Contents

Realizable and Unrealizable Specifications of Reactive Systems

Martín Abadi* Leslie Lamport* Pierre Wolper†

1 Introduction

A specification is useless if it cannot be realized by any concrete implementation. There are obvious reasons why it might be unrealizable: it might require the computation of a nonrecursive function, or it might be logically inconsistent. A more subtle danger is specifying the behavior of part of the universe outside the implementor's control. This source of unrealizability is most likely to infect specifications of concurrent systems or reactive systems [HP85]. It is this source of unrealizability that concerns us.

Formally, a specification is a set of sequences of states, which represents the set of allowed behaviors of a system. We do not distinguish between specifications and programs; a Pascal program and a temporal logic specification are both specifications, although one is at a lower level than the other. (Some may wonder which is the lower-level one.) A specification $\mathbf{S_1}$ is said to implement a specification $\mathbf{S_2}$ iff (if and only if) it allows fewer behaviors than $\mathbf{S_2}$.

We define a class of *realizable* specifications. It includes all specifications that can be implemented by physically possible systems, but also some that have no real implementations for reasons that do not concern us—for example, because they presume the computation of nonrecursive functions.

In general, deciding whether a specification is realizable may be difficult. For a specification that includes only safety properties, determining realizability is easy in principle. A specification is unrealizable iff it constrains the environment. A safety property asserts that something bad does not happen. It constrains the environment iff there is some sequence of states in which the environment makes the bad thing happen. However, for liveness requirements, which assert that something good must eventually happen, it is not easy to determine if they constrain the environment.

To study realizability, we consider a specification to be a type of infinite game of perfect information [Mar75], where the system plays against the environment and wins if it produces a correct behavior. Under hypotheses justified by previous work [AL88], we prove that specifications are determined games, meaning that one of the players

*Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA.

†Institut Montefiore, B28, Université de Liège, B-4000 Liège Sart-Tilman, Belgium. The work of this author was done in part while visiting the Digital Equipment Corporation Systems Research Center and was partly supported by the Esprit basic research action SPEC (3096).

has a winning strategy. Hence, a specification is realizable iff it can be implemented for each known, deterministic environment. This in turn implies that the realizability problem is $\mathbf{\Delta}^1_2$—easier than could be expected, but still harder than the consistency problem, which is in $\mathbf{\Sigma}^1_1$.

As a special case, we consider finite-state processes linked by synchronous communication, in the style of CCS [Mil80] and CSP [Hoa85]. Specifications consist of finite-state machines plus liveness conditions expressed by Büchi automata [Buc62, VW86a] or temporal logic formulas [Pnu81, MW84, VW86a]. We show that the realizability problem for these specifications is hard for PSPACE with respect to logspace reductions and can be solved in EXPTIME. Our algorithm to check realizability yields a program that satisfies the specification. By contrast, the consistency problem for such specifications is complete in NLOGSPACE [SVW87].

2 The General Case

In this section, we study realizability in a general model. We first present our model and show that, under simple hypotheses, specifications in this model allow only Borel sets in a suitable topology on the set of behaviors. We then define realizability, meaning realizability in any arbitrary environment, and weak realizability, meaning realizability in any deterministic environment. Prima facie, weak realizability is weaker than realizability. However, we prove that the two concepts coincide for Borel specifications and we bound the complexity of realizability.

2.1 Specifications

We now describe our model. Since it is similar to ones used in previous work [Lam86, AL88], the description will be brief.

Informally, a specification describes the sequences of states that some object under observation can go through. The object could be a screen, a register, or a sheet of paper. For instance, in the specification of a factoring program, a state might consist of the values in an input register and an output register, and the specification might describe all sequences in which a state where the input register contains a number is followed by a state where the output register contains that number's largest proper factor.

Formally, a *state* is an element of a set Σ, the *state space*. A *behavior* (over Σ) is the interleaving of two equal-length sequences: a sequence of states and a sequence of *labels* from the alphabet $\Omega = \{\mu, \epsilon, \lambda\}$. We require that the first label be ϵ, and that the label between two consecutive states be λ iff the two states are identical. Intuitively, a behavior is a sequence of states, together with attributions for state changes. The label μ means that the system modifies the state, ϵ means that the environment does, and λ means that there is no change (though there may be an invisible change "inside" the system or the environment). The environment chooses the initial state. The empty behavior is denoted by Λ, and $\cdot$ denotes concatenation.

In the following definitions, σ denotes an arbitrary behavior (finite or infinite), with labels $l_0, l_1, l_2, \ldots$ and states $s_0, s_1, s_2, \ldots$, which we write

$$\xrightarrow{l_0} s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots.$$

If the length of σ is greater than m, then $\sigma|_m$ denotes the prefix

$$\xrightarrow{l_0} s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \ldots \xrightarrow{l_{m-1}} s_{m-1}$$

$\sigma_{@m}$ denotes the state s_m, and $who_m(\sigma)$ denotes the label l_m.

