# Symbolic Controller Synthesis for Discrete and Timed Systems*

Eugene Asarin[1] Oded Maler[2] Amir Pnueli[3]

[1] Institute for Information Transmission Problems, 19 Ermolovoy st., Moscow, Russia, asarin@ippi.msk.su
[2] Spectre – Verimag, Miniparc-zirst, 38330 Montbonnot, France, Oded.Maler@imag.fr
[3] Dept. of Computer Science, Weizmann Inst. Rehovot 76100, Israel, amir@wisdom.weizmann.ac.il

**Abstract.** This paper presents algorithms for the *symbolic* synthesis of discrete and real-time controllers. At the semantic level the controller is synthesized by finding a winning strategy for certain games defined by automata or by timed-automata. The algorithms for finding such strategies need, this way or another, to search the state-space of the system which grows exponentially with the number of components. Symbolic methods allow such a search to be conducted without necessarily enumerating the state-space. This is achieved by representing sets of states using formulae (syntactic objects) over state variables. Although in the worst case such methods are as bad as enumerative ones, many huge practical problems can be treated by fine-tuned symbolic methods. In this paper the scope of these methods is extended from analysis to synthesis and from purely discrete systems to real-time systems.
We believe that these results will pave the way for the application of program synthesis techniques to the construction of real-time embedded systems from their specifications and to a solution of other related design problems associated with real-time systems in general and asynchronous circuits in particular.

## 1 Introduction

Apart from the different underlying state-spaces and time domains, perhaps the largest difference between control theory and computer science[4] lies in the relative weight of *analysis* and *synthesis* methods. Analysis can be roughly stated as: *Will this happen?* while synthesis as: *How can we make this happen?*

---

* This research was supported in part by the European Community projects BRA-REACT(6021), HYBRID EC-US-043 and INTAS-94-697 as well as by Research Grant #93-012-884 of Russian Foundation of Fundamental Research. Verimag is a joint laboratory of cnrs, inpg, ujf and verilog sa. spectre is a project of inria.

[4] We take computer science to denote here the community of those who want to reason formally about the behavior of computers.

There is nothing inherent in the nature of the discrete or the continuous that makes control people more centered around synthesis and informaticians around analysis (also called "verification"). The study of "passive" analysis of continuous systems is simply not called "control" but rather the theory of dynamical systems, differential equations, etc. On the other hand, the attempts to build automatic program synthesis (or "derivation") methods, although numerous, have mostly been considered as sci-fi dreams, given the common belief that analysis/verification is already hard enough. One line of research in program synthesis concentrated on non-reactive programs, i.e. systems that operate in a "static" environment, e.g. [MWa80]. In this framework a program is derived as a constructive proof of an existential statement which constitutes the specification. In spite of the absence of external disturbances, this problem is much harder than the one we consider because no a-priori program structure is assumed.

For reactive systems (systems that maintain an ongoing interaction with a dynamic environment) the synthesis problem has been posed as early as 1957 by Church [Chu63] in the context of digital circuits. Church's problem was solved by Büchi and Landweber [BL69] (a readable exposition of their result appeared in [TB73]). More modern efforts toward automatic synthesis have been made by various authors such as [EC82], [MWo84], [PR89-a], [PR89-b], [ALW89] or [WD91], but apart from some impressive theoretical results (in particular, complexity bounds) the work on synthesis remained marginal compared to the vast literature on verification and, as far as we know, has not been transferred from academia to industry. Interestingly, a large body of work dealing with discrete synthesis came from outsiders to computer science, namely the DEDS model of Ramadge and Wonham [RW89] in which the controller can inhibit certain transitions of an automaton in order to achieve some behavioral specifications.

Meanwhile there have been some breakthroughs in the verification area. The analysis problem, although intractable in terms of worst-case asymptotic complexity, became feasible for industrial size problems, especially in hardware. This success is due to the use of *symbolic* methods [BCM$^+$93], [McM93] that do not transform the description of the system into an enormous "flat" automaton but rather represent the transition relation as a formula over the state variables. Given such a formula $\mathcal{T}$ and a formula $P$ describing some subset $F$ of the state-space one can calculate a new formula $P'$ characterizing the set $F'$ of successors (or predecessors) of $F$. The goal of this paper is to discuss the transfer of this technology from analysis to synthesis and from discrete to real-time systems. The only work along similar lines we are aware of is that of Hoffmann and Wong-Toi [HW92-a], [HW92-b], [BHG$^+$93] who introduce symbolic methods into the Ramadge-Wonham model. Another interesting approach to introducing syntactics into controller synthesis is reported in [DV94]. Although our paper presents no new theoretical result, we hope that it will contribute to a better understanding of the nature of real-time discrete control and its associated computational problems.

