Synthesis from Knowledge-Based Specifications* (Extended Abstract)

Ron van der Meyden[1] and Moshe Y. Vardi[2]

[1] Computing Sciences
University of Technology, Sydney
P.O. Box 123, Broadway,
NSW 2007, Australia
ron@socs.uts.edu.au
http://www-staff.socs.uts.edu.au/~ron
[2] Department of Computer Science
Mail Stop 132, Rice University
6100 S. Main Street
Houston, TX 77005-1892, U.S.A
vardi@cs.rice.edu
http://www.cs.rice.edu/~vardi

Abstract. In program synthesis, we transform a specification into a program that is guaranteed to satisfy the specification. In synthesis of reactive systems, the environment in which the program operates may behave nondeterministically, e.g., by generating different sequences of inputs in different runs of the system. To satisfy the specification, the program needs to act so that the specification holds in every computation generated by its interaction with the environment. Often, the program cannot observe all attributes of its environment. In this case, we should transform a specification into a program whose behavior depends only on the observable history of the computation. This is called *synthesis with incomplete information*. In such a setting, it is desirable to have a *knowledge-based specification*, which can refer to the uncertainty the program has about the environment's behavior. In this work we solve the problem of synthesis with incomplete information with respect to specifications in the logic of knowledge and time. We show that the problem has the same worst-case complexity as synthesis with complete information.

1 Introduction

One of the most significant developments in the area of design verification over the last decade is the development of of algorithmic methods for verifying temporal specifica-

* Work begun while both authors were visitors at the DIMACS Special Year on Logic and Algorithms. Work of the first author supported by an Australian Research Council Large Grant. Work of the second author supported in part by NSF grants CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation. Thanks to Kai Engelhardt and Yoram Moses for their comments on earlier versions of this paper.

tions of *finite-state* designs [4]. The significance of this follows from the fact that a considerable number of the communication and synchronization protocols studied in the literature are in essence finite-state programs or can be abstracted as finite-state programs. A frequent criticism against this approach, however, is that verification is done *after* substantial resources have already been invested in the development of the design. Since designs invariably contain errors, verification simply becomes part of the debugging process. The critics argue that the desired goal is to use the specification in the design development process in order to guarantee the design of correct programs. This is called *program synthesis*.

The classical approach to program synthesis is to extract a program from a proof that the specification is satisfiable. For reactive programs, the specification is typically a temporal formula describing the allowable behaviors of the program [22]. Emerson and Clarke [8] and Manna and Wolper [23] showed how to extract programs from (finite representations of) models of the formula. In the late 1980s, several researchers realized that the classical approach is well suited to *closed* systems, but not to *open* systems [7, 29, 1]. In open systems the program interacts with the environment. A correct program should be able to handle arbitrary actions of the environment. If one applies the techniques of [8, 23] to open systems, one obtains programs that can handle only some actions of the environment.

Pnueli and Rosner [29], Abadi, Lamport and Wolper [1], and Dill [7] argued that the right way to approach synthesis of open systems is to consider the situation as a (possibly infinite) game between the environment and the program. A correct program can then be viewed as a winning strategy in this game. It turns out that satisfiability of the specification is not sufficient to guarantee the existence of such a strategy. Abadi et al. called specifications for which winning strategies exist *realizable*. A winning strategy can be viewed as an infinite tree. In those papers it is shown how the specification can be transformed into a tree automaton such that a program is realizable precisely when this tree automaton is nonempty, i.e., it accepts some infinite tree. This yields a decision procedure for realizability.

The works discussed so far deal with situations in which the program has complete information about the actions taken by the environment. This is called synthesis with *complete information*. Often, the program does not have complete information about its environment. Thus, the actions of the program can depend only on the "visible" part of the computation. Synthesizing such programs is called synthesis with *incomplete information*. The difficulty of synthesis with incomplete information follows from the fact that while in the complete-information case the strategy tree and the computation tree coincide, this is no longer the case when we have incomplete information. Algorithms for synthesis were extended to handle incomplete information in [30, 35, 2, 16, 33, 17].

It is important to note that temporal logic specifications cannot refer to the uncertainty of the program about the environment, since the logic has no construct for referring to such uncertainty. It has been observed, however, that designers of open systems often reason explicitly in terms of uncertainty [13]. A typical example is a rule of the form "send an acknowledgement as soon as you *know* that the message has been received". For this reason, it has been proposed in [14] to use epistemic logic as a specification language for open systems with incomplete information. When dealing with ongoing behavior in systems with incomplete information, a combination of temporal and epistemic logic can refer to both behavior and uncertainty [21, 19]. In such a logic the above rule can be formalized by the formula $\Box(K\mathsf{received} \rightarrow \mathsf{ack})$, where $\Box$ is the temporal connective "always", K is the epistemic modality indicating knowledge, and received and ack are atomic propositions.

Reasoning about open systems at the *knowledge level* allows us to abstract away from many concrete details of the systems we are considering. It is often more intuitive to think in terms of the high-level concepts when we design a protocol, and then translate these intuitions into a concrete program, based on the particular properties of the setting we are considering. This style of program development will generally allow us to modify the program more easily when considering a setting with different properties, such as different communication topologies, different guarantees about the reliability of various components of the system, and the like. See [11] for many citations of papers that offer examples of knowledge-level analysis of open systems with incomplete information. To be able to translate, however, these high-level intuitions into a concrete program one has to be able to check that the given specification is realizable in the sense described above.

Our goal in this paper is to extend the program synthesis framework to temporal-epistemic specification. The difficulty that we face is that all previous program-synthesis algorithms attempt to construct strategy trees that realize the given specification. Such trees, however, refer to temporal behavior only and they do not contain enough information to interpret the epistemic constructs. (We note that this difficulty is different than the difficulty faced when one attempts to extend synthesis with incomplete information to *branching-time* specification [17], and the solution described there cannot be applied to knowledge-based specifications.) Our key technical tool is the definition of *finitely labelled* trees that contain information about both temporal behavior and epistemic uncertainty. Our main result is that we can extend the program synthesis framework to handle knowledge-based specification with no increase in worst-case computational complexity.

2 Definitions

In this section we define the formal framework within which we will study the problem of synthesis from knowledge-based specifications, provide semantics for the logic of knowledge and time in this framework, and define the realizability problem.

Systems will be decomposed in our framework into two components: the program, or *protocol* being run, and the remainder of the system, which we call the *environment* within which this protocol operates. We begin by presenting a model, from [25], for the environment. This model is an adaption of the notion of *context* of Fagin et al. [11]. Our main result in this paper is restricted to the case of a single agent, but as we will state a result in Section 5 that applies in a more general setting, we define the model assuming a finite number of agents.

