An Automata-Theoretic Approach to Fair Realizability and Synthesis

(Preliminary Report)

Moshe Y. Vardi[1]*

Rice University, Department of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, U.S.A. Email: vardi@cs.rice.edu

Abstract. Over the last few years, the automata-theoretic approach to realizability checking and synthesis of reactive modules, developed by Pnueli and Rosner, by Abadi, Lamport, and Wolper, and by Dill and Wong-Toi, has been quite successful, handling both the synchronous and the asynchronous cases. In this approach one transforms the specification into a tree automaton. The specification is realizable if this tree automaton is nonempty, i.e., accepts some infinite tree, and a finite representation of an infinite tree accepted by this automaton can be viewed as a finite-state program realizing the specification. Anuchitanukul and Manna argued that the automata-theoretic approach cannot handle realizability checking and synthesis under fairness assumptions. In this paper we show to the contrary that the automata-theoretic approach can handle realizability checking and synthesis under a variety of fairness assumption.

1 Introduction

One of the most significant developments in the area of program verification over the last decade is the development of of algorithmic methods for verifying temporal specification of *finite-state* programs [CES86, LP85, QS81] (see overview in [CGL93]). The significance of this follows from the fact that a significant number of the communication and synchronization protocols studied in the literature are in essence finite-state programs or can be abstracted as finite-state program [Liu89, Rud87]. A frequent criticism against this approach, however, is that verification is done *after* significant resources have already been invested in the development of the program. Since programs invariably contains errors, verification simply becomes part of the debugging process. The critics argue that the desired goal is to use the specification in the program 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 functional programs, the specification is typically a first-order formula describing the desired relationship between inputs and outputs. In this case, a program that meets the specification can be extracted from a constructive proof that the formula is satisfiable [MW80]. For reactive programs, the specification is typically a temporal formula describing the allowable behaviors of the program. Emerson

* Part of this work was done at the IBM Almaden Research Center

and Clarke [EC82] and Manna and Wolper [MW84] 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 [Dil89, PR89a, ALW89]. 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 [EC82, MW84] to open systems, one obtains programs that can handle only certain actions of the environment.

Pnueli and Rosner [PR89a], Abadi et al. [ALW89], and Dill [Dil89] argued that the right way to approach synthesis of reactive programs is to consider the situation as a (possibly infinite) game between the environment and the program. A correct program can be then 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 exists *realizable*. A winning strategy can be viewed as an infinite tree. In both 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, and a finite representation of an infinite tree accepted by this automaton can be viewed as a finite-state program realizing the specification. (This is closely related to the approach taken by Büchi and Landweber [BL69] to solve Church's *solvability problem* [Chu63].) In [PR89b, WTD91] it was shown how this approach to program synthesis can be carried out in a variety of settings, both synchronous and asynchronous.

Anuchitanukul and Manna [AM94] argued that the automata-theoretic approach described above cannot handle realizability checking and synthesis under fairness assumptions. Fairness assumptions are assumptions that exclude certain "pathological" behaviors by the environment, such as *always* failing a link whenever a transmission is attempted across it [Fra86]. The problem of realizability checking and synthesis under fairness assumptions was first raised in [ALW89]. To quote from [AM94],

> "Since the synthesized program does not exists at the time the specification is written, there is no location or transition in the program to refer to in the specification. We cannot encode in the specification the assumption that the synthesized program is to be executed fairly. This problem arises regardless of the choice of the specification language."

Anuchitanukul and Manna presented realizability-checking and synthesis algorithms that handle fairness assumptions and do not use the automata-theoretic approach described above.

In this paper we show to the contrary that the automata-theoretic approach can handle realizability checking and synthesis under various fairness assumptions (cf. [LPS81]). In the same spirit as in [WTD91], we show that the original specification can be transformed to a constraint on the infinite branches of the strategy tree by a sequence of transformations of ω-automata. The key to these transformations is the closure of the set of ω-regular languages under complementation and projection [Büc62]. Once we have an automaton that constrains the branches of the strategy tree, we can follow the approach of [PR89a, ALW89] and transform this automaton into a tree automaton and then check for nonemptiness of the latter. The advantage of our solution over the solution in [AM94]

is that we avoid the need to implement their special algorithms. Any implementation of the algorithms in [PR89a, ALW89, WTD91] can be easily extended using our approach to handle a large class of fairness assumptions. We also show that realizability checking under fairness assumptions is complete for 2EXPTIME, even when the specification is given by a Büchi automaton; in contrast, "vanilla" realizability checking can be done in exponential time when the specification is given by a Büchi automaton [PR89a, ALW89].