When σ is a finite behavior, we say that σ is *stutter-free* iff $l_i \neq \lambda$ for all i. When σ is an infinite behavior, we say that σ is *stutter-free* iff, for all i, either $l_i \neq \lambda$ or $l_j = \lambda$ for all $j \geq i$. We let $\natural\sigma$ be the stutter-free sequence obtained by replacing every maximal finite subsequence

$$s_i \xrightarrow{\lambda} s_i \xrightarrow{\lambda} \ldots \xrightarrow{\lambda} s_i$$

with the single element s_i. We define $\sigma \simeq \tau$ to mean that $\natural\sigma = \natural\tau$, and $\Gamma\sigma$ to be the set $\{\tau : \tau \simeq \sigma\}$. If P is a set of behaviors, $\Gamma(P)$ is the set $\{\tau \in \Gamma\sigma : \sigma \in P\}$. A set of behaviors P is *closed under stuttering* iff $P = \Gamma(P)$. A *property* (over Σ) is a set of infinite behaviors closed under stuttering. Closure under stuttering is essential for relating specifications at different levels of abstraction [Lam83]. An infinite behavior is *terminating* iff at most finitely many of its labels differ from λ.

For the set of labels $\Omega = \{\mu, \epsilon, \lambda\}$ and any state space Σ, let $(\Omega, \Sigma)^*$, $(\Omega, \Sigma)^+$, and $(\Omega, \Sigma)^\omega$ denote the sets of all finite, finite nonempty, and infinite behaviors over Σ, respectively. An infinite sequence $\sigma_0, \sigma_1, \sigma_2, \ldots$ of behaviors in $(\Omega, \Sigma)^\omega$ is said to *converge* to the behavior σ in $(\Omega, \Sigma)^\omega$ iff for all $m \geq 0$ there exists an $n \geq 0$ such that $\sigma_i|_m = \sigma|_m$ for all $i \geq n$. In this case, we let $\lim \sigma_i$ be σ. The definition of convergence determines a topology on $(\Omega, \Sigma)^\omega$.

We use the following standard topological notions [Mar77]. Let σ be a behavior in $(\Omega, \Sigma)^\omega$ and let P be a subset of $(\Omega, \Sigma)^\omega$. We say that σ is a *limit point* of P iff there exist elements σ_i in P such that $\lim \sigma_i = \sigma$. The set P is *closed* iff P contains all its limit points. The *closure* of P, denoted $\overline{P}$, consists of all limit points of P. A set is *open* iff its complement is closed. The class of *Borel* sets is the smallest class that contains the open sets and is closed under complementation and countable union. The $\mathbf{\Sigma}_1^1$ sets are the projections of Borel sets (these Borel sets are taken in a state space "of higher dimension," with additional state components, which projection deletes); the $\mathbf{\Pi}_1^1$ sets are the complements of $\mathbf{\Sigma}_1^1$ sets; the $\mathbf{\Sigma}_2^1$ sets are the projections of $\mathbf{\Pi}_1^1$ sets; the $\mathbf{\Pi}_2^1$ sets are the complements of $\mathbf{\Sigma}_2^1$ sets; $\mathbf{\Delta}_2^1$ is the intersection of $\mathbf{\Sigma}_2^1$ and $\mathbf{\Pi}_2^1$. With each topological class, we associate a class of problems—for example, a $\mathbf{\Sigma}_1^1$ problem is one that can be reduced to the membership problem for a $\mathbf{\Sigma}_1^1$ set.

A *specification* consists of a state space together with a property over the state space. We consider a general method of writing specifications that uses internal, *auxiliary variables*. To describe a property O over the state space Σ, we first choose a new state space Σ_I and define a property P over $\Sigma \times \Sigma_I$. We call P the *complete property* of the specification. Let $\Pi_{[\Sigma_I]}$ be the projection function that erases the Σ_I components of the states. We then define the property O to be the closure under stuttering of the projection of P—that is, the set $\Gamma(\Pi_{[\Sigma_I]}(P))$ of behaviors. We denote $\Gamma(\Pi_{[\Sigma_I]}(P))$ by $\tilde{\Pi}(P)$ and call it the *external* part of the complete property P.

The set Σ_I of internal states is used to simplify the specification. For example, in a queue specification, an element of Σ would describe the state of the input and output devices, while an element of Σ_I might be a list of values describing the contents of the queue. However, all specifications that prescribe the same input/output behavior should be considered equivalent, even if they mention different internal data structures.

A convenient way of expressing the complete property P is as the conjunction of a safety property M [AS86] and a supplementary property L. The supplementary property can be arbitrary and typically expresses liveness requirements [AS85]. We advocate the use of transition axioms or state machines for expressing safety properties, and the use of suitable logics, such as various temporal logics [Pnu81], for expressing supplementary properties. Topologically, a safety property is a closed set. In most formalisms, such as temporal logics, it is either impossible or unnatural to express properties that are not Borel sets. We therefore assume that the supplementary properties are Borel sets. This implies that the complete property is also Borel.

Note that even if the complete property is a Borel set, the external part of the complete property is not always a Borel set—it is a $\mathbf{\Sigma}_1^1$ set. However, we will show that under reasonable and useful restrictions on its specification (from [AL88]), the external part of a property is also a Borel set.