Intuitively, we model the environment as a finite-state transition system, with the transitions labelled by the agents' actions. For each agent $i = 1 \dots n$ let ACT_i be a set of *actions* associated with agent i. We will also consider the environment as able to perform actions, so assume additionally a set ACT_e of actions for the environment. A *joint action* will consist of an action for each agent and an action for the environment, i.e., the set of joint actions is the cartesian product $ACT = ACT_e \times ACT_1 \times \ldots \times ACT_n$.

Suppose we are given such a set of actions, together with a set of *Prop* of atomic propositions. Define a *finite interpreted environment for n agents* to be a tuple E of the form $\langle S_e, I_e, P_e, \tau, O_1, \ldots, O_n, \pi_e \rangle$ where the components are as follows:

1. S_e is a finite set of *states of the environment*. Intuitively, states of the environment may encode such information as messages in transit, failure of components, etc. and possibly the values of certain local variables maintained by the agents.
2. I_e is a subset of S_e, representing the possible *initial states* of the environment.
3. $P_e : S_e \to \mathcal{P}(ACT_e)$ is a function, called the *protocol of the environment,* mapping states to subsets of the set ACT_e of actions performable by the environment. Intuitively, $P_e(s)$ represents the set of actions that may be performed by the environment when the system is in state s. We assume that this set is nonempty for all $s \in S_e$.
4. τ is a function mapping joint actions $\mathbf{a} \in ACT$ to state transition functions $\tau(\mathbf{a}) : S_e \to S_e$. Intuitively, when the joint action $\mathbf{a}$ is performed in the state s, the resulting state of the environment is $\tau(\mathbf{a})(s)$.
5. For each $i = 1 \dots n$, the component O_i is a function, called the *observation function of agent i,* mapping the set of states S_e to some set $\mathcal{O}$. If s is a global state then $O_i(s)$ will be called the *observation* of agent i in the state s.
6. $\pi_e : S_e \to \{0,1\}^{Prop}$ is an *interpretation*, mapping each state to an assignment of truth values to the atomic propositions in $Prop$.

A *run* r of an environment E is an *infinite* sequence $s_0, s_1, \dots$ of states such that $s_0 \in I_e$ and for all $m \geq 0$ there exists a joint action $\mathbf{a} = \langle a_e, a_1, \dots, a_n \rangle$ such that $s_{m+1} = \tau(\mathbf{a})(s_m)$ and $a_e \in P_e(s_m)$. For $m \geq 0$ we write $r_e(m)$ for s_m. For $k \leq m$ we also write $r[k..m]$ for the sequence $s_k \dots s_m$ and $r[m..]$ for $s_m s_{m+1} \dots$.

A *point* is a tuple (r, m), where r is a run and m a natural number. Intuitively, a point identifies a particular instant of time along the history described by the run. A run r' will be said to be a run *through* a point (r, m) if $r[0..m] = r'[0..m]$. Intuitively, this is the case when the two runs r and r' describe the same sequence of events up to time m.

Runs of an environment provide sufficient structure for the interpretation of formulae of linear temporal logic. To interpret formulae involving knowledge, we need additional structure. Knowledge arises not from a single run, but from the position a run occupies within the collection of all possible runs of the system under study. Following [11], define a *system* to be a set $\mathcal{R}$ of runs and an *interpreted system* to be a tuple $\mathcal{I} = (\mathcal{R}, \pi)$ consisting of a system $\mathcal{R}$ together with an interpretation function π mapping the points of runs in $\mathcal{R}$ to assignments of truth value to the propositions in $Prop$.

All the interpreted systems we deal with in this paper will have all runs drawn from the same environment, and the interpretation π derived from the interpretation of the environment by means of the equation $\pi(r,m)(p) = \pi_e(r_e(m))(p)$, where (r, m) is a point and p an atomic proposition. That is, the value of a proposition at a point of a run is determined from the state of the environment at that point, as described by the environment generating the run.

The definition of run presented above is a slight modification of the definitions of Fagin et al. [11]. Roughly corresponding to our notion of state of the environment is their notion of a *global state*, which has additional structure. Specifically, a global state identifies a *local state* for each agent, which plays a crucial role in the semantics of knowledge. We have avoided the use of such extra structure in our states because we focus on just one particular definition of local states that may be represented in the general framework of [11].

In particular, we will work with respect to a *synchronous perfect-recall* semantics of knowledge. Given a run $r = s_0, s_1 \dots$ of an environment with observation functions

O_i, we define the *local state of agent i at time $m \geq 0$* to be the sequence $r_i(m) = O_i(s_0)\ldots O_i(s_m)$. That is, the local state of an agent at a point in a run consists of a complete record of the observations the agent has made up to that point.

These local states may be used to define for each agent i a relation $\sim_i$ of *indistinguishability* on points, by $(r,m) \sim_i (r',m')$ if $r_i(m) = r'_i(m')$. Intuititively, when $(r,m) \sim_i (r',m')$, agent i has failed to receive enough information to time m in run r and time m' in run r' to determine whether it is on one situation or the other. Clearly, each $\sim_i$ is an equivalence relation. The use of the term "synchronous" above is due to the fact that an agent is able to determine the time simply by counting the number of observations in its local state. This is reflected in the fact that if $(r,m) \sim_i (r',m')$, we must have $m = m'$.

To specify systems, we will use a propositional multimodal language for knowledge and linear time based on a set *Prop* of atomic propositions, with formulae generated by the modalities $\bigcirc$ (next time), U (until), and a knowledge operator K_i for each agent $i = 1\ldots n$. More precisely, the set of formulae of the language is defined as follows: each atomic proposition $p \in Prop$ is a formula, and if φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\bigcirc\varphi$, $\varphi U \psi$, $K_i\varphi$ for each $i = 1\ldots n$. As usual, we use the abbrevations $\Diamond\varphi$ for $\mathbf{true}\, U\, \varphi$, and $\Box\varphi$ for $\neg\Diamond\neg\varphi$.

The semantics of this language is defined as follows. Suppose we are given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of runs of environment E and π is determined from the environment as described above. We define satisfaction of a formula φ at a point (r,m) of a run in $\mathcal{R}$, denoted $\mathcal{I},(r,m) \models \varphi$, inductively on the structure of φ. The cases for the temporal fragment of the language are standard:

1. $\mathcal{I},(r,m) \models p$, where p is an atomic proposition, if $\pi(r,m)(p) = 1$,
2. $\mathcal{I},(r,m) \models \varphi_1 \wedge \varphi_2$, if $\mathcal{I},(r,m) \models \varphi_1$ and $\mathcal{I},(r,m) \models \varphi_2$,
3. $\mathcal{I},(r,m) \models \neg\varphi$, if not $\mathcal{I},(r,m) \models \varphi$,
4. $\mathcal{I},(r,m) \models \bigcirc\varphi$, if $\mathcal{I},(r,m+1) \models \varphi$,
5. $\mathcal{I},(r,m) \models \varphi_1 U \varphi_2$, if there exists $k \geq m$ such that $\mathcal{I},(r,k) \models \varphi_2$ and $\mathcal{I},(r,l) \models \varphi_1$ for all l with $m \leq l < k$.

