irrespective of the actually intended receiver, modelling interception of messages). Agent I may then store the message in the local variable `intercepted`, even if it cannot decrypt the message. On the other hand, if the message is encrypted with its own public key, it may analyze the data part and possibly extract the nonces used by agents A and B.

The second alternative represents the emission of a message of the first type to agent B (there is obviously no point in sending such a message to either A or I). There are two subcases: first, agent I may have previously intercepted such a message and may replay it from its store. Second, it may construct the message from the data at its disposal, possibly using nonces learnt from messages received earlier. It may pretend that the message was sent by either A or I (it would not make sense to masquerade as B). The remaining alternatives model the emission of the other message types and are similar.

3.3.3. *Analysis of the model.* We want to verify that whenever both agents A and B have successfully performed a run of the protocol then A believes to share the secret with B iff B believes to share the secret with A. This property is expressed by the **PTL** formula

$$\Box(\mathit{partnerA} \neq 0 \wedge \mathit{partnerB} \neq 0 \Rightarrow (\mathit{partnerA} = \mathit{agentB} \Leftrightarrow \mathit{partnerB} = \mathit{agentA}))$$

(All PROMELA variables are initialized to 0.) Given the model of the three agents and this formula, SPIN runs for a few seconds and reports a behavior that violates the property and constitutes an attack on the protocol. To visualize this attack, the message sequence chart shown in figure 6 is produced that makes it easy to track the communication history: Alice initiates a run of the protocol with Intruder who in turn (masquerading as A) starts a run with Bob, using the nonce received from A. B replies with his nonce, encrypted with A's key. Now, agent I cannot decrypt this message, but forwards it to A. Unsuspecting, since she finds her nonce from the first message, A declares success and returns the second nonce to her partner I. This time, I is able to decrypt the message, extract the nonce, and send it to B, who also declares success. The result is a configuration where A believes to share the secret with I, but B believes to share it with A. To avoid this attack, the second message in the protocol should explicitly identify the sender. A variant where the second message is replaced by $\langle \mathrm{B}, N_A, N_B \rangle_A$ satisfies the desired property, as SPIN confirms (but of course the analysis of the simplified model does not by itself establish the integrity of the modified protocol).

3.4. **CTL model checking.** Model checking algorithms for **CTL** have traditionally used a completely different approach based on fixed-point computations [ClEmSi86, ClGrLo96]: Define the *satisfaction set* $[\![\varphi]\!]$ of a **CTL** formula φ in a transition system $\mathcal{T}$ as the set of states s such that $\mathcal{T}, s \models \varphi$. The model checking problem can then be restated as deciding whether $I \subseteq [\![\varphi]\!]$ holds. Satisfaction sets can be recursively computed as follows:

$$\begin{array}{rcl}
[\![v]\!] & = & \{s : s(v) = \mathit{tt}\} \qquad (\text{for } v \in \mathcal{V}) \\
[\![\neg\varphi]\!] & = & S \setminus [\![\varphi]\!] \\
[\![\varphi \vee \psi]\!] & = & [\![\varphi]\!] \cup [\![\psi]\!] \\
[\![\exists\circ\varphi]\!] & = & \delta^{-1}([\![\varphi]\!]) = \{s : t \in [\![\varphi]\!] \text{ for some } t \text{ s.t. } (s,t) \in \delta\} \\
[\![\exists\Box\varphi]\!] & = & \nu\big(\lambda X.[\![\varphi]\!] \cap \delta^{-1}(X)\big) \\
[\![\exists(\varphi \textbf{ until } \psi)]\!] & = & \mu\big(\lambda X.[\![\psi]\!] \cup ([\![\varphi]\!] \cap \delta^{-1}(X))\big)
\end{array}$$

where νf (resp., μf) denote the greatest (resp., smallest) fixed point of a function f that maps sets of states to sets of states. These fixed points can effectively be computed because S is finite. The definitions follow the recursive characterizations of the **CTL** operators:

$$\begin{array}{rcl}
\exists\Box\varphi & \Leftrightarrow & \varphi \wedge \exists\circ\exists\Box\varphi \\
\exists(\varphi \textbf{ until } \psi) & \Leftrightarrow & \psi \vee (\varphi \wedge \exists\circ\exists(\varphi \textbf{ until } \psi)
\end{array}$$

which are easily verified from the **CTL** semantics. The algorithms given in [ClEmSi86] have complexity linear in the product of the sizes of $\mathcal{T}$ and φ. Therefore, the exponential blow-up incurred for **PTL** by the translation to Büchi automata can be avoided for **CTL**. Again, note that the dominant factor is the size of the transition system, and that the **CTL** model checking problem is again exponential (PSPACE-complete) when the description of the model is considered as the input given to the model checker. The algorithm can be modified to respect fairness conditions, which are not expressible as **CTL** formulas.

The set-based algorithm outlined above opens the way for so-called *symbolic* model checking algorithms that avoid the explicit enumeration of system states, but represent sets of states using compact data structures such as binary decision diagrams [Bry86]. Symbolic **CTL** model checking has first been implemented in the SMV system [McM93]. It performs very well when the state space of the system is sufficiently regular, and when the fixed-point computations converge quickly, and important practical systems have been analysed in this way [ClGrHi93]. Some common data structures such as queues are difficult to represent as BDDs, and special-purpose representations have been proposed. Symbolic model checking can also be used for **PTL** specifications [ClGrHa94]; the idea is to introduce a new state variables v_ψ for every subformula ψ of the original specification, and to express the transition and acceptance conditions in terms of these new variables.

It is difficult to compare symbolic model checkers such as SMV and explicit-state model checkers with partial-order reductions such as SPIN because they are optimized for different types of systems. In practice, SMV tends to perform better for hardware or synchronous systems with relatively short data paths, while SPIN often does better for software or asynchronous systems with several processes that operate independently.

More recently, algorithms for **CTL** model checking based on a class of alternating tree automata have been developed and implemented [BeVaWo94, Le99] whose complexity matches that of the fixed-point algorithm described above.

4. ABSTRACTION TECHNIQUES

The model checking techniques described previously operate on finite-state models because the entire state space has to be stored and traversed. There has also been active research on the model checking of restricted classes of infinite-state systems [Mo96, Es96, EsFiMay99], but this can be made to work only if the state space is sufficiently regular for interesting properties to be decidable. Although physical systems are ultimately finite-state systems, they may well be too big to be amenable to model checking. Besides, it is often useful to model systems as having infinitely many states when one is not interested in the specific limitations imposed by concrete hardware. Examples include synchronization protocols for an arbitrary number of processes or communication protocols over networks with arbitrary buffering capacity. In this section, we discuss abstraction techniques that allow properties to be verified for large, and even infinite-state, models by verifying a small, finite-state abstraction. The abstract model is constructed semi-automatically, relying on abstract interpretation and/or on interactive theorem provers. However, the runs of the abstract system are analysed automatically using model checking.

4.1. **Boolean abstractions.** *Abstract interpretation* [CouCou77] provides a generic framework for the construction of abstract models. It requires the state spaces of the concrete and abstract models to be related by a *Galois connection*, a pair (α, γ) of functions where the abstraction function α associates an element of a lattice of abstract states to each concrete state, whereas the concretization function γ maps every abstract state to a set of concrete states such that α and γ are consistent with each other. Lattice-theoretic operations on the abstract states correspond to set-theoretic operations on sets of concrete states. In our application, we are interested in verifying properties expressed in temporal logic, and it is convenient to choose the Boolean algebra generated by a finite set of *basic assertions* $\mathcal{V}$ as the abstract domain. The choice of suitable basic assertions in general requires creativity on the side of the verifier. Obviously, the state predicates that appear in the formula to be verified should be included in the set of basic assertions such that it can be interpreted as a **PTL** formula over $\mathcal{V}$. State predicates that occur as preconditions of actions are also natural candidates for inclusion into $\mathcal{V}$. In more complicated situations, it is necessary to invent auxiliary predicates that may be suggested by an informal correctness argument.

var n : nat **where** $n > 0$		

process P1
 loop
  thk_1 : **await** $even(n)$;
  eat_1 : $n := n$ **div** 2
 endloop
endprocess
	$\|$	**process** P2
		 loop
		  thk_2 : **await** $\neg even(n)$;
		  eat_2 : $n := 3 * n + 1$
		 endloop
		endprocess

FIGURE 7. Dining mathematicians protocol.