2 Basic Definitions

In this section we describe, with minor modification, the framework of [AM94].

A *behavior* over a set S of states is an ω-word over the alphabet $S \times \{0,1\}$. The intended meaning is that the behavior $\langle s_0, a_0\rangle, \langle s_1, a_1\rangle, \ldots$ describes a sequence $s_0, s_1, \ldots$ of observable states that the system goes through, where the transition from s_{i-1} to s_i was caused by the environment when $a_i = 0$ and by the program when $a_i = 1$. When $a_i = 1$, we say that the program is scheduled at i. The set of behaviors over S is denoted by $bhv(S)$. Given a behavior $\sigma = \langle s_0, a_0\rangle, \langle s_1, a_1\rangle, \ldots$, we write $state(i, \sigma)$ to denote s_i and $sched(i, \sigma)$ to denote a_i. It is convenient to assume that $sched(0, \sigma) = 0$, i.e., the transition to the first state s_0 is always caused by the environment.

The program does not see all the states that the system goes through. The program only sees the state of the system when it is scheduled to make a transition and when it has caused a transition. Given a behavior $\sigma = \langle s_0, a_0\rangle, \langle s_1, a_1\rangle, \ldots$, if $a_i = 1$, then s_{i-1} is *pre-visible* and s_i is *post-visible*. Intuitively, s_{i-1} is a state in which the program is about to cause a transition and s_i is a state that the system entered as a result of a transition caused by the program; both states are visible to the program. We call $\langle s_{i-1}, s_i\rangle$ the *visible pair* at i. Let $i_0 < i_1 < \ldots < i_k$ be the sequence of points at which the program is scheduled before i (i.e., $a_{i_j} = 1$ for $j = 1, \ldots, k$). The *visible history* of σ at i, denote by $vis(i, \sigma)$, is the sequence of visible pairs before i follows by s_{i-1}, i.e., $\langle s_{i_1-1}, s_{i_1}\rangle, \langle s_{i_2-1}, s_{i_2}\rangle, \ldots, \langle s_{i_k-1}, s_{i_k}\rangle, s_{i-1}$. For example, if if $a_1 = a_2 = 1$, then the visible history at $i = 3$ is $\langle s_0, s_1\rangle, \langle s_1, s_2\rangle, s_2$. The repetition of the states in the visible history stems from the fact that the program cannot tell whether the two occurrences of, say, s_1 happened at the same time or not. It is clear that the action of the program at i should depend only on $vis(i, \sigma)$. Note that visible histories can be viewed as words in $(S^2)^*S$.

A *program* $f : (S^2)^*S \to S$ is a partial function from visible histories to states. The idea is that if the program is scheduled at a point at which the visible history is h, then the program will cause a change into the state $f(h)$. Thus, the actions of the program depend only on the history visible to the program. If $f(h)$ is defined, then the program is *enabled* when its visible history is h. Thus, if σ is a behavior, then we say that the program is enabled at i if $f(vis(i, \sigma))$ is defined, denoted $f(vis(i, \sigma))\downarrow$.

A behavior σ is a *run* of a program f if $s_i = f(vis(i, \sigma))$ for all i such that $a_i = 1$. That is, all the state transitions caused by the program are consistent with the program f. A run σ of f is *impartial*, if the program is scheduled infinite often in σ, i.e., if $sched(i, \sigma) = 1$ for infinitely many i's. A run σ of f is *just*, if the program is scheduled infinitely often if it is enabled continuously, i.e., if there exists some i_0 such that the program is enabled at all $i > i_0$, then $sched(j, \sigma) = 1$ for infinitely many j's. A run

σ of f is *fair*, if the program is scheduled infinitely often when it is enabled infinitely often, i.e., if the program is enabled at infinitely many i's, then $sched(j, \sigma) = 1$ for infinitely many i's. Note that if f is empty (i.e., nowhere defined), then the program is never enabled, and any behavior where the program is never scheduled is a fair run of f.

