Synthesizing Processes and Schedulers from Temporal Specifications*

Howard Wong-Toi and David L. Dill
Department of Computer Science
Stanford University

Abstract

We examine two closely related problems: synthesizing processes to satisfy temporal specifications of reactive systems, and the synthesis of a scheduler to interact with and control a group of processes in order to meet a specification. Processes communicate through shared and distributed variables, either synchronously or asynchronously. In the finite state case, processes and specifications are arbitrary ω-regular languages, and both synthesis problems are solvable in doubly exponential time and space. The framework we present is flexible enough to incorporate dense real time into the model of concurrency, thereby allowing us to study the synthesis of real-time processes and schedulers. Real-time implementability and scheduling are also doubly exponential.

1 Introduction

A specification for a reactive process defines the desired ongoing behavior of the process along with its environment (the other processes with which it is connected). If the implementer of a process does not have control over the behavior of its environment, it may not be possible to meet the specification. For example, if a specification says that "x is always 1", and the environment can assign x the value 2, the specification is not implementable — the process can set x to 1, but cannot prevent it from taking other values.

The fundamental problem is to derive an implementation that always satisfies the specification in the face of adversarial behavior by the environment. Intuitively, an implementation is a winning strategy in the game against the environment. When the specification can be modeled as a finite automaton on infinite words, or a formula from a logic interpreted over infinite words, the synthesis problem is decidable. The decision procedures consist of reductions to the Church *solvability problem*, [Chu63, BL69] which can be solved using automata on infinite trees [Rab72].

Our processes have three different types of variables: shared, read-only and "distributed" (variables that can be read by other processes but written only by one). We solve the problem of process synthesis under both synchronous and asynchronous paral-

*Supported by NSF under grant MIP-8858807.

lel execution[1]. These results extend, unify, and simplify previous work [PR89a, PR89b, ALW89]. Next, we solve the problem of synthesizing a *scheduler*, which controls a collection of processes so as to satisfy a specification. For example, a critical region between processors could be enforced by a scheduling strategy instead of explicit synchronization within the processes. We also discuss ways to solve the same problem when there are additional constraints on the scheduler (such as processor availability). Finally, we show how these methods can also be applied to meet general real-time requirements.

For arbitrary untimed ω-regular processes and specifications the implementability problem is doubly exponential in the length of a temporal logical formula for the specification. Scheduling is doubly exponential in the number of processes as well, but only singly exponential in their size. If there is any implementation, the algorithm produces one of at most doubly exponential size. Real-time implementability and scheduling are also doubly exponential.

A more detailed presentation of this material appears in [WD91].

2 Background

In the last two years, several people have discovered the relevance of the solvability problem for determining the implementability of a linear temporal specification. In 1988, Dill considered it as a well-formedness condition on models of asynchronous circuits, but did not consider synthesis *per se* [Dil89]. Later, Pnueli and Rosner, [PR89a, PR89b], solved the problem of synthesizing concurrent reactive processes (in both synchronous and asynchronous models of concurrency) from a linear temporal logic specification. This solution was applicable to systems with only *distributed variables*. Shared variables were not considered.

Abadi, Lamport and Wolper did model shared variables, but their *computers* had the ability to observe *every* change made to the state, including all changes made by the environment [ALW89]. This is inappropriate for asynchronous parallelism, since in reality an implementation can base its actions only on the state changes that it can observe by reading variables. It does not have full observability of the entire system. Our implementations use only information directly inferable from their own actions.

We simplify considerably Pnueli and Rosner's solution for the asynchronous case. Their original solution involved transformations on several different representations of the specification. In our case, all transformations are performed on automata. The key simplifying idea is that the reductions used in solving implementability can be derived from projections that are used in the definitions of parallel composition.

3 Processes

We choose an action-based model of execution as opposed to a state-based approach because a process is always completely aware of its actions, but often ignorant of the

[1]By synchronous parallelism, we mean that processes run in "lock-step"; by asynchronous parallelism, we mean that processes run at arbitrary speeds. Elsewhere, these terms are used incompatibly to mean unbuffered and buffered communication between processes.

global state of the system. Our model of a single execution of a process is an infinite sequence of *sets* of actions (the reading and writing of variables). A process is assumed to be acting in an environment of other agents, with the only communication being indirectly through the values of the variables.