Given a (possibly infinite-state) transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta, \alpha)$ and a finite set $\mathcal{V}$ of basic assertions that can be evaluated over S, we say that $\overline{\mathcal{T}} = (\overline{S}, \overline{I}, \mathcal{A}, \overline{\delta}, \overline{\alpha})$ is a Boolean abstraction of $\mathcal{T}$ with respect to $\mathcal{V}$ if the following conditions are satisfied:

- $\overline{S} = 2^{\mathcal{V}}$; we interpret a state $p \in \overline{S}$ as the conjunction of those basic assertions $v \in \mathcal{V}$ that are contained in p and the negations of those in $\mathcal{V} \setminus p$.
- $p \in \overline{I}$ if there exists some concrete state $s \in I$ such that p holds of s.
- $(p, q) \in \delta$ if there exist concrete states $(s, t) \in \delta$ such that p holds of s and q holds of t.
- $a \in \overline{\alpha}(p, q)$ if there exist concrete states $s, t \in S$ such that $a \in \alpha(s, t)$, p holds of s, and q holds of t.

Proposition 8. *Assume that $\mathcal{T}$ is a transition system and that $\overline{\mathcal{T}}$ is a Boolean abstraction of $\mathcal{T}$ with respect to some set $\mathcal{V}$ of basic assertions. For every run $\rho = r_0 r_1 \ldots$ of $\mathcal{T}$ there is a run $\overline{\rho} = \overline{r_0}\,\overline{r_1} \ldots$ of $\overline{\mathcal{T}}$ such that $\overline{r_i}$ holds of r_i, for all $i \in \mathbb{N}$.*

Proposition 8 asserts that every run of the concrete system $\mathcal{T}$ has a corresponding run in the abstract system $\overline{\mathcal{T}}$. In particular, any run that violates some **PTL** formula φ built from the set $\mathcal{V}$ of basic assertions has an abstract counterpart in $\overline{\mathcal{T}}$, which also violates φ. Conversely, any property that can be verified for $\overline{\mathcal{T}}$ also holds for $\mathcal{T}$. In fact, it is not necessary to choose the full powerset of $\mathcal{V}$ as the set of abstract states: it is enough that $\overline{S}$ contains abstract states that represent every reachable concrete state.

4.2. **Example: the dining mathematicians.** The "dining mathematicians" problem due to [DamGrGe94] serves to illustrate the use of Boolean abstractions. Figure 7 illustrates the protocol: two processes P1 and P2 alternate between "thinking" and "eating" states. To ensure exclusive use of the dining room, they synchronize via a shared variable n, initialized to some positive natural number. Process P1 may eat only if n is even, whereas P2 must wait for n to become odd. When leaving the "eating" state, P1 and P2 update n such that eventually the other process will be able to eat.

We wish to verify the following properties:

$$
\begin{array}{llll}
(Pos) & \Box(n > 0) & (Excl) & \Box\neg(eat_1 \wedge eat_2) \\
(Live_1) & \Box\Diamond\, eat_1 & (Live_2) & \Box\Diamond\, eat_2
\end{array}
$$

The first two formulas assert invariance properties stating that n is always positive and that the two processes are never simultaneously at the "eating" state. The remaining two formulas assert the liveness properties that each process will eat infinitely often ("starvation freedom"). The protocol and the properties under investigation suggest to choose the set $\mathcal{V} = \{eat_1, eat_2, n > 0, even(n)\}$ as the set of basic assertions. A Boolean abstraction of the original protocol (more precisely, the reachable part of such an abstraction) is shown in figure 8, where the action labelling has been omitted. The states marked q_0 and q_1 are the