The semantics of the knowledge operators is defined by

6. $\mathcal{I},(r,m) \models K_i\varphi$, if $\mathcal{I},(r',m') \models \varphi$ for all points (r',m') of $\mathcal{I}$ satisfying $(r',m') \sim_i (r,m)$

That is, an agent knows a formula to be true if this formula holds at all points that it is unable to distinguish from the actual point. This definition follows the general framework for the semantics of knowledge proposed by Halpern and Moses [14]. We use the particular equivalence relations $\sim_i$ obtained from the assumption of synchronous perfect recall, but the same semantics for knowledge applies for other ways of defining local states, and hence the relations $\sim_i$. We refer the reader to [14, 11] for further background on this topic.

The systems $\mathcal{I}$ we will be interested in will not have completely arbitrary sets of runs, but rather will have sets of runs that arise from the agents running some program, or protocol, within a given environment. Intuitively, an agent's actions in such a program should depend on the information it has been able to obtain about the environment, but no more. We have used observations to model the agent's source of information about the environment. The maximum information that an agent has about the environment at a point (r,m) is given by the local state $r_i(m)$. Thus, it is natural to model an agent's program as assigning to each local state of the agent an action for that agent. We define a *protocol* for agent i to be a function $P_i : \mathcal{O}^+ \to ACT_i$. A *joint protocol* $\mathbf{P}$ is a tuple $\langle P_1, \ldots, P_n \rangle$, where each P_i is a protocol for agent i.

The systems we consider will consist of all the runs in which at each point of time each agent behaves as required by its protocol. As usual, we also require that the environment follows its own protocol. Formally, the *system generated by a joint protocol* $\mathbf{P}$ *in environment* E is the set $\mathcal{R}(\mathbf{P}, E)$ of all runs r of E such that for all $m \geq 0$ we have $r_e(m+1) = \tau(\mathbf{a})(r_e(m))$, where $\mathbf{a}$ is the joint action $\langle P_e(r_e(m)), P_1(r_1(m)), \ldots, P_n(r_n(m))\rangle$. The *interpreted system generated by a joint protocol* $\mathbf{P}$ *in environment* E is the interpreted system $\mathcal{I}(\mathbf{P}, E) = (\mathcal{R}(\mathbf{P}, E), \pi)$, where π is the interpretation derived from the environment E as described above.

Finally, we may define the relation between specifications and implementations that is our main topic of study. We say that a joint protocol $\mathbf{P}$ *realizes* a specification φ in an environment E if for all runs r of $\mathcal{I}(\mathbf{P}, E)$ we have $\mathcal{I}, (r, 0) \models \varphi$. A specification φ is *realizable* in environment E if there exists a joint protocol $\mathbf{P}$ that realizes φ in E. The following example illustrates the framework and provides examples of realizable and unrealizable formulae.

Example 1. Consider a timed toggle switch with two positions (on, off), with a light intended to indicate the position. If the light is on, then the switch must be in the on position. However, the light is faulty, so it might be off when the switch is on. Suppose that there is a single agent that has two actions: "toggle" and "do nothing". If the agent toggles, the switch changes position. If the agent does nothing, the toggle either stays in the same position or, if it is on, may timeout and switch to off automatically. The timer is unreliable, so the timeout may happen any time the switch is on, or never, even if the switch remains on forever. The agent observes only the light, not the toggle position.

This system may be represented as an environment with states consisting of pairs $\langle t, l \rangle$, where t is a boolean variable indicating the toggle position and l is a boolean variable representing the light, subject to the constraint that $l = 0$ if $t = 0$. The agent's observation function is given by $O_1(\langle t, l \rangle) = l$. To represent the effect of the agent's actions on the state, write T for the toggle action and Λ for the agent's null action. The environment's actions may be taken to be pairs (u, v) where u and v are boolean variables indicating, respectively, that the environment times out the toggle, and that it switches the light on (provided the switch is on). Thus the transition function is given by $\tau(\langle (u, v), a_1 \rangle)(\langle t, l \rangle) = \langle t', l' \rangle$ where (i) $t' = \bar{t}$ if either $a_1 = T$ or $t = u = 1$, else $t' = t$, and (ii) $l' = 1$ iff $t' = 1$ and $v = 1$.

If "`toggle-on`" is the proposition true in states $\langle t, l \rangle$ where $t = 1$, then the formula $\Box(K_1 \texttt{toggle-on} \vee K_1 \neg\texttt{toggle-on})$ expresses that the agent knows at all times whether or not the toggle is on. This formula is realizable when the initial states of the environment are those in which the toggle is on (and the light is either on or off). The protocol by which the agent realizes this formula is that in which it performs T at all steps. Since it has perfect recall it can determine whether the toggle is on or off by checking if it has made (respectively) an odd or an even number of observations.

However, the same formula is not realizable if all states are initial. In this case, if the light is off at time 0, the agent cannot know whether the switch is on. As it has had at time 0 no opportunity to influence the state of the environment through its actions, this is the case whatever the agent's protocol. $\Box$

3 A Characterization of Realizability

In this section we characterize realizability in environments for a single agent in terms of the existence of a certain type of labelled tree. Intuitively, the nodes of this tree correspond to the local states of the agent, and the label at a node is intended to express (i) the relevant knowledge of the agent and (ii) the action the agent performs when in the corresponding local state.

Consider $\mathcal{O}^*$, the set of all finite sequences of observations of agent 1, including the empty sequence. This set may be viewed as an infinite tree, where the root is the null sequence and the successors of a vertex $v \in \mathcal{O}^*$ are the vertices $v \cdot o$, where $o \in \mathcal{O}$ is an observation. A *labelling* of $\mathcal{O}^*$ is a function $\mathcal{T} : \mathcal{O}^* \to L$ for some set L. We call $\mathcal{T}$ a *labelled tree*. We will work with trees in which the labels are constructed from the states of the environment, a formula ψ and the actions of the agent. Define an *atom* for a formula ψ to be a mapping X from the set of all subformulae of ψ to $\{0,1\}$. A *knowledge set* for ψ in E is a set of pairs of the form (X, s), where X is an atom of ψ and s is a state of E. Take $L_{\psi,E}$ to be the set of all pairs of the form (K, a) where K is a knowledge set for ψ in E and a is an action of agent 1. We will consider trees that are labellings of $\mathcal{O}^*$ by $L_{\psi,E}$. We will call such a tree a *labelled tree for* ψ *and* E.