Suppose that $\mathcal{P}$ is an infinite set of primitive process names. Then every process P has a *type* describing what basic components it has and the variables it can access. Its type is a quadruple $\mathcal{T}_P = (Procs, \mathcal{D}_P, \mathcal{S}_P, \mathcal{E}_P)$, where $Procs$ is a subset of $\mathcal{P}$, $\mathcal{D}_P$ is a set of (owned) *distributed variables* which it can write and other processes can only read, $\mathcal{S}_P$ is a set of *shared variables* which may be written and read by P and other processes, and $\mathcal{E}_P$ is a set of *external* variables, which are read-only. We omit the subscripts when no confusion arises and abbreviate $\mathcal{D} \cup \mathcal{S} \cup \mathcal{E}$ by $\mathcal{V}$. $Procs$ is used to record which primitive process performs each action; this seems necessary for dealing with shared variables[2]. We assume all variables range over a common domain, Dom. When discussing finite-state processes, we assume Dom is finite.

The process P has a set of *primitive actions*, A_P, consisting of *reads* of the form $read(P_i, x, v)$ and *writes* of the form $write(P_i, x, v)$, where P_i is a primitive agent name in $\mathcal{P}$ and v is some value in the domain of the variable x. A *composite action* is a set of primitive actions. These correspond to primitive actions occurring *simultaneously*. The empty set of actions represents nothing happening and is called *skip*.

A *state*, s, is a function assigning a value to every variable in $\mathcal{V}$. An initial condition is a function mapping from $\mathcal{D}$ to $\mathcal{V}$. Let σ_i be the ith element of σ. An *action run* of P corresponds to a trace of actions that P may perform, in which P's distributed variables change only if P writes to them, but shared and external variables can change arbitrarily. Formally, an action run $a = a_1, a_2, \ldots$ of type $(Proc, \mathcal{D}, \mathcal{S}, \mathcal{E})$ is an infinite sequence of composite actions of P, satisfying the following consistency condition: there exists an infinite sequence of states $s = s_1, s_2, \ldots$ such that

(*read-values*) if $read(-, x, v)$ occurs in a_i, then x must have the value v in s_i,

(*write-values*) for every $x \in \mathcal{D}$, if a $write(-, x, v)$ occurs in a_i, x's value in s_{i+1} must be v' where some $write(-, x, v')$ occurs in a_i, and, if no $write(-, x, v)$ occurs in a_i, then $s_i(x) = s_{i+1}(x)$, and,

(*type*) whenever $write(-, x, v)$ occurs in a, x is in $\mathcal{D} \cup \mathcal{S}$.

Intuitively, if several writes to x occur simultaneously, x arbitrarily takes the value of one of them. An action run is consistent with an initial condition I if there is a satisfying sequence of states starting with state s_1 which agrees with I on $\mathcal{D}$.

The behavior of a process P is a triple $(\mathcal{T}_P, I_P, Runs_P)$, where $\mathcal{T}_P$ is its type, I_P its initial condition and $Runs_P$ (usually denoted $R(P)$) a set of action runs of type $\mathcal{T}_P$ consistent with its I. We sometimes use P as an abbreviation for $R(P)$.

3.1 Synchronous parallelism

The behaviors of two processes can be combined to yield the behavior that results when they run synchronously (in "lock step"). The runs of $P = P_1 \parallel P_2$ are essentially those

[2]The usual way of dealing with shared variables is have two labels for actions: π if performed by the process or ε if by the environment[BKP84]. But for scheduling, it helps to record exactly which component executed each action.

runs of the correct type which look like runs to both P_1 and P_2. P is defined only when $Proc_1$ and $Proc_2$ are disjoint, $\mathcal{D}_1$ and $\mathcal{D}_2 \cup \mathcal{S}_2$ are disjoint, and $\mathcal{D}_2$ and $\mathcal{D}_1 \cup \mathcal{S}_1$ are disjoint. Let del_Q be the projection on action runs that deletes all atomic actions made by agents in Q. Formally, $P = (\mathcal{T}_P, I_P, R(P))$ is defined as follows:

$\mathcal{T}_P = (Proc, \mathcal{D}, \mathcal{S}, \mathcal{E})$ where $Proc = Proc_1 \cup Proc_2$, $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2$, $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ and
$\mathcal{E} = (\mathcal{E}_1 \cup \mathcal{E}_2) - (\mathcal{D} \cup \mathcal{S}) = (\mathcal{E}_1 - (\mathcal{D}_2 \cup \mathcal{S}_2)) \cup (\mathcal{E}_2 - (\mathcal{D}_1 \cup \mathcal{S}_1))$.