The first restriction requires that the external part $\tilde{\Pi}(\overline{P})$ of the closure of the complete property, $\overline{P}$, is closed. In particular, if the complete property P is a safety property, then so is $\tilde{\Pi}(\overline{P})$. This requirement follows from a stronger *finite-invisible-nondeterminism* condition, which states that, given any finite number of steps of a behavior allowed by $\tilde{\Pi}(\overline{P})$, there are only a finite number of possible choices for its internal state component.

The second restriction, which has been called *internal continuity*, requires that if a behavior is allowed by $\overline{P}$ and its external part is allowed by O, then the behavior is allowed by P as well. In particular, internal continuity holds when the supplementary property L mentions only the state space Σ and does not depend on the Σ_I state component.

A specification satisfying these two restrictions is said to be *regular.*

Definition 1 *The specification* $\mathbf{S}$*, with property* O *and complete property* P*, is regular iff* $\tilde{\Pi}(\overline{P})$ *is closed and* $\overline{P} \cap \tilde{\Pi}^{-1}(O) \subseteq P$.

Theorem 1 *All regular specifications define Borel properties.*

This theorem provides a partial answer to questions on the expressive power of regular specifications that were left open in previous work [AL88]. Its proof is omitted.

In the rest of Section 2, we assume that the state space Σ is fixed once and for all. We make no assumptions about how specifications are written and simply identify a specification with the property that it specifies.

2.2 Realizable Specifications

Intuitively, a specification is realizable if there exists a physical device that implements it. We start by defining an abstract computing device, which we call a *computer* for short.

Definition 2 *A computer f is a partial function from $(\Omega, \Sigma)^+$ to Σ such that, for all s, σ, τ, i, and j, if $f(\sigma|_i) = s$ and $\sigma|_i \simeq \tau|_j$ then $f(\tau|_j) = s$ as well, and $f(\sigma|_{i+1}) \neq \sigma_{@i}$.*

A computer is a function that decides state changes, on the basis of a finite initial fragment of a behavior. The first condition requires the function to be invariant under stuttering. The second condition requires that it introduce no stutters.

Definition 3 *A run of a computer f is an infinite behavior σ such that, for all i, if $who_i(\sigma) = \mu$ then $f(\sigma|_i)$ is defined and equals $\sigma_{@i}$, and if $f(\sigma|_i)$ is defined then $who_j(\sigma) \neq \lambda$ for some $j \geq i$. The set of runs of f is denoted by $R(f)$.*

This definition guarantees that the computer causes all changes that are attributed to it, and that infinite stuttering is impossible when the computer can cause a change. An additional condition, asserting that the computer always gets a chance to take a move, is considered below.

We say that a realization is the set of behaviors that can be generated with a computer, and a realizable specification is one that is implemented by some realization.

Definition 4 *A realization is a specification with property $R(f)$, for some computer f.*

Definition 5 *Specification* $\mathbf{S_1}$ *implements specification* $\mathbf{S_2}$ *if* $\mathbf{S_1} \subseteq \mathbf{S_2}$.

Definition 6 *A specification is realizable iff there exists a realization that implements it.*

The definition of realizability might seem overly restrictive because it requires implementation by a deterministic computing device. However, it is easy to check that permitting our computers to be nondeterministic would yield an equivalent definition. Our definition is actually quite liberal—for example, our computers may compute nonrecursive functions. Realizability, as we have defined it, is a necessary but not sufficient condition for the existence of a real implementation.

Realizability has many expected properties. For example, we can define a parallel-composition operation and prove that it preserves realizability.

2.3 Realizability Under Fairness

In our definition of realizability, there is no fairness condition to insure that the computer gets a chance to do anything. Thus, a specification requiring that a register eventually equals 1 is unrealizable because a continual stream of environment actions can prevent the system from ever setting the register. This specification might be realizable if only "fair runs" were allowed. One possible definition of "fair" is:

Definition 7 *A fair run of a computer f is a run of f such that if, for some i, $f(\sigma|_j)$ is defined for all $j \geq i$, then $who_j(\sigma) = \mu$ for some $j \geq i$. The set of fair runs of f is denoted by $R_F(f)$.*

Realizability under fairness is then defined in the obvious way. It is a weaker requirement than realizability.

Proposition 1 *All realizable specifications are realizable under fairness, but not conversely.*

The concept of realizability under fairness can be reduced to the usual concept of realizability. A specification $\mathbf{S}$ is realizable under fairness iff a certain specification $\mathbf{S}^*$ is realizable; $\mathbf{S}^*$ is obtained from $\mathbf{S}$ simply by adding some behaviors where the environment shuts out the system.

