Realizability and Synthesis of Reactive Modules

Anuchit Anuchitanukul Zohar Manna

Computer Science Department
Stanford University
Stanford, CA 94305
anuchit@cs.stanford.edu

April 4, 1994

Abstract

We present two algorithms: a realizability-checking algorithm and a synthesis algorithm. Given a specification of reactive asynchronous modules expressed in propositional ETL (Extended Temporal Logic), the realizability-checking algorithm decides whether the specification has an actual implementation, under the assumptions of a random environment and fair execution. It also creates a structure which can then be transformed by the synthesis algorithm into a program, represented as a labeled finite automaton. Unlike previous approaches, the realizability-checking algorithm can handle fairness assumptions. The realizability-checking algorithm is incremental and it directly manipulates formulas in linear temporal logic without having to transform into a branching-time logic or other representations.

1 Introduction

The problem of automatic program synthesis has been previously studied in many different frameworks. For functional programs, the specification is a first-order formula expressing the desired relationship between inputs and outputs, where the synthesized program can be extracted from a constructive proof of the formula [MW80].

Later, [EC82] and [MW84] extended the approach to reactive programs. The synthesized program is extracted from a proof of the satisfiability of the specification. However, the reactive programs considered in the approach do not have any interaction with the environment, that is, they are *closed* systems.

The effort to synthesize reactive modules, i.e., *open* systems, was first reported in [PR89a]. In that paper, the synthesis of reactive *synchronous* modules from a specification in linear-time temporal logic is linked to the problem of

checking the validity of a branching-time temporal formula obtained by transforming the original specification.

The restriction to synchronous systems (or the game of perfect information) was removed in [PR89b] where the problem of synthesizing asynchronous systems is considered. In that work, a linear-time temporal specification is transformed into a formula in branching-time temporal logic by introducing read and write variables, and by adding constraints on the variables.

Several notions of realizability were introduced and studied in [ALW89]. For the finite case, the approach taken is similar to the automata approach of [PR89a,b]. Because of the choice of the specification, the method can check realizability in a more general sense, that is, when the behavior of the environment is restricted. However, [ALW89] only considered synchronous systems.

In [WD91], the approach of [PR89b] was extended to handle shared variables and the restriction on read and write sequences was relaxed. The paper also generalizes [ALW89] to include the asynchronous and real-time cases. The main technique used is still by transformations from one representation (automata) to another.

The assumption shared by all of the works mentioned previously is that the problem of realizability can be solved simply by transforming the specification (under some representation) into an automata representation (tree automaton) and then checking for non-emptiness. We argue that this assumption is not valid when we want to solve a more general realizability problem. There are properties which cannot be encoded into a specification, such as the assumption of fairness. We would like to be able to determine whether a specification is realizable, assuming that the execution of the system and the environment is fair. Since the synthesized program does not exist 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. Therefore, any realizability-checking and synthesis algorithms must handle this explicitly. Although [ALW89] did define a notion of realizability under fairness (and a theorem relating it to realizability without fairness), no solution was provided.

In this paper, we present two algorithms, one for realizability checking and the other for synthesis. The specification language we study is the Extended Temporal Logic (ETL) described in [Wo83]. We also introduce a scheduling variable μ following the approach in [BKP84]. Although useful for expressing specifications and for extending the algorithm to handle sequential composition, μ is not essential to the algorithms. The realizability-checking algorithm is based on the tableau decision procedure described in [Wo85]. Given a specification, the first algorithm checks for *strong* realizability under fairness and random environment assumptions and generates a structure called a *realizability graph.* The synthesis algorithm takes the generated realizability graph and produces a program which satisfies the specification. Unlike other approaches, the synthesis

algorithm will generate a more general class of programs which may have some disabled transitions. The transitions in the generated program may be labeled weakly or strongly fair as necessary. Since the realizability-checking algorithm is a tableau-based algorithm, it manipulates only formulas of linear temporal logic, which are subformulas of the original specification.