Given such a labelled tree $\mathcal{T}$, we may define the functions K, mapping $\mathcal{O}^*$ to knowledge sets, and P, mapping $\mathcal{O}^*$ to actions of agent 1, such that for all $v \in \mathcal{O}^*$ we have $\mathcal{T}(v) = (K(v), P(v))$. Note that P is a protocol for agent 1. This protocol generates an interpreted system $\mathcal{I}(P, E)$ in the given environment E. Intuitively, we are interested in trees in which the $K(v)$ describe the states of knowledge of the agent in this system. We now set about stating some constraints on the labels in the tree $\mathcal{T}$ that are intended to ensure this is the case.

Suppose we are given a sequence of states $r = s_0 s_1 \dots$ and a vertex v of $\mathcal{T}$ with $v = w \cdot O_1(s_0)$ for some w. Then we obtain a branch $v_0 v_1 \dots$ of $\mathcal{T}$, where $v_0 = v$ and $v_m = v_{m-1} \cdot O_1(s_m)$ for $m > 0$. We say that r is a *run of* $\mathcal{T}$ *from* v if there exists an atom X such that $(X, s_0) \in K(v)$, and for each $m \geq 0$ there exists an action $a_e \in P_e(s_m)$ such that $s_{m+1} = \tau(\langle a_e, P(v_m)\rangle)(s_m)$. That is, the actions of agent 1 labelling the branch corresponding to r, together with some choice of the environment's actions, generate the sequence of states in the run.

We now define a relation $\models^*$ on points of the runs from vertices of $\mathcal{T}$. This relation interprets subformulae of ψ by treating the temporal operators as usual, but referring to the knowledge sets to interpret formulae involving knowledge. Intuitively, $\mathcal{T}, v, (r, m) \models^* \varphi$ asserts that the formula φ "holds" at the mth vertex v_m reached from v along r, as described above. More formally, this relation is defined by means of the following recursion:

1. $\mathcal{T}, v, (r, m) \models^* p$ if $\pi_e(r_e(m), p) = 1$
2. $\mathcal{T}, v, (r, m) \models^* \bigcirc\varphi$ if $\mathcal{T}, v, (r, m+1) \models^* \varphi$
3. $\mathcal{T}, v, (r, m) \models^* \varphi_1 U \varphi_2$ if there exists $k \geq m$ such that $\mathcal{T}, v, (r, l) \models^* \varphi_1$ for $m \leq l < k$ and $\mathcal{T}, v, (r, k) \models^* \varphi_2$.
4. $\mathcal{T}, v, (r, m) \models^* K_1\varphi$ if $X(\varphi) = 1$ for all $(X, s) \in K(v_m)$, where v_m is determined as above.

We use the abbreviation $\mathcal{T}, (r, m) \models^* \varphi$ for $\mathcal{T}, r_1(0), (r, m) \models^* \varphi$. (The choice of the vertex $r_1(0)$ here is not really significant: it is not difficult to show that for all $k \leq m$ we have $\mathcal{T}, (r, m) \models^* \varphi$ iff $\mathcal{T}, r_1(k), (r[k..], m-k) \models^* \varphi$.)

Define a labelled tree $\mathcal{T}$ for ψ and E to be *acceptable* if it satisfies the following conditions:

(Real) For all observations o, and for all $(X, s) \in K(o)$, we have $X(\psi) = 1$.

(Init) For initial states $s \in I_e$, there exists an atom X for ψ such that (X, s) is in $K(O_1(s))$.

(Obs) For all observations o and all vertices v of $\mathcal{T}$, we have $O_1(s) = o$ for all $(X, s) \in K(v \cdot o)$.

(Pred) For all observations o, for all vertices v other than the root, and for all $(X, s) \in K(v \cdot o)$, there exists $(Y, t) \in K(v)$ and an action $a_e \in P_e(t)$ such that $s = \tau(\langle a_e, P(v)\rangle)(t)$.

(Succ) For all vertices v other than the root, for all $(X, s) \in K(v)$ and for all $a_e \in P_e(s)$, if $t = \tau(\langle a_e, P(v)\rangle)(s)$ then there exists an atom Y such that $(Y, t) \in K(v \cdot O_1(t))$.

(Ksound) For all vertices v (other than the root) and all $(X, s) \in K(v)$, there exists a run r from v such that $r_e(0) = s$ and for all subformulae φ of ψ we have $\mathcal{T}, v, (r, 0) \models^* \varphi$ iff $X(\varphi) = 1$.

(Kcomp) For all vertices v and all runs r from v there exists $(X, s) \in K(v)$ such that $r_e(0) = s$ and for all subformulae φ of ψ we have $\mathcal{T}, v, (r, 0) \models^* \varphi$ iff $X(\varphi) = 1$.

The following theorem provides the characterization of realizability of knowledge-based specifications that forms the basis for our synthesis procedure.

Theorem 1. *A specification ψ for a single agent is realizable in the environment E iff there exists an acceptable labelled tree for ψ in E.*

Proof: (Sketch) We first show that if there exists an acceptable tree then the specification is realizable. Suppose $\mathcal{T}$ is an acceptable tree for ψ in E. Let P be the protocol for agent 1 derived from this tree, and let $\mathcal{I}$ be the system generated by P in E. We show that for all points (r, m) of $\mathcal{I}$ and all subformulae φ of ψ we have $\mathcal{T}, (r, m) \models^* \varphi$ iff $\mathcal{I}, (r, m) \models \varphi$. It follows from this using Init, Kcomp and Real that P realizes ψ in E.

Next, we show that if ψ is realizable in E then there exists an acceptable tree for ψ and E. Suppose that the protocol P for agent 1 realizes ψ in E. We construct a labelled tree $\mathcal{T}$ as follows. Let $\mathcal{I}$ be the system generated by P in E. If (r, m) is a point of $\mathcal{I}$, define the atom $X(r, m)$ by $X(r, m)(\varphi) = 1$ iff $\mathcal{I}, (r, m) \models \varphi$. Define the function f to map the point (r, m) of $\mathcal{I}$ to the pair $(X(r, m), r_e(m))$. For all v in $\mathcal{O}^+$, define $K(v)$ to be the set of all $f(r, m)$, where (r, m) is a point of $\mathcal{I}$ with $r_1(m) = v$. Define $\mathcal{T}$ by $\mathcal{T}(v) = (K(v), P(v))$ for each $v \in \mathcal{O}^+$. (The label of the root can be chosen arbitrarily.) We may then show that $\mathcal{T}$ is an acceptable tree for ψ and E. ∎

In the next section, we show how this result can be used to yield an automata-theoretic procedure for constructing a realization of a specification.