I_P is the extension of both I_{P_1} and I_{P_2}, i.e. I_P has domain D and agrees with I_{P_1} on D_1 and I_{P_2} on D_2.

$$\begin{aligned} R(P) &= del_{P_2}{}^{-1}(R(P_1)) \cap del_{P_1}{}^{-1}(R(P_2)) \\ &= \{r \mid r \text{ is a consistent run of type } \mathcal{T}_P,\ del_{P_2}(r) \in R(P_1) \text{ and } del_{P_1}(r) \in R(P_2)\}. \end{aligned}$$

Because del_{P_2} removes all P_2 actions, it deletes everything unobservable to P_1, leaving exactly what P_1 observes. Thus $del_{P_2}(r)$ is in $R(P_1)$ precisely when P_1's view of the global system run r is a run of P_1.

3.2 Generalized parallelism

A run is in the parallel composition of P_1 and P_2 when it is observed as a run by each process. Process observability is represented by functions f_{P_i} mapping consistent runs of type $\mathcal{T}_{P_1 \| P_2}$ to sets of consistent runs of type $\mathcal{T}_{P_i}$[3]. In synchronous parallelism, we had del_{P_2} for f_{P_1} and del_{P_1} for f_{P_2}.

$$\begin{aligned} R(P_1 \| P_2) &= f_{P_1}{}^{-1}(R(P_1)) \cap f_{P_2}{}^{-1}(R(P_2)) \\ &= \{r \mid r \text{ is a consistent run of type } \mathcal{T}_{P_1 \| P_2},\ f_{P_1}(r) \subseteq R(P_1) \text{ and } f_{P_2}(r) \subseteq R(P_2)\} \end{aligned}$$

3.3 Asynchronous parallelism

We often want the behaviors of processes that run in parallel *asynchronously.* Inserting an arbitrary (finite) number of *skip* actions between the non-*skip* actions of a run of a process allows other processes to perform as many or as few actions as they like between the process's non-trivial actions. The *delskip* operation removes the *skip* actions of P from an action run. The *stuttering closure* of an action run a is $stut(a) = \{a' \mid delskip(a') = delskip(a)\} = delskip^{-1}(delskip(a))$, see [ALW89]. This definition is extended to sets of action runs. For convenience, we use $R_{AS}(P)$ to refer to $stut(R(P))$, the asynchronous runs, or asynchronous closure of P. Asynchronous parallelism, $\|_{AS}$, is defined as synchronous parallelism on the asynchronous closure of the processes. That is, $R(P_1 \|_{AS} P_2) = del_{P_2}{}^{-1}(R_{AS}(P_1)) \cap del_{P_1}{}^{-1}(R_{AS}(P_2))$. This is a special case of generalized parallelism, with $delskip^{-1} \circ delskip \circ del_{P_2}$ for f_{P_1}.

4 Implementability

A reactive system consists of processes which communicate among one another and that are intended to run forever. Suppose P is a reactive process where the only other agent is a completely unpredictable environment over which it has no control. The interaction between the implementation and the environment can be viewed as a game. Each player

[3]It may be that P_i's view of a run is not uniquely determined.

takes a turn extending a computation. The implementation wins the game if the resulting computation satisfies the specification. Moves in the game are actions, the *reads* and *writes* of variables. If the process always wins the game, it is an implementation of the specification.

A *strategy* is a special kind of process which decides its next *move* from its history of actions. Its moves are *read* requests of *individual* variables, and *write* requests of individual variables *with* the values to be written. Notice that a strategy is restricted to accessing only one variable at a time. Formally, a strategy S of type $(\{S\}, \mathcal{D}, \mathcal{S}, \mathcal{E})$ is a function $g_S : A_S^* \rightarrow Moves_S$, where $Moves_S = \{read(var) \mid var \in \mathcal{S} \cup \mathcal{E}\} \cup \{write(var, value) \mid var \in \mathcal{D} \cup \mathcal{S}, value \in Dom\}$. Because the adversarial environment can choose to return any value in response to a *read* action by the strategy, $R(S)$ is the set of *all* sequences of actions $a = a_1, a_2, \ldots$ such that

1. $a_{i+1} = read(S, var, value)$ for some $value \in Dom$ whenever $g_S(a_1, a_2, \ldots, a_i) = read(var)$, and
2. $a_{i+1} = write(S, var, value)$ whenever $g_S(a_1, a_2, \ldots, a_i) = write(var, value)$.

