Alcom-FT Technical Report Series
ALCOMFT-TR-01-161

A Computational Model of Interaction in Embedded Systems*

Jan van Leeuwen[1] Jiří Wiedermann[2]

[1] Department of Computer Science, Utrecht University,
Padualaan 14, 3584 CH Utrecht, the Netherlands.
[2] Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic.

Abstract. Embedded systems behave very differently from classical machine models: they interact with an unpredictable environment, they never terminate, and learn over time. The behavior of embedded systems has been well-studied in concurrency theory but Wegner [29, 30] recently appealed for a new theory, claiming that the computational features of interaction are not adequately captured by traditional models of computation.
We describe a simple model of interactive computing consisting of one component C and one environment E, interacting using single streams of input and output signals. The model enables us to study the computational implications of interaction. Viewing components as interactive transducers, we show that the interactive recognition and generation capabilities of components are identical. We also show that, in the given model, all interactively computable functions are limit-continuous. An interesting inversion theorem is proved. Several connections to the theory of ω-automata are pointed out, placing the subject area in a classical framework.

1 Introduction

We will study some aspects of embedded systems from a computability-theoretic viewpoint. The term 'embedded systems' encompasses a broad range of hardware and software systems that are considered to form an integral part of a larger system (see e.g. [18]). Embedded systems communicate and compute, in interaction with an environment. Embedded systems behave very differently from traditional models of computation: their input is unpredictable and is not specified in advance, they never terminate (unless a fault occurs), and they 'learn' over time. In this paper we aim to derive some general results for the kind of interactive computing behavior which (components of) embedded systems can exhibit.

The purpose of an embedded system is usually not to compute some final result, but to react to or interact with the environment in which the system is placed and to maintain a well-defined action-reaction behavior. In the late nineteen seventies and early nineteen eighties, this behavior of systems received much attention in the formal specification and verification of concurrent processes (see Milner [11, 12]). The term 'reactive systems' was introduced to refer to systems with this kind of behavior. Pnueli [14] (p. 511) writes:

> *"Reactivity characterizes the nature of interaction between the system and its environment. It states that this interaction is not restricted to accepting inputs on initiation and producing outputs on termination. In particular, it allows some of the inputs to depend on intermediate outputs."*

Reactive systems clearly are 'infinite state' systems and have a behavior that is evolving over time. One can also say that their program is changing over time.

Wegner [29, 30] (see also Wegner and Goldin [32]) recently called for a more computational view of reactive systems, claiming that they have a richer behavior than 'algorithms' as we know

* This research was partially supported by GA ČR grant No. 201/00/1489 and by EC Contract IST-1999-14186 (Project ALCOM-FT). A preliminary version of this report appeared in [24].

them. He even challenged Church's Thesis by proclaiming that Turing machines cannot adequately model the interactive computing behavior of reactive systems. Wegner [30] (p. 318) writes:

> *"The intuition that computing corresponds to formal computability by Turing machines ... breaks down when the notion of what is computable is broadened to include interaction. Though Church's thesis is valid in the narrow sense that Turing machines express the behavior of algorithms, the broader assertion that algorithms precisely capture what can be computed is invalid."*

For a discussion of Wegner's claim from the viewpoint of computability theory, we refer to Prasse and Rittgen [16]). Irrespective of whether the claim is valid, it is interesting to look at some of the computational implications of reactive, or interactive, computing. The formal aspects of Wegner's theory of interaction are studied in e.g. Wegner and Goldin [31, 33].

In this paper we give a simple model of interactive computing, consisting of one component C and one environment E interacting using single streams of input and output signals over a simple alphabet. The notion of 'component' that we use is very similar to Broy's [1] but we restrict ourselves to deterministic components only. We identify a special condition, referred to as the interactiveness condition, which we will impose throughout. Loosely speaking, the condition states that C is guaranteed to give some meaningful output within finite time any time after receiving a meaningful input from E and vice versa. The model is described in detail in section 2. We note that our aim is not to formally specify any processes but to analyze the capabilities of the model from the perspective of computability theory.

In the model we prove a number of general results for the interactive computing behaviour which a component C can exhibit, assuming that E can behave arbitrarily and unpredictably. In most results we assume that C is a program with unbounded memory, with a memory contents that is building up over time and that is never erased (unless the component explicitly does so). This compares to the use of persistent Turing machines by Goldin [4] (see also Goldin and Wegner [5]) and Kosub [8]. No special assumptions are made about the 'speed' at which C and E can operate and generate responses. In sections 3 and 4 we show a number of simple results that indicate how interactive computing can lead to nonrecursive behaviour.

Viewing components as interactive transducers of the signals that they receive from their environment we show in section 5 that, using suitable definitions, recognition and generation coincide just like they do in the case of Turing machines. The proof is more intricate than in the latter case and depends on some of the special operational assumptions in the model. Finally, in section 6 we define a general notion of interactively computable functions. We prove that interactively computable functions are limit-continuous, using a suitable extension of the notion of continuity known from the semantics of computable functions. We also prove an interesting inversion theorem which states that interactively computable 1-1 functions have interactively computable inverses.

The study of 'machines' working on infinite input streams (ω-words) is by no means new and has a sizable literature, with the first studies dating back to the nineteen sixties and seventies (cf. Thomas [20], Staiger [19]). In the model studied in the present paper a number of features are added that are meant to better capture some intuitive notions of interactiveness, inspired by Wegner's papers. It will appear that, in terms of the classical theory of ω-languages, the interactively recognizable (or: generated) languages are topologically closed and that the interactively computable functions are 'continuous mappings', considering the domain $\{0,1\}^\omega$ as a topological space with the normal product (Cantor) topology. These connections provide the theory of interactive computing with a firm basis in the known framework of ω-computations.

In the remainder of this paper we assume some familiarity with the rudiments of computability theory [17, 13].

2 A Simple Model of Interactive Computation

Let C be a component (a software agent or a device) that interacts with an environment E. We assume that C and E interact by exchanging signals (symbols). Although general interactive systems do not need to have a limit on the nature and the size of the signals that they exchange, we assume here that the signals are taken from a fixed and finite alphabet. More precisely:

> (*Alphabet*) C and E interact by exchanging symbols from the alphabet $\Sigma = \{0, 1, \tau, \sharp\}$.

Here 0 and 1 are the ordinary bits, τ is the 'silent' or empty symbol, and $\sharp$ is the fault or error symbol. Instead of the bits 0 and 1, one could use any larger (finite) set of symbols but this is easily coded back into binary form. We assume that the interaction is triggered by E.

In order to describe the interactions between C and E we assume a uniform time-scale of discrete moments $0, 1, 2 \ldots$. C and E are assumed to work synchronously, in the following sense. At any time t, E can send a symbol of Σ to C and C can send a symbol of Σ to E. It is possible that E or C remains 'silent' for a certain amount a time, i.e. that either of them does not send any active signal during some consecutive time moments. During these moments E or C is assumed to send the symbol τ, just to 'record' this. For the symbol $\sharp$ a special convention is used:

> (*Fault rule*) If C receives a symbol $\sharp$ from E, then C will output a $\sharp$ in finite time after that as well (and vice versa).

If no $\sharp$'s are exchanged, the interaction between E and C is called fault-free (error-free).

Some further assumptions are necessary. First of all, we assume that when E (C) sends a signal to C (E) during time t, then C (E) 'knows' this signal from the next time-moment onward. (This does not necessarily mean that E or C has processed the symbol in any meaningful way by time $t + 1$, but we do assume that the signal has entered their circuitry somehow.) Second, to disambiguate the interaction even further, we assume that the interaction is always initiated by E, i.e., at any moment, E sends its signal (if any) to C first and C sends its signal (if any) to E next. It means that the communication between E and C can be described by the two sequences $e = e_0 e_1 \ldots e_t \ldots$ and $c = c_0 c_1 \ldots c_t \ldots$, with e_t denoting the signal that E sends to C at time t and c_t denoting the signal that C sends to E at time t. Here e may also be regarded as the 'interactive input sequence' and c as the corresponding 'interactive output sequence'. When E or C is silent at time t, then the corresponding symbol is τ ('empty'). If two infinite sequences e and c correspond to the actual interactions that take place over time, we say that the sequences represent an *interactive computation* of C, i.e. of C in response to the (unpredictable) environment E. C is called an interactive component. We let $\overline{e}$ and $\overline{c}$ denote the sequences e and c *without* the τ's.