4 An Algorithm for Realizability

4.1 Automata on Infinite Words

The types of finite automata on infinite words we consider are those defined by Büchi [5]. A (nondeterministic) automaton on words is a tuple $\mathcal{A} = \langle \Sigma, S, S_0, \rho, \alpha \rangle$, where Σ is a finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is a set of starting states, $\rho : S \times \Sigma \to 2^S$ is a (nondeterministic) transition function, and α is an acceptance condition. A Büchi acceptance condition is a set $F \subseteq S$.

A *run* r of $\mathcal{A}$ over a infinite word $w = a_0 a_1 \cdots$, is a sequence $s_0, s_1, \cdots$, where $s_0 \in S_0$ and $s_i \in \rho(s_{i-1}, a_{i-1})$, for all $i \geq 1$. Let $\mathit{inf}(r)$ denote the set of states in Q

that appear in $r(\rho)$ infinitely often. The run r satisfies a Büchi condition F if there is some state in F that repeats infinitely often in r, i.e., $F \cap \mathit{inf}(r) \neq \emptyset$. The run r is *accepting* if it satisfies the acceptance condition, and the infinite word w is *accepted* by $\mathcal{A}$ if there is an accepting run of A over w. The set of infinite words accepted by $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A})$.

The following theorem establishes the correspondence between temporal formulae and Büchi automata.

Proposition 1. [32] *Given a temporal formula φ over a set Prop of propositions, one can build a Büchi automaton $\mathcal{A}_\varphi = \langle 2^{Prop}, S, S_0, \rho, F\rangle$, where $|S| \leq 2^{O(|\varphi|)}$, such that $\mathcal{L}(\mathcal{A}_\varphi)$ is exactly the set of computations satisfying the formula φ.*

4.2 Alternating Automata on Infinite Trees

Alternating tree automata generalize nondeterministic tree automata and were first introduced in [26]. They have recently found usage in computer-aided verification [34]. An alternating tree automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha\rangle$ runs on Σ-labelled Υ-trees (i.e., mappings from Υ^* to Σ). It consists of a finite set Q of states, an initial state $q_0 \in Q$, a transition function δ, and an acceptance condition α (a condition that defines a subset of Q^ω).

For a set D, let $\mathcal{B}^+(D)$ be the set of positive Boolean formulae over D; i.e., Boolean formulae built from elements in D using $\wedge$ and $\vee$, where we also allow the formulae **true** and **false**. For a set $C \subseteq D$ and a formula $\theta \in \mathcal{B}^+(D)$, we say that C *satisfies* θ iff assigning **true** to elements in C and assigning **false** to elements in $D \setminus C$ makes θ true.

The transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(\Upsilon \times Q)$ maps a state and an input letter to a formula that suggests a new configuration for the automaton. A *run* of an alternating automaton $\mathcal{A}$ on an input Σ-labelled Υ-tree $\mathcal{T}$ is a tree $\langle T_r, r\rangle$ in which the root is labelled by q_0 and every other node is labelled by an element of $\Upsilon^* \times Q$. Each node of T_r corresponds to a node of Υ^*. A node in T_r, labelled by (x, q), describes a copy of the automaton that reads the node x of Υ^* and visits the state q. Formally, $\langle T_r, r\rangle$ is a Σ_r-labeled tree where $\Sigma_r = \Upsilon^* \times Q$ and $\langle T_r, r\rangle$ satisfies the following:

1. $\epsilon \in T_r$ and $r(\epsilon) = (\epsilon, q_0)$.
2. Let $y \in T_r$ with $r(y) = (x, q)$ and $\delta(q, \mathcal{T}(x)) = \theta$. Then there is a (possibly empty) set $S = \{(c_1, q_1), \ldots, (c_n, q_n)\} \subseteq \Upsilon \times Q$, such that the following hold:
   - S satisfies θ, and
   - for all $1 \leq i \leq n$, we have $y \cdot i \in T_r$ and $r(y \cdot i) = (x \cdot c_i, q_i)$.

For example, if $\langle T, V\rangle$ is a $\{0,1\}$-tree with $V(\epsilon) = a$ and $\delta(q_0, a) = ((0, q_1) \vee (0, q_2)) \wedge ((0, q_3) \vee (1, q_2))$, then the nodes of $\langle T_r, r\rangle$ at level 1 include the label $(0, q_1)$ or $(0, q_2)$, and include the label $(0, q_3)$ or $(1, q_2)$.

Each infinite path ρ in $\langle T_r, r\rangle$ is labelled by a word $r(\rho)$ in Q^ω. A run $\langle T_r, r\rangle$ is accepting iff all its infinite paths satisfy the acceptance condition. Let $\mathit{inf}(\rho)$ denote the set of states in Q that appear in $r(\rho)$ infinitely often. In a Büchi acceptance condition, $\alpha \subseteq Q$ and an infinite path ρ satisfies an acceptance condition α if $\alpha \cap \mathit{inf}(\rho) \neq \emptyset$, In a co-Büchi acceptance condition, $\alpha \subseteq Q$ and an infinite path ρ satisfies an acceptance condition α if $\alpha \cap \mathit{inf}(\rho) = \emptyset$, In a *Rabin* acceptance condition, $\alpha \subseteq 2^Q \times 2^Q$, and an infinite path ρ satisfies an acceptance condition $\alpha = \{\langle G_1, B_1\rangle, \ldots, \langle G_m, B_m\rangle\}$ iff there exists $1 \leq i \leq m$ for which $\mathit{inf}(\rho) \cap G_i \neq \emptyset$ and $\mathit{inf}(\rho) \cap B_i = \emptyset$. As with nondeterministic automata, an automaton accepts a tree iff there exists an accepting run on it. We denote by $\mathcal{L}(\mathcal{A})$ the language of the automaton $\mathcal{A}$; i.e., the set of all labelled trees that $\mathcal{A}$ accepts. $\mathcal{A}$ is *empty* if $\mathcal{L}(\mathcal{A}) = \emptyset$.

Nondeterministic tree automata are a special case of alternating tree automata. An automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$ is nondeterministic if, for each state $q \in Q$ and letter $a \in \Sigma$, the formula $\delta(q, a)$ does not contain two pairs (c, q_1) and (c, q_2), where $q_1 \neq q_2$, that are conjunctively related (i.e., both appear in the same disjunct of the disjunctive normal form of $\delta(q, a)$). Intuitively, it means that the automaton cannot send two distinct copies in the same direction [26].