The strategy plays against an environment E of type $(\{E\}, \mathcal{E}, \mathcal{S}, \mathcal{D})$ whose actions runs are simply all the action runs for its type. It is an implementation of the specification *Spec* if every run of the combined system is in the specification, i.e. $R(S \parallel E) \subseteq Spec$. In this case it is a winning strategy for the game.

4.1 The Read-Write Game

Most of the games in this paper are games of partial information. The implementation cannot directly observe the entire system; the only information it knows about the environment has been inferred from reading the environment's variables. A solution to the implementability problem proceeds in two steps: reducing the game of partial information to one of complete information, and then solving the game of complete information, the *read-write game*. Runs in this game are traces of strategy actions only, and therefore reflect exactly what the strategy observes. A read-write game is stated as: Given a set of action runs W of type $(\{S\}, \mathcal{D}, \mathcal{S}, \mathcal{E})$, is there a strategy of the same type whose action runs are all in W, i.e. $R(S) \subseteq W$?

If the problem is finite state (the behaviors of the processes and the specification can be represented as ω-regular sets), the read-write game is decidable. This result follows almost immediately from the decidability of the Church-Büchi solvability problem, [BL69, Rab72].

Theorem 1 *If A is a nondeterministic Büchi automaton (NBA) defining Spec, a set of action runs for S over k variables, and A has n states, then the read-write game for Spec is decidable in $2^{O[(kn)^2 \cdot log(kn)]}$ time. If there is a winning strategy for Spec, the algorithm produces one of size $2^{O[(kn)^2 \cdot log(kn)]}$.*

Proof: The solution is a variant on the algorithm for "synchronous implementability" in [PR89a]. We construct from A a tree automaton that accepts exactly the trees that correspond to winning strategies. Hence to solve the game, we need only test for emptiness of the tree automaton [EJ88]. □

4.2 Cores

It is usual for a process to have only partial observability of the entire system; it cannot observe all the environment's actions. Suppose the environment owns the variables x_1 and x_2. While the process is reading x_1, the environment may be changing the value of x_2. In an asynchronous system the environment may even change x_2 an arbitrary number of times without the process noticing. Suppose two global system runs, r_1 and r_2, both look like the action run r to the implementation. The implementation only observes r and cannot distinguish whether r_1 or r_2 actually occurred. Thus a strategy or implementation should only generate action runs for which *all* corresponding global runs lie in the specification.

We assume the strategy has observability function f_S. The core of a global specification $Spec$, $core(Spec)$, is the set of (local) strategy action runs whose inverse images under f_S are all in the specification, i.e. $core(Spec) = \{a \mid f_S^{-1}(a) \subseteq Spec\}$.

Lemma 1 *Let Spec a global specification and S be a strategy, interacting with the environment under observability function f_S. The following are equivalent:*
1) S is an implementation of Spec,
2) $R(S \parallel E) \subseteq Spec$,
3) $f_S^{-1}(R(S)) \subseteq Spec$,
4) $R(S) \subseteq core(Spec)$,
5) S wins the read-write game over $core(Spec)$. □

Therefore implementability can be constructively reduced to the read-write game, provided we have an algorithm for finding the core of a specification. Since $core(Spec) = \{r \mid {f_S}^{-1}(r) \subseteq Spec\}$, its complement is $\{r \mid {f_S}^{-1}(r) \cap \overline{Spec} \neq \emptyset\}$. Thus a run r is in the complement of the core if it has a preimage under f_S in $\overline{Spec}$. Hence $core(Spec) = \overline{f_S(\overline{Spec})}$. The core can be found by complementation and substitution.

4.3 Synchronous Implementability

Until now, we have not considered any particular representation of the specification. We want to express ω-regular properties of variables ranging over *finite non-boolean* domains. Rather than defining a temporal logic with finite-valued variables, for convenience we express properties in *propositional* temporal logic (PTL). We encode each bit-value of a variable as a separate proposition.