The interactiveness of E and C is assumed to manifest itself as follows. First of all, we assume that the signal C sends to E during time t depends deterministically on the internal state of C at time $t-1$ and on the signal that C received at that time. We assume that C acts according to some program that evolves deterministically over time, in a way depending on the history of its past interaction with E. We assume likewise that the signal E sends to C during time t depends on what E remembers from the earlier interaction with C and on c_{t-1}, but now also on its unpredictable 'mood' or situation (which may vary over time) which can lead it to send any symbol it wants. Thus, E can be totally indeterministic and unpredictable in generating its response. For later reference we write this as $E_{t-1}(c_{t-1}) \ni e_t$, where E_{t-1} represents all 'knowledge' that E possesses at the moment that it generates the response for output to C at time t and all situations that can lead it to generate e_t (which will be an unpredictable choice from the symbols from Σ).

Note that at time t, E and C can in principle be assumed to know the sequences of signals that they have sent at previous times as well. Thus, for example, C's output at time t will depend on C's program, on $e_0 e_1 \ldots e_{t-1}$ and, implicitly, on $c_0 c_1 \ldots c_{t-1}$. The same holds for E, except that one would also have to know the 'situations' of E that underly its unpredictable response at the separate time moments. We assume that E and C somehow generate their e_0 and c_0 signals spontaneously, with C always generating c_0 deterministically e.g. for example always as τ.

We assume the following *property* as being characteristic for interactive computations: E sends signals to C infinitely often, and C is guaranteed to give a meaningful (non-empty) output within finite time any time after receiving an input signal from E. More precisely:

(*Interactiveness*) For all times t, when E sends a non-empty signal to C at time t, then C sends a non-empty signal to E at some time t' with $t' > t$ (and vice versa).

The condition of interactiveness is assumed throughout this paper. Note that, in the definition of interactiveness, we do not make any special assumption about the causality between the signal sent (by E or C) at time t and the signal sent (by C or E respectively) at time t'. With the condition of interactiveness into effect, the behavior of a component C with respect to E is a relation on infinite strings over Σ. It consists of the (deterministic) responses that C may have to all possible behaviors that E may exhibit. We assume that E sends a *nonempty* signal at least once, thus always triggering an interaction sequence with infinitely many nonempty signals.

Definition 1. *An* interaction pair *of C and E is any pair (e, c) such that $e = e_0 e_1 \ldots e_t \ldots$ and $c = c_0 c_1 \ldots c_t \ldots$ represent an interactive computation of C in response to E.*

In the model we make the following further assumptions about the internal operations of C, aside from the fact that it acts deterministically: C has only one channel of interaction with E, C admits multi-threading (allowing it to run several processes simultaneously), and C has a fixed but otherwise arbitrary speed of operation (i.e. any non-zero speed is allowed which comes with the component). As a consequence it will be possible for C to have a foreground process doing e.g. the I/O-operations very quickly and have one or more background processes running at a slower pace simultaneously. The following crucial assumption will be made also, as in classical computability theory:

(Unbounded component memory) C works without any a priori bounds on the amount of available memory, i.e. its memory space is potentially infinite.

In particular C's memory is never reset during an interactive computation, unless its program explicitly does so. We allow C to build up an 'infinite' database of knowledge that it can consult in the course of its interactions with the environment.

The environment E can behave like an adversary and can send signals in any arbitrary, unpredictable and non-algorithmic way. We do assume it has the full choice of symbols from Σ at its disposal at all times. The fact that E can respond in many different ways at any given moment accounts for the variety of output sequences C can produce (generate), when we consider all possible behaviors later on. We assume that, in principle, E can generate any possible signal at any moment but do not concern ourselves with the way E actually comes up with a specific signal in any given circumstance.

Considerable care is needed in dealing with sequences of τ's. A sequence of 'silent steps' can often be meaningful to C or E, e.g. for internal computation or timing purposes. It means that a given infinite sequence of non-empty signals may lead to a multitude of different transductions c, depending on the way E cares to intersperse the sequence with 'empty' signals. One should note that E is fully unpredictable in this respect. However, the assumed interactiveness forces E to keep sequences of intermittent empty signals finite. For the purposes of this paper we assume that the environment E sends a non-empty signal at every moment in time, i.e. $e = \overline{e}$ for all sequences of environment input that we consider in this model. The assumption simplifies the presentation but does not affect the generality of the model.

(Full environmental activity) At all times t, E sends a non-empty signal to C.

Note that we retain the possibility for C to emit τ's and thus to do internal computations for some time without giving output, even though the assumed interactiveness will force C to give some non-empty output after finite time.

Despite the assumed generality, it is conceivable that E is constraint in some way and can generate at some or any given moment only a *selection* of the possible signals from Σ. (We assume that the infinity of the interaction is never in danger, i.e., there should always be at least one allowable non-empty symbol that E can send.) In such environments a component may be confronted with and thus act on only a, possibly very irregular, subset of the possible input sequences. If this is the case, one may wish to assume that the constraint behavior of E can be checked algorithmically afterwards (i.e. every time after E has generated a response).

> (*Algorithmicity of environmental input*) When an arbitrary infinite sequence over Σ is supplied as input to C, symbol after symbol, it can be algorithmically verified alongside of it whether this sequence could have been input by E, taking into account the stepwise interaction of C and E and any constraint which may have restricted E's choice of signals at any given moment.

Algorithmicity means that there is some program $\mathcal{E}$ which evolves over time just like the program of C does and which answers, possibly after some finite delay, whether $E_{t-1}(c_{t-1}) \ni e_t$ or not, given t and e_t as input and knowing the whole interaction history up until $t-1$ (and assuming the given sequence was correct up until then). The assumption of algorithmicity does not interfere with or change the unpredictability of E as it generates its responses in any 'run' during the lifetime of a component. The assumption only implies that, regardless of whatever situation E is in when generating its inputs to C and acting on the responses interactively, there is an algorithmic way to *verify afterwards* that a given sequence *could* have been generated by E e.g. in a simulation. As soon as the sequence deviates and becomes inconsistent with E's possible actions, the verifier is assumed to output an error message from that point onward indicating (in the simulation) that the current sequence is not allowable. Without constraints there is no need for a special verifier, but when constraints are in effect there is.

A component C that interacts with its environment according to the given assumptions, will be called an *interactive machine* or a *interactive component*. We assume that its operation is governed by an evolving process that can be effectively simulated (given a simulation of any behavior of E).

It will be helpful to describe an interactive computation of C and E also as a mapping (transduction) of strings e (of environment inputs) to strings c (of component responses). In this way C acts as an ω-transducer on infinite strings, with the special incremental and interactive behavior as described here (and with game-theoretic connotations). To appreciate the following definition, recall that $e = \overline{e}$.

Definition 2. *The behavior of C with respect to E is the set $T_C = \{(e, \overline{c}) | (e, c)$ is an interaction pair of C and E $\}$. If (e, c) is an interaction pair of C and E, then we also write $T_C(e) = \overline{c}$ and say that $\overline{c}$ is the interactive transduction of e by C.*

Definition 3. *A relation T on infinite strings is called interactively computable if and only if there is an interactive component C such that $T = T_C$.*

Seemingly simple transductions may be impossible in interactive computation, due to the strict requirement of 'interactiveness' and the unpredictability of the environment. Let $0^\star$ denote the set of finite sequences of 0's (including the empty sequence), $\{0, 1\}^\star$ the set of all finite strings over $\{0, 1\}$, and $\{0, 1\}^\omega$ the set of infinite strings over the alphabet $\{0, 1\}$.

Example 1. We claim that no (interactive) component C can exist that transduces its inputs such that input streams that happen to be of the form $1\alpha 1\beta 1\gamma$ are transduced to $1\beta 1\alpha 1\gamma$, with $\alpha, \beta \in 0^\star$ and $\gamma \in \{0, 1\}^\omega$. Note that the mapping would amount to swapping the first and the second block of zeroes in a string starting with a 1, *empty blocks allowed.* Suppose that there was a component C that could do this. Consider how C would respond to an input $100\ldots$ sent to it by E, assuming that E keeps sending 0's until further notice. By interactiveness, C must send a non-empty signal to E at some time, and we may assume without loss of generality that the first signal it sends is a 1. But, by interactiveness C must generate further non-empty signals. Denote the second non-empty symbol which C sends by σ. Now let E act as follows. If $\sigma = 0$ (meaning that C's output starts with 10), then let E switch to sending 11 (implying that β is empty) and anything it wants after that. If $\sigma = 1$ (meaning that C's output starts with 11), then let E switch to sending 101 (implying that $\beta = 0$) and anything it wants after that. If $\sigma = \sharp$, the computation clearly is not fault-free. It follows that in all cases C has been fooled into sending the wrong output.

The example crucially depends on the fact that C cannot predict that the input will be of the special form specified for the transduction.