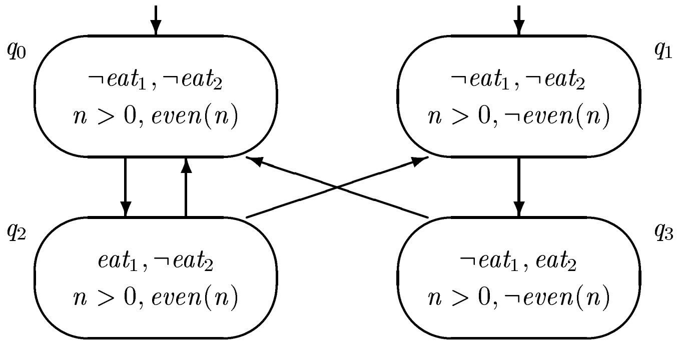

FIGURE 8. An abstraction of the dining mathematicians protocol.

initial states of the abstraction, since the initial condition does not specify whether initially n is even or odd. The transitions between the abstract states correspond to all possible transitions of the concrete system. For example, from the concrete states represented by the abstract state q_2 the only possible action is the eat_1 action of process P1. However, the information that n is a positive even number is not enough to conclude whether n will be even or odd after division by 2. In this simple example, the abstraction can be computed automatically, relying on elementary arithmetical facts such as

$$n > 0 \wedge \neg even(n) \Rightarrow even(3 * n + 1)$$

Evaluating the formulas listed above over the abstract model, we immediately see that (Pos) and $(Excl)$ hold since all reachable states contain the formula $n > 0$, and none contains both eat_1 and eat_2. Similarly, formula $(Live_1)$ is satisfied because state q_2, which satisfies eat_1, is reached from any other state after finitely many steps. On the other hand, formula $(Live_2)$ cannot be verified over the given abstraction: the run that alternates between states q_0 and q_2 never visits a state that satisfies eat_2. In fact, $(Live_2)$ can not be verified over any finite-state abstraction without using additional information: For all $k \in \mathbb{N}$, when $n = 2^k$, process P1 may eat k times in succession before P2 gets a chance to eat. Hence, any finite-state abstraction of the protocol must contain a cycle that does not contain a state satisfying eat_2 and therefore invalidates property $(Live_2)$.

4.3. **Strengthening boolean abstractions.** The situation with property $(Live_2)$ of the dining mathematicians protocol is quite typical of proofs by abstraction: the identification of distinct concrete states in the abstract model introduces cycles, which invalidate liveness properties. In hand proofs, well-founded orderings are used to verify that the system eventually exits from the cycle. Fortunately, this idea can also be applied to break cycles in the abstract model. Assume given a valuation function $\nu : S \to D$ from the states of the concrete transition system into some well-founded ordering $(D, \preceq)$. An edge labelling $\overline{\nu} : \overline{\delta} \to \{\prec, \preceq, -\}$ is a *wfo annotation* for a Boolean abstraction $\overline{\mathcal{T}} = (\overline{S}, \overline{I}, \mathcal{A}, \overline{\delta}, \overline{\alpha})$ and ν if

- $\overline{\nu}(p, q) = \prec$ implies that $\nu(t) \prec \nu(s)$ for all concrete states $(s, t) \in \delta$ such that p holds of s and q holds of t, and
- $\overline{\nu}(p, q) = \preceq$ implies that $\nu(t) \preceq \nu(s)$ for all concrete states $(s, t) \in \delta$ such that p holds of s and q holds of t.