Let $runs(f)$ (resp., $runs_{imp}(f)$, $runs_{just}(f)$, $runs_{fair}(f)$ be the set of all runs (resp., impartial runs, just runs, fair runs) of f. A set B of allowable behaviors is *realized* (resp., *impartially realized, justly realized, fairly realized*) by a program f if $runs(f) \subseteq B$ (resp., $runs_{imp}(f) \subseteq B$, $runs_{just}(f) \subseteq B$, $runs_{fair}(f) \subseteq B$). We say that B is *realizable* under a fairness condition Θ if it is Θ-realized by some program f. Thus, B is realizable (under the appropriate fairness assumption) if there is a program that guarantees that, regardless of the transitions caused by the environment and the environment scheduling choices, the resulting behavior (under the appropriate fairness assumption) will be in B. The intuition is that the specification can constrain the program, but cannot constrain the environment. A specification that constrains the environment is unrealizable.

We allow for a variety of formalisms in which the set B of allowable behavior is expressed: LTL [Pnu77], ETL [Wol83], temporal fixpoint calculus [Var88] and the like. The crucial point is that we should be able to convert the specification into a Büchi automaton [VW94]. Thus, we assume that we are given a Büchi automaton $\mathcal{A}$ over the alphabet $S \times \{0, 1\}$ such that $B = L_\omega(\mathcal{A})$, i.e., B is the set of infinite words accepted by $\mathcal{A}$. (The cost of going from a temporal logic formula to an equivalent Büchi automaton is typically exponential [VW94, SV89]. We will come back to this point later.) We will often talk about realizing $\mathcal{A}$ instead of talking about realizing $L_\omega(\mathcal{A})$.

Note that since a specification represents a set of behaviors, it can refer not only to the sequence of states that the system goes through but also to the sequence of scheduling decisions in the behavior. (For example, if B is described by a temporal formula, then, as in [AM94], we allow a special proposition μ that holds at point i of a behavior σ precisely when $sched(i, \sigma) = 1$.) Thus, the specification can enforce impartiality by requiring that the program be scheduled infinitely often. It follows that impartial realizability is a special case of standard realizability. In contrast, the specification cannot enforce justice or fairness. This is because these assumptions involve the enabling status of the program that is yet to be synthesized! This led Anuchitanukul and Manna [AM94] to the conclusion that the automata-theoretic approach is not applicable to realizability checking and synthesis under fairness assumptions. We show in the next section how this difficulty can be overcome.

Remark. Our definition of behaviors assumes that at every point either the program or the environment causes a transition. This does not allow for the possibility of *stuttering*, where the system does not change its state. It is argued in [Lam83] that specifications ought to be written at an abstract enough level to make them insensitive to stuttering.

To capture that in our model we can allow a behavior to be an ω-word $\langle s_0, a_0\rangle$, $\langle s_1, a_1\rangle, \ldots$ over $S \times \{0, 1, 2\}$. As before $a_i = 0$ means that the system causes the transition, and $a_i = 1$ means the the program causes the transition. Here $a_i = 2$ means that the system is stuttering, i.e, $a_i = 2$ iff $s_i = s_{i-1}$. We can now require that B be *closed under stuttering* [AL91], i.e., $\langle s_0, a_0\rangle, \ldots, \langle s_i, a_i\rangle, \langle s_{i+1}, a_{i+1}\rangle, \ldots$ is in B iff $\langle s_0, a_0\rangle, \ldots, \langle s_i, a_i\rangle, \langle s_i, 2\rangle, \langle s_{i+1}, a_{i+1}\rangle, \ldots$ is in B. Our approach to fair realizability

and synthesis can be easily extended to this framework. Details will be provided in the full paper.

3 Realizability Checking and Synthesis

3.1 Program Trees

Consider a program $f : (S^2)^*S \to S$. Let $w \in (S^2)^*$ be a sequence of pairs of states. We say that f is *enabled* in a state $s \in S$ with respect to w if $f(ws)$ is defined. We define $enable_f(w)$ to the set of states in which f is enabled with respect to w, i.e, $enable_f(w) = \{s \in S \mid f(ws) \downarrow\}$. We assume, without loss of generality, that if $s \notin enable_f(w)$, then $enable_f(u) = \emptyset$ whenever $u = w\langle s, s'\rangle \ldots \langle t, t'\rangle$. That is, if f is not enabled in s with respect to w, then we need not care about its value on any extension of ws.