The rest of the paper is organized as follows. In section 2 we set the stage for the discrete case and give semantic and symbolic characterizations of winning strategies. The derivation of a discrete scheduler in section 3 illustrates the idea. Timed automata and real-time games are introduced in section 4 along with a solution of the symbolic synthesis problem. In section 5 we demonstrate a solution of a real-time version of the scheduler problem. Finally we mention application and implementation issues.

## 2 Discrete Systems

### 2.1 The Core Idea behind Discrete Synthesis

The most fruitful approach to discrete program synthesis is to view the ongoing interaction between the system one wants to design and the environment[5] in which it is supposed to operate as some variant of the von Neumann-Morgenstern discrete games. A strategy for a given game is a rule that tells the controller how to choose between several possible actions in any game position. A strategy is good if the controller, by following these rules, always wins (according to a given definition of winning) no matter what the environment does [PR89-a], [NYY92].

The existence (and extraction) of a strategy for finite games is done using the max-min principle of [NM44], disguised sometimes as searching AND-OR trees or as the elimination of an alternating pair of the logical quantifiers $\exists$ and $\forall$. This principle is illustrated using the game at the left part of figure 1. In this game the controller starts from position 0 and can choose between the two actions $a_1$ and $a_2$. Then the environment can choose between $b_1$ and $b_2$. The winning condition is specified via some subset $F$ of $\{1,2,3,4\}$. A run of the game is winning if it ends up in an element of $F$. Suppose $F = \{1,4\}$ – in this case the first player has no winning strategy at state 0 because if it chooses $a_1$, the adversary can take $b_2$ and reach state 2. If it chooses $a_2$ the adversary can reach state 3 by taking $b_1$. Hence, 0 is not a winning position. If, on the other hand, we consider a game with the same transition structure but with $F = \{1,2\}$ then there is a winning strategy as the controller can, by making $a_1$, "force" the environment into $F$.

The convention that each player plays in *its own turn* can be replaced by having a simultaneous move. In this case there are no two types of states and the transition between states is made by a *joint action* of the two players as in the right of figure 1. *This is the convention that we adopt in this paper* and hence, the notions of "first" and "second" player bear no ordinal meaning.[6] For games on continuous time the notion of turns becomes anyway meaningless.

Some readers may wonder whether such trivialities deserve a relatively long verbal exposition. We consider this to be the *essence* of any synthesis algorithm,

[5] As one control theoretician once said: "you CS people call *environment* everything that lies outside the computer".

[6] It can be shown that every von Neumann-Morgenstern game can be converted into an equivalent simultaneous action game.

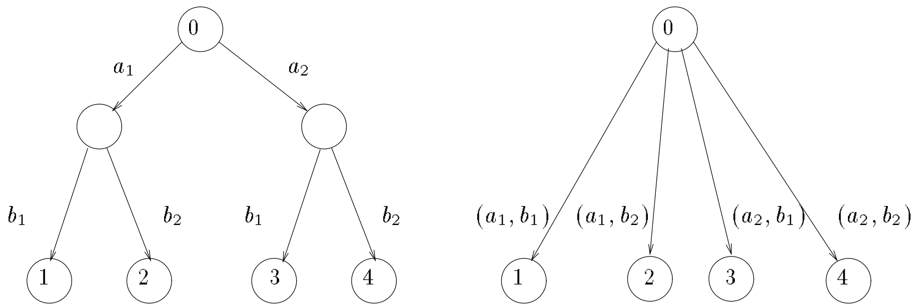


**Fig. 1.** A simple game.

a fact which is sometimes obscured by fancy complicated constructs. The mathematical formulation of this notion for a game with a state-space $Q$ is via an operator $\pi : 2^Q \rightarrow 2^Q$ assigning for every $F \subseteq Q$ the set $\pi(F)$ denoting its *controllable predecessors*, that is, the set of states from which the controller can force its adversary into $F$. In the example above $\{0\} \notin \pi(\{1,4\})$ and $\{0\} \in \pi(\{1,2\})$. Calculating this operator, together with some set-theoretical operations constitutes the core of any synthesis algorithm.

### 2.2 Semantic Version

**Definition 1 (Game Automaton).** *A game automaton is $\mathcal{A} = (Q, \delta, q_0, F)$ where $Q = Q_x \times Q_y$ is a set of states, $\delta : Q \rightarrow 2^Q$ is the transition relation, $q_0 \in Q$ is the initial state of the game and $F \subseteq Q$ is a set of accepting states. We assume that $\delta$ admits a decomposition into $\delta_x : Q \rightarrow 2^{Q_x}$ and $\delta_y : Q \rightarrow 2^{Q_y}$ such that for every $q \in Q$, $\delta(q) = \delta_x(q) \times \delta_y(q)$.*

One can see that the game automaton is a product of two automata communicating via their states, that is, the transition made by the $X$-automaton may depend on the current state of the $Y$-automaton and vice versa.