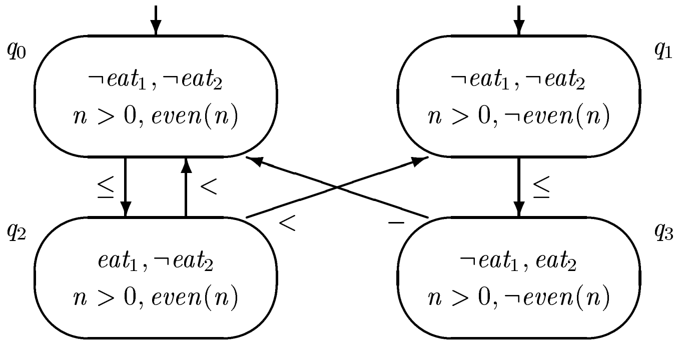

FIGURE 9. A wfo annotation for the abstraction of figure 8.

The idea is that for every $\prec$-labelled edge that is taken in some run of the abstract system, the valuation function ν strictly decreases for every corresponding run in the concrete system, and similarly for $\preceq$-labelled edges. The principle of well-founded orderings implies that only finitely many transitions that decrease ν are possible between any two transitions that do not decrease ν. Now define the **PTL**-formulas

$$Decr \equiv \bigvee_{\overline{\nu}(p,q)=\prec} p \wedge \circ q \qquad NonDecr \equiv \bigvee_{\overline{\nu}(p,q)=-} p \wedge \circ q$$

Then the principle of well-founded orderings justifies the abstract system to be strengthened by the formula $\Box\Diamond Decr \Rightarrow \Box\Diamond NonDecr$. That is, any property verified for $\overline{\mathcal{T}}$ under this additional hypothesis holds of the concrete system.

For the dining mathematicians protocol, we define $\nu(s)$ as the value of n at state s, and choose the standard ordering $\leq$ on the natural numbers as our well-founded ordering. The annotation $\overline{\nu}$ shown in figure 9 is a wfo annotation for the abstraction of figure 8; this is again easily verified using simple arithmetical facts such as

$$n > 0 \wedge even(n) \Rightarrow n \textbf{ div } 2 < n$$

The annotation $\overline{\nu}$ is used to break the cycle between states q_0 and q_2, along which the valuation decreases infinitely often, but never increases. In fact, property $(Live_2)$ can be verified by model checking under the hypothesis $\Box\Diamond Decr \Rightarrow \Box\Diamond NonDecr$.

In general, several well-founded orderings may be used in combination, and invariants may be necessary to establish the soundness of a wfo annotation. Similar strengthening techniques allow fairness conditions associated with the concrete system to be recovered in the abstract system [Me97]. We have successfully employed these techniques to verify, among other examples, a reader-writer algorithm developed at Siemens Corporate Research, and a self-stabilizing protocol due to Dijkstra [Me98]. The latter example required rather intricate proofs (performed using Isabelle [Pau94]) to establish the soundness of the abstraction. Still, the separation between interactive proofs in ordinary first-order logic that justify the soundness of abstractions and the use of temporal logic model checking to verify the temporal properties of the system resulted in substantial savings over an alternative proof [QaSha98] performed in PVS.

Abstractions for branching-time temporal logics, where also possibility properties asserting the existence of certain branches are to be verified in the abstract system, require the use of so-called $\forall\exists$ abstractions of transitions [DamGrGe94], besides the $\exists\exists$ abstractions considered here. These can also be useful to establish counterexamples for the concrete system when analysing of **PTL** properties. In *deductive model checking* [SipUrMa96] the abstract system is constructed in a process of stepwise refinement, starting from a trivial abstraction and the Büchi automaton for the negation of the property.

Acknowledgement. I gratefully acknowledge the helpful suggestions of an anonymous referee.