The program f can be represented by a labeled tree τ_f. The (non-root) nodes of τ_f are labeled by elements of S^2, i.e., a node of τ_f is labeled by a pair of states. With each node x of τ_f we also associate a *valuation* $v(x) \in (S^2)^*$. We define τ_f and v as follows:

- The root ϵ is labeled by ϵ.[2]
- Consider a node x with $v(x) = w$. Then x has $|S'|$ children, where $S' = enable_f(w)$. For every $t \in S'$, the node x has a child y labeled with $\langle t, f(wt)\rangle$, and $v(y) = w\langle t, f(wt)\rangle$.

Thus, the labels of the nodes of τ_f encode the values of the program f, and the valuation of a node is simply a concatenation of the labels along the branch leading to that node.

Conversely, suppose that we are given given a labeled tree τ where the root ϵ is labeled by ϵ, and all other nodes are labeled by elements of S^2. Suppose also that the set of children of a node x has unique first components, i.e., if y_1 and y_2 are children of x and are labeled by $\langle t_1, t_1'\rangle$ and $\langle t_2, t_2'\rangle$, then $t_1 \neq t_2$. We call such a tree a *program tree*. We define the valuation $v(x)$ for all nodes x as follows: (a) $v(\epsilon) = \epsilon$, and (b) if $v(x) = w$ and y is a child of x labeled by is $\langle t, t'\rangle$, then $v(y) = w\langle t, t'\rangle$. We now can define a program f_τ as follows: if x is a node with a child y labeled by $\langle t, t'\rangle$, then $f_\tau(v(x)t) = t'$. The program f_τ is undefined everywhere else.

Consider a node x in a program tree τ with children labeled by $\langle t_1, t_1'\rangle, \ldots \langle t_k, t_k'\rangle$. It is easy to see that $enabled_{f_\tau}(v(x) = \{t_1, \ldots, t_k\}$. It is technically useful to have the enabling information be made explicit. To that end, we consider *augmented* program trees, where we add to each label a subset $S' \subseteq S$. Thus, the root has a label of the form (ϵ, S') and all other nodes have labels of the form $(\langle t, t'\rangle, S')$. If a node x is labeled by the set S' and its children are labeled by $(\langle t_1, t_1'\rangle, S_1), \ldots (\langle t_k, t_k'\rangle, S_k)$, then $S' = \{t_1, \ldots, t_k\}$. Thus, a node x in an augmented program tree τ is labeled by the set $enabled_{f_\tau}(v(x))$. From now on we refer to augmented program trees only.

Our approach is going to be as follows. We are given a Büchi automaton $\mathcal{A}$. Through a series of automata-theoretic transformations, we will transform $\mathcal{A}$ into a tree automaton $\mathcal{T}_\mathcal{A}$ that accept all augmented program trees that realize (under the appropriate fairness condition) $\mathcal{A}$. Checking realizability would then be reduced to checking nonemptiness of $\mathcal{T}_\mathcal{A}$.

[2] ϵ denotes the empty word.

3.2 Annotated Behaviors

Recall that a behavior over a state set S is an ω-word over the alphabet $S \times \{0,1\}$. An *annotated* behavior over S is an ω-word over the alphabet $S \times \{0,1\} \times \{0,1\}$ that obeys certain conditions. The intended meaning is that the annotated behavior $\langle s_0, a_0, b_0\rangle, \langle s_1, a_1, b_1\rangle, \ldots$ describes a behavior $\langle s_0, a_0\rangle, \langle s_1, a_1\rangle, \ldots$, where $b_i = 1$ precisely when the program is enabled at i. The conditions that the annotated behavior has to satisfy are as follows:

- If $a_i = 1$ then $b_i = 1$. Intuitively, the program can be scheduled only if it is enabled.
- If for some $i < j$ we have that $s_i = s_j$ and $a_k = 0$ for $i \leq k \leq j+1$, then $b_{i+1} = b_{j+1}$. Intuitively, as long as the program does not get scheduled, only the last state in its visible history can change, which implies that the enabling annotations have to be consistent.

While we cannot talk about justice or fairness of behaviors, we can talk about justice and fairness of annotated behaviors. Let $\alpha = \langle s_0, a_0, b_0\rangle, \langle s_1, a_1, b_1\rangle, \ldots$ be an annotated behavior. We say that α is just if $a_i = 1$ for infinitely many i's when $b_j = 1$ for all j greater than some j_0. We say that α is fair if $a_i = 1$ for infinitely many i's when $b_j = 1$ for infinitely many j.