Proposition 2. [27] *Given an alternating Rabin automaton with n states and m pairs, we can translate it into an equivalent nondeterministic Rabin automaton with $(mn)^{O(mn)}$ states and mn pairs.*

Proposition 3. [10, 29, 18] *Emptiness of a nondeterministic Rabin automaton with n states and m pairs over an alphabet with l letters can be tested in time $(lmn)^{O(m)}$.*

4.3 Realizability

We now derive an automata-theoretic algorithm for realizability for knowledge-based specifications involving a single agent.

Theorem 2. *There is an algorithm that constructs for a given specification ψ and an environment E an nondeterministic Rabin automaton $\mathcal{A}_{\psi,E}$ such that $\mathcal{A}_{\psi,E}$ accepts precisely the acceptable trees for ψ in E. $\mathcal{A}_{\psi,E}$ has $2^{||E|| \cdot 2^{O(||\psi||)}}$ states and $||E|| \cdot 2^{O(||\psi||)}$ pairs.*

Proof: (sketch) The inputs to the automaton $\mathcal{A}_{\psi,E}$ are $L_{\psi,E}$-labeled trees. Note that the size of $L_{\psi,E}$ is exponential in the number of states in E and doubly exponential in the length of ψ. To check that an input tree $\mathcal{T}$ is acceptable, the automaton has to check that it satisfies the properties Real, Init, Obs, Pred, Succ, Ksound, and Kcomp. We describe automata that check these properties; $\mathcal{A}_{\psi,E}$ is obtained as the intersection of these automata. The two non-trivial cases are Ksound and Kcomp.

To check Ksound, an alternating automaton guesses, for all vertices v (other than the root) and all $(X, s) \in K(v)$, a run r from v such that $r_e(0) = s$ and for all subformulae φ of ψ we have $\mathcal{T}, v, (r, 0) \models^* \varphi$ iff $X(\varphi) = 1$. A formula ξ can be viewed as a temporal formula by considering every subformula $K\theta$ as a new proposition. Consider the formula ψ_X that is obtained by taking the conjunction of subformulae of ψ or their negation according to X. We consider ψ_X as a temporal formula and appeal to Theorem 1 to construct a Büchi automaton A_{ψ_X} that check whether ψ_X is satisfied by sequence of truth assignments to its extended set of propositions (i.e., atomic propositions and subformulae of the form $K\theta$). Thus, the automaton guesses a sequence $v_0, v_1, \ldots$ of nodes in the tree and a sequence $(X_0, s_0), (X_1, s_1), \ldots$ of atom-state pairs such that $v_0 = v$, $X_0 = X$, $s_0 = s$, v_{i+1} is a child of v_i, $(X_i, s_i) \in K(v_i)$, and $s_{i+1} = \tau(\langle a_e, P(v_i) \rangle)(s_i)$ for some $a_e \in P_e(s_i)$. It then emulates A_{ψ_X} and checks that the sequence $X_0, X_1, \ldots$ is accepted. This automaton has $||E|| \cdot 2^{O(||\psi||)}$ states and a Büchi acceptance condition.

Instead of checking that Kcomp holds, we construct an alternating automaton that checks that Kcomp is violated, since alternating automata can be complemented by dualizing their transition function (i.e., switching $\vee$ and $\wedge$ as well as **true** and **false**) and complementing the acceptance condition [26]. The automaton guesses a vertex v

and a run r from v such that for no $(X, s) \in K(v)$ we have that $r_e(0) = s$ and for all subformulae φ of ψ we have $\mathcal{T}, v, (r, 0) \models^* \varphi$ iff $X(\varphi) = 1$. We already saw how the automaton guesses a run; it guesses a sequence $v_0, v_1, \ldots$ of nodes in the tree and a sequence $(X_0, s_0), (X_1, s_1), \ldots$ of atom-state pairs such that $v_0 = v$, $(Y_0, s_0) \in K(v)$ for some atom Y_0, but $(X_0, s_0) \notin K(v)$, v_{i+1} is a child of v_i, $(Y_i, s_i) \in K(v_i)$ for some atom Y_i, and $s_{i+1} = \tau(\langle a_e, P(v_i)\rangle)(s_i)$ for some $a_e \in P_e(s_i)$. It then emulates $A_{\psi_{X_0}}$ and checks that the sequence $X_0, X_1, \ldots$ is accepted. This automaton has $||E|| \cdot 2^{O(||\psi||)}$ states. After complementing it, it has a co-Büchi acceptance condition.

We now apply Proposition 2 to the alternating automata that check Ksound and Kcomp to get nondeterministic Rabin automata with $2^{||E|| \cdot 2^{O(||\psi||)}}$ states and $||E|| \cdot 2^{O(||\psi||)}$ pairs. ∎

Corollary 1. *There is an algorithm that decides whether a formula ψ is realizable in an environment E in time $2^{O(||E||)} \cdot 2^{2^{O(||\psi||)}}$.*

Proof: By Theorem 2, ψ is realizable in E iff $\mathcal{L}(\mathcal{A}_{\psi,E}) \neq \emptyset$. The claim now follows by Proposition 3, since $\mathcal{A}_{\psi,E}$ has Rabin automata with $2^{||E|| \cdot 2^{O(||\psi||)}}$ states and $||E|| \cdot 2^{O(||\psi||)}$ pairs and the alphabet has $2^{||E|| \cdot 2^{O(||\psi||)}}$ letters. ∎

We note that it is shown in [29] that realizability of temporal formulae with complete information is already 2EXPTIME-hard. Thus, the bound in Corollary 1 is essentially optimal.

So far our focus was on realizability. Recall, however, that if $\mathcal{T}$ is an acceptable tree for ψ in E, then the protocol P for agent 1 derived from this tree realizes ψ in E. The emptiness-testing algorithm used in the realizability test (per Proposition 3) does more than just test emptiness. When the automaton is nonempty the algorithm returns a *finitely-generated* tree, which, as shown in [6], can be viewed as a finite-state protocol. We return to this point in the following section.

5 Knowledge in the Implementation

In this section we remark upon a subtle point concerning the states of knowledge attained in protocols realizing a specification. As these remarks apply equally to the general multi-agent framework we have defined, we return to this context.

We have defined local states, hence the semantics of knowledge, using the assumption of synchronous perfect recall, which involves an infinite space of local states. A protocol realizing a specification is not required to have perfect recall, and could well be represented (like the protocol synthesized by our procedure) using a finite set of states. The sense in which such a protocol satisfies the conditions on knowledge stated by the specification is the following: an agent that follows the actions prescribed by the protocol, but computes its knowledge based on the full record of its observations, satisfies this specification. Thus, although we may have a finite-state protocol, it appears that we have not in actuality eliminated the need to maintain an unbounded log of all the agent's observations. If this is so then the system is better characterized as consisting of an infinite state space coupled to a finite-state controller.