REFERENCES

[AlHenKu97] Rajeev **Alur**, Thomas A. **Henzinger**, Orna **Kupferman**, Alternating-time temporal logic, *in:* 38th IEEE Symposium on Foundations of Computer Science, 1997, p. 100–109

[BeHaPn83] M. **Ben-Ari**, J. **Halpern**, A. **Pnueli**, The temporal logic of branching time, Acta Informatica 20 (1983), p. 207–226

[BeVaWo94] O. **Bernholtz**, M. **Vardi**, P. **Wolper**, An Automata-Theoretic Approach to Branching-Time Model Checking, *in:* D. L. Dill (*ed.*), 6th International Conference on Computer Aided Verification (CAV'94), Stanford 1994, Berlin 1994 [Lecture Notes in Computer Science 818], p. 142–155

[Bry86] R. E. **Bryant**, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers C-35 (1986), p. 677–691

[Bü62] J. R. **Büchi**, On a decision method in restricted second-order arithmetics, *in:* International Congress on Logic, Method and Philosophy of Science, Stanford 1962, p. 1–12

[ClEmSi86] E. M. **Clarke**, E. A. **Emerson**, A. P. **Sistla**, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (1986), p. 244–263

[ClGrHa94] E. M. **Clarke**, O. **Grumberg**, K. **Hamaguchi**, Another look at LTL model checking, *in:* D. L. Dill (*ed.*), 6th International Conference on Computer Aided Verification (CAV'94), Stanford 1994, Berlin 1994 [Lecture Notes in Computer Science 818], p. 415–427

[ClGrHi93] E. M. **Clarke**, O. **Grumberg**, H. **Hiraishi**, S. **Jha**, D. E. **Long**, K. L. McMillan, L. A. **Ness**, Verification of the Futurebus+ Cache Coherence Protocol, *in:* D. Agnew, L. Claesen, R. Camposano (*eds.*), IFIP Conference on Computer Hardware Description Languages and their Applications, Ottawa 1993, Amsterdam 1993, p. 5–20

[ClGrLo96] Edmund M. **Clarke**, Orna **Grumberg**, David E. **Long**, Model checking, *in:* Manfred Broy (*ed.*), Deductive Program Design, Berlin 1996 [NATO ASI Series F-152], p. 305–350

[CourVaWoYa92] C. **Courcoubetis**, M. **Vardi**, P. **Wolper**, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal methods in system design 1 (1992), p. 275–288

[CouCou77] Patrick **Cousot**, Radhia **Cousot**, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *in:* 4th ACM Symposium on Principles of Programming Languages, Los Angeles 1977, p. 238–252

[DamGrGe94] Dennis **Dams**, Orna **Grumberg**, Rob **Gerth**, Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*, *in:* Ernst-Rüdiger Olderog (*ed.*), Programming Concepts, Methods, and Calculi (PROCOMET '94), Amsterdam 1994 [IFIP Transactions], p. 561–581

[DanGiuVa99] M. **Daniele**, F. **Giunchiglia**, M. **Vardi**, Improved automata generation for linear temporal logic, *in:* N. Halbwachs, D. Peled (*eds.*), Computer Aided Verification (CAV'99), Trento, Italy, 1999, Berlin 1999 [Lecture Notes in Computer Science 1633], p. 249–260

[Em80] E. Allen **Emerson**, Edmund C. **Clarke**, Characterizing correctness properties of parallel programs using fixpoints, *in:* 7th International Colloquium on Automata, Languages and Programming, Berlin 1980 [Lecture Notes in Computer Science 85], p. 169–181

[Em90] E. Allen **Emerson**, Temporal and modal logic, *in :* Jan van Leeuwen (*ed.*), Formal models and semantics, Handbook of theoretical computer science, Volume B, Amsterdam 1990, p. 997–1071

[Es96] Javier **Esparza**, More Infinite Results, *in:* 1st International Workshop on the Verification of Infinite State Systems, Pisa, Italy, 1996 Electronic Notes in Theoretical Computer Science 5

[EsFiMay99] Javier **Esparza**, Alain **Finkel**, Richard **Mayr**, On the verification of broadcast protocols, *in:* G. Longo (*ed.*), 14th IEEE Symposium on Logic in Computer Science, Trento, Italy, 1999, Washington, 1999, p. 352–359

[GePeVaWo95] R. **Gerth**, D. **Peled**, M. **Vardi**, P. **Wolper**, Simple on-the-fly automatic verification of linear temporal logic, *in:* Protocol Specification, Testing, and Verification, Warsaw, Poland, 1995, London, 1995, p. 3–18