Proposition 1. *Given a set S of m states, we can construct Büchi automata A_{just} (resp., A_{fair}) of size $2^{O(m)}$ that accept precisely all just (resp. fair) annotated behaviors.*

Proof sketch: A_{just} ensures justice by nondeterministically ensuring that $a_i = 1$ infinitely often or that $b_i = 0$ infinitely often. A_{fair} ensures fairness by nondeterministically ensuring that $a_i = 1$ infinitely often or that $b_i = 0$ only finitely often. The exponential blow up comes from having to verify the second clause in the definition of annotated behaviors, as the automaton has to remember the set of enabled states in an interval along which the program does not get scheduled. ∎

As we will see, Proposition 1 is the only place where the fairness conditions enters our algorithm. Thus, we can handle any fairness condition as long as the analog of Proposition 1 holds for this condition. (This is in contrast to the algorithms in [AM94] where the fairness condition is hardwired into the algorithm.) Thus, from now on we will not distinguish between justice and fairness and we will use the term fairness generically.

Let $\alpha = \langle s_0, a_0, b_0\rangle, \langle s_1, a_1, b_1\rangle, \ldots$ be an annotated behavior. The behavior of α, denoted $bhv(\alpha)$, is the behavior $\langle s_0, a_0\rangle, \langle s_1, a_1\rangle, \ldots$, i.e., we simply project out the enabling annotation. Recall that we are given a set B of allowable behaviors. We say that α is *B-bad* if $bhv(\alpha) \notin B$. If $B = L_\omega(\mathcal{A})$ for some Büchi automaton $\mathcal{A}$, then we say *$\mathcal{A}$-bad* instead of B-bad.

Proposition 2. *Given a Büchi automaton $\mathcal{A}$ of size n over a state set S of size m, we can construct a Büchi automaton of size $2^{O(mn \log n)}$ that accept precisely all $\mathcal{A}$-bad fair annotated behaviors.*

Proof sketch: We first construct an automaton $\mathcal{A}'$ that accepts an ω-word α over $S \times \{0,1\} \times \{0,1\}$ if $bhv(\alpha) \in L_\omega(\mathcal{A})$. Clearly, $\mathcal{A}'$ has the same size as $\mathcal{A}$. The desired

automaton is obtained by taking the intersection of the automaton given by Proposition 1 with the complement of the automaton $\mathcal{A}'$. The bound follows from Safra's $2^{O(n \log n)}$ bound for complementing Büchi automata [Saf88]. ∎

Let τ be a program tree. Consider now a (finite or infinite) maximal branch β of τ. Let the branch be $x_0, x_1, \ldots$, where $x_0 = \epsilon$ is labeled by (ϵ, S_0), x_i is the child of x_{i-1} and is labeled by $(\langle t_i, t_i' \rangle, S_i)$ (note that $t_i \in S_{i-1}$). Note that $v(x_0) = \epsilon$ and $v(x_{i+1}) = v(x_i)\langle t_{i+1}, t_{i+1}' \rangle$. This branch represents a sequence of "decisions" by the program. The branch can be finite if its last node x_k is labeled by $(\langle s_k, s_k' \rangle, \emptyset)$.

Let $\alpha = \langle s_0, a_0, b_0 \rangle, \langle s_1, a_1, b_1 \rangle, \ldots$ be an annotated behavior, and let $\sigma = bhv(\alpha)$. We say that σ is *consistent* with the branch β if the transitions in σ agree with the information represented by β. Let $0 < i_1 < i_2 < \ldots$ be the sequence of points along σ in which the program is scheduled, i.e., the sequence of points i_j such that $a_{i_j} = 1$. We require the following for $j = 0, 1, \ldots$:

- $s_{i_j - 1} = t_j$
- $s_{i_j} = t_j'$

These conditions require that the visible history traced along σ coincides with the trace of states along β. Note that if β is finite, then the program is scheduled in σ only finitely many times. We say that α is consistent with β if σ is consistent with β and in addition the enabling annotations are consistent with the enabling information in β, that is,

- if $k < i_1$, then $b_k = 1$ iff $s_{k-1} \in S_0$, and
- if $i_j < k < i_{j+1}$, then $b_k = 1$ iff $s_{k-1} \in S_j$.