A *run* of the game is any infinite sequence $\xi = q[1], q[2], \ldots$ such that $q[1] = q_0$ and for every $i$, $q[i+1] \in \delta(q[i])$. The first player wins a run of the game if the run always stays in $F$. Otherwise it loses the run.

**Remark 1**: We have considered one of the several possible definitions of winning, namely the $\Box$-condition (also known as "safety", or "closed" game). One can imagine the "dual" game as viewed from the second player's perspective – this player wins a run of the game if it succeeds to steer the run at least once into $Q - F$. These are called $\Diamond$-games ("eventuality", "open"). Since the emphasis in this paper is on real-time and on symbolic methods, we will not refer here to more complicated winning conditions – see [Tho94] for a recent survey.

A strategy for the first player is a transition relation $\delta_x^* : Q \rightarrow 2^{Q_x}$ such that $\delta_x^*(q) \subseteq \delta_x(q)$ for every $q \in Q$. This can be viewed as a rule telling the controller which actions it should take while being at any $q \in Q$. A strategy can be deterministic ($|\delta_x^*(q)| = 1$) but it need not be so – in case we need later to implement it, we can fix arbitrarily some $q' \in \delta^*(q)$. When we replace $\delta_x$ by $\delta_x^*$ we obtain a more restricted automaton $\mathcal{A}^*$ with $\delta^* \subseteq \delta$. A strategy $\delta_x^*$ is a winning one if all the runs of $\mathcal{A}^*$ are winning.

The search for a strategy is performed via the determination of the set $F^*$ of *winning states*. This is done iteratively starting with $F_0 = F$. Every iteration we create $F_{i+1}$ by removing from $F_i$ all the states from which the first player *cannot* force the game to stay in $F_i$. It is not hard to see that a state $q$ belongs to $F_i$ if, starting from $q$, the controller can stay in $F$ for at least $i$ steps. This procedure converges to the set $F^*$ of winning states. If $q_0 \in F^*$ the controller has a winning strategy. Formally:

**Definition 2 (Controllable Predecessors).** *Let $\mathcal{A} = (Q, \delta, q_0, F)$ be a game automaton. We define a function $\pi : 2^Q \mapsto 2^Q$ as*

$$\pi(P) = \{q : \exists q'_x \in \delta_x(q) \, \forall q'_y \in \delta_y(q) \, (q'_x, q'_y) \in P\}$$

The algorithm for calculating the winning states works as follows:

**Algorithm 1 (Synthesis for Discrete $\Box$-Games).**

$$\begin{array}{l} F_0 := F \\ \textbf{for } i = 0, 1, \ldots, \textbf{ repeat} \\ \qquad F_{i+1} := F_i \cap \pi(F_i) \\ \textbf{until } F_{i+1} = F_i \end{array}$$

The algorithm is illustrated in figure 2. The strategy for every $q \in Q$ is extracted as follows:

$$\delta^*(q) = \begin{cases} \emptyset & \text{if } q \notin F^* \\ \{q'_x : \forall q'_y \in \delta_y(q) \, (q'_x, q'_y) \in F^*\} & \text{otherwise} \end{cases}$$

Note that in the first case ($q \notin F^*$) we do not care because anyway we cannot win after arriving to such a state. This might be different had we been interested in more quantitative notions of winning, e.g. expected probability.

**Remark 2**: We extract the strategy *after* having completed the calculation of the winning states. We could have done it incrementally by removing losing transitions after each iteration. In games with winning condition other than $\Box$, it might be better (and sometimes even necessary) to calculate $\delta^*$ iteratively along with the calculation of $F^*$.

This is all that needs to be said about safety games. More complex games require more complicated iterative procedures and sometimes they cannot be won by simple strategies (those that depend only on $q$) but rather need some more information on the history of the game (as a continuous controller needs sometime to know the sign of the derivative of the state-variable). We refer the reader again to the survey [Tho94] as well as to [TW94].

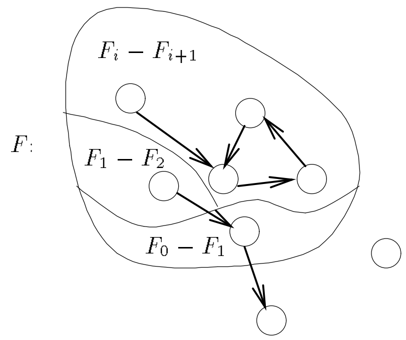


**Fig. 2.** An illustration of algorithm 1.

### 2.3 Symbolic Version

So far we have considered the state-space of $\mathcal{A}$ to be an amorphous set $Q$ void of any structure. Large complex systems, however, are usually composed of smaller sub-systems, and the global state-space is the Cartesian product of the local ones (and so is the transition relation). Instead of creating this huge automaton and applying analysis or synthesis algorithms to it, symbolic methods keep the transition relation in a syntactic form and operate on it.