Theorem 2 *Let* $\mathbf{S}$ *be a specification, and let* $\mathbf{S}^* = \mathbf{S} \cup \{R(f) | \, R_F(f) \subseteq \mathbf{S}\}$. *Then* $\mathbf{S}$ *is realizable under fairness iff* $\mathbf{S}^*$ *is realizable.*

2.4 Weakly Realizable Specifications

A specification is realizable iff there exists a single computer that implements the specification in a totally unpredictable environment. One might think that this definition is too strong because it does not take into account knowledge that the implementor might have of what the environment can actually do. We therefore introduce a weaker notion of realizability, in which the implementor knows exactly how the environment will behave. Knowledge of the environment is expressed formally by describing it as a (deterministic) computer.

Definition 8 *An environment computer* h *is a partial function from* $(\Omega, \Sigma)^*$ *to* Σ *such that, for all* s, σ, τ, i, *and* j, *if* $h(\sigma|_i) = s$ *and* $\sigma|_i \simeq \tau|_j$ *then* $h(\tau|_j) = s$ *as well,* $h(\sigma|_{i+1}) \neq \sigma_{@i}$, *and* $h(\Lambda)$ *is defined.*

Definition 9 *An environment run of an environment computer* h *is a behavior* σ *such that, for all* i, *if* $who_i(\sigma) = \epsilon$ *then* $h(\sigma|_i)$ *is defined and equals* $\sigma_{@i}$, *and if* $h(\sigma|_i)$ *is defined then* $who_j(\sigma) = \epsilon$ *for some* $j \geq i$ *and* $who_k(\sigma) = \lambda$ *for all* k *such that* $i \leq k \leq j$. *The set of environment runs of* h *is denoted by* $ER(h)$.

Definitions 2 and 8 differ in that environment computers are defined on the empty behavior while "ordinary" computers are not—in other words, the environment chooses the initial state, not the system. Definition 9 guarantees that the environment computer is the first one to cause a change when it decides to. In other words, the environment has priority over the system.

Given the environment computer, an implementor needs to implement only a single behavior. Thus, a specification is realizable for a particular environment computer iff it allows some run of that computer.

Definition 10 *The specification* $\mathbf{S}$ *is weakly realizable if* $ER(h) \cap \mathbf{S} \neq \emptyset$ *for all environment computers* h.

Realizability implies weak realizability. On the other hand, it is conceivable that some specification can be implemented in each deterministic environment, with full information about the environment, but cannot be implemented in a totally unpredictable, arbitrary environment. We show below that, for Borel specifications, realizability and weak realizability are actually equivalent.

2.5 Games and Specifications

A specification can be viewed as the rules for a two-player game, where the system, playing against the environment, must cause a correct behavior to be produced. We now make this correspondence between games and specifications precise and reap a few results from known theorems on infinite games.

Infinite games of perfect information have been considered in mathematics, starting in Polish taverns in the 1930's [Mau81, GS53]. Recently, infinite games of perfect information have received much attention in descriptive set theory [Mar77]. In one of the typical scenarios, players I and II alternately produce moves; if the sequence of moves belongs to a certain *payoff set* A, then player II wins, otherwise player I wins. A *strategy* is a function that tells either player what its next move should be on the basis of previous moves. A *winning strategy* for player II (or I) is one that guarantees that the sequence obtained is in A (or not in A).

There are obvious differences between the course of a game between two players and the running of a concurrent system in an environment. Most noticeably, the system and the environment do not take turns as politely as the players in a game. On the contrary, the environment is free to act as it pleases and when it pleases. At best, that the environment acts only at particular times can be proved as a lemma, or stated as an explicit assumption. (For synchronous systems, where such an assumption is justified, the games we discuss can be simplified.) The possibility of stuttering introduces a second difference between specifications and games. The repetition of a state in a behavior does not matter for the purposes of correctness, but the presence of an idle turn in the course of a game might matter in naming a winner.

In view of these differences, it may seem reasonable to introduce a new class of games, perhaps similar to the Davis-Dubins games, where player I has freedom in choosing when to act [Dav64]. We find it more convenient to define the games that correspond to specifications as a special case of the basic scenario we have outlined.

For a specification $\mathbf{S}$, we define a game $G(\mathbf{S})$ as follows. The two players, ϵ and μ, alternately produce elements of Σ, with player ϵ starting. A move is a stutter if the element played is identical to the previous one played; a move is proper otherwise. An idle turn is a pair of consecutive stutters, one by ϵ and then one by μ. To each sequence of moves, we associate the behavior obtained by labeling the proper moves with either ϵ or μ, depending on who made them, and labeling all stutters with λ. Given a sequence of moves, player μ wins iff the behavior associated with the sequence is allowed by $\mathbf{S}$ and no proper move of μ immediately follows a proper move of ϵ.

Intuitively, the environment plays whenever it wishes. A stutter indicates that it would let the system play. The system is not allowed to make proper moves at any other time. The system wins if it has never violated this rule and if the specification allows the sequence of moves.

Despite some technical difficulties related to stuttering, the notions of realizability and weak realizability find simple expressions in terms of games.

Theorem 3 *The specification* $\mathbf{S}$ *is realizable iff player* μ *has a winning strategy in the game* $G(\mathbf{S})$*. The specification* $\mathbf{S}$ *is weakly realizable iff player* ϵ *does not have a winning strategy in the game* $G(\mathbf{S})$*.*

This correspondence leads to new insights on realizability. First, using the Axiom of Choice, Gale and Stewart have constructed games which are not determined—that is, where neither of the players has a winning strategy [GS53]. From this result, one derives the existence of weakly realizable but not realizable specifications.