3 Interactive relations and programs

Given a sequence $u \in \{0,1\}^\omega$ and $t \geq 0$, let $pref_t(u)$ be the length-t prefix of u. For finite or infinite strings u we write $x \prec y$ if x is a finite (and strict) prefix of u. We paraphrase the common definition of monotonic functions (cf. [34]) for the case of partial functions as follows.

Definition 4. *A partial function $g : \{0,1\}^\star \to \{0,1\}^\star$ is called* monotonic *if for all $x, y \in \{0,1\}^\star$, if $x \prec y$ and $g(y)$ is defined then $g(x)$ is defined as well and $g(x) \preceq g(y)$.*

The following observation captures some aspects of the computational model in terms of monotonic functions.

Theorem 1. *If a relation $T \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$ is interactively computable, then there exists a classically computable, monotonic partial function $g : \{0,1\}^\star \to \{0,1\}^\star$ such that for all $(u,v) \in \{0,1\}^\omega \times \{0,1\}^\omega$, $(u,v) \in T$ if and only if for all $t \geq 0$ $g(pref_t(u))$ is defined, $\lim_{t\to\infty} |g(pref_t(u))| = \infty$ and for all $t \geq 0$ $g(pref_t(u)) \prec v$.*

Proof. Let $T = T_C$. We define g by designing a Turing machine M_g for it. Given an arbitrary finite string $x = x_0 x_1 \ldots x_{t-1} \in \{0,1\}^\star$ on its input tape, M_g operates as follows. M_g simulates C using the 'program' of C, feeding it the consecutive symbols of x as input and checking every time it does so whether the next symbol is an input signal that E *could* have given on the basis of the interaction with C up until this moment. To check this, M_g employs the verifier $\mathcal{E}$ which exists by the assumed algorithmicity of E (and which evolves along with the simulation). As long as no inconsistency is detected, M_g continues with the simulation of the interaction of E and C. Whenever the simulation leads C to output a signal 0 or 1, M_g writes the corresponding symbol to its output tape. When the simulation leads C to output a τ, M_g writes nothing. When the simulation leads C to output a $\sharp$ or when the verifier detects that the input is not consistent with E's possible behavior, then M_g is sent into an indefinite loop. If M_g has successfully completed the simulation up to and including the (simulation of the) processing of the final input symbol x_{t-1}, then M_g halts. It follows that M_g terminates if and only if x is a valid beginning of an interaction of E with C, with C's response appearing on the output tape if it halts. The result now follows by observing what properties of g are needed to capture that $(u,v) \in T$ in terms of M_g's action on the prefixes of u. The constraints capture the interactiveness of C and E and the fact that the interaction must be indefinite. It is clear from the construction that g is monotonic.

For at least one type of interactively computable relation can the given observation be turned into a complete characterisation. Let a relation $T \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$ be called *total* if for every $u \in \{0,1\}^\omega$ there exists a $v \in \{0,1\}^\omega$ such that $(u,v) \in T$. Note that the behaviours of interactive components in environments without constraints are always total relations. In the following result the monotonicity of g is *not* assumed beforehand.

Theorem 2. *Let $T \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$ be a total relation. T is interactively computable if and only if there exists a classically computable total function $g : \{0,1\}^\star \to \{0,1\}^\star$ such that for all $(u,v) \in \{0,1\}^\omega \times \{0,1\}^\omega$, $(u,v) \in T$ if and only if $\lim_{t\to\infty} |g(pref_t(u))| = \infty$ and for all $t \geq 0$ $g(pref_t(u)) \prec v$.*

Proof. The 'only if' part follows by inspecting the proof of theorem 1 again. If T is total, then the constructed function g is seen to be total and the stated conditions are satisfied.

For the 'if' part, assume that $T \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$ is a total relation, that g is a computable total function and that for all $(u,v) \in \{0,1\}^\omega \times \{0,1\}^\omega$, $(u,v) \in T$ if and only if $\lim_{t\to\infty} |g(pref_t(u))| = \infty$ and for all integers $t \geq 0$ $g(pref_t(u)) \prec v$. To prove that T is interactively computable, design a component C that operates as follows. (The totality of T implies that we only have unconstrained environments.)

While E feeds input, a foreground process of C keeps buffering the input symbols in a queue $q = q_0 q_1 \ldots q_t$ with $t \to \infty$. Let $r \in \{0,1\}^\star$ be the (finite) output generated by C at any given moment. We will maintain the following invariant: q is a prefix of u and r a prefix of v, for some pair $(u, v) \in T$. Letting q grow into 'u' by the input from E, we let r grow into 'v' by letting C carry out the following background process P every once in a while. C keeps a counter c_q that is initialized to 1 before any call to P has occurred. C outputs 'empty' signals as long as a call to P is running.

When called, P copies the length-c_q prefix of q into the variable x, it increments c_q by 1, and computes $g(x)$ using the subroutine for g. (Note that the string now in x extends the string on which the previous call of P operated by precisely one symbol.) By totality of g the subroutine ends in finitely many steps. Let $y = g(x)$ be the output string. By totality of T and the second condition on g only two cases can occur: $r \prec y$ or $y \preceq r$. If $r \prec y$, then C outputs the symbols by which y extends r one after the other, updates r to account for the new output, and calls P again after it has done so. If $y \preceq r$, C does not generate any special output and simply moves on to another call of P, provided at least one further input symbol has entered the queue in the meantime (which will be so by the assumed environmental activity). Note that every call to P maintains the invariant.

Because $\lim_{t\to\infty} |g(pref_t(u))| = \infty$, there will be infinitely many calls to P in which the case '$r \prec y$' occurs. Thus r will grow to infinity, with the output generated by C being precisely $\lim_{t\to\infty} r = v$.

For total relations $T \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$, we say that T *is 'implied in the limit' by* g if T and g are related as in theorem 2. Combining theorems 1 and 2 we obtain:

Corollary 1. *For total relations $T \subseteq \{0,1\}^\omega \times \{0,1\}^\omega$, if T is implied in the limit by some classically computable total function $g : \{0,1\}^\star \to \{0,1\}^\star$ then it is mplied in the limit by a classically computable total function $g : \{0,1\}^\star \to \{0,1\}^\star$ that is monotonic.*

The corollary expresses the relationship between interactive computability and monotonicity very succinctly: *the interactively computable total relations are precisely the relations implied in the limit by classically computable, monotonic total functions on* $\{0,1\}^\star$. We will return to this characterisation of interactive computations in Section 6.

It is realistic to assume that the initial specification of C is a program, written in some acceptable programming system. For example, the internal operation of C could be modeled by a persistent Turing machine of some sort (as in Wegner and Goldin [31] and Goldin [4]). In our model, the underlying program itself may evolve as well. It is easily argued that interactiveness, as a property of arbitrary component programs, is recursively undecidable, by reducing from the Halting Problem. The following stronger but still elementary statement can be observed, thanking discussions with B. Rovan.

Theorem 3. *The set of interactive programs is not recursively enumerable.*

Proof. Let $S = \{\pi | \pi$ is the program of a component C that is interactive$\}$. Suppose that S is recursively enumerable. We use a simple diagonal argument to obtain a contradiction. Let $\pi_1, \pi_2, \ldots$ be a recursive enumeration of S. Consider the programs π_i and observe how they operate when the environment just feeds 1's to them (without empty signals). We now construct the following program π, designed to be different from all π_i's. We let π react the same way regardless of the input that it receives from E, but it is useful to imagine it working under the same environment input of all 1's. Let r and r_i denote the finite sequences of output generated by π and π_i in the course of the computation.

As soon as π receives signals from E, it starts. Now π proceeds in *stages*, starting with stage 1. During stage i, π simulates the interactive computation of π_i until its output sequence $\overline{r_i}$ has become longer than $\overline{r}$ (the output of π so far). During the simulation π only outputs τ's. If, during the simulation of π_i, E would decide to stop inputting 1's based on π_i's response and switch to giving an input signal different from a 1, then we stop the simulation, at this point.

Now consider the cases that can occur in the simulation. Assume that E could input 1's all the way (in the simulation of π_i). Then the situation that $\overline{r_i}$ becomes longer than $\overline{r}$ will occur. We can assume w.l.o.g. that, when this happens, $\overline{r_i} = \alpha\delta$ with $|\alpha| = |\overline{r}|$ and $\delta \in \{0,1\}$. At this point, let π output a signal $\delta^c \in \{0,1\}$ different from δ (making $\overline{r}$ into $\overline{r}\delta^c$). If E could not/did not input 1's the whole way (in the simulation), let π output any non-empty symbol, say a 1. After this, let π go to stage $i+1$.