2 Definitions

A [infinite] *behavior* σ over a state space Σ is a pair $\langle \sigma_v, \sigma_s \rangle$ of two equal-length [infinite] sequences: a sequence of *states* $\sigma_v = s_0 s_1 s_2 \ldots$ where $s_i \in \Sigma$ and a *scheduling* sequence $\sigma_s = a_0 a_1 a_2 \ldots$ where $a_i \in \{0, 1\}$. We denote the set of all infinite behaviors from Σ by $Bhv(\Sigma)$, or Bhv if Σ is clear from the context, and the set of all finite behaviors by $Bhv_{fin}(\Sigma)$.

We can represent a behavior $\langle s_0 s_1 s_2 \ldots, a_0 a_1 a_2 \ldots \rangle$ pictorially as

$$\xrightarrow{a_0} s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$$

The intended meaning is that the move from s_i to s_{i+1} is caused by the environment if $a_{i+1} = 0$, and by the system if $a_{i+1} = 1$. Since we always assume that the environment chooses the initial state, we require that the scheduling sequence always begins with 0, i.e., $a_0 = 0$.

Given a behavior $\sigma = \langle s_0 s_1 s_2 \ldots, a_0 a_1 a_2 \ldots \rangle$, we write $State(i, \sigma)$ to denote s_i and $Sched(i, \sigma)$ to denote a_i. If $\sigma = \langle \sigma_v, \sigma_s \rangle$, then $\sigma|_i$ denotes a behavior $\langle \sigma_v|_i, \sigma_s|_i \rangle$ where $\sigma_v|_i$ $[\sigma_s|_i]$ is the prefix of σ_v $[\sigma_s]$ of length i.

Let $\Pi^0 : Bhv_{fin} \mapsto \Sigma^*$ be a function that maps a finite behavior σ to a subsequence of states which are caused by the system, namely, all $State(i, \sigma)$ where $Sched(i, \sigma) = 1$. Let $\Pi^1 : Bhv_{fin} \mapsto \Sigma^*$ be a function that maps a finite behavior σ to a subsequence of states which are observed by the system (precede a system state), that is, all $State(i, \sigma)$ where $Sched(i + 1, \sigma)$ is defined and $Sched(i + 1, \sigma) = 1$, or $i + 1$ is the length of σ.

A *computer* $f : \Sigma^* \times \Sigma^* \mapsto \Sigma$ is a partial function which takes a history of all the states the system caused and all the states the system observed and selects a state as the next move of the system. A *run* of a computer f is an infinite behavior such that for all i, if $Sched(i, \sigma) = 1$ then $f(\Pi^0(\sigma|_i), \Pi^1(\sigma|_i))$ is defined and equal to $State(i, \sigma)$. Therefore, a behavior is a run of a computer if every system move is the result of f computed with the information regarding the system's own moves and all the moves the system has observed in the past.

A run σ of f is *weakly fair* iff for all j, if $f(\Pi^0(\sigma|_i), \Pi^1(\sigma|_i))$ is defined for all $i \geq j$, then $Sched(k, \sigma) = 1$ for some $k \geq j$, i.e., if f is continuously enabled beyond a certain point, it has to be taken eventually. Similarly, a run σ is *strongly fair* iff for all j, if for all $j' \geq j$ there exists $i \geq j'$ such that $f(\Pi^0(\sigma|_i), \Pi^1(\sigma|_i))$ is defined, then $Sched(k, \sigma) = 1$ for some $k \geq j$.

Let $Run_{sf}(f)$ be all possible strongly fair runs and $Run_{wf}(f)$ be all possible weakly fair runs of the computer f. A set B of behaviors is *realizable* (under

fairness and random environment assumptions) iff there exists a computer f such that $Run_{sf}(f) \subseteq B$. If only weak fairness is assumed, B is realizable iff there exists a computer f such that $Run_{wf}(f) \subseteq B$.