Now there are situations in which we can dispense with the observation logs, leaving just the finite-state controller. This holds when, although we state the specification in knowledge-theoretic terms, we are more concerned with the *behavior* of the synthesized system than the information encoded in its states. For example, Halpern and Zuck [15] give a knowledge-based specification (in the form of a knowledge-based program) of solutions to a sequence transmission problem. They start with the assumption of perfect recall, but their ultimate interest is to develop implementations for this specification that optimize the memory maintained by agents while preserving their behaviour. One of the implementations they consider, the alternating-bit protocol [3], is a finite-state protocol.

Although in some cases one is concerned only with behavior, in others what one has in mind in writing a knowledge-based specification is to construct an implementation whose states have the information-theoretic property expressed. This is the case when the states of knowledge in question function as an output of the system, or provide inputs to some larger module. For example, we might specify that a controller for a nuclear reactor must keep the reactor temperature below a certain level and must also know of a critical level of radiation whenever this condition holds, with the intention that this information be provided to the operator. In this case it will not do to implement the specification according to its behavioral component alone, since this might lose the attribute, knowledge of radioactivity, that we wish to present as an output.

Clearly, we could always ensure that the knowledge properties specified are available in the implementation by taking the implementation to consist of both the finite-state controller and the log of all the agent's observations. Such an implementation is rather inefficient. Can we do better? One attempt to do so would be simply to take the implementation to consist just of the protocol, and to compute knowledge on the basis of the protocol states.

To make this idea precise, we adopt the following model of a protocol and the knowledge it encodes. We suppose that agent i's protocol is represented as an automaton $A_i = \langle Q_i, q_i, \mu_i, \alpha_i \rangle$, where Q_i is the set of *protocol states*, $q_i \in Q$ is the initial state, $\mu_i : Q_i \times \mathcal{O} \to Q_i$ is the state transition function, used to update the protocol state given an observation in $\mathcal{O}$, and $\alpha_i : Q_i \to ACT_i$ is a function mapping each state to an action of the agent. As usual, we define the state reached after a sequence σ of inputs (i.e., observations of the agent) by $A_i(\epsilon) = q_i$ and $A_i(\sigma \cdot o) = \mu_i(A_i(\sigma), o)$. We may then define the protocol itself, as a function from sequences of observations to actions, by $P_{A_i}(\sigma) = \alpha_i(A_i(\sigma))$.

Suppose we are given a tuple $A = \langle A_1, \ldots, A_n \rangle$ of automata representing the protocols of agents $1 \ldots n$. To interpret the knowledge operators with respect to the states of such these automata, we first define for each agent i an indistinguishability relation $\approx_i^A$ on points, based on the states of the automata A_i rather than the perfect-recall local states used for the relation $\sim_i$ above. That is, we define $(r, m) \approx_i^A (r', m')$ to hold when $A_i(r_i(m)) = A_i(r'_i(m'))$. We may now define the semantics of knowledge exactly as we did using the relation $\sim_i$. To distinguish the two interpretations, we introduce new knowledge modalities K_i^A, and define $\mathcal{I}, (r, m) \models K_i^A \varphi$ if $\mathcal{I}, (r', m') \models \varphi$ for all points (r', m') of $\mathcal{I}$ satisfying $(r', m') \approx_i^A (r, m)$. We may now formulate the proposal above as follows. Suppose a specification φ is realized in an environment E by a joint protocol $\mathbf{P}_A$, represented by the automata A. Is it then the case that this joint protocol realizes in E the specification φ_A obtained from φ by replacing (recursively) each subformula $K_i \psi$ with $K_i^A \psi$? It is not, as the following example shows.

Example 2. The protocol in Example 1, which performs the toggle action at all steps, can be represented by an automaton A with a single state. This protocol realizes the specification $\Box(K_1 \texttt{toggle-on} \lor K_1 \neg\texttt{toggle-on})$. However, with respect to the automaton A, the formula $\Box(K_1^A \texttt{toggle-on} \lor K_1^A \neg\texttt{toggle-on})$ is false at time 0 in a run generated by the protocol. For, at even numbered points on these runs the toggle is on and at odd points the toggle is off, and the single state does not suffice to distinguish the two. □

Nevertheless, a slight modification of the proposal makes it possible to ensure that the protocols realizing a specification have the desired information theoretic property. All that is required is to reflect an agent's knowledge according to the perfect-recall definition in its behavior. To do so, we first modify the environment so that an agent is provided with actions that allow it to assert what it knows, and then add a constraint to the specification that requires agents to assert their knowledge truthfully.

We will just sketch the construction here, and provide further details in the full version of the paper. Suppose for each agent i, we have a finite set Φ_i containing all the knowledge formulae of the form $K_i\varphi$ that we wish the implementation to preserve. If such a set contains a formula with a subformula $K_j\varphi'$ for some j, then we require that Φ_j contains this subformula. Let Φ be the union of the Φ_i. The modification of the environment involves adding to each agent's actions a component in which the agent asserts a subset of Φ_i. That is, we take ACT'_i to be $ACT_i \times \mathcal{P}(\Phi_i)$. We also modify the states S_e of the environment to be the set $S_e \times \mathcal{P}(\Phi_1) \times \ldots \times \mathcal{P}(\Phi_n)$. We take the effect of agent i's action $\langle a_i, \Psi_i\rangle$ to be to have a_i act on the state component from S_e exactly as in the environment E, but to additionally record the set Ψ_i in the appropriate component of the state. This makes it possible to extend the language by introducing for each formula $\psi \in \Phi_i$ an atomic proposition "$\mathrm{said}_i(\psi)$", with semantics given by $\pi'_e((s, \Psi_1, \ldots, \Psi_n), \mathrm{said}_i(\psi)) = 1$ iff $\psi \in \Psi_i$. Call the resulting environment E'.

Suppose now that we are given a specification φ, for which we wish to view the knowledge formulae in the set Φ as outputs of the system. Define $\mathrm{Say}(\Phi)$ to be the formula $\bigwedge_{K_i\psi\in\Phi} \Box(K_i\psi \equiv \bigcirc\ \mathrm{said}_i(K_i\psi))$, which asserts that agents say what they know (according to perfect recall.) Additionally, define $\mathrm{Know}(\Phi)$ to be the formula $\bigwedge_{K_i\psi\in\Phi} \Box(K_i\psi \equiv K_i^A\psi))$, which says that each agent knows a fact in Φ according to its protocol just when it knows this fact using perfect recall. We then have the following result.