Let a behavior $\sigma = \langle s_0, a_0 \rangle, \langle s_1, a_1 \rangle, \ldots$ be a run of a program f. Define $b_i = 1$ if f is enabled at i and $b_i = 0$ otherwise for $0 \leq i < \infty$. Then $\langle s_0, a_0, b_0 \rangle, \langle s_1, a_1, b_1 \rangle, \ldots$ is called the *annotation* by f of σ, denoted $annot_f(\sigma)$. Clearly, $bhv(annot_f(\sigma)) = \sigma$.

Proposition 3. *Let σ be a run of a program f. Then there is a unique branch β of τ_f that is consistent with σ. Furthermore, $annot_f(\sigma)$ is the unique annotated behavior consistent with β.*

Proof sketch: Let $\sigma = \langle s_0, a_0 \rangle, \langle s_1, a_1 \rangle, \ldots$. Let $0 < i_1 < i_2 < \ldots$ be the sequence of points along σ in which the program is scheduled, i.e., the sequence of points i_j such that $a_{i_j} = 1$. We define the branch $\beta = x_0, x_1, \ldots$ inductively such that $v(x_0) = \epsilon$ and $v(x_j) = \langle s_{i_1 - 1}, s_{i_1} \rangle, \ldots \langle s_{i_j - 1}, s_{i_j} \rangle$. Clearly, $x_0 = \epsilon$. Suppose that x_j has been defined. Then $s_{i_{j+1}-1} \in enable_f(v(x_j))$ and $s_{i_{j+1}} = f(v(x_j)s_{i_{j+1}-1})$. Thus, x_j has a child y labeled with $\langle s_{i_{j+1}-1}, s_{i_{j+1}} \rangle$ and $v(y) = v(x_j)\langle s_{i_{j+1}-1}, s_{i_{j+1}} \rangle$. We take x_{j+1} to be y.

It is easy to see that β is the unique branch satisfying for $j = 0, 1, \ldots$:

- $s_{i_j - 1} = t_j$
- $s_{i_j} = t_j'$

where $\langle t_j, t_j', S_j \rangle$ is the label of x_j. The definition of annotated behavior now guarantees that $annot_f(\sigma)$ is the unique annotated behavior consistent with β. ∎

From Proposition 3 we can derive a characterization of programs that fairly realize a set B of allowable behaviors. We say that a branch β is B-bad if it is consistent with a B-bad fair annotated behavior.

Theorem 4. *A program f fairly realizes a set B of behaviors iff τ_f has no B-bad branches.*

Proof sketch: *Only if*: Let β be a branch of τ_f that is consistent with a B-bad fair annotated behavior α. Then $bhv(\alpha)$ is also consistent with β. By Proposition 3, $\alpha = annot_f(bhv(\alpha))$. Thus, $bhv(\alpha)$ is a fair run of f that is not in B.

If: Suppose that f does not fairly realize B. Then there is a fair run σ of f that is not in B. By Proposition 3, $annot_f(\sigma)$ is a fair B-bad annotated behavior that is consistent with a branch β of τ_f. ∎

3.3 The Algorithm

To find if there exists a program tree that fairly realizes a Büchi automaton $\mathcal{A}$ we will construct a tree automaton that accepts all augmented program trees that fairly realize $\mathcal{A}$. By Theorem 4, an augmented program tree τ realizes f if no branch of τ is $\mathcal{A}$-bad. We already obtained in Proposition 2 an automaton that accepts precisely all $\mathcal{A}$-bad behaviors. We will transform it into an automaton that accepts $\mathcal{A}$-bad branches.

Consider again a branch β of τ. Let the branch be $x_0, x_1, \ldots$, where $x_0 = \epsilon$, x_i is labeled by $(\langle t_i, t_i' \rangle, S_i)$, and x_{i+1} is a child of x_i. In checking whether β is consistent with an annotated behavior, we are only concerned with the *description* of β, which is the sequence of labels along β, i.e., $(\epsilon, S_0), (\langle t_1, t_1' \rangle, S_1), \ldots$ is the description of β. Note that a branch description is a word in $S(S^2 \times 2^S)^* \cup S(S^2 \times 2^S)^\omega$. We say that a branch description is *$\mathcal{A}$-bad* if it is the description of a $\mathcal{A}$-bad branch.