3 Preliminaries

3.1 Specification Language

The specification language studied here is Extended Temporal Logic (ETL) augmented with a special predicate μ. The use of ETL and μ is not necessary for the realizability-checking and synthesis algorithms. Clearly, the algorithms can handle any subset of the language, including ordinary propositional temporal logic specifications without μ. Adding μ to the language is necessary, however, to express some common forms of specifications such as mutual exclusion. Without μ, we would have to separate the environment assumption and the system property. With μ, the whole specification can be expressed in a single formula.

In ETL, there are infinitely many temporal operators. Each corresponds to a non-terminal symbol of a right-linear grammar. A right-linear grammar G is a tuple (V_N, V_T, P) such that

- $V_N = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$ is a finite set of non-terminal symbols.
- $V_T = \{t_1, \ldots, t_n\}$ is a finite set of terminal symbols.
- P is a finite set of production rules of the forms $\mathcal{G}_i \rightarrow t_j$ or $\mathcal{G}_i \rightarrow t_j \mathcal{G}_k$ where $\mathcal{G}_i, \mathcal{G}_k \in V_N$ and $t_j \in V_T$.

For each non-terminal symbol $\mathcal{G}_i$, the corresponding temporal operator $\mathcal{G}_i(\phi_1, \ldots, \phi_n)$ has exactly n arguments (n is the number of terminal symbols).

Given a set $\mathcal{P}$ of propositions and a truth-value assignment function $\pi : \Sigma \mapsto 2^{\mathcal{P}}$, the semantics of a formula on an *infinite* behavior σ is defined as follows:

- $\sigma \models \phi$ iff $\langle \sigma, 0 \rangle \models \phi$.
- $\langle \sigma, i \rangle \models p$ iff $p \in \pi(State(i, \sigma))$, for any proposition $p \in \mathcal{P}$.
- $\langle \sigma, i \rangle \models \mu$ iff $Sched(i, \sigma) = 1$.
- $\langle \sigma, i \rangle \models \bigcirc \phi$ iff $\langle \sigma, i+1 \rangle \models \phi$.
- $\langle \sigma, i \rangle \models \mathcal{G}(\phi_1, \ldots, \phi_n)$ iff there is a word (finite or infinite) $w = t_{n_0} t_{n_1} t_{n_2} \ldots$ (each $t_{n_j} \in V_T$), generated by $\mathcal{G}$, and for all $j \geq 0$, $\langle \sigma, i+j \rangle \models \phi_{n_j}$.
- Other cases ($\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, and $\neg\phi$) are standard.

Clearly, any formula ϕ defines a set of infinite behaviors B which satisfy the formula, i.e., $\sigma \models \phi$ iff $\sigma \in B$. Therefore, we define a specification to be realizable if the corresponding set of behaviors is realizable.

3.2 Elementary Formulas

A formula is called *elementary* if it is either

- an atomic formula, i.e., an atomic proposition (including μ) or its negation, or
- a next formula, i.e., a formula that has $\bigcirc$ as its main connective.

3.3 Decomposition Rules

The following decomposition rules are used in the tableau graph construction algorithm to decompose non-elementary formulas. The meaning of a decomposition rule is that in order to satisfy the formula on the left hand side, one of the sets on the right hand side must be satisfied.

- $(\phi_1 \vee \phi_2) \Longrightarrow \{\{\phi_1\}, \{\phi_2\}\}$
- $(\phi_1 \wedge \phi_2) \Longrightarrow \{\{\phi_1, \phi_2\}\}$
- $\neg(\phi_1 \vee \phi_2) \Longrightarrow \{\{\neg\phi_1, \neg\phi_2\}\}$
- $\neg(\phi_1 \wedge \phi_2) \Longrightarrow \{\{\neg\phi_1\}, \{\neg\phi_2\}\}$
- $(\neg\neg\phi) \Longrightarrow \{\{\phi\}\}$
- $(\neg \bigcirc \phi) \Longrightarrow \{\{\bigcirc\neg\phi\}\}$
- For an ETL grammar operator $\mathcal{G}(\phi_1, \ldots, \phi_n)$ with grammar productions of the form: $\mathcal{G} \rightarrow t_{a_i}\ \mathcal{G}_{b_i}$ where $1 \leq i \leq l$ is the index of the production rules of $\mathcal{G}$, $t_{a_i} \in V_T$ and $\mathcal{G}_{b_i} \in V_N$ (which may or may not be present), we have the following decomposition rules:

$$\mathcal{G}(\phi_1, \ldots, \phi_n) \Longrightarrow \bigcup_{1 \leq i \leq l} \{\{\phi_{a_i}, \bigcirc\mathcal{G}_{b_i}(\phi_1, \ldots, \phi_n)\}\}$$

$$\neg\mathcal{G}(\phi_1, \ldots, \phi_n) \Longrightarrow \{ \bigcup_{1 \leq i \leq l} \{\neg\phi_{a_i} \vee \bigcirc\neg\mathcal{G}_{b_i}(\phi_1, \ldots, \phi_n)\}\}$$

3.4 Tableau Graph

Before we proceed to describe the realizability-checking algorithm, we will briefly explain a tableau graph construction similar to that in [Wo85]. A tableau graph is a directed graph in which each node n is labeled with a set of formulas, denoted by $\Phi(n)$.

- A node n in a tableau graph is called a *state* node iff $\Phi(n)$ contains only elementary formulas.

- A node n is *environment-compatible* iff $\mu \notin \Phi(n)$.
- Similarly, a node n is *system-compatible* iff $\neg\mu \notin \Phi(n)$.

Given a formula $\hat{\psi}$ to be checked for satisfiability, the tableau graph for $\hat{\psi}$ is created as follows:

First,

1. create a node *(root)* and label it with $\{\hat{\psi}\}$.

Repeatedly apply steps 2 and 3.

2. If a node n, with no successor, contains a non-elementary formula ϕ in its label $\Phi(n)$, and if the decomposition rule for ϕ is $\phi \Longrightarrow \{S_1, \ldots, S_t\}$, then for each set of formulas S_i, create a successor of n and label it with $(\Phi(n) - \{\phi\}) \cup S_i$. However, if there is a node with the same label already, then just connect n to the existing node.
3. For a state node n with label $\Phi(n)$, create (if no duplication occurs) a successor of n and label it with $\{\phi \mid \bigcirc\phi \in \Phi(n)\}$.

Finally,

4. Remove all inconsistent nodes (the nodes containing a proposition p and its negation $\neg p$).

A loop in a tableau graph is called a *self-supporting* loop if for any state node n in the loop, there is a finite path in the loop starting from n such that all formulas of the form $\bigcirc\mathcal{G}(\ldots)$ in $\Phi(n)$ are *fulfilled* on the path. A formula $\bigcirc\neg\mathcal{G}(\ldots)$ with the decomposition rule,

$$\neg\mathcal{G}(\phi_1, \ldots, \phi_n) \Longrightarrow \{ \bigcup_{1 \le i \le l} \{\neg\phi_{a_i} \vee \bigcirc\neg\mathcal{G}_{b_i}(\phi_1, \ldots, \phi_n)\}\}$$

is *fulfilled* at a state n if the next state n' on the path contains a term from each of the disjunctions of the decomposition rule and if the term is $\bigcirc\neg\mathcal{G}_{b_i}(\ldots)$ then it is also fulfilled at n' (i.e. at the next state down the path).

3.5 Maximally Consistent Subsets

Given a set of state nodes N, a subset $N_{mcs} \subseteq N$ is *maximally consistent* if both of the following conditions are satisfied:

- **Consistent:** It is not the case that for some proposition p other than μ and for some nodes $n_1, n_2 \in N_{mcs}$, both $p \in \Phi(n_1)$ and $\neg p \in \Phi(n_2)$. In other words, the union of all the observable atomic formulas (which are all atomic formulas except μ and $\neg\mu$) in the labels of the nodes in N_{mcs} is consistent.
- **Maximal:** There is no other subset $N' \subseteq N$ such that N' satisfies the above condition (consistent) and $N_{mcs} \subset N'$.