Lemma 2 *Let ϕ be a PTL formula for Spec. Then there is an algorithm to find a deterministic Rabin automaton A accepting the $core_S(Spec)$.*

Proof: We proceed in three steps. From ϕ construct a NBA B for $\overline{Spec}$, ([WVS83]). Find a NBA B' for $\overline{core_S(Spec)}$, (apply the homomorphism $f_s = del_E$). Complement and determinize B' to get the desired Rabin automaton A ([EJ89]). □

Theorem 2 *Let ϕ be a PTL formula of length n_0 for the property Spec. There is a $2^{O[(k\cdot|Dom|)^3\cdot 2^{(3n_0)}]}$ algorithm for testing the synchronous implementability of Spec. If Spec is implementable, it yields a $2^{O[(k\cdot|Dom|)^3\cdot 2^{(3n_0)}]}$ folded representation of a strategy for Spec.*

Proof: Because of Lemma 1, we can follow the algorithm of Lemma 2, then solve the read-write game. The complexity result follows from analysis of the tableau procedure of [WVS83], the simultaneous complementation and determinization algorithm of [EJ89] and the test for emptiness of [EJ88]. □

4.4 Asynchronous Implementability

The analogous results from the last section all hold; the observability function is merely changed from del_E to $f_{AS} = delskip^{-1} \circ delskip \circ del_E$. Performing the substitution for *delskip* is similar to way ϵ-transitions are removed from finite state automata.

Theorem 3 *Let ϕ be a PTL formula of length n_0, defining the set Spec . The* asynchronous *implementability of Spec is solvable with complexity* $2^{O[(k\cdot|Dom|)^3\cdot 2^{(3n_0)}]}$. □

5 Scheduling

The ideas in the previous section can be used to find a *scheduler*, which is a process designed to provide external coordination for other processes. Such a situation may arise in the fields of distributed computing, networking, hardware design and operating systems. For example, we may be given information about the processes to be run and asked to schedule them to ensure mutual exclusion of their critical sections or guarantee some form of liveness. The scheduling problem is stated as follows: For processes $P_1, \ldots, P_n$ and specification *Spec*, is there a strategy S such that $P_1 \parallel \cdots \parallel P_n \parallel S$ satisfies *Spec*?

Let $P_1 \parallel \cdots \parallel P_n$ be collectively called P. We assume $P \parallel S$ is a system with no external variables. A truly external environment can always be modeled as another primitive process within P. By definition, $R(P \parallel S) = f_P{}^{-1}(R(P)) \cap f_S{}^{-1}(R(S))$. Thus, $P \parallel S$ satisfies *Spec* iff $f_S{}^{-1}(R(S)) \subseteq \overline{f_P{}^{-1}(R(P))} \cup Spec$. This problem is essentially implementability with a modified specification, $Spec' = \overline{f_P{}^{-1}(R(P))} \cup Spec$, (see Lemma 1). Intuitively, *Spec'* asserts that if the environment behaves like the processes we need to schedule, then the original specification *Spec* must hold.

5.1 The Finite State Case

For both synchronous and asynchronous parallelism, the processes can be effectively composed and *Spec'* computed, which implies that the scheduling problem is decidable. The complexity is the same as that for implementability except for an extra multiplicative factor of p^n for the composition of the processes.

Theorem 4 (Scheduling) *Assume the system has k variables over a finite domain Dom. Let $\{P_1, \ldots, P_n\}$ be a set of processes, where each P_i is defined by a NBA A_i of size $\leq p$. Let Spec, a property of action runs with agents $P_1, \ldots, P_n$ and S, be defined by a PTL formula of length n_0. Then there is a $2^{O[(k\cdot|Dom|\cdot p^n)^3\cdot 2^{(3n_0)}]}$ algorithm to solve the scheduling problem, giving a $2^{O[(k\cdot|Dom|\cdot p^n)^3\cdot 2^{(3n_0)}]}$ state scheduler, if any scheduler exists.* □

6 Transformations on specifications

Most specifications are only expected to be met when the processes or the environment behave in a certain way. For example, a mutual exclusion program may only guarantee progress for each process requesting entry to its critical section if every process has a terminating critical section, *Term_Crit* $\Rightarrow$ *Prog*. An environmental assumption can be handled simply by adding it as an antecedent to the specification.