[GodWo94] P. **Godefroid**, P. **Wolper**, A partial approach to model checking, Information and Computation 110 (1994), p. 305–326

[Ho91] Gerard **Holzmann**, Design and Validation of Computer Protocols, Englewood Cliffs NJ 1991

[HoPe94] Gerard **Holzmann**, Doron **Peled**, An improvement in formal verification, *in:* IFIP WG 6.1 Conference on Formal Description Techniques, Bern, Switzerland, 1994, London 1994, p. 197–214

[HopUll79] John E. **Hopcroft**, Jeffrey D. **Ullman**, Introduction to automata theory, languages, and computation, Reading MA 1979

[Kl98] Nils **Klarlund**, Mona & Fido: The Logic-Automaton Connection in Practice, *in:* M. Nielsen, W. Thomas (*eds.*), Computer Science Logic, CSL '97, Berlin 1997 [Lecture Notes in Computer Science 1414], p. 311–326

[Ko83] Dexter **Kozen**, Results on the propositional mu-calculus, Theoretical Computer Science 27 (1983), p. 333–354

[Krö87] Fred **Kröger**, Temporal Logic of Programs, Berlin 1987 [EATCS Monographs on Theoretical Computer Science 8]

[KuVa97] Orna **Kupferman**, Moshe Y. **Vardi**, Weak alternating automata are not so weak, *in:* 5th Israeli Symposium on Theory of Computing and Systems, Ramat Gan 1997, p. 147–158

[La83] Leslie **Lamport**, What Good is Temporal Logic?, *in:* R. E. A. Mason (*ed.*), Information Processing 83: Proceedings of the IFIP 9th World Congress, Amsterdam 1983, p. 657–668

[Le99] Martin **Leucker**, Model Checking Games for the Alternation Free mu-Calculus and Alternating Automata, *in:* A. Voronkov (*ed.*), 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99), Berlin 1999 [Lecture Notes in Computer Science 1705], p. 77–91

[LiPnZu85] Orna **Lichtenstein**, Amir **Pnueli**, Lenore **Zuck**, The glory of the past, *in:* Rohit Parikh (*ed.*), Logics of Programs, Berlin 1985 [Lecture Notes in Computer Science 193], p. 196–218

[Lo96] Gavin **Lowe**, Breaking and fixing the Needham-Schroeder public key protocol using FDR, *in:* T. Margaria and B. Steffen (*eds.*), Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), Berlin 1996 [Lecture Notes in Computer Science 1055], p. 147–166

[MaPn92] Zohar **Manna**, Amir **Pnueli**, The temporal logic of reactive and concurrent systems—Specification, New York 1992

[McM93] K. L. **McMillan**, Symbolic Model Checking, Dordrecht 1993

[McM97] K. L. **McMillan**, A compositional rule for hardware design refinement, *in:* O. Grumberg(*ed.*), 9th International Conference on Computer Aided Verification (CAV'97), Berlin 1997 [Lecture Notes in Computer Science 1254], p. 24–35

[Me97] Stephan **Merz**, Rules for abstraction, *in:* R. K. Shyamasundar, K. Ueda (*eds.*), Advances in Computing Science—ASIAN'97, Kathmandu, Nepal 1997, Berlin 1997 [Lecture Notes in Computer Science 1345], p. 32–45

[Me98] Stephan **Merz**, On the verification of a self-stabilizing algorithm, *typed notes* 1998 *available at:* `http://www.pst.informatik.uni-muenchen.de/~merz/papers/dijkstra.ps.gz`

[Mo96] F. **Moller**, Infinite Results, *in:* U. Montanari, V. Sassone (*eds.*), 7th International Conference on Concurrency Theory (CONCUR'96), Pisa, Italy, 1996, Berlin 1996 [Lecture Notes in Computer Science 1119], p. 195–216