By interactiveness, every stage of π is finite and thus π itself is interactive. The construction guarantees that π is different from every π_i (by comparing how the programs react when the environment sends 1's all the time) and thus $\pi \notin S$. This contradicts the definition of S.

In the further sections we study the given model of interactive computing from three different perspectives: recognition, generation and translation (i.e. the computation of mappings on infinite binary strings).

4 Interactive Recognition

Embedded systems typically perform a task in *monitoring*, i.e. the online perception or recognition of patterns in infinite streams of signals from the environment. The notion of recognition is well-known and well-studied in the theory of ω-automata (cf. [9, 19–21]) and is usually based on the automaton passing an infinite number of times through one or more 'accepting' states during the processing of the infinite input string. In interactive systems this is not detectable and thus this kind of criterion is not applicable. Instead, a component is normally placed in an environment that has to follow a certain specification and the component has to 'observe' that this specification is adhered to. This motivates the following definition.

Definition 5. *An infinite string $\alpha \in \{0,1\}^\omega$ is said to be recognized by C if $(\alpha, 1^\omega) \in T_C$.*

The definition states that, in interactive computation, an infinite string α is recognized if C outputs a 1 every once in a while and *no* other symbols except τ's in between, where E generates the infinite string α as input during the computation. As a criterion it is closely related to the notion of *1'-acceptance* of ω-words by ω-automata due to Landweber [9] which requires an ω-automaton to accept by *always* staying in a designated subset of states while processing the infinite input string, and which thus finds a new and natural interpretation in our context. Note that in our definition we need *no* assumption on any internal structure of the machine or component.

In interactive computation, a recognized string can never contain a $\sharp$ because in finite time it would lead C to output a $\sharp$ as well, causing C to reject the input from E. We can also assume that C does not output any $\sharp$ by itself either for, if it did, we might as well have it output a 0 instead without affecting the definition of what is recognized and what is not.

Definition 6. *The set interactively recognized by C with respect to E is the set $J_C = \{\alpha \in \{0,1\}^\omega | \alpha$ is recognized by C $\}$.*

Note that interactiveness is required on all infinite inputs, i.e. also on the strings that are not recognized. As a curious fact we observe that when it comes to recognition we may assume that C makes no silent steps: letting C output a 1 whenever it wants to output a τ does not affect the set of strings recognized by the component.

Definition 7. *A set $J \subseteq \{0,1\}^\omega$ is called* interactively recognizable *if there exists a component C such that $J = J_C$.*

Interactive recognizability as defined directly corresponds to Landweber's notion of *1'-definability* for ω-automata ([9], see also [19]).

Considering Wegner's claim that interactive computing is more powerful than classical computation (cf. section 1), the question arises whether this is reflected somehow in the recognition power of interactive components. To large extent the power of interactive computation comes from the infinite behavior, but at the same time it comes with new limitations as well. We prove a number of results that all have their analogies for ω-automata but which we argue here for the case of interactive components. The requirement of interactiveness apparently enables components to see 'patterns' within predictable finite time only, as shown in the following examples (assuming unconstrained environments).

Lemma 1. *The following sets are interactively recognizable:*
(i) $J = \{\alpha \in \{0,1\}^\omega | \alpha \text{ contains at most } k \text{ ones}\}$*, for any fixed integer* k*,*
(ii) $J = \{\alpha \in \{0,1\}^\omega | \alpha \text{ has a } 1 \text{ in all the prime number positions and nowhere else}\}$.
The following sets are not *interactively recognizable:*
(iii) $J = \{\alpha \in \{0,1\}^\omega | \alpha \text{ contains finitely many } 1\text{'s}\}$*,*
(iv) $J = \{\alpha \in \{0,1\}^\omega | \alpha \text{ contains infinitely many } 1\text{'s}\}$*,*
(v) $J = \{\alpha \in \{0,1\}^\omega | \alpha \text{ contains precisely } k \text{ ones}\}$*, for any fixed integer* $k \geq 1$*,*
(vi) $J = \{\alpha \in \{0,1\}^\omega | \alpha \text{ contains at least } k \text{ ones}\}$*, for any fixed integer* $k \geq 1$.

Proof. *(i)* Let C output a 1 with every 0 that it receives from E, and let it continue doing so until after the k'th 1 that it sees. Let C switch to outputting 0's after it receives the $(k+1)$-st 1. C is interactive and precisely recognizes the set J.
(ii) Left to the reader.
(iii) Suppose there was an interactive component C that recognized J. Let E input 1's. By interactiveness C must generate a non-empty signal σ at some moment in time. E can now fool C as follows. If $\sigma = 0$, then let E switch to inputting 0's from this moment onward: it means that the resulting input belongs to J but C does not respond with all 1's. If $\sigma = 1$, then let E continue to input 1's. Possibly C outputs a few more 1's but there must come a moment that it outputs a 0. If it didn't then C would recognize the string $1^\omega \notin J$. As soon as C outputs a 0, let E switch to inputting 0's from this moment onward: it means that the resulting input still belongs to J but C does not recognize it properly. Contradiction.
(iv) Suppose there was an interactive component C that recognized J. Let E input 0's. Now argue as in the preceding case.
(v) Suppose there was an interactive component C that recognized J, the set of infinite strings with precisely k 1's. Let E input $k-1$ 1's followed by all 0's for a while from then onward. By interactiveness C must generate a non-empty signal σ at some moment in time. E can now fool C as follows. If $\sigma = 0$, then let E send a 1 followed by all 0's from then onward: the input string clearly belongs to J but isn't recognized properly by C. If $\sigma = 1$, then let E continue to send 0's. Possibly C outputs a few more 1's but there must come a moment that it outputs a 0. If it didn't then C would recognize the string $1^{k-1}0^\omega \notin J$. As soon as C outputs a 0, let E switch to inputting a 1 followed by all 0's from then onward: the input string again clearly belongs to J but isn't recognized properly by C.
(vi) Analogous to *(v)*. With $k = 1$ this example was shown not to be 1'-definable in [2], lemma 7.17 (b).

The proof of lemma 1 is based the following underlying fact: if $J \subseteq \{0,1\}^\omega$ contains $\alpha 0^\star 1 M$ as a sublanguage for some nonempty set $M \subseteq \{0,1\}^\omega$ but does *not* contain $\alpha 0^\omega$, for some finite $\alpha \in \{0,1\}^\star$ then J is not interactively recognizable. We will return to this observation in a different context in example 2.

The power of interactive recognition is expressed in the following observations. We assume again that the internal operation of the components that we consider is specified by some program in an acceptable programming system and that the components operate in an unconstrained environment.

Theorem 4. $J = \{0^n 1 \{0,1\}^\omega | n \in A\} \cup 0^\omega$ *is interactively recognizable if and only if* A *is the complement of a recursively enumerable set.*

Proof. Let J be of the given form and let C interactively recognize the strings of J. Observe that C must have the following behavior: if E has sent input $0^n 1$ at some point in time, then $n \in A$ if and only C recognizes the string no matter what further input signals follow. Likewise $n \notin A$ if and only if C does *not* recognize the string, no matter what further signals follow. Let π be the program of C.

Now recursively enumerate the complement of A as follows. Enumerate the integers n and for every n simulate π on the input $0^n 1$ from the environment, using any extension of the string when these inputs are called for by π (noting here that by assumption E is able to generate any such extension, regardless of the course of the computation and its interactive experiences). After the simulation of C has received the complete 0^n1 as input, output n if, or otherwise as soon as, the simulation of C has led to an output symbol 0. The latter happens only for the elements of the complement of A.

Conversely, let A be the complement of a recursively enumerable set. Let π be the program enumerating A's complement $\overline{A}$. Design a component C that operates as follows. If the first symbol that it receives is a 1, then C outputs 0's forever. (The case $n = 0$ cannot occur as we are only considering subsets of N.) If the first symbol that it receives is a 0, then C outputs 1's until it receives a first 1. If no 1 is ever received, it effectively means that C recognizes 0^ω. If C does receive a 1, let $n \geq 0$ be the number of 0's that it has received until this first 1. Now C switches to the program π that enumerates $\overline{A}$. C continues to output 1's while it is running π, until it encounters n in the enumeration. If n is encountered, C stops running π and starts outputting 0's instead. Clearly C recognizes $0^n1\ldots$ precisely if $n \notin \overline{A}$, i.e. if $n \in A$. This shows that C recognizes J.