3.6 Maximally Negation-Consistent Subsets

For a set of state nodes N, a subset $N_{mncs} \subseteq N$ is *maximally negation-consistent* if both of the following conditions are satisfied:

- **Negation-consistent:** There exists a function f which maps each node $n \in N_{mncs}$ to an atomic formula $f(n) \in \Phi(n)$ which is not μ or $\neg\mu$, and the set $P = \{\neg f(n) \mid n \in N_{mncs}\}$ is consistent. The set P is called the *falsifying* set for N_{mncs}.
- **Maximal:** There is no other subset $N' \subseteq N$ such that N' satisfies the above condition (negation-consistent) and $N_{mncs} \subset N'$.

3.7 Realizability Graph

The structure created by the realizability-checking algorithm is called a *realizability graph*. A realizability graph is a directed bipartite graph $(V_s, V_n, E_{sn}, E_{ns})$ where

- V_s is a set of nodes called *R-state* nodes and labeled by a *node-label* which is a set of tableau graph nodes and a *write-label* which is a set of atomic formulas.
- V_n is a set of nodes called *R-non-state* nodes and labeled by a node-label.
- V_{sn} is a set of links from R-state nodes to R-non-state nodes.
- V_{ns} is a set of links from R-non-state nodes to R-state nodes.

3.8 Embedding

An increasing sequence $d_0 \ldots d_l$ of integers is an *embedding* of a path [loop] $n_0 \ldots n_k$ in a tableau graph into a path [loop] $v_0 \ldots v_l$ in a realizability graph if both of the following conditions hold:

- for all $0 \leq i \leq l$, n_{d_i} is in the node-label of v_i.
- for all n_j, if $j \neq d_i$ for all $0 \leq i \leq l$, then n_j is environment-compatible.

It is straightforward to extend the definition to allow the embedding of an infinite path in a tableau graph into a (finite or infinite) path in a realizability graph.

4 Realizability-Checking Algorithm

The key idea in the algorithm is that the realizability graph represents a game between the system and the environment in which the environment can make any finite number of moves after a system's move. This is represented by the alternate levels of R-state and R-non-state nodes. Given a formula ψ to be tested for realizability, the algorithm construct a tableau graph for the negation of ψ. To "win the game", the environment must try to force the execution to stay on a path in the tableau graph which falsifies ψ; whereas the system must try to push the execution out of such path.

We start constructing the realizability graph from an R-state node which contains the root node n_{root} of the tableau graph. Since the environment can make any number of moves, it may try to follow any path in the tableau graph from n_{root}. Without the complete knowledge of all the moves the environment makes, the system cannot determine which path the environment has taken. It can only use the information from the state it observes when it is scheduled to run, to determine a *set* of all state nodes accessible from n_{root} the path might have led into. In the worst case, such s set will be a maximally consistent subset of all accessible state nodes. Therefore, we construct an R-non-state successor of the R-state node, for each maximally consistent subset. For its own move, the system must try to push the execution out of any path which the environment might follow (and win) afterward. The best move that the system can possibly make is to falsify as many successor nodes of the nodes in the node-label of the R-non-state node and in essence, to limit the possible paths left for the environment to follow. This is the reason why we compute the maximally negation-consistent subsets and the falsifying set of atomic formulas. The remaining nodes which are not falsified can be computed by subtracting the maximally negation-consistent subsets from the set of all successors of the nodes in the R-non-state node. For each best move possible, we create an R-state successor of the R-non-state successor, put the remaining nodes in its node-label and continue expanding the realizability graph from the new R-state node.

In the algorithm, at each R-state node v_s, we compute a set *Disabled* by collecting all state nodes in the labels of every deleted R-non-state successor of v_s. When an R-non-state successor v_{ns} is deleted, it means the system will not be able to satisfy the specification by making a transition from v_s through v_{ns}. Therefore, we should consider such a transition "disabled". As a result, we put every state node in the deleted R-non-state node into the set *Disabled* because the environment can choose to move into some states in which the transition through the deleted R-non-state node is disabled.

Finally, we also have to check at each R-state node that the environment cannot win by remaining in a loop containing disabled state nodes.