Theorem 4 *Some specifications are weakly realizable but not realizable.*

All the examples we know of specifications that are weakly realizable but not realizable seem artificial. For more common specifications, realizability coincides with weak realizability, and thus appears as a robust notion. Martin has proved that all Borel games are determined [Mar75], which implies:

Theorem 5 *Weak realizability and realizability are equivalent conditions on Borel specifications.*

Martin's theorem involves extremely complex strategies [Fri71]. However, the strategies for safety properties are much simpler than those for arbitrary properties. Moreover, in important "finite cases", the result can be refined to provide finite-state strategies [BL69, GH82]. Thus, issues of realizability are relatively simple for safety specifications and propositional specifications, and become more complex as intricate liveness requirements are introduced.

From Theorem 5, we derive the following complexity result.

Corollary 1 *For Borel specifications, the realizability problem is* $\mathbf{\Delta}^1_2$.

This complexity upper bound is smaller than could be expected in general, thanks to determinacy, and it is the best possible—obviously, realizability of Borel specifications is hard for both $\mathbf{\Sigma}^1_1$ and $\mathbf{\Pi}^1_1$. However, even for Borel specifications, the complexity of the realizability problem remains higher than that of the consistency problem, which is $\mathbf{\Sigma}^1_1$.

3 The Finite Case

A common method of writing specifications is to use finite-state transition systems or finite automata. This method makes possible the automatic verification of a specification by exploring the state space. Unfortunately, whereas transition systems are well suited for specifying safety properties, they are not adequate for specifying liveness properties. This is because transition systems can specify which transitions are or are not possible (a safety property) but, as such, are incapable of stating that some transition should eventually be taken (a liveness property).

One way to express liveness properties is to add to the transition system a restriction on its infinite behaviors. For instance, one could state that in all infinite behaviors some action should be taken infinitely often. This can be done either by using an automaton on infinite words—such as a Büchi automaton [Buc62]—or by using a propositional temporal logic formula [Pnu81, MW84], which can be converted to a Büchi automaton [WVS83, VW88]. The specification of a process is then the combination of a finite automaton describing the allowed transitions of the system and a Büchi automaton restricting the infinite behaviors of the system. As one would expect, a finite automaton is a realizable specification. However, once the infinite behaviors of a finite automaton are restricted, the realizability question is much more delicate.

In this section, we define what a finite-state process is and give a semantics for processes that is the basis of our definition of realizability. We then describe an algorithm for deciding realizability and discuss the implications of realizability for finite-state verification.

3.1 Process Definition

We explore a framework suitable for the verification of finite-state processes. Our framework is in the tradition of CCS [Mil80] and TCSP [Hoa85] in that we use handshaking as a communication mechanism. However, we only use a simple process description language. Basic processes are finite automata, and parallel composition is the only operation on processes. This is not really a restriction, since finite-state CCS or TCSP programs can be systematically transformed into transition systems [Mil80, Mil84, Old85]. One substantial difference between our framework and more usual ones is that we consider the infinite behaviors of the processes and allow a restriction on these behaviors as part of the specification of processes. The syntax and semantics we use are closer to those of TCSP than to those of CCS.

Processes are defined over an alphabet of communication actions Σ. In addition to transitions corresponding to actions, processes can take silent transitions labeled by the silent (internal) action τ. A process specification is a pair $P = (P_t, P_i)$ consisting of a *finite-state transition system* P_t and an *infinitary restriction* P_i limiting the infinite sequences of communication actions of the process. The finite-state transition system P_t is a quadruple $(\Sigma, S_t, \rho_t, s_{0t})$, where

- Σ is the alphabet of communication actions,
- S_t is a finite set of states,
- $\rho_t : S_t \times (\Sigma \cup \{\tau\}) \rightarrow 2^{S_t}$ is a transition relation that for each state and action gives the possible next states,
- $s_{0t} \in S_t$ is the initial state of the process.

The infinitary restriction P_i is a finite automaton on infinite words (a Büchi automaton) on the alphabet Σ—i.e., the alphabet of communication actions of the process, not including the silent action. This automaton defines a subset of Σ^ω. We require that all infinite behaviors of the process are in this subset.

Formally, the infinitary restriction P_i is a quintuple $(\Sigma, S_i, \rho_i, s_{0i}, F_i)$, where

- Σ is the communication alphabet of the process,
- S_i is a finite set of states,
- $\rho_i : S_i \times \Sigma \rightarrow 2^{S_i}$ is a nondeterministic transition function,

- $s_{0i} \in S_i$ is a starting state,
- $F_i \subseteq S_i$ is a set of designated states.