Suppose a system is defined using a set $X = \{x_1, \ldots, x_k\}$ of Boolean variables. Hence $Q = \{0,1\}^k$ and every subset $F$ of $Q$ can be described by one (or more) Boolean formula $P$ over $X$. A transition relation $R \subseteq \{0,1\}^k \times \{0,1\}^k$ can be written as a formula over a set $X \cup X'$ of variables where each $x_i'$ represents the value of $x_i$ in the "next" state.

A symbolic analysis method consists of a class $\mathcal{F}$ of syntactic objects (formulae) covering all the subsets of $Q$, such that for every set-theoretic operation on $2^Q$ there is a corresponding semantics-preserving operation on $\mathcal{F}$. In particular one needs a class of objects in which there are syntactic operations that correspond to: 1) standard set-theoretic operations, 2) the predecessor operator $\pi$, and 3) equality testing. This is all one needs in order to perform algorithm 1. Certain classes of syntactic objects, such as ordered BDDs [Bry86], have a canonicity property, namely one-to-one mapping between syntactic objects and the sets they denote – this makes equality testing trivial. In our exposition we will not insist on a particular representation (which is an implementation question) but rather use arbitrary Boolean formulae. Note that elimination of quantifiers is a simple syntactic operation for Boolean formulae:

$$\exists x_i\, P(x_1, \ldots, x_i, \ldots, x_k) = P(x_1 \ldots, 0, \ldots, x_k) \vee P(x_1, \ldots, 1, \ldots x_k)$$

and

$$\forall x_i\, P(x_1, \ldots, x_i, \ldots, x_k) = P(x_1 \ldots, 0, \ldots, x_k) \wedge P(x_1, \ldots, 1, \ldots x_k)$$

In order to adapt the symbolic method for synthesis we must first introduce some notion of interaction between automata and "ownership" of state-variables. Intuitively every process has its own set of local variables which cannot be changed by other processes. On the other hand, it may "read" the state of other processes and base its decision concerning which transition to make on the values of these variables. Syntactically, if the corresponding sets of variables are $X$ and $Y$, the respective transition formulae of the two automata can be written as $\mathcal{T}_X(X,Y,X')$ and $\mathcal{T}_Y(Y,X,Y')$. When we compose them together we obtain a closed system where both $X$ and $Y$ are internal variables and the transition formula is $\mathcal{T}(X,Y,X',Y') = \mathcal{T}_X(X,Y,X') \wedge \mathcal{T}_Y(Y,X,Y')$.

In a control setting we let a variable set $X$ denote the controllable variables, namely the variables owned by the controller and which he can change (or refrain from changing). The other set $Y$ consists of environmental variables which the controller cannot influence directly (however, since $\mathcal{T}_Y$ depends on $X$, the controller can influence them indirectly, which is essentially what control is all about).

We use quantifiers of the form $\exists X$ or $\forall X$ as an abbreviation for $\exists x_1 \exists x_2 \ldots$, etc., and assume without loss of generality that the transition relation is complete. Given $\mathcal{T}_X$, $\mathcal{T}_Y$ and a set $F$ expressed by a formula $P(X,Y)$, the syntactic predecessor operator is defined as follows:

$$\pi(P)(X,Y) = \exists X' \left[\mathcal{T}_X(X,Y,X') \wedge \forall Y' \left(\mathcal{T}_Y(Y,X,Y') \Rightarrow P(X',Y')\right)\right]$$

To rephrase it verbally, the immediate controllable predecessor of $P$ are all the states from which there is an $X$ "action" such that for every $Y$ action the resulting state satisfies $P$.

Having all the other ingredients of a symbolic method we can plug it into algorithm 1 and converge to a formula $P^*$ characterizing all the winning states. The controller is derived from $\mathcal{T}_x$, $\mathcal{T}_y$ and $P^*$ as follows:

$$\mathcal{T}_X^*(X,Y,X') = \mathcal{T}_X(X,Y,X') \wedge P^*(X,Y) \wedge \forall Y' \left(\mathcal{T}_Y(Y,X,Y') \Rightarrow P^*(X',Y')\right)$$

The transition relation expressed by $\mathcal{T}_X^*$ is a subset of the one expressed by $\mathcal{T}_X$ and is obtained by restricting the possible values for $X'$. The transition relation of the whole system after the controller is synthesized is expressed by $\mathcal{T}^*(X,Y,X',Y') = \mathcal{T}_X^*(X,Y,X') \wedge \mathcal{T}_Y(Y,X,Y')$. All its runs are winning for the first player.

This is all the story. The question whether there exists an efficient implementation scheme allowing large-scale synthesis is an empirical open question. Some positive evidence is reported in [BHG+93].