4.1 Main procedure

1. First, create a tableau graph *Glb* for the formula $\neg\psi$ where ψ is the formula to be tested for realizability.
2. Create an R-state node *(root)* and label it with the set $\{n_{root}\}$ where n_{root} is the root node of *Glb*.
3. Call the subroutine *Expand*, passing the root node as its parameter, to expand the realizability graph in a depth-first fashion.
4. Finally, check if the root node of the final realizability graph is deleted. If it is not deleted, then the formula ψ is realizable. Otherwise, it is unrealizable.

4.2 Subroutine *Expand* (Realizability Graph Construction)

Given an R-state node v_s with a node-label $L(v_s)$, expand the realizability graph as follows:

1. If $L(v_s)$ is empty, then do nothing and return.
2. Let N_{acc} be the set of all state nodes n_k accessible from some $n_0 \in L(v_s)$ through some path $n_0 \ldots n_k$ in *Glb* such that for all $0 < i \leq k$, n_i is an environment-compatible node.
3. If there is a node in N_{acc} which contains only atomic formulas, then delete v_s and return from *Expand*.
4. Set *Disabled* to be the empty set.
5. For each maximally consistent subset N_{mcs} of N_{acc},
   (a) Create an R-non-state node v_{ns} as a successor of v_s and label v_{ns} by N_{mcs}.
   (b) Let N' be the set of all system-compatible state nodes n_k accessible from some $n_0 \in N_{mcs}$ through a path $n_0 \ldots n_k$ where for all $0 < i < k$, n_i is not a state node.
   (c) For each maximally negation-consistent subset N_{mncs} of N' and the corresponding falsifying set P of atomic formulas,
      i. Create an R-state node as a successor of v_{ns} and label it by a node-label $N' - N_{mncs}$ and a write-label P. Then, recursively call *Expand* on the new node.
      ii. However, if there is an R-state node v'_s with the same node-label and write-label, and if, in addition, the node v'_s itself is marked, *"satisfied"*, then connect v_{ns} to v'_s. If v'_s is not marked *"satisfied"*,

then check whether there exists a self-supporting loop in *Glb* that can be embedded into the loop $v_s \ldots v'_s$. If there is no such loop in *Glb*, connect v_{ns} to v'_s.

(d) If there is no successor to v_{ns}, delete v_{ns} and add all the nodes in N_{mcs} (the node-label of v_{ns}) to the set variable *Disabled*.

6. Check if there is a self-supporting loop in *Glb* which is accessible from a node in $L(v_s)$ through a path consisting only of environment-compatible nodes, and all state nodes in the loop are environment-compatible and in the set *Disabled*. If there is, then delete v_s. Otherwise, mark v_s *"satisfied"* and return.

   If only weak fairness is allowed in the definition of realizability that we are checking, we only have to look for a self-supporting loop with *at least one* state node in *Disabled*.

5 Synthesis Algorithm

To simplify the presentation, we choose to represent the synthesized module by a labeled finite automaton. A *module automaton* is a tuple $\langle S, \delta, s_0, l \rangle$ where S is a finite set of states, $\delta : S \times 2^{\mathcal{P}} \mapsto 2^S$ the transition relation, $s_0 \in S$ the initial state and $l : S \mapsto 2^{\mathcal{P}}$ the labeling function. A run r is a sequence (finite or infinite) of states from S starting with s_0. We will write $r[k]$ to denote the k-th state in the sequence r and $|r|$ to denote the length of r. A behavior σ with a truth-value assignment $\pi : \Sigma \mapsto 2^{\mathcal{P}}$ is accepted by the automaton iff there is a run r such that for every k, if $r[k+1]$ is defined then $\pi((\Pi^0(\sigma))[k]) = l(r[k+1])$ and $r[k+1] \in \delta(r[k], \pi((\Pi^1(\sigma))[k]))$. With weak fairness, a behavior σ is accepted iff in addition to the previous conditions, if r is finite then for some j, there exist infinitely many $i \geq j$, such that $\delta(r[|r|], \pi(State(i, \sigma))) = \emptyset$. A similar acceptance condition can be defined for the case of strong fairness.