A *run* of P_i over an infinite word $w = a_1 a_2 \ldots$ is an infinite sequence $s_0, s_1, \ldots$, where $s_0 = s_{0i}$ and $s_j \in \rho(s_{j-1}, a_j)$, for all $j \geq 1$. A run $s_0, s_1, \ldots$ is *accepting* iff there is some designated state that repeats infinitely often—that is, iff for some $s \in F_i$ there are infinitely many j's such that $s_j = s$. The infinite word w is *accepted* by P_i iff there is an accepting run of A over w. The set of denumerable words accepted by P_i is denoted $L^\omega(P_i)$.

We now define a parallel composition operation on processes that corresponds to the concurrent execution of two processes with handshaking on events common to both. Let $P_1 = (P_{t1}, P_{i1})$ and $P_2 = (P_{t2}, P_{i2})$ be two finite-state processes, where $P_{t1} = (\Sigma_1, S_{t1}, \rho_{t1}, s_{0t1})$, $P_{t2} = (\Sigma_2, S_{t2}, \rho_{t2}, s_{0t2})$, and P_{i1} and P_{i2} are Büchi automata. The process $P_1 \parallel P_2$ is the process $((P_{t1} \parallel P_{t2}), (P_{i1} \parallel P_{i2}))$. The finite-state transition system $(P_{t1} \parallel P_{t2})$ is obtained as usual by taking the product of the transition systems P_{t1} and P_{t2}, synchronizing on actions common to both processes. The infinitary restriction $(P_{i1} \parallel P_{i2})$ is the Büchi automaton that accepts all infinite words over the alphabet $\Sigma_1 \cup \Sigma_2$ whose projections on the alphabets of P_1 and P_2 are in the sets accepted by P_{i1} and P_{i2}, respectively. In other words, the parallel composition of the infinitary restrictions is a Büchi automaton that accepts all infinite sequences compatible with both component processes.

3.2 Semantics

To define realizability, we need to interpret processes in an abstract semantic domain. To choose the semantic domain, we take into account the three properties of processes that we want to observe:

- their infinite behaviors,
- their terminating finite behaviors (whether they are intended to terminate or result in deadlock),
- their possibility of diverging (producing an infinite sequence of internal actions).

To simplify our discussion, we consider only nondiverging processes. Note that parallel composition preserves nondivergence, since it does not introduce any hiding of actions.

We want our semantics to be fully abstract with respect to parallel composition. That is, we want to be able to determine the semantics of a composed process from its parts, and we want our semantics to be the weakest one compatible with this requirement.

Main [Mai86] and Hennessy [Hen87] have shown that if we restrict attention only to finite behaviors, then the process semantics satisfying our requirement are essentially failure semantics [Hoa85]. The failures of a process are the pairs (s, X), where s is a sequence of external actions of the process and X is a set of actions the process can refuse after executing s. Because we also care about the infinite behaviors of processes, our semantic domain for a process defined on the alphabet Σ is $2^{FAILURES_\Sigma} \times 2^{\Sigma^\omega}$. In other words, the semantics of a process is a set of failures and a set of infinite words.

Now, we associate an element (F, I) of the semantic domain with a process $P = (P_t, P_i)$. We use the usual definitions [BHR84, Hoa85] to associate a set F of failures with the transition system P_t. We denote by $L^\omega(P_t)$ the set of infinite behaviors allowed by P_t. These are the infinite sequences of visible actions that can be generated by the transition system P_t viewed as a Büchi automaton whose set of accepting states is the whole set of states. The set I of infinite sequences is defined to be the intersection of $L^\omega(P_t)$ and $L^\omega(P_i)$.

Finally, an order on the semantic domain represents the implementation relation.

Definition 11 *Let P_1 and P_2 be processes whose semantics are respectively (F_1, I_1) and (F_2, I_2). Then we have that $P_1 \leq P_2$ (P_1 implements P_2) iff $F_1 \subseteq F_2$ and $I_1 \subseteq I_2$.*[1]

Note that the fewer failures a process has, the fewer possible behaviors it has. An example is given in the next section.

3.3 Realizability

In the context of finite-state automata, we take transition systems without infinitary restrictions to be directly implementable.

Definition 12 *A process specification (P_t, P_i) is a realization iff $L^\omega(P_t) \subseteq L^\omega(P_i)$.*

In other words, realizations are processes whose infinitary restrictions do not actually constrain the behavior of the process.

Definition 13 *A process specification P is realizable iff there exists a realization P' such that $P' \leq P$.*

It is natural for a process with a vacuous infinitary restriction to be a realization. We allow only these realizations because an infinitary restriction, given as a Büchi automaton, is not directly implementable. It is not enough to say that something should happen infinitely often to have a program; a program should choose one or more specific ways of realizing this condition.

One might think that our notion of realizable process is too restrictive because we allow only finite-state transition systems as realizations. It is conceivable that a process described by a finite-state transition automaton and a Büchi automaton would be realizable by an infinite-state transition system, but not by a finite-state one. We show in Section 3.4 that this situation cannot arise.

Intuitively, a specification is realizable iff there exists a program that guarantees that all infinite behaviors are in the required set without introducing any new failures. The fact that we cannot introduce new failures reflects the requirement that a realizable specification can be implemented without constraining the environment. In other words, a process specification is realizable iff its infinitary requirement can be implemented by making choices within the internal nondeterminism of the finite-state transition system.