[Mu63] D. E. **Muller**, Infinite Sequences and Finite Machines, *in:* 4th Annual Symposium on Switching Circuit Theory and Logical Design, New York 1963, p. 3–16

[MuSaSch88] D. E. **Muller**, A. **Saoudi**, P. E. **Schupp**, Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time, *in:* 3rd IEEE Symposium on Logic in Computer Science, 1988, p. 422–427

[Pau94] Lawrence C. **Paulson**, Isabelle: A Generic Theorem Prover, Berlin 1994 [Lecture Notes in Computer Science 828]

[Pau99] Lawrence C. **Paulson**, Proving security protocols correct, *in:* G. Longo (*ed.*), 14th IEEE Symposium on Logic in Computer Science, Trento, Italy, 1999, Washington 1999, p. 370–383

[PenGeKui∞] W. **Penczek**, R. **Gerth**, R. **Kuiper**, Partial order reductions preserving simulations, *to appear* 1999

[Pn77] Amir **Pnueli**, The temporal logic of programs, *in:* Proceedings of the 18th Annual Symposium on the Foundations of Computer Science, 1977, p. 46–57

[QaSha98] Shaz **Qadeer**, Natarajan **Shankar**, Verifying a self-stabilizing mutual exclusion algorithm, *in:* David Gries, Willem-Paul de Roever (*eds.*), Programming Concepts and Methods, Shelter Island, NY 1998, p. 424–443

[QueiSi81] J. P. **Queille**, J. **Sifakis**, Specification and verification of concurrent systems in Cesar, *in:* 5th International Symposium on Programming, Berlin 1981 [Lecture Notes in Computer Science 137], p. 337–351

[Sa88] Shmuel **Safra**, On the complexity of ω-automata, *in:* 29th IEEE Symposium on Foundations of Computer Science, 1988, p. 319–327

[SipUrMa96] H. B. **Sipma**, T. E. **Uribe**, Z. **Manna**, Deductive model checking, *in:* R. Alur and T. Henzinger (*eds.*), 8th International Conference on Computer-Aided Verification. Berlin 1996 [Lecture Notes in Computer Science 1102], p. 208–219

[Sis85] A. P. **Sistla**, E. M. **Clarke**, The complexity of propositional linear temporal logic, Journal of the ACM 32 (1985), p. 733–749

[St92] Colin **Stirling**, Modal and temporal logics, *in:* S. Abramsky, D. Gabbay, T. Maibaum (*eds.*), Handbook of Logic in Computer Science, Oxford 1992, p. 477–563

[Th90] Wolfgang **Thomas**, Automata on infinite objects, *in:* Jan van Leeuwen (*ed.*), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Amsterdam 1990, p. 133–194

[Th97] Wolfgang **Thomas**, Languages, automata, and logic, *in:* G. Rozenberg, A. Salomaa (*eds.*), Handbook of Formal Language Theory, Volume III, Berlin 1997, p. 389–455

[Th∞] Wolfgang **Thomas**, Complementation of Büchi automata revisited, *to appear*, *available at* `http://www-i7.informatik.rwth-aachen.de/~thomas/publications.html`

[Val90] A. **Valmari**, A stubborn attack on state explosion, *in:* 2nd International Workshop on Computer Aided Verification, Berlin 1990 [Lecture Notes in Computer Science 531], p. 156–165

[Va95] Moshe Y. **Vardi**, Alternating Automata and Program Verification, *in:* Jan van Leeuwen (*ed.*), Computer Science Today, Berlin 1995 [Lecture Notes in Computer Science 1000], p. 471–485

[VaWo94] M. Y. **Vardi**, P. **Wolper**, Reasoning about infinite computations, Information and Computation 115 (1994), p. 1–37

[Wo83] Pierre **Wolper**, Temporal logic can be more expressive, Information and Control 56 (1983), p. 72–93

INSTITUT FÜR INFORMATIK, UNIVERSITÄT MÜNCHEN, GERMANY
E-mail address: `merz@informatik.uni-muenchen.de`