Proposition 5. *Given a Büchi automaton $\mathcal{A}$ of size n over a state set S of size m, we can construct a Büchi [3] automaton $\mathcal{A}_{bad}$ of size $2^{O(mn \log n)}$ that accept precisely all $\mathcal{A}$-bad branch descriptions.*

Proof sketch: $\mathcal{A}_{bad}$ guesses an annotated behavior and checks that it is $\mathcal{A}$-bad and fair and that it is consistent with the branch description. $\mathcal{A}$-badness and fairness can be checked by simulating the automaton described in Proposition 2. ∎

The final step is to go from the automaton given in Proposition 5 to an automaton on program trees. As in [WTD91], we co-determinize [Saf88, EJ89] the automaton that accepts $\mathcal{A}$-bad branch descriptions. This yields a deterministic Rabin automaton that reject precisely all $\mathcal{A}$-bad branch descriptions. The size of this automaton is doubly exponential in the size of S and $\mathcal{A}$, but it has only an exponential number of pairs. As in [PR89a], this automaton is then transformed into a Rabin tree automaton $\mathcal{T}_\mathcal{A}$ that accepts all program trees that fairly realize $\mathcal{A}$. Thus, $\mathcal{A}$ is fairly realizable iff $\mathcal{T}_\mathcal{A}$ accepts some program tree. Thus, the realizability problem has been reduced to the nonemptiness problem for Rabin tree automata. The latter problem can be solved in time polynomial in the number of states of the automaton and exponential in the number of pairs [EJ88, PR89a]. We have obtained:

[3] There is a subtlety here, since Büchi automata accept only ω-words and branch descriptions can be finite. We take here the convention that the Büchi acceptance condition is used to define both acceptance of ω-words and of finite words.

Theorem 6. *The fair realizability problem is decidable in doubly exponential time.*

If the program is fairly realizable, then the nonemptiness algorithm not only says that the automaton $\mathcal{T}_{\mathcal{A}}$ is nonempty, it also yields a representation of a tree accepted by $\mathcal{T}_{\mathcal{A}}$, which is essentially a deterministic sub-automaton of $\mathcal{T}_{\mathcal{A}}$ [Eme85, VS85]. As observed in [PR89a], this sub-automaton can be viewed as a finite-state program fairly realizing f. Thus, the algorithm for fair realizability also solves the fair synthesis problem.

Remark.

1. It may seem as if our upper bound depends on the set B of allowable behaviors to be represented by an automaton $\mathcal{A}$. Suppose instead that we are given a linear temporal formula ϕ. Translating ϕ into an equivalent Büchi automaton $\mathcal{A}_\phi$ involves an exponential blow-up. Thus, it looks as if our algorithm in this case has a triply exponential running time!

   To address this issue we have to analyze carefully the sources for the doubly exponential complexity of our algorithm. The first exponential blow-up came when we used complementation to construct an automaton for $\mathcal{A}$-bad fair annotated behaviors (Proposition 2). The second exponential blow-up came when we co-determinized the automaton that accepted $\mathcal{A}$-bad branch descriptions. The second exponential blow-up is unavoidable and applies even when B is represented by a temporal formula. There is no need, however, to use complementation in the proof of Proposition 2 when we start with a linear temporal formula; we can simply use the automaton $\mathcal{A}_{\neg\phi}$ instead of complementing the automaton $\mathcal{A}_\phi$. Thus, Theorem 6 holds even when the specification consists of a linear temporal formula rather than a Büchi automaton.
2. The algorithm described above is doubly exponential in both the size n of the specification automaton $\mathcal{A}$ and the size m of the state set S. As commented in the first part of this remark, the second exponential blow-up results from the co-determinization construction. The first exponential blow up in m, however, results from the needs to add enabling information to the label (Proposition 2). This seems wasteful, since a program tree contains the enabling information even without augmenting the labels. Since, however, our construction of the tree automaton starts with a construction of a Büchi ω-automaton (Proposition 5), this blow-up seems unavoidable.

   Nevertheless, this blow-up is avoidable. Instead of defining consistency of a branch with an annotated behavior, we can define consistency of a program tree with an annotated behavior. We can then say that a program tree is $\mathcal{A}$-bad if it is consistent with an $\mathcal{A}$-bad fair annotated behavior. It then can be shown that given a Büchi automaton $\mathcal{A}$ of size n over a state set S of size m, we can construct a Büchi tree automaton with $m2^{O(n \log n)}$ states that accept precisely all $\mathcal{A}$-bad program trees. Notice that this involves an exponential blow-up in n, but not in m. Finally, by complementing this Büchi tree automaton [Kla92], we get a Rabin tree automaton whose size is doubly exponential in n, but only exponential in m. Thus, working directly with tree automata avoids one exponential blow-up in m. Details will be given in the full paper.