## 3 Example: A Discrete Scheduler

Suppose we have two identical processes with the corresponding state variables $S_1$ and $S_2$. Each of them can be either in $I$ (idle) or $W$ (waiting). A process can be at $S_i = I$ as long as it wishes and can generate a request (move to $S_i = W$)

only if at least 3 time units have elapsed since the previous request. It can move from $S_i = W$ to $S_i = I$ whenever the scheduler gives it a permission by letting the variable $G_i = 1$ (we assume here that the service is immediate). For modeling this behavior we use for each $i$ a variable $C_i$ ranging over $\{0, 1, 2, 3\}$ measuring the number of steps since the previous request. The system is depicted in figure 3 and the product of $S_i$ and $C_i$ is the automaton of figure 4. The whole system consists of a product $S_1 \circ C_1 \circ S_2 \circ C_2 \circ G$ where $G$ is the automaton for the scheduler that we want to synthesize. It is a 2-variable (4-state) automaton that decides the values of $G_1$ and $G_2$. We have not drawn the whole system (the formulae are sufficiently large).

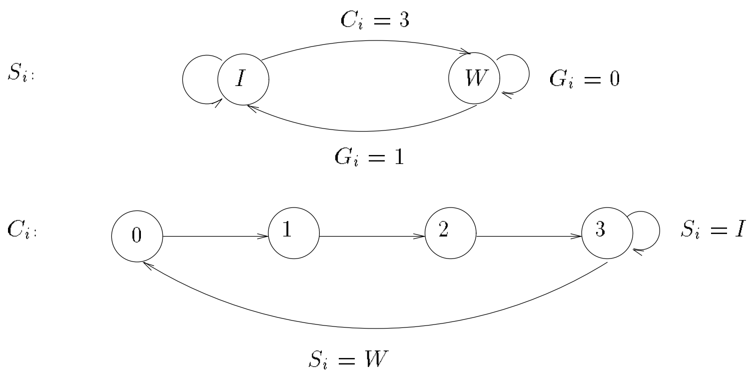


**Fig. 3.** The automata for process $i \in \{1, 2\}$, responsible for the variables $S_i$ and $C_i$. Unlabeled transitions can be made unconditionally.

The transition formula $\mathcal{T}_y$ appears below. Since we start with the most liberal controller that allows everything, we have $\mathcal{T}_x = \mathbf{true}$ and $\mathcal{T} = \mathcal{T}_y$ does not mention the variables $G'_1$ and $G'_2$.

$$
\begin{array}{l}
\mathcal{T}_y(S_1, C_1, S_2, C_2, G_1, G_2, S'_1, C'_1, S'_2, C'_2) = \\
\\
(S_1 = I \wedge (S'_1 = I \vee (C_1 = 3 \wedge S'_1 = W)) \vee \\
S_1 = W \wedge (G_1 = 1 \wedge S'_1 = I \vee G_1 = 0 \wedge S'_1 = W)) \wedge \\
(C_1 = 0 \wedge C'_1 = 1 \vee C_1 = 1 \wedge C'_1 = 2 \vee C_1 = 2 \wedge C'_1 = 3 \vee \\
C_1 = 3 \wedge (S_1 = I \wedge C'_1 = 3 \vee S_1 = W \wedge C'_1 = 0)) \wedge \\
(S_2 = I \wedge (S'_2 = I \vee (C_2 = 3 \wedge S'_2 = W)) \vee \\
S_2 = W \wedge (G_2 = 1 \wedge S'_2 = I \vee G_2 = 0 \wedge S'_2 = W)) \wedge \\
(C_2 = 0 \wedge C'_2 = 1 \vee C_2 = 1 \wedge C'_2 = 2 \vee C_2 = 2 \wedge C'_2 = 3 \vee \\
C_2 = 3 \wedge (S_2 = I \wedge C'_2 = 3 \vee S_2 = W \wedge C'_2 = 0))
\end{array}
$$

The performance specifications (winning conditions) are that no process will wait in $W$ more than 2 time units, and that mutual exclusion is satisfied, i.e.

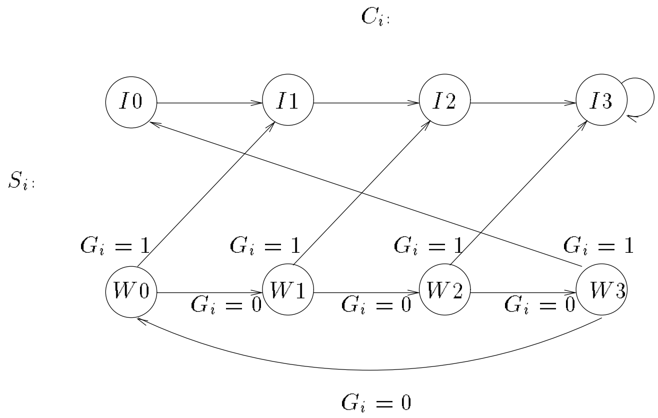


**Fig. 4.** The automaton $S_i \circ C_i$ for $i \in \{1, 2\}$. Note that the only "external" variable this automaton refers to is $G_i$.