Additional requirements on the scheduler can be added to specifications. For example, suppose we have 10 processes to schedule, but only 5 processors available. Then we must disallow any scheduler that permits more than 5 processes to execute at any time. To handle this, the scheduler needs substantial control over the processes. One way to achieve this control is to add a new boolean *signal variable* $\{go_i\}$ for each process P_i. These new variables are owned by the scheduler. The process constraints C_P would assert that no P_i ever makes a non-*skip* action when the signal go_i is false. We then add to the scheduler the constraint "no more than 5 go_i signals are true at any time".

A constraint on the scheduler C_{sched} can be enforced by changing the specification *Spec* to $C_{sched} \cap Spec$. The same technique can be used for more general resource constraints, for example, "processes 5 and 6 can never run on processors A and B at the same time."

7 Real-Time Processes and Specifications

Synchronous parallelism requires all processes to advance at the same time, while asynchronous parallelism is completely speed independent. It is often the case that processes are not interacting synchronously, but nor are they completely asynchronous — something is known about timing in the system. Knowledge of the relative timing of events may make it easier to implement a specification.

We consider *dense time*, where events may occur at arbitrarily close times; time is interpreted over the nonnegative real numbers, **R**. The execution of concurrent real-time processes is modeled by *timed traces* [AD90], which are infinite sequences together with times at which each event occurs. We use the *timed Büchi automata (TBA)* of [AD90] to model specifications and processes.

7.1 Real-Time Implementability

The methodology for solving the implementability and scheduling problems for untimed processes and specifications also works when timing is introduced. Using the techniques of section 6, timing assumptions on the environment and timing requirements on the implementation may all be written into the specification. Thus a strategy has no timing information itself and may be synthesized from the core of the specification, exactly as above for the untimed case. To solve real-time implementability then, we must show how to derive the core from a timed specification.

Timed processes are the natural extension of untimed processes, in that runs are now *timed* traces. Let *Untime* be the function that removes the timing information from a timed trace. The observability function for P_1 in $P_1 \parallel_t P_2$ is given by $f_{P_1} = Untime \circ delskip^{-1} \circ delskip \circ del_{P_2}$. A strategy S is a timed implementation of *Spec* iff $Untime^{-1}(S) \parallel_t Untime^{-1}(E) \subseteq Spec$. It is clear that S implements *Spec* if it wins the read-write game over $core_t(Spec) = \{r \mid {f_S}^{-1}(r) \subseteq Spec\}$.

Lemma 3 *Given a TBA* $A' = \langle \Sigma, S, s_0, F, C, \delta \rangle$ *for* $\overline{Spec}$, *there is an* $O(|A|^3)$ *algorithm to find a NBA B for* $\overline{core(Spec)}$. *B has size* $O(|C|! \cdot |\Sigma| \cdot |S|^2 \cdot 2^d)$, *where d is the number* *of bits in the binary encoding of A''s timing constants.* □

Because TBAs are not closed under complementation, we allow only specifications given as *deterministic* timed automata (DTA), which have at most one run for any timed trace. However DTA with Büchi acceptance conditions are not as expressive as DTA with Muller, Streett and Rabin acceptance condition. All these deterministic automata can be complemented, yielding TBA with polynomially many states[4]. Timing conditions on the implementation and the environment can also be included as antecedents to the original specification. An implementation of $Spec' = \overline{T_{imp} \cap T_e} \cup Spec$ implements $Spec$ under the timing assumptions T_{imp} and T_e.

Theorem 5 (Real-time Implementability) *Let A_{imp} and A_e be TBA for timing requirements T_{imp} and T_e on the implementation and the environment respectively. Let A_{Spec} be a DTA[4] for the specification. Then the real-time implementability problem is solvable in time exponential in the number of variables and number of states of the automata but doubly exponential in the number of bits to encode the timing constants.* □

7.2 Real-time Scheduling

The problem of scheduling a collection of activities to meet hard real-time deadlines has been studied extensively [CS88]. Previous work in this area considers scheduling a set of tasks (which may or may not be known in advance) to meet completion deadlines, usually on a single processor. These problems are usually at least NP-hard.

Here we are scheduling *reactive* processes which communicate and coordinate among themselves. This problem is not easily modelled as a collection of tasks to be scheduled. Furthermore, the expressiveness of ω-regular languages enables us to handle a far wider range of timing properties than the simple meeting of completion deadlines. For example, restrictions can be enforced on the ordering of completion times and the differences between completion times. Our specifications include, for example, "if y is set to 1 sufficiently often, it will eventually always be reset to 0 within 2 seconds".