For sets $J \subseteq \{0,1\}^\omega$, let $Init(J)$ be the set of all finite prefixes of strings from J. Theorem 4 leads to an important observation which shows that in interactive computation the initial parts of an environment input do not necessarily hold any clue about the recognizability of the input 'in the limit', as one would expect. The result parallels the one for 1'-definable ω-Turing machine languages by Cohen and Gold [2], theorem 7.22.

Corollary 2. *There are interactively recognizable sets J such that Init(J) is not recursively enumerable.*

Proof. Consider the set $J = \{0^n1\{0,1\}^\omega | n \in A\} \cup 0^\omega$ for an arbitrary non-recursively enumerable set A whose complement is recursively enumerable (cf. Rogers [17]). By theorem 4, J is interactively recognizable. Note that $Init(J) \cap 0^\star 1 = \{0^n1 | n \in A\}$. Hence, if $Init(J)$ were recursively enumerable, then so would A be. Contradiction.

A further characterisation of interactive recognizability is implied by the following result. For 1'-definable ω-Turing machine languages the fact was shown Cohen and Gold [2], theorem 7.16 (b).

Theorem 5. *$J \subseteq \{0,1\}^\omega$ is interactively recognizable if and only if there exists a recursively enumerable language $A \subseteq \{0,1\}^\star$ such that $J = \{u \in \{0,1\}^\omega | u$ has no prefix in A$\}$.*

Proof. Let $J \subseteq \{0,1\}^\omega$ be interactively recognizable, and C a component that interactively recognizes J. Let A consist of all strings $\alpha \in \{0,1\}^\star$ that lead C to output a 0, after E has interactively fed it α and C has output only τ's and 1's so far. (Thus, α leads C to output its first 0.) By simulating and dovetailing the interactive computations between E and C on all possible finite input segments, A is seen to be recursively enumerable (using our assumptions). J precisely consist of all strings $\in \{0,1\}^\omega$ that do not begin with a string in A.

Conversely, let $A \subseteq \{0,1\}^\star$ be recursively enumerable and J as defined. Design a component C that operates as follows. As soon as C receives input, it starts buffering the input in a queue q. At the same time it starts the recursive enumeration of A and it starts outputting 1's. Every time the enumeration of A outputs a string α, C adds it to a list L_A. Every once in a while, C checks whether any prefix of the current q happens to occur in L_A. If this is the case, C stops the enumeration and switches to outputting 0's from here onwards. Otherwise C continues with the procedure, and keeps on outputting 1's. Clearly C is interactive, and C recognizes precisely the set J.

The given characterisation together with Cohen and Gold's result show that, with unconstrained environments and recursive C's, *interactive recognizability and 1'-definability essentially coincide.* Theorem 5 has another consequence when we view $\{0,1\}^\omega$ as a topological space with the usual product or Cantor topology (cf. [9]). The following result was observed by Landweber [9] (cor. 3.2) for 1'-definable ω-regular languages and Staiger (cf. [19]) for 1'-definable (deterministic) ω-Turing machine languages. Recall that an open set $L \subseteq \{0,1\}^\omega$ is said to have a (minimal) basis $B \subseteq \{0,1\}^\star$ if $L = B\{0,1\}^\omega$ (and B is prefix-free).

Corollary 3. *$J \subseteq \{0,1\}^\omega$ is interactively recognizable if and only if J is closed and $\overline{J}$ has a recursively enumerable basis.*

Finally we note some rules for constructing new interactively recognizable sets from old ones. Again a similar result exists for 1'-definable ω-Turing machine languages, see [2], theorem 7.20 (a). The proofs here are tedious because the outputs of a component may feed back to E.

Lemma 2. *The family of interactively recognizable sets of infinite strings is*
(i) closed under $\cup$ and $\cap$, but
(ii) not *closed under ω-complement (i.e. complement in $\{0,1\}^\omega$).*

Proof. (i) We only prove closure under $\cup$, leaving the similar argument for closure under $\cap$ to the reader. Let J_1 and J_2 be interactively recognized by components C_1 and C_2, respectively. A component C recognizing $J_1 \cup J_2$ is obtained as follows. C buffers the input that it receives from E in a queue q, symbol after symbol. In conjunction with this, C simulates the programs of both C_1 and C_2 simultaneously, simulating the input from E by the consecutive symbols from q. C keeps C_1 in the foreground and outputs what C_1 outputs *until* the environment input (which can be influenced by C_1's output) is about to be inconsistent with q or C_1 is about to output a 0 for the first time. (C outputs finitely many extra τ's along the way to account for the simulation overhead). If the simulation never reaches a point where this occurs, then C works completely like C_1 all the way and recognizes the input as an element of J_1. Every element of J_1 can be recognized this way.

If the simulation does reach a point where one of the two situations occurs, then C tries to switch to C_2. In case the environment input was about to become inconsistent with q (due to C_1's output and E's response to it), C checks whether the environment input in the simulation of C_2 is still consistent with q. If it is, it subsequently checks whether C_2 (running in the background) has output a 0 in the simulation so far. If not, C switches to the simulation of C_2, otherwise it switches to outputting 0's from this moment onward, effectively rejecting the whole input string. In case C interrupted the simulation because C_1 was about to output a 0 for the first time, then C does *not* output the 0 but makes a similar check as described before, to see if it can bring the simulation of C_2 to the foreground and switch. If the simulation switches successfully to C_2, then the same constraints continue to be observed. Clearly, if no further exception is reached, then C works completely like C_2 all the way and recognizes the input as an element of J_2. Note that every element of $J_2 \backslash J_1$ can be recognized this way.

It is easily seen that C is interactive. Note also that, when C switches from C_1 to C_2 as described, both C_1 and C_2 must have been outputting τ's and 1's until this point and thus, when the simulation of C_2 takes over, it is like C_2 has been running from the beginning as far as the recognition is concerned. C recognizes precisely the set $J_1 \cup J_2$.
(ii) Consider the set $J = 0^\omega \cup 0^\star 10^\omega$. By lemma 1 (i) the set is interactively recognizable, but lemma 1 (vi) shows that its ω-complement is not.

The given results shed some light on the power of interactive recognition, and expose the connections to the theory of ω-automata. The results follow with minimal assumptions on the internal structure of components, in particular we need no concrete Turing machine inside.

5 Interactive Generation

Embedded components typically also perform tasks in *controlling* other components. This involves the online translation of infinite streams into other, more specific streams of signals. In this section we consider what infinite streams of signals an interactive component can *generate*. The notion of generation is well-known in automata theory and related to matters of definability and expressibility, but it seems not to have been looked at very much in the theory of ω-automata (cf. Staiger [19]). Our aim will be to prove that generation and recognition are *duals* in our model. The following definition is intuitive.

Definition 8. *An infinite string $\beta \in \{0,1\}^\omega$ is said to be generated by C if there exists an environment input $\alpha \in \{0,1\}^\omega$ such that $(\alpha, \beta) \in T_C$.*

Unlike the case for recognition (cf. section 4) one cannot simplify the output capabilities for components C now. In particular one has to allow that C outputs $\sharp$-symbols, for example when it wants to signify that its generation process has gotten off on some wrong track. If C outputs a $\sharp$-symbol, it will automatically trigger E to produce a $\sharp$ by by assumption some finite time later and thus invalidate the current run. Note that strings that contain a $\sharp$ are *not* considered to be validly generated.

Definition 9. *The set interactively generated by component C is the set $L_C = \{\beta \in \{0,1\}^\omega | \beta$ is generated by C $\}$.*

Formally, the definition should constrain the strings β to those strings that can be generated *using allowable inputs α from E only.* Observe that, contrary to recognition, C may need to make silent steps while generating. It means that interactive generation is not necessarily an online process. Nevertheless, if C satisfies the interactiveness condition, the generation process will output non-empty signals with finite delay only.

Definition 10. *A set $L \subseteq \{0,1\}^\omega$ is called* interactively generable *if there exists an interactive component C such that $L = L_C$.*

In the context of generating ω-strings, it is of interest to know what finite *prefixes* an interactive component C can generate. To this end we consider the following problem:

> (*Reachability*) Given an interactive component C and a finite string $\gamma \in \{0,1\}^\star$, is there an interactive computation of C such that the sequence of non-silent symbols generated and output by C at some finite moment equals γ.