3.4 A Lower Bound

Following the doubly exponential bound for satisfiability of CTL* in [VS85], it is shown in [PR89a] that the realizability is 2EXPTIME-complete. That lower bound, however, depends crucially on using linear temporal logic as the specification language. If one starts directly with a Büchi automaton, then the running time of the algorithms in [PR89a, ALW89] is singly exponential rather than doubly exponential. In contrast, our algorithm has a doubly exponential running time even when we start with a Büchi automaton. We now show that this complexity is unavoidable.

Theorem 7. *The fair realizability problem with respect to Büchi automata is 2EXPTIME-complete.*

Proof sketch: As in [VS85], we encode the halting problem for alternating exponential-space Turing machines [CKS81]. The program tree encodes the computation tree of the alternating machine. The difficulty is that every branch represent a sequence of machine configuration of size 2^n. In [VS85, PR89a] the succinctness of temporal logic was used to describe such sequences by formulas of size $O(n)$. It seems impossible, however, to describe such sequence by automata of size $O(n)$.

The key to overcoming this difficulty is the second-order universal quantification hidden in Theorem 4. (See [SVW87] for another demonstration of the power of second-order quantification.) According to the theorem, β is a branch of a program tree that realizes $\mathcal{A}$ if *all* fair annotated behaviors that are consistent with $\mathcal{A}$ are $\mathcal{A}$-good (i.e., not $\mathcal{A}$-bad). Thus, it is possible to construct an automaton $\mathcal{A}$ of size $O(n)$ such that if all fair annotated behaviors that are consistent with β are $\mathcal{A}$-good, then β is a correct sequence of machine configurations of size 2^n. Details will be given in the full paper. ∎

4 Concluding Remarks

Contrary to the argument in [AM94], we have demonstrated that the automata-theoretic approach can handle fair realizability checking and synthesis. The advantage of this approach is that it is a natural extension of previous works [PR89a, ALW89, Dil89, PR89b, WTD91] and can deal with a variety of fairness conditions. It will be interesting to compare the precise complexity of our algorithm against the complexity of the algorithm in [AM94] (where no complexity analysis is offered).

Acknowledgement

I'd like to thank the members of the program committee for their helpful comments on a previous draft of this paper. I am also grateful to Martin Abadi for an enlightening discussion on the basic model of the paper and on the appropriate way to model stuttering.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[ALW89] 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.

[AM94] A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In *Computer-Aided Verification, Proc. 6th Int'l Workshop*, pages 156–169, Stanford, California, June 1994. Springer-Verlag, Lecture Notes in Computer Science 818.

[BL69] J.R. Büchi and L.HG. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.

[Büc62] 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.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CGL93] E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, Lecture Notes in Computer Science, pages 124–175. Springer-Verlag, 1993.

[Chu63] A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pages 23–35. institut Mittag-Leffler, 1963.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.

[Dil89] D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[EJ89] E.A. Emerson and C. Jutla. On simultaneously determinizing and complementing ω-automata. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 333–342, 1989.

[Eme85] E.A. Emerson. Automata, tableaux, and temporal logics. In *Proc. Workshop on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 79–87. Springer-Verlag, 1985.

[Fra86] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[Kla92] N. Klarlund. Progress measures, immediate determinacy, and a subset construction for tree automata. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 382–393, Santa Cruz Juan, 1992.

[Lam83] L. Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*. IFIP, North-Holland, 1983.

[Liu89] M.T. Liu. Protocol engineering. *Advances in Computing*, 29:79–195, 1989.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPS81] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness – the ethic of concurrent termination. In *Proc. 8th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, July 1981.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[MW84] 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.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, Austin, Januery 1989.

[PR89b] 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.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[Rud87] H. Rudin. Network protocols and tools to help produce them. *Annual Review of Computer Science*, 2:291–316, 1987.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[SV89] S. Safra and M.Y. Vardi. On ω-automata and temporal logic. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 127–137, Seattle, May 1989.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Var88] M.Y. Vardi. A temporal fixpoint calculus. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 250–259, San Diego, January 1988.

[VS85] M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

[WTD91] 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. 1991.