Given a realizability graph, we will synthesize a module automaton which implements the specification. First, for each R-state node v, create a state $s_v \in S$ for the automaton. The initial state s_0 corresponds to the root node of the realizability graph. For the labeling function l, let $l(s_v)$ be the write-label of v.

For each s_v and each $x \in 2^{\mathcal{P}}$, recall the set N_{acc} of all state nodes accessible from the nodes in the node-label $L(v)$ of the R-state node v. Find the largest subset $N \subseteq L(v)$ such that for every state node $n \in N$, all the propositions $p \in \Phi(n)$ are in x and there is no $p \in x$ such that $\neg p \in \Phi(n)$. An R-non-state successor v_{ns} of v is said to *cover* x iff $N \subseteq L(v_{ns})$ where $L(v_{ns})$ is the node-label of v_{ns}. Let V be the set of all R-state successors of the R-non-state successor v_{ns} of v which covers x. Then $\delta(s_v, x) = \{s_{v_s} \in S \mid v_s \in V\}$.

6 Correctness and Completeness

Proposition 6.1 *(Correctness) If A is the module automaton synthesized after checking the realizability of the specification formula ψ under strong [weak] fairness, then for all behaviors σ accepted by A under strong [weak] fairness, $\sigma \models \psi$.*

Proof Outline: Suppose there were a behavior σ accepted by A under strong fairness but $\sigma \not\models \psi$. We will prove that this leads to a contradiction. First, from $\sigma \not\models \psi$, then $\sigma \models \neg\psi$ and we can show that σ can be embedded into a path in the tableau graph *Glb*. The path starts from the root node of *Glb* and may be either finite or infinite. A behavior σ can be embedded into a path $n_0 n_1 n_2 \ldots$ in the tableau graph iff for all state nodes n_i, if n_i is the j-th state node in the path, then for all $\phi \in \Phi(n_i)$, $\langle \sigma, j \rangle \models \phi$. Next, we can show that the path $n_0 n_1 n_2 \ldots$ can be embedded into a path $v_0 v_1 v_2 \ldots$ in the realizability graph, using the assumption that σ is accepted by A. We use the fact that we compute the maximally negation-consistent subset in the realizability-checking algorithm to show (by induction) the existence of the part of the embedding from an R-non-state node to an R-state node and the fact that we compute the largest subset $N \subseteq L(v)$ for each R-state node v in the synthesis algorithm for the part of the embedding from an R-state node to an R-non-state node. If the path $n_0 n_1 \ldots$ in *Glb* is finite, then it must be the case that the last state node n_t in the path must contain only atomic formulas, because $\sigma \models \neg\psi$. It implies that n_t is accessible (in N_{acc}) from some node in the label of the last R-state node v_l of the path $v_0 v_1 \ldots$. If that is the case, v_l would have been deleted in step 3 of the realizability-checking algorithm, a contradiction.

If $n_0 n_1 \ldots$ is infinite but $v_0 v_1 \ldots v_l$ is finite, then we can also derive a contradiction by showing that for the case of strong [weak] fairness, there must be a self-supporting loop within $n_0 n_1 \ldots$ such that all [some] state nodes in the loop are environment-compatible and in the set *Disabled*. The essential step is to use the fact that σ is accepted under strong [weak] fairness and to show from the properties of maximally consistent subsets that all state nodes n_i in the loop must be in the set *Disabled* if $\delta(s_{v_l}, \pi(n_i)) = \emptyset$. However, if such a self-supporting loop exists, then v_l would have been deleted in step 6.

Similarly, we can also derive a contradiction in the case when both $n_0 n_1 \ldots$ and $v_0 v_1 \ldots$ are infinite, by showing that the loops in $v_0 v_1 \ldots$ would have been eliminated in step 5.(c).ii.