Lemma 3. *The reachability problem for interactive components C is effectively decidable.*

Proof. Let C and γ be given. Consider the (infinite) binary tree $\mathcal{T}$ with left branches labeled 0 and right branches labeled 1. Every node q of $\mathcal{T}$ represents a finite input of E, namely the string α_q of 0's and 1's leading from the root of $\mathcal{T}$ to q, and every finite input that E can provide is so represented. Label/prune $\mathcal{T}$ as follows. Label the root by 'n'. Work through the unlabeled nodes q level by level down the tree and simulate C while E supplies α_q as input to C, halting the simulation when E reaches the end of α_q or when E wants to deviate from giving the next symbol of α_q as input based on C's response. Then do the following:

- label q by 'Y' and prune the tree below it if the simulation at q leads C to output a string r such that γ is a prefix of $\overline{r}$,
- label q by 'N' and prune the tree below it if the simulation at q leads C to output a string r of which γ is not a prefix (which certainly can be decided as soon as $|\overline{r}| \geq |\gamma|$),
- label q by 'N' and prune the tree below it if the simulation halts before E could input all of α_q, and
- just label q by 'n' otherwise (and thus the subtree at q is *Not* pruned yet in this case).

Denote the pruned tree by $\overline{\mathcal{T}}$. Clearly the reachability problem is equivalent to the problem of deciding whether there exists a Y-labeled node in $\overline{\mathcal{T}}$.

We claim that $\overline{\mathcal{T}}$ is finite and, hence, that the algorithm terminates in finitely many steps. Suppose $\overline{\mathcal{T}}$ was infinite. By *König's Unendlichkeitslemma* ([6,7]) it follows that in this case $\overline{\mathcal{T}}$ must contain an infinite path from the root down. But by interactiveness the simulations of C along this path must eventually either halt or lead to output strings r with $|\overline{r}|$ exceeding any fixed bound. This means that some node on the path must lead the algorithm to prune the tree below it, contradicting the fact that the remainder of the path is still in $\overline{\mathcal{T}}$.

Because $\overline{\mathcal{T}}$ is finite, it can be decided in finite time whether there exists a Y-labeled node in it and thus whether γ can be obtained as a finite output of C.

We will now show that the fundamental law that 'what can be generated can be recognized and vice versa' holds in our model of interactive computing. We prove it in two steps.

Lemma 4. *For all sets $J \subseteq \{0,1\}^\omega$, if J is interactively generable then J is interactively recognizable.*

Proof. Let J be interactively generated by means of some component C, i.e. $J = L_C$. To show that J can be interactively recognized, design the following component C'. Let the input from E be β. C' buffers the input that it receives from E symbol after symbol, and goes through the following cycle of activity: it takes the string γ that is currently in the buffer, decides whether γ is reachable for C by applying the procedure of lemma 3, and outputs a 1 if it is and a 0 if it is not. This cycle is repeated forever, each time taking the new (and longer) string γ that is in the buffer whenever a new cycle is executed.

Because the reachability problem is decidable in finite time, C' is an interactive component. Clearly, if an ω-string β belongs to J then all its prefixes are reachable for C, and C' recognizes it. Conversely, if an ω-string β is recognized by C' then it must be interactively generated by C and hence belong to J. We argue this point somewhat more precisely, referring to the construction in lemma 3.

Suppose that β is recognized by C'. Take a new instance $\mathcal{S}$ of the infinite binary tree and label its nodes as follows. Every time C' carries out its cycle on a next string γ (a longer prefix of β) it runs the labeling/pruning algorithm of lemma 3 on a copy of $\mathcal{T}$ to completion and identifies one or more nodes that are to be labeled Y. (This follows because, by assumption, C' identifies every prefix that it checks as reachable.) Copy the labels 'Y' to the corresponding nodes of $\mathcal{S}$. Do not label a node again if it was already labeled at an earlier stage. This process will lead to infinitely many Y-labeled nodes in $\mathcal{S}$, because the prefixes of β that C' checks and finds reachable have a length going to infinity (and this leads to Y-labeled nodes lower and lower in the tree even though some overlaps may occur). By *König's Unendlichkeitslemma*, $\mathcal{S}$ must contain an infinite path from the root down with the property that every node on the path has a Y-labeled node as descendent. Let $alpha \in \{0,1\}^\omega$ be the infinite string corresponding to this path. It follows from the definition of the Y-label that C transduces α to β and hence that $\beta \in J$.

Lemma 5. *For all sets $J \subseteq \{0,1\}^\omega$, if J is interactively recognizable then J is interactively generable.*

Proof. Let J be interactively recognizable. Let C be an interactive component such that $J = J_C$. To show that J can be interactively generated, design the following component C'. C' buffers the input that receives from E symbol after symbol, and copies it to output as well (at a slower pace perhaps). At the same time C' runs a simulation of C in the background, inputting the symbols from the buffer to C one after the other as if they were directly input from E. By algorithmicity it can be checked alongside of this whether the input is indeed a sequence that E could input when taking the responses of C into account.

Let C' continue to copy input to output as long as (a) no inconsistency between the buffered input and the verification of E arises and (b) the simulation of C outputs only τ's and 1's. If anything else occurs, C' switches to outputting $\sharp$'s. C' is clearly interactive, and the generated strings are precisely those that C recognizes.

The lemmas lead to the following basic result.

Theorem 6. *For all sets $J \subseteq \{0,1\}^\omega$, J is interactively generable if and only if J is interactively recognizable.*

The theorem gives evidence that the concepts of interactive recognition and generation as intuitively defined, are well-chosen. Besides being of interest in its own right, theorem 6 also enables one to prove that certain sets are not interactively recognizable by techniques different from Section 4.

Example 2. Consider the set $J = 0^{\star}1 \cdot M$ with $M \subseteq \{0,1\}^{\omega}$ an arbitrary nonempty (countable or uncountable) set of infinite binary strings. We show that J is not interactively recognizable, by showing that it is not interactively *generable*. Suppose J could be generated by some interactive component C. Consider the infinite binary tree $\mathcal{T}$ as defined in lemma 3. The paths from the root down to the individual nodes q represent the prefixes of the potential infinite inputs of E to C. Label the root by 'n'. Working through the nodes q below it level after level, consider the interaction of E and C, with E supplying input α_q symbol after symbol. As soon as C outputs a 1, label q with 'Y' and prune the tree below it. (Note that this will automatically limit the Y-labeling step to the cases where C outputs a 1 for the first time and precisely after inputting all of the string α_q.) If E has input α_q and C has not given any nonempty output except perhaps some 0's, then label q by 'n' but do not prune the tree below it. If E cannot complete the input segment because its behavior leads it to generate a symbol different from what the α_q prescribes at some time, then label q by N and prune the tree below it. Let $\overline{\mathcal{T}}$ denote the resulting labeled and pruned tree.

We claim that $\overline{\mathcal{T}}$ is finite. Suppose $\overline{\mathcal{T}}$ was infinite. Then here is an infinite path leading from the root down, necessarily consisting only of nodes that are labeled 'n'. The infinite binary sequence corresponding to this path must be a viable input of E to C (even taking the interaction into account), and must lead to output 0^{ω}. This string is not a member of J, a contradiction. Thus $\overline{\mathcal{T}}$ is finite.

Considering the leaves of $\overline{\mathcal{T}}$, let l be the length of the longest sequence of 0's in any generated output of the form $0 \ldots 01$ as recorded in any Y-labeled node. Such an l exists. It follows from the construction that $0^{l+1}1\beta$ can*not* be generated by C, for any $\beta \in M$, contradicting the fact that all these string are in J. Thus J is not interactively generable.

6 Interactive Translations

As an additional task, embedded components typically perform the online translation of infinite streams into other infinite streams of signals. We consider this in more detail, viewing components as interactive transducers and viewing the translations (or: transductions) they realize as interactive mappings defined on infinite strings of 0's and 1's. The related notion of ω-transduction in the theory of ω-automata has received quite some attention before (cf. Staiger [19]), but mostly in the context of ω-regular languages only. In this section we present some basic observations on interactive mappings. Let C be an interactive component. Let T_C be the behavior of C (cf. definition 2).

Definition 11. *The interactive mapping computed by C is the mapping $f_C : \{0,1\}^{\omega} \to \{0,1\}^{\omega}$ defined by the following property: $f_C(\alpha) = \beta$ if and only if $(\alpha, \beta) \in T_C$.*

Generally speaking, an interactive mapping is a partial function defined over infinite binary strings. If $f_C(\alpha) = \beta$ (defined), then in response to input α, C actually outputs a sequence $r \in \{0,1,\tau\}^{\omega}$ such that $\overline{r} = \beta$.

Definition 12. *A partial function $f : \{0,1\}^{\omega} \to \{0,1\}^{\omega}$ is called interactively computable if there exists an interactive component C such that $f = f_C$.*

Computable functions on infinite strings should be continuous in the sense that, anytime after some finite prefix has been input, any further extension of the input should only lead to an extension of the output generated so far and vice versa, without retraction of any earlier output signals. Interactively computable functions clearly all have this property on defined values, which can be more precisely formulated as follows. We paraphrase the classical definition of continuous functions (cf. [34]) for the case of functions on infinite strings.