Proposition 4. *The formula $\varphi \wedge \mathrm{Know}(\Phi)$ is realizable in E iff the formula $\varphi \wedge \mathrm{Say}(\Phi)$ is realizable in E'. Moreover, there exists a finite-state realization of one iff there exists a finite-state realization of the other.*

Intuitively, this result holds because the implementation can only behave as specified by $\varphi \wedge \mathrm{Say}(\Phi)$ if the protocol states encode the relevant knowledge. This result shows that, provided some care is taken in writing specifications, the realizability framework we have defined in this paper is capable of handling both the case in which agents are required simply to behave as if they had perfect recall, and the case in which agents are required both to behave in this fashion *and* encode certain perfect-recall knowledge in their protocol states.

In particular, in the single agent case, if we apply the synthesis procedure of the previous section to the specification $\varphi \wedge \mathrm{Say}(\Phi)$, we obtain a protocol that represents knowledge defined according to the perfect-recall semantics, but using only a finite number of states.

6 Conclusion

The techniques we have developed in this extended abstract are able to handle a number of generalizations of the model we have considered.

Two possible views regarding the nature of time induce two types of temporal logics [20, 9]. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. In *branching* temporal logics, each moment in time may split into various possible futures. The algorithm described in this paper handles linear-time knowledge-based specification. We show in the full paper how it can be extended to handle branching-time knowledge-based specifications (for this extension we consider nondeterministic protocols, in which the agent may have a choice of actions at each point in time). Moreover, we also show that this extension makes it possible to use our framework for the automated construction (in the single agent case) of implementations of *knowledge-based programs* [11, 12]. (These are programs in which an agent's actions are determined from its state of knowledge.)

We have been able to treat the case of single agent knowledge-based specifications in this paper. Is it possible to generalize our results to the multi-agent case? In general, it is not. Using ideas from Peterson and Reif's study of the complexity of multi-agent games of incomplete information [28], Pnueli and Rosner [31] show that realizability of linear temporal logic specifications in the context of two agents with incomplete information is undecidable. This result immediately applies to our more expressive specification language.

However, there are limited classes of situations in which realizability of temporal specifications for more than one agent with incomplete information is decidable, and for which one still obtains finite-state implementations. Pnueli and Rosner [31] show that this is the case for *hierarchical* agents. Roughly, this corresponds in our model to the assumption that the observation functions O_i have the property that for all states s and t of the environment, if $O_i(s) = O_i(t)$ then $O_{i+1}(s) = O_{i+1}(t)$. Intuitively, this means that agent 1 makes more detailed observations than agent 2, which in turn makes more detailed observations than agent 3, etc. Pnueli and Rosner's results suggest that realizability of knowledge-based specifications in hierarchical environments may also be decidable. We do not yet know if this is the case. It appears that the techniques we have developed in the present paper are too weak to resolve this question, but we are presently studying a more powerful automaton model that may lead to its resolution.

Other restrictions on the environment suggest themselves as candidates for generalization of our results. For example, whereas atemporal knowledge-based programs (in which conditions do not involve temporal operators) do not have finite-state implementations in general [25], in *broadcast* environments this is guaranteed [24]. Again, this suggests that realizability of knowledge specifications in broadcast environments is worth investigation, particularly as this is a very natural and applicable model.

Finally, one could also consider definitions of knowledge other than the perfect-recall interpretation that we have treated in this paper. In particular, one open question is whether it is decidable to determine the existence of a finite state automaton A realizing a specification stated using the knowledge operator K_1^A. The result of Section 5 provides only a sufficient condition for this.

References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372, pages 1–17. Lecture Notes in Computer Science, Springer-Verlag, July 1989.

2. A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In *Computer-Aided Verification, Proc. 6th Int'l Conference*, pages 156–169, Stanford, California, June 1994. Springer-Verlag, Lecture Notes in Computer Science 818.
3. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.
4. I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, Stanford, June 1994.
5. J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.
6. J.R. Büchi and L.HG. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.
7. D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.
8. E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
9. E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.
10. E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 368–377, White Plains, October 1988.
11. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.
12. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.
13. J. Y. Halpern. Using reasoning about knowledge to analyze distributed systems. In J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, editors, *Annual Review of Computer Science, Vol. 2*, pages 37–68. Annual Reviews Inc., Palo Alto, Calif., 1987.
14. J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.
15. J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.
16. R. Kumar and M.A. Shayman. Supervisory control of nondeterministic systems under partial observation and decentralization. *SIAM Journal of Control and Optimization*, 1995.
17. O. Kupferman and M.Y. Vardi. Synthesis with incomplete informatio. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997.
18. O. Kupferman and M.Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proc. 30 ACM Symp. on Theory of Computing*, pages 224–233, 1998.
19. R. E. Ladner and J. H. Reif. The logic of distributed protocols (preliminary report). In J. Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference*, pages 207–222. Morgan Kaufmann, San Francisco, Calif., 1986.

20. L. Lamport. "Sometimes" is sometimes "not never": on the temporal logic of programs. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 164–185, 1980.
21. D. Lehmann. Knowledge, common knowledge, and related puzzles. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 62–67, 1984.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.
23. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.
24. R. van der Meyden. Finite state implementations of knowledge-based programs. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer LNCS No. 1180, pages 262–273, Hyderabad, India, December 1996.
25. R. van der Meyden. Knowledge based programs: On the complexity of perfect recall in finite environments (extended abstract). In *Proceedings of the Conference on Theoretical Aspects of Rationality and Knowledge*, pages 31–50, 1996.
26. D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54,:267–276, 1987.
27. D.E. Muller and P.E. Schupp. Simulating aternating tree automata by nondeterministic automata: New results and new proofs of theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141:69–107, 1995.
28. G.L. Peterson and J.H. Reif. Multiple-person alternation. In *Proc. 20st IEEE Symposium on Foundation of Computer Science*, pages 348–363, 1979.
29. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, Austin, January 1989.
30. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372, pages 652–671. Lecture Notes in Computer Science, Springer-Verlag, July 1989.
31. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symposium on Foundation of Computer Science*, pages 746–757, 1990.
32. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
33. M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *Computer Aided Verification, Proc. 7th Int'l Conf.*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer-Verlag, Berlin, 1995.
34. M.Y. Vardi. Alternating automata – unifying truth and validity checking for temporal logics. In W. McCune, editor, *Proc. 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 191–206. Springer-Verlag, Berlin, july 1997.
35. H. Wong-Toi and D.L. Dill. Synthesizing processes and schedulers from temporal specifications. In E.M. Clarke and R.P. Kurshan, editors, *Computer-Aided Verification'90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 177–186. AMS, 1991.