that at least one of $G_1, G_2$ is zero. This can be expressed by the formula

$$\begin{array}{l} P(S_1, C_1, S_2, C_2, G_1, G_2) = \\ \neg(S_1 = W \wedge C_1 = 2 \vee S_2 = W \wedge C_2 = W \vee G_1 = 1 \wedge G_2 = 1) \end{array}$$

By performing the algorithm we obtain the following sequence of formulae:

$$\begin{array}{ll} P_0 = & \neg(S_1 = W \wedge C_1 = 2 \vee S_2 = W \wedge C_2 = W \vee G_1 = 1 \wedge G_2 = 1) \\ P_1 = P_0 \wedge & \neg(G_1 = 1 \wedge G_2 = 1) \\ P_2 = P_1 \wedge & \neg(S_1 = W \wedge C_1 = 1 \wedge G_1 = 0 \vee S_2 = W \wedge C_2 = 1 \wedge G_2 = 0) \\ P_3 = P_2 \wedge & \neg(S_1 = W \wedge C_1 = 0 \wedge G_1 = 0 \wedge S_2 = W \wedge C_2 = 0 \wedge G_2 = 0) \end{array}$$

The first iteration excludes violation of mutual exclusion. The second iteration excludes the case where for some $i$, $C_i = 1$ and $G_i = 0$ – in this case the next step will take us to a bad state $C_i = 2 \wedge S_i = W$. Finally the third iteration excludes the states where the two processes have moved to $W$ and the scheduler has not allocated the resource to at least one of them. The resulting controller, appearing in figure 5, is specificed by the formula $\mathcal{T}_x^*$:

$$\begin{array}{l} \mathcal{T}_x^*(G_1, G_2, S_1, C_1, S_2, C_2, G_1', G_2') = \\ \\ ((S_1 = W \wedge C_1 = 0) \Rightarrow G_1' = 1) \wedge \\ ((S_2 = W \wedge C_2 = 0) \Rightarrow G_2' = 1) \wedge \\ (G_1 = 0 \wedge S_1 = W \wedge C_1 = 3 \wedge \\ G_2 = 0 \wedge S_2 = W \wedge C_2 = 3) \Rightarrow (G_1' = 1 \vee G_2' = 1) \end{array}$$

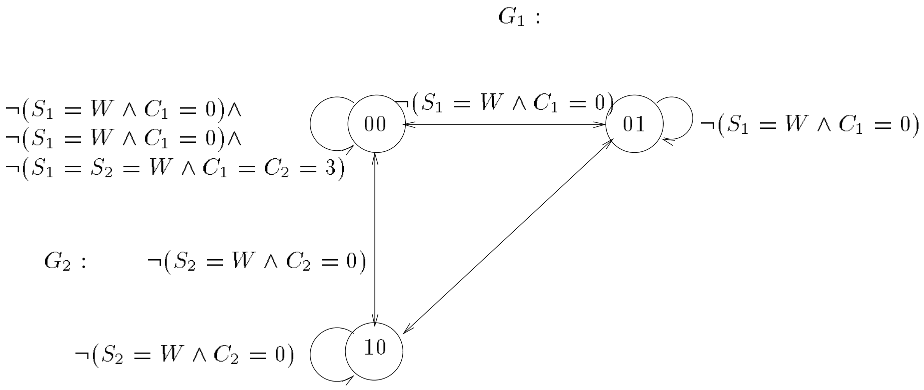


**Fig. 5.** The resulting scheduler obtained by restricting the complete 4-state automaton.

## 4 Timed Systems

### 4.1 Real-Time Games

In real-time games the outcome of the players' actions depend also on their timing because performing the same action "now" or "later" might have completely different consequences. For such games we take the model of *timed automata* [AD94], in which automata are equipped with auxiliary continuous variables called *clocks* which grow uniformly when the automaton is in some state. The clocks interact with the transitions by participating in pre-conditions (guards) for certain transitions and they are possibly reset when some transitions are taken.

In this continuous-time setting, a player might choose at a given moment to wait some time $t$ and *then* take a transition. Unlike purely-discrete games, it should consider not only what the adversary can do *after* this action but also the possibility that the latter will *not wait* for $t$ time, and perform an action at some $t' < t$. Thus the two-person game becomes a three-player game in which Time can interfere in favor of both other players.

While synthesizing a controller for timed automata one should be careful not letting any of the players win by "Zenonism", that is, by preventing the time from progressing as does the Tortoise in its race against Achilles.