Definition 13. *A partial function $f : \{0,1\}^{\omega} \to \{0,1\}^{\omega}$ is called limit-continuous if there exists a classically computable partial function $g : \{0,1\}^{\star} \to \{0,1\}^{\star}$ such that the following conditions are satisfied: (1) g is monotonic, and (2) for all strictly increasing chains $u_1 \prec u_2 \prec \cdots \prec u_t \prec \ldots$ with $u_t \in \{0,1\}^{\star}$ for $t \geq 1$, one has $f(\lim_{t\to\infty} u_t) = \lim_{t\to\infty} g(u_t)$.*

In condition (2) the identity is assumed to hold as soon as one of the sides is defined. Note that for all strictly increasing chains $u_1 \prec u_2 \prec \cdots \prec u_t \prec \ldots$ of binary strings, $\lim_{t\to\infty} u_t$ always exists as an infinite string in $\{0,1\}^\omega$.

Clearly, monotonic functions map chains into chains, if they are defined on all elements of a chain. In general, monotonic functions do not necessarily map strictly increasing chains into chains that are strictly increasing again. Definition 13 implies however that if a total function f is limit-continuous, then the underlying g must be total as well and map strictly increasing chains into ultimately increasing chains. (In the terminology of [19] sect 2.2, g is 'totally unbounded'.) Using theorem 1 and 2, one easily concludes the following facts.

Theorem 7. *If $f : \{0,1\}^\omega \to \{0,1\}^\omega$ is interactively computable, then f is limit-continuous.*

Theorem 8. *Let $f : \{0,1\}^\omega \to \{0,1\}^\omega$ be a total function. Then f is interactively computable if and only if f is limit-continuous.*

Several other properties of interactively computable functions are of interest. The following observation is elementary but is spelled out in some detail so as to show how the assumptions in our model of interactive computing play a role and how only generic properties of the internal functioning of components are needed. In the following results we do not assume any of the interactively computable functions to be total. Let $\circ$ denote composition of functions.

Theorem 9. *If f and g are interactively computable, then so is $f \circ g$.*

Proof. Let $f = f_{C'}$ and $g = f_C$. To show that $f \circ g$ is interactively computable, design a component C'' that works as follows. C'' runs a foreground process that works exactly like C. On top of that it runs a verifier that observes the incoming symbols and the output of C and verifies that the input is consistent with the behavior E would or could have (which can be done by algorithmicity). Note that this is necessary, because the output of the foreground process is not visible to E directly, and we have to make sure that the interaction between E and C is simulated correctly. If the verifier ever observes an inconsistency, C'' immediately stops the foreground process and outputs $\sharp$'s from this moment onward.

The foreground process feeds its output into an internal buffer $\mathcal{B}$, which only records the non-τ symbols. C'' runs a background process that takes its inputs from $\mathcal{B}$ and simulates the operation of C' just like the foreground process did with C. In particular it (also) runs the verifier to see that the input taken from $\mathcal{B}$ is consistent with the behavior of E, including its response to the output of C' (which can be done by algorithmicity of E again). The background process cannot make steps with every time-tick like C' would. Instead it has to follow/operate on the time-ticks defined by the appearance of symbols in $\mathcal{B}$, to adequately simulate the environmental activity and keep the same timing relationships between E's input and the action of C'. The output of the background process, i.e. of the simulation of C', is the output of C''.

It is easily verified that C'' must be interactive. Whenever an inconsistency in the simulated actions of C and C' is discovered, a $\sharp$ is generated and fed into the further simulation and thus eventually to output. Note that the whole process is triggered by the input from E to C'', i.e. to the simulation of C and only this input has a variable aspect. Internally everything runs deterministically (aside from any unpredictable time-delays). It is easily seen that C'' correctly computes the value of $f \circ g$ on the input stream supplied by E.

The following result is more tedious and relies on the machinery which we developed in the previous section. The result shows that the inverses of 1-1 mappings defined by interactive components are interactively computable again.

Theorem 10. *Let f be interactively computable and $1-1$. Then f^{-1} is interactively computable as well.*

Proof. Let $f = f_C$ and assume f is $1-1$. If $f(\alpha) = \beta$ (defined) then $f^{-1}(\beta) = \alpha$. Design a component C' to realize the mapping of β's into α's as follows.

Let the input supplied so far be γ, a finite prefix of 'β'. Assume the environment supplies further input symbols in its own way, reveiling to C' the longer and longer prefixes γ of the β to which an original under f is sought. Let C' buffer γ internally. We want the output σ of C' at any point to be a finite (and growing) prefix of 'α' (ignoring any τ's in σ). Let this be the case at some point. Let C' do the following, as more and more symbols are coming in and reveal more and more of β and outputting τ's until it knows better.

The dilemma C' faces is whether to output a 0 or a 1 (or, of course, a $\sharp$). In other words, C' must somehow decide whether $\sigma 0$ or $\sigma 1$ is the next longer prefix of the original α under f as β is unfolding. We argue that this is indeed decidable in finite time. The idea is to look 'into the future' and see which of the two possibilities survives. To achieve it, create a process P_b that explores the future for σb, for every $b \in \{0, 1\}$. Remember that symbols continue to come into γ.

P_b works on the infinite binary tree $\mathcal{T}$ defined in lemma 3. Remember that every node q of $\mathcal{T}$ corresponds to a finite string α_q, consisting of the 0's and 1's on the path from the root down to q. P_b labels the root by 'Y'. Then it works through the unlabeled nodes q level by level down the tree, testing for every node q whether the string $\sigma b \alpha_q$ is output (i.e. is reached) by C *as it operates on (a prefix of) the string* γ, i.e. on a prefix of β. (P_b does this in the usual way, by running the simulation of the interactive computation of C and E and using the algorithmicity of E to properly test for the corresponding behavior of E on C's output.) If $\sigma b \alpha_q$ is reached, then label q by 'Y'. If the output of C does not reach the end of $\sigma b \alpha_q$ but is consistent with it as far as it gets, then P_b waits (at q) and only continues the simulation when more symbols have come into γ. (By interactivity, γ will eventually be long enough for C to give an output at least as long as $\sigma b \alpha_q$.) If the output of C begins to differ from $\sigma b \alpha_q$ before the end of $\sigma b \alpha_q$ is reached, then label q by 'N' and prune the tree below it. If the simulation runs into an inconsistency between E's behavior and the γ that is input, then label q by 'N' and prune the tree below it as well. If P_b reaches a tree level where all nodes have been pruned away, it stops. Denote the tree as it gets labeled by P_b by $\mathcal{T}_b$.

Let C' run P_0 and P_1 'in parallel'. We claim that one of the two processes must stop in finite time. Suppose that neither of the two stopped in finite time. Then $\mathcal{T}_0$ and $\mathcal{T}_1$ would both turn into infinite trees as γ extends to 'infinity' (i.e. turns into the infinite string β). By the *Unendlichkeitslemma*, $\mathcal{T}_0$ will contain an infinite path δ_0 and likewise $\mathcal{T}_1$ will contain an infinite path δ_1. This clearly implies that both $\sigma 0 \delta_0$ and $\sigma 1 \delta_1$ would be mapped by C to β, which contradicts that f is $1-1$. It follows that at least one of the processes P_0 and P_1 must stop in finite time. (Stated in another way, the process that explores the wrong prefix of α will die out in finite time.) Note that both processes could stop, which happens at some point in case the limit string β has *no* original under f.

Thus letting C' run P_0 and P_1 in parallel, do the following as soon as one of the processes stops. If both processes stop, C' outputs $\sharp$. If P_0 stopped but P_1 did not, then output 1. If P_1 stopped but P_0 did not, then output 0. If C' output a b (0 or 1), then it repeats the whole procedure with σ replaced by σb. If it output a $\sharp$ it continues to output $\sharp$'s from this moment onwards and does not repeat the above procedure anymore. It is easily seen that C' is interactive and that it computes precisely the inverse of f.

7 Conclusion

Interactive computing and its formal study has received much attention since the late nineteen sixties, usually within the framework of reactive systems. In this paper we considered a simple model of interactive computing in embedded systems, consisting of one component and an 'environment' acting together on infinite streams of input and output symbols that are exchanged in an online manner. The motivation stems from the renewed interest for the computation-theoretic capabilities of interactive computing, in particular by Wegner's claim (cf. [29]) that 'interactive computing is more powerful than algorithms'.