Let us examine some examples. In the system of Figure 1, the failure set consists

[1] Most orders that have been defined on semantic domains similar to ours are in the opposite direction, which is more natural when dealing with semantic issues. We choose the direction that corresponds to implementation (less nondeterminism).

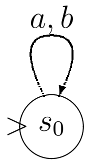

Figure 1: A transition system.

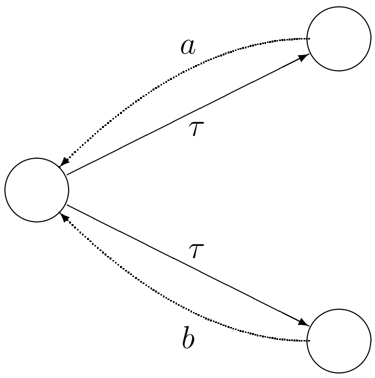

Figure 2: Another transition system.

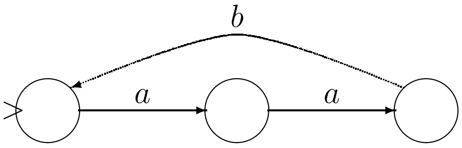

Figure 3: A realization.

of all failures of the form $(\{a, b\}^\star, \emptyset)$. The set of infinite behaviors generated by the transition system is $\{a, b\}^\omega$. This set of failures has no well defined (i.e., corresponding to a process) subset of failures. Thus, if we add to this transition system an infinitary requirement in the form of a Büchi automaton, then the specification is realizable only if the Büchi automaton accepts all words in $\{a, b\}^\omega$ (is universal).

On the other hand, the transition system of Figure 2 has as its set of failures all elements of the form $(\{a, b\}^\star, \emptyset)$, $(\{a, b\}^\star, \{a\})$, and $(\{a, b\}^\star, \{b\})$. Consequently, it forms a realizable specification in conjunction with any nonempty infinitary restriction. A typical example of such an infinitary restriction is the requirement that a and b appear infinitely often in all infinite behaviors. This is realized by the transition system of Figure 3.

As a final remark, note that the parallel composition of realizable processes is a realizable process. Indeed, the parallel composition of the realizations of the component processes is a realization of the composed process.

3.4 Deciding Realizability

We now address the problem of deciding whether a process specified by a finite-state transition system and a Büchi automaton is realizable. First, notice that this is a different problem than deciding whether the specification is consistent. Consistency simply means that the set of infinite behaviors of the process is nonempty. As the examples of the previous section show, this can be the case even if the specification is unrealizable. To solve the consistency problem, one can check that the product of the transition system and the Büchi automaton is nonempty, which can be done in polynomial time—in fact, the problem is complete for NLOGSPACE [SVW87].

The realizability problem is much harder, but it is decidable in EXPTIME:

Theorem 6 *Given a process specification $P = (P_t, P_i)$, where P_t is a finite-state transition system and P_i is a Büchi automaton, it is possible to decide in exponential time whether the specification P is realizable.*

The idea of the proof is, following the ideas in [VW86b], to build a tree automaton on infinite trees that accepts an infinite tree iff it is a realization of the specification P. The automaton checks on the one hand that all failures appearing in the infinite tree are failures of P_t, and on the other hand that all infinite paths through the tree are accepted by the Büchi automaton P_i. For the latter step, we use a result of Safra [Saf88] to construct a deterministic version of the Büchi automaton. This yields a Rabin automaton [Rab69] with a number of states exponential in the size of the Büchi automaton, but with a linear number of accepting pairs. The tree automaton is thus a Rabin tree automaton with a number of states exponential in the size of (P_t, P_i) and a linear number of accepting pairs.

One then checks whether this tree automaton is nonempty. Using results of Emerson and Jutla [EJ88], this can done in time polynomial in the size of the tree automaton. Our algorithm for checking realizability is thus in EXPTIME.

If the tree automaton is indeed nonempty, it follows by the results of Hossley and Rackoff [HR72] that there is a finitely generated tree accepted by the automaton. This implies that if a process has an infinite realization, then it also has a finite-state realization.

Theorem 7 *Given a process specification $P = (P_t, P_i)$, where P_t is a finite-state transition system and P_i is a Büchi automaton, the problem of deciding whether P is realizable is logspace hard for polynomial space.*

This follows from the example of Figure 1, which shows that the universality problem for Büchi automata, which is PSPACE-complete [SVW87], can be reduced to the realizability problem.

3.5 Realizability and Verification

One can verify that a set of finite-state processes operating concurrently satisfies a requirement by verifying that their parallel composition satisfies it. Such a verification can tell us that we have a correct abstract representation of the processes to be implemented. But, doing this with no regard for realizability can lead to one of two unpleasant consequences. The first is that, during the implementation, someone notices that the specified processes are not implementable. The abstract description should then be revised and the verification redone. The second, and more dangerous, possible consequence is that the implementation corresponds to a different abstract description—for example, that of Figure 2 instead of Figure 1. In this case, of course, the verification of the abstract descriptions is meaningless, and the implemented system may exhibit unexpected behaviors such as deadlocks.