We deal only with static scheduling, where the characteristics of the processes are known beforehand. Specifications must be given as *deterministic* TBA, but the processes to be scheduled and the timing constraints on the scheduler and the environment may be arbitrary TBA. Real-time scheduling then reduces to real-time implementability over the property "if the environment behaves like the processes, and all components in the system satisfy their timing constraints, then the specification is met".

Theorem 6 (Real-time Scheduling) *Let $\{P_1, \ldots, P_n\}$ be a set of processes with each P_i defined by the TBA A_i of size $\leq p$. Let Spec be given by a DTBA A_{Spec} of size n_{Spec}, and let T_{imp} and T_e be timing constraints on the implementation and the environment defined by TBA A_{imp} and A_e of sizes n_{imp} and n_e respectively. Let c be the total number of clocks and d the number of bits in the encoding of all the timing constants. The complexity of real-time scheduling is $2^{O([c! \cdot p^n \cdot k \cdot |Dom|(n_{imp} \cdot n_e \cdot n_{Spec})^2 \cdot 2^d]^3)}$.* □

[4]The number of acceptance pairs of a Rabin automaton must be logarithmic in the number of states.

8 Conclusion

The idea of finding strategies for games is not only applicable to the specific problem of synthesizing an implementation from a temporal specification, but can also be applied to other problems such as scheduling. Although the doubly exponential algorithms presented here may prove to be useful for very small systems, easier special cases of the problem will have to be discovered for real practicality. The problem of synthesizing a scheduler to meet a specification closely resembles the *supervisory control problem* in the study of discrete event systems, which historically has fallen in the domain of control theory. The relations between these two problems should be studied in greater detail.

Acknowledgements

We thank Amir Pnueli for a helpful discussion, and Elizabeth Wolf for careful reading of a draft of the paper.

References

[AD90] R. Alur and D. Dill, "Automata for modeling real-time systems", ICALP 1990.

[ALW89] M. Abadi, L. Lamport, P. Wolper, "Realizable and unrealizable specifications of reactive systems", *International Colloquium on Automata, Languages, and Programming,* Lecture Notes in Computer Science, 1989, Springer-Verlag

[BKP84] H. Barringer, R. Kuiper and A. Pnueli, "Now you can compose temporal logic specifications", *Proceedings of the ACM Symposium on Theory of Computing*, 1984, pp. 51–63.

[BL69] J. R. Büchi and L. H. Landweber, "Solving sequential conditions by finite-state strategies", Transactions of the American Mathematical Society, 138, 1969, pp. 295–311.

[CS88] S-C. Cheng and J. A. Stankovic, "Scheduling algorithms for hard real-time systems - a brief survey", in *Hard Real-Time Systems*, IEEE Press, 1988, pp. 150–173.

[Chu63] A. Church, "Logic, arithmetic, and automata", in *Proceedings of the International Congress of Mathematicians, 1962*, Institut Mittag-Leffler, 1963, pp. 23–35.

[Dil89] D.L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits", MIT Press, 1989.

[EJ88] E.A. Emerson, C.S. Jutla, "The complexity of tree automata and logics of programs", *Proc. of the 29th IEEE Symp. on Foundations of Computer Science*, 1988, pp. 328–337.

[EJ89] E.A. Emerson, C.S. Jutla, "On simultaneously determinizing and complementing ω-automata", *Pro. of the Symp. on Logic in Computer Science*, 1989, pp. 333–342.

[PR89a] A. Pnueli, R. Rosner, "On the synthesis of a reactive module", *Proc. 16th ACM Symp. Principles of Programming Languages*, 1989, pp. 179–190.

[PR89b] A. Pnueli, R. Rosner, "On the synthesis of an asynchronous reactive module", *International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science Vol 372, 1989, Springer-Verlag

[Rab72] M. O. Rabin, "Automata on infinite objects and church's problem", *Regional Conference Series in Mathematics*, Vol. 13, American Mathematical Society, 1972.

[WVS83] P. Wolper, M.Y. Vardi, A.P. Sistla, "Reasoning about infinite computation paths", *Proc. of the 24th IEEE Symp. on Foundations of Computer Science*, 1983, pp. 185–194.

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