### 4.2 Semantic Version

For the sake of readers not familiar with timed automata we start with an informal illustration of the behavior of these creatures. Consider the timed automaton of figure 6. It has two states and two clocks $z_1$ and $z_2$. Suppose it starts operating in the configuration $(q_1, 0, 0)$ (the two last coordinates denote the values of the clocks). Then it can stay at $q_1$ as long as the staying condition for $q_1$ is true, namely $z_1 < 2$. Meanwhile the values of the clocks grow and the set of all configurations reachable from $(q_1, 0, 0)$ without leaving $q_1$ is $\{(q_1, t, t) : 0 \leq t \leq 2\}$. However, after one second, the condition $z_1 \geq 1$ (the guard of the transition from $q_1$ to $q_2$) is satisfied and the automaton can move to $q_2$ while setting $z_2$ to 0. Hence the additional reachable configurations are $\{(q_2, t, 0) : 1 \leq t \leq 2\}$. Having entered $q_2$ in one of these configurations, the automaton can either stay there as long as $z_1 < 5 \wedge z_2 < 3$ or can unconditionally move to $(q_1, 0, 0)$, etc.

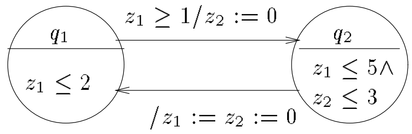


**Fig. 6.** A timed automaton.

Since the state-space of timed automata contains real-variables, we have an infinite-state automaton and a purely-semantic approach, where all states and transitions are enumerated, is impossible. We will use notation such as $T_{qq'}$ to denote the set of values in the clock space such that a transition from $q$ to $q' \neq q$ is possible ("guards"). Similarly, $T_{qq}$ denotes the set of clock values for which the automaton can stay in $q$ ("staying conditions"). In timed automata such sets are restricted to be $k$-polyhedral subsets of $(\mathbb{R}^+)^d$, that is, the class of sets obtainable by set-theoretic operations from half-spaces of the form $\{(v_1, \ldots, v_d) : v_i \leq c\}$, $\{(v_1, \ldots, v_d) : v_i < c\}$, $\{(v_1, \ldots, v_d) : v_i - v_j \leq c\}$ or $\{(v_1, \ldots, v_d) : v_i - v_j < c\}$ for some integer $c \in \{0, \ldots, k\}$, where $k$ is some positive integer. In fact, we can use $c \in \{0, r, 2r \ldots, kr\}$ for some positive rational $r$. These sets constitute the finite *region graph* [AD94] whose properties underly all analysis methods for timed automata. Since we model interaction between two automata, the guards and staying conditions of one automaton may depend, in addition, on the state of the other automaton and thus can be a union of sets of the form $\{q_i\} \times L_i$ with $L_i \subseteq (\mathbb{R}^+)^d$. We will call such sets $k$-polyhedral as well.

A function $f : \mathbb{R}^d \to \mathbb{R}^d$ is a *reset* function if it sets some of its arguments to 0 and leaves the others intact. We will use $R_{qq'}$ to denote the reset function associated with every pair of states. Without loss of generality we assume that there is only one transition associated with every ordered pair of states. Finally, for $z \in (\mathbb{R}^+)^d$ we use $z + t$ to denote $z + t \cdot \mathbf{1}$ where $\mathbf{1} = (1, 1, \ldots, 1)$ is a $d$-dimensional unit vector.

**Definition 3 (Timed Game Automaton).** *A timed game automaton is* $\mathcal{A} = (Q, Z, \delta_x, \delta_y, q_0, F)$ *such that*

- $Q = Q_x \times Q_y$ *is a discrete set,*
- $Z = Z_x \times Z_y = (\mathbb{R}^+)^d$ *is the clock space* ($Q \times Z$ *is the configuration space),*
- $\delta_x : Q \times Z \times \mathbb{R}^+ \rightarrow 2^{Q_x \times Z_x}$ *and*
- $\delta_y : Q \times Z \times \mathbb{R}^+ \rightarrow 2^{Q_y \times Z_y}$ *are the transition relations for the two players,*
- $q_0 \in Q$, *and*
- $F \subseteq Q \times Z$ *is a set of accepting configurations.*