Parrow [Par85] has introduced an interesting "infinitary process algebra". It is essentially an extension of CCS with restrictions on infinite behaviors expressed by temporal logic formulas or Büchi automata. The theory of this algebra is quite well developed and is used for verification. However, the concept of realizable specifications is not considered, which makes the verifications done in this framework potentially meaningless.

4 Conclusion

Realizability has only recently received attention. One might think that the realizability problem would have been addressed by work on synthesizing concurrent programs from temporal logic specifications—for example, by Emerson and Clarke [EC82] or Manna and Wolper [MW84]. However, these approaches avoid the issue of realizability by dealing only with closed systems, in which there is no environment.

The work of Pnueli and Rosner [PR89a, PR89b] probably comes the closest to describing the realizability problem as we understand it. There, one finds an elegant approach for synthesizing reactive modules from finite-state specifications. Their synthesis method uses automata on trees and is similar to our method for checking realizability in the finite-state case. However, our method is a little more general because we check realizability with respect to a transition system, so we can check realizability in an environment whose behavior is restricted, not just in an arbitrary environment.

Even if one is not interested in automatic synthesis (which rarely produces usable results), it is important to know that not all specifications are realizable and to be able to distinguish between realizable and unrealizable specifications. How easy it is to avoid writing unrealizable specifications provides a new criterion for judging specification styles.

Acknowledgments

We are grateful to Leo Harrington for his help with Theorem 1, and to Yuri Gurevich and Mark Manasse for interesting discussions on determinacy.

References

[AL88] Martín Abadi and Leslie Lamport. *The Existence of Refinement Mappings.* Research Report SRC29, Digital Equipment Corporation, Systems Research Center, August 1988. A short version of this paper appeared in the *Proceedings of the Third Annual Symposium on Logic in Computer Science.*

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[AS86] Bowen Alpern and Fred B. Schneider. *Recognizing Safety and Liveness.* Technical Report TR86-727, Department of Computer Science, Cornell University, January 1986.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(7):560–599, 1984.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[Buc62] 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 University Press, Stanford, 1962.

[Dav64] Morton Davis. Infinite games of perfect information. In M. Dresher, L. S. Shapley, and A. W. Tucker, editors, *Advances in game theory*, pages 85–101, Princeton University Press, Princeton, New Jersey, 1964.

[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. Allen Emerson and Charanjit 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.

[Fri71] Harvey Friedman. Higher set theory and mathematical practice. *Annals of Mathematical Logic*, 2:326–357, 1971.

[GH82] Yuri Gurevich and Leo Harrington. Trees, automata, and games. In *Proceedings of the 14th Symposium on Theory of Computing*, pages 60–65, ACM, May 1982.

[GS53] D. Gale and F.M. Stewart. Infinite games with perfect information. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the theory of games, Volume 2*, pages 245–266, Princeton University Press, Princeton, New Jersey, 1953.

[Hen87] M. Hennessy. Why testing equivalence is natural. April 1987. Handwritten note.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP85] David Harel and Amir Pnueli. *On the development of reactive systems*, pages 477–498. *NATO ASI Series, F13*, Springer-Verlag, 1985.

[HR72] R. Hossley and C. W. Rackoff. The emptiness problem for automata on infinite trees. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 121–124, 1972.

[Lam83] Leslie 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, Paris, September 1983.

[Lam86] Leslie Lamport. *A simple approach to specifiying and verifying concurrent systems*. Research Report 15, Digital Equipment Corporation, Systems Research Center, December 1986.

[Mai86] Michael G. Main. *Demons, Catastrophies and Communicating Processes.* Technical Report CU-CS-343-86, Department of Computer Science, University of Colorado, July 1986.

[Mar75] Donald A. Martin. Borel determinacy. *Annals of Mathematics*, 102:363–371, 1975.

[Mar77] Donald A. Martin. Descriptive set theory: projective sets. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 783–815, North-Holland Publishing Co., 1977.

[Mau81] R. Daniel Mauldin, editor. *The Scottish Book.* Birkhäuser, Boston, 1981.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, 1980.

[Mil84] R. Milner. A complete inference system for a class of regular behaviours. *Journal of Compututer and System Science*, 28:439–466, 1984.

[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.

[Old85] E. R. Olderog. Process theory: semantics, specification and verification. In *Proc. Advanced School on Current Trends in Concurrency*, pages 442–509, Volume 224, LNCS, Springer-Verlag, Berlin, 1985.

[Par85] J. Parrow. *Fairness Properties in Process Algebra.* PhD thesis, University of Uppsala, Sweden, 1985.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

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

[PR89b] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of ICALP 89*, Stresa, July 1989.

[Rab69] M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

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

[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.

[VW86a] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW86b] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.

[VW88] M. Y. Vardi and P. Wolper. Reasoning about infinite computation paths. 1988. To appear.

[WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, Tucson, 1983.