In the model we have tried to identify a number of properties which one would intuitively ascribe to a component of an embedded system that interacts with the environment in which it is placed. In [25] we have carried this further, to model some interactive features of the Internet and of 'global computing'. In the latter case, the model includes the possibility of letting external information enter into the interaction process and of many components influencing each other. In the present report we have concentrated purely on the property of 'interactiveness' for a single component, implying that both the component and its environment always react within some (unspecified) finite time. As components operate on infinite inputs, there are various connections to the classical theory of ω-automata.

We have given definitions of interactive recognition, generation and translation that are inspired by realistic considerations of how the various tasks would proceed in an interactive setting. The definition of interactive recognition leads to a useful, machine-independent analogue of the notion of 1'-definability as known for ω-automata. The definitions allow a proof that interactive recognition and interactive generation are equally powerful in the given model of interactive computation. We also proved that (total) functions are interactively computable if and only they are limit-continuous, using a simple extension of the common definitions of continuity. Among the further results we showed that interactively computable (partial) functions that are $1-1$ have interactively computable inverses. Many interesting problems seem to remain in the further analysis of the model.

In the paper we do not comment on Wegner's claims. We have merely attempted to identify the power of interactive computation in a simple model, following the style of classical computability theory. We hope the analysis of the model adds to the understanding of interactive computing and the possible implications for complexity theory. The further analysis of the model quickly leads to the consideration of non-uniformly evolving, interactive machines and programs. The prospects of a theory that takes this into account are sketched in [25].

Acknowledgements
We are grateful to D.Q. Goldin and T. Muntean for several useful comments and suggestions. Preliminary versions of the ideas in this report were included in presentations at IFIP-TCS'2000 [23] and MFCS'2000 [24]. We thank J-E. Pin for useful discussions on the theory of ω-automata.

References

1. M. Broy. A logical basis for modular software and systems engineering, in: B. Rovan (Ed.), *SOFSEM'98: Theory and Practice of Informatics*, Proc. 25th Conference on Current Trends, Lecture Notes in Computer Science, Vol. 1521, Springer-Verlag, Berlin, 1998, pp. 19-35.
2. R.S. Cohen, A.Y. Gold. ω-Computations on Turing machines, *Theor. Comput. Sci.* 6 (1978) 1-23.
3. J. Engelfriet, H.J. Hoogeboom. X-automata on ω-words, *Theor. Comput. Sci.* 110 (1993) 1-51.
4. D.Q. Goldin. Persistent Turing machines as a model of interactive computation, in: K-D. Schewe and B. Thalheim (Eds.), *Foundations of Information and Knowledge Systems*, Proc. First Int. Symposium (FoIKS 2000), Lecture Notes in Computer Science, vol. 1762, Springer-Verlag, Berlin, 2000, pp. 116-135.
5. D. Goldin, P. Wegner. Persistence as a form of interaction, Techn. Rep. CS-98-07, Dept. of Computer Science, Brown University, Providence, RI, 1998.
6. D. König. Sur les correspondances multivoques des ensembles, *Fundam. Math.* 8 (1926) 114-134.
7. D. König. Über eine Schlussweise aus dem endlichen ins Unendliche (Punktmengen. – Kartenfärben. — Verwantschaftsbeziehungen. – Schachspiel), Acta Litt. Sci. (Sectio Sci. Math.) 3 (1927) 121-130.
8. S. Kosub. Persistent computations, Technical Report No. 217, Institut für Informatik, Julius-Maximilians-Universität Würzburg, 1998.
9. L.H. Landweber. Decision problems for ω-automata, *Math. Systems Theory* 3 (1969) 376-384.
10. Z. Manna, A. Pnueli. Models for reactivity, *Acta Informatica* 30 (1993) 609-678.
11. R. Milner. *A calculus of communicating systems*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin, 1980.
12. R. Milner. Elements of interaction, *C.ACM* 36:1 (1993) 78-89.
13. P. Odifreddi. *Classical recursion theory – The theory of functions and sets of natural numbers*, North-Holland/Elsevier Science Publishers, Amsterdam, 1989.

14. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends, in: J.W. de Bakker, W.-P. de Roever and G. Rozenberg, *Current Trends in Concurrency*, Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, 1986, pp. 510-585.
15. A. Pnueli. Specification and development of reactive systems, in: H.-J. Kugler (Ed.), *Information Processing 86*, Proceedings IFIP 10th World Computer Congress, Elsevier Science Publishers (North-Holland), Amsterdam, 1986, pp. 845-858.
16. M. Prasse, P. Rittgen. Why Church's Thesis still holds. Some notes on Peter Wegner's tracts on interaction and computability, *The Computer Journal* 41 (1998) 357-362.
17. H. Rogers. *Theory of recursive functions and effective computability*, McGraw-Hill, New York, 1967.
18. G. Rozenberg, F.W. Vaandrager (Eds.). *Lectures on embedded systems*, Lecture Notes in Computer Science, Vol. 1494, Springer-Verlag, Berlin, 1998.
19. L. Staiger. ω-Languages, in: G. Rozenberg and A. Salomaa (Eds.), *Handbook of Formal Languages*, Vol. 3: *Beyond Words*, Chapter 6, Springer-Verlag, Berlin, 1997, pp. 339-387.
20. W. Thomas. Automata on infinite objects, in: J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. B: *Models and Semantics*, Elsevier Science Publishers, Amsterdam, 1990, pp. 135-191.
21. W. Thomas. Languages, automata, and logic, in: G. Rozenberg and A. Salomaa (Eds.), *Handbook of Formal Languages*, Vol. 3: *Beyond Words*, Chapter 7, Springer-Verlag, Berlin, 1997, pp. 389-455.
22. B.A. Trakhtenbrot. Automata and their interaction: definitional suggestions, in: G. Ciobanu and G. Păun (Eds.), *Fundamentals of Computation Theory*, Proc. 12th International Symposium (FCT'99), Lecture Notes in Computer Science, Vol. 1684, Springer-Verlag, Berlin, 1999, pp. 54-89.
23. J. van Leeuwen, J. Wiedermann. On the power of interactive computing, in: J.van Leeuwen et al (Eds), *Theoretical Computer Science - Exploring New Frontiers of Theoretical Computer Science*, Proc. IFIP TCS'2000 Conference, Lecture Notes in Computer Science Vol. 1872, Springer-Verlag, Berlin, 2000, pp 619-623.
24. J. van Leeuwen, J. Wiedermann. On algorithms and interaction, in: M. Nielsen and B. Rovan (Eds), *Mathematical Foundations of Computer Science 2000*, 25th Int. Symposium (MFCS'2000), Lecture Notes in Computer Science Vol. 1893, Springer-Verlag, Berlin, 2000, pp. 99-112.
25. J. van Leeuwen, J. Wiedermann. The Turing machine paradigm in contemporary computing, Techn. Report UU-CS-2000-33, Dept of Computer Science, Utrecht University (2000), also in: B. Enquist and W. Schmidt (Eds), Mathematics Unlimited - 2001 and Beyond, Springer-Verlag, 2001 (to appear).
26. K. Wagner, L. Staiger. Recursive ω-languages, in: M. Karpinsky (Ed.), *Fundamentals of Computation Theory*, Proc. 1977 Int. FCT-Conference, Lecture Notes in Computer Science, Vol. 56, Springer-Verlag, Berlin, 1977, pp. 532-537.
27. P. Wegner. Interactive foundations of object-based programming, *IEEE Computer* 28:10 (1995) 70-72.
28. P. Wegner. Interaction as a basis for empirical computer science, *Comput. Surv.* 27 (1995) 45-48.
29. P. Wegner. Why interaction is more powerful than algorithms, *C.ACM* 40 (1997) 80-91.
30. P. Wegner. Interactive foundations of computing, *Theor. Comp. Sci.* 192 (1998) 315-351.
31. P. Wegner, D. Goldin. Co-inductive models of finite computing agents, in: B. Jacobs and J. Rutten (Eds.), *CMCS'99-Coalgebraic Methods in Computer Science*, TCS: Electronic Notes in Theoretical Computer Science, Vol. 19, Elsevier, 1999.
32. P. Wegner, D. Goldin. Interaction as a framework for modeling, in: P. Chen *et al.* (Eds.), *Conceptual Modeling - Current Issues and Future Directions*, Lecture Notes in Computer Science, Vol. 1565, Springer-Verlag, Berlin, 1999, pp 243-257.
33. P. Wegner, D.Q. Goldin. Interaction, computability, and Church's thesis, *The Computer Journal* 1999 (to appear).
34. G. Winskel. *The formal semantics of programming languages: an introduction*, The MIT Press, Cambridge (Mass.), 1993.