*It is required that* $\delta_x$ *and* $\delta_y$ *admit the following decomposition: For every* $q_x, q'_x \in Q_x$ *and* $q_y, q'_y \in Q_y$, *let* $T_{q_x q'_x} \subseteq Q_y \times Z$ *and* $T_{q_y q'_y} \subseteq Q_x \times Z$ *be k-polyhedral sets and let* $R_{q_x q'_x} : Z_x \rightarrow Z_x$ *and* $R_{q_y q'_y} : Z_y \rightarrow Z_y$ *be reset functions. Then for every* $(q_x, z_x) \in Q_x \times Z_x$, $(q_y, z_y) \in Q_y \times Z_y$:

$$\delta_x((q_x, z_x, q_y, z_y), t) = \left\{ \begin{array}{l} (q'_x, z'_x) : \\ \forall t' \in [0, t)\, (q_y, z + t') \in T_{q_x q_x} \wedge \\ (q_y, z + t) \in T_{q_x q'_x} \wedge z'_x = R_{q_x q'_x}(z + t) \} \end{array} \right\}$$

$$\delta_y((q_y, z_y, q_x, z_x), t) = \left\{ \begin{array}{l} (q'_y, z'_y) : \\ \forall t' \in [0, t)\, (q_x, z + t') \in T_{q_y q_y} \wedge \\ (q_x, z + t) \in T_{q_y q'_y} \wedge z'_y = R_{q_y q'_y}(z + t) \} \end{array} \right\}$$

The meaning of $\delta_x((q_x, z_x, q_y, z_y), t)$ is the set of $Q_x \times Z_x$ configuration the first player can reach by waiting $t$ time, and then making *at most* one transition, given that the other player has done nothing meanwhile. The meaning of $\delta_y$ is symmetric. This allows us to define the predecessors operator rather simply:

**Definition 4 (Timed Controllable Predecessors).** *For a given timed game automaton* $\mathcal{A} = (Q, Z, \delta_x, \delta_y, q_0, F)$ *we define a function* $\pi : 2^{Q \times Z} \mapsto 2^{Q \times Z}$ *as*

$$\pi(F) = \left\{ \begin{array}{l} (q_x, z_x, q_y, z_y) : \\ \exists t \geq 0\, \exists (q'_x, z'_x) \in \delta_x((q_x, z_x, q_y, z_y), t) \\ \quad \forall t' \leq t\, \forall (q'_y, z'_y) \in \delta_y((q_y, z_y, q_x, z_x), t')\ (q_x, z_x + t', q'_y, z'_y) \in F \wedge \\ \quad\quad t = t' \Rightarrow (q'_x, z'_x, q'_y, z'_y) \in F \wedge \\ \quad t < \infty \Rightarrow q'_x \neq q_x \end{array} \right\}$$

Verbally this means that from the configuration $(q_x, z_x, q_y, z_y)$ the controller can force the game to stay in $F$ by taking a transition after waiting $t$ such that whatever transition the environment can take during the interval $[0, t)$ will not steer the game out of $F$. This is the essence of the definition (lines 2 and 3). Line 4 takes care of the special case where $t' = t$ and the two players make their transition simultaneously. The last line makes sure that the first player will not play Zenonist tricks, i.e. will try to prevent the progress of time without taking any transition. He is allowed to refrain from action only if it chooses $t = \infty$. We assume, initially, that $\delta_x$ is strongly non-Zeno, i.e. there is minimal period of time $d$ such that every two discrete transitions must be separated by an interval of at least $d$. This condition can be relaxed into a condition on cycles, but, as observed in the context of asynchronous circuits [MP95], you really do not need to interleave two discrete transitions in zero time.

An important fact about this operator (first stated explicitly in [MPS95], but really follows immediately from region-graph properties): *The class of $k$-polyhedral sets is closed under $\pi$.* This means that algorithm 1, when initiated with a $k$-polyhedral set $F$, is guaranteed to converge to a fixed point $F^*$ as the number of $k$-polyhedral sets is finite. The strategy $\delta_x^*$, which is a restriction of $\delta_x$, can be obtained by restricting the sets $T_{q_x q_x'}$ as follows:

$$T^*_{q_x q_x'} = T_{q_x q_x'} \cap \{(q_y, z_x, z_y) : \delta_x((q_x, z_x, q_y, z_y), 0) \subseteq F^*\}$$

This approach to real-time synthesis has been first presented in [MPS95]. Alternative approaches, e.g. [OW90], [BW93] are based on a discrete time model. Wong-Toi and Hoffmann [WH92] use timed automata, but then they discretize the system into an untimed automaton (essentially the region graph) and synthesize the controller using discrete symbolic methods.

### 4.3 Symbolic Version

For timed automata we need syntactic objects to represent subsets of the binary hypercube (discrete sets of states) as well as $k$-polyhedral subsets of the Euclidean space (which cannot be enumerated anyway). Systems of *linear inequalities* and sets of *vertices* are among the syntactic objects used to represent such sets. A very useful representation by Dill's *difference bounds matrices* [Dil89]. This representation is employed in the timed automata analysis tool KRONOS, developed at VERIMAG [DOY94], and it does not have a canonicity property (unless the set is convex). Following our presentation of the discrete case, we will not commit ourselves to this or that representation formalism but rather use arbitrary linear inequalities. The ideas in this section are adapted from the symbolic analysis methods for timed automata [HNSY94], [ACD93].

As before, we will use discrete sets of variables $X$ and $