Proposition 6.2 *(Termination) The realizability-checking algorithm always terminates.*

Proof Outline: There are only finitely many possible R-state and R-non-state nodes. Therefore, it is not possible to keep expanding the realizability graph forever. It is also clear that there are only finitely many maximally consistent and maximally negation-consistent subsets at any time in the algorithm.

Proposition 6.3 *(Completeness) The formula ψ is realizable iff the root node of the realizability graph is not deleted.*

Proof Outline: One direction of the proof, showing that if the root node of the realizability graph is not deleted then ψ is realizable, is straightforward from the proposition 1 (correctness).

In the other direction, we assume that ψ is realizable and show that the root node of the realizability graph is not deleted. Since ψ is realizable, there exists a function f which realizes it.

We have to define an embedding of a behavior into the realizability graph. A behavior σ can be embedded into a finite or infinite path $v_0 v_1 \ldots$ starting from the root in the realizability graph iff there exists an increasing sequence of integers $d_0 d_1 \ldots$ such that all of the following conditions are true:

- for all $i \geq 0$ and $n \in L(v_{2i})$, there is a formula $\phi \in \Phi(n)$ such that $\langle \sigma, d_i \rangle \models \neg\phi$,
- for all $i > 0$, $\langle \sigma, d_i \rangle \models \mu$,
- for all $i > 0$ and $n \in L(v_{2i+1})$, there is a formula $\phi \in \Phi(n)$ such that $\langle \sigma, d_i - 1 \rangle \models \neg\phi$.

An R-state node is called *reachable* iff there is a behavior of f which can be embedded into some path passing through the node. It is easy to see that the root node must be reachable. We want to show that some of the reachable nodes including the root are not deleted.

First, we can show that a reachable R-state node must not be deleted in step 3; otherwise, we can easily construct a behavior of f which falsifies ψ.

Next, we can show that for every reachable R-state node v and every self-supporting environment-compatible loops accessible from a node in the node-label $L(v)$ of v, there exists a node n in the loop such that for every R-non-state successor v_{ns} of v which contains n in the node-label, there is an R-state successor v' of v_{ns} which is also reachable. Again, we can prove this by showing that if such n does not exist, we can construct a fair behavior of f which falsifies ψ. We also use the fact that the label of v_{ns} is a maximally consistent set to show the existence of the embedding into a path through v_{ns}.

Finally, we can show that for any loop of reachable R-state nodes, if there is a self-supporting loop in *Glb* which can be embedded into it as in step 5.(c).ii, there is a reachable R-state node in the loop which is not deleted as the result of breaking the loop of R-state nodes in step 5.(c).ii. We prove this by considering a behavior of f which can be embedded into a path passing through this loop of reachable R-state nodes. Clearly, the path cannot remain within the loop forever or the behavior will not satisfy ψ.

Acknowledgements

We thank Allen Emerson for his very useful comments and Howard Wong-Toi, Eddie Chang, Nikolaj Bjorner and Henny Sipma for fruitful discussions and for carefully reading the drafts of this paper.

References

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. *Proc. 16th Int. Colloq. Aut. Lang. and Prog.* Lec. Notes in Comp. Sci. 372, Springer-Verlag, Berlin, 1–17, 1989.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. *Proc. 16th ACM Symp. Theory of Comp.*, 51–63, 1984.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comp. Prog.*, 2(3):241–266, 1982.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Prog. of Lang. and Sys.*, 2(1):90–121, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal-logic specifications. *ACM Trans. on Prog. Lang. and Sys.*, 6(1):68–93, 1984.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. *Proc. 16th Int. Colloq. Aut. Lang. Prog.* Lec. Notes in Comp. Sci. 372, Springer-Verlag, Berlin, 652–671, 1989.

[WD91] H. Wong-Toi and D.L. Dill. Synthesizing processes and schedulers from temporal specifications, *Computer-Aided Verification (Proc. CAV90 Workshop)*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 3 (American Mathematical Society, 1991).

[Wo83] P. Wolper. Temporal logic can be more expressive. *Info. and Cont.*, 56:72–99, 1983.

[Wo85] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Anal.*, 28:119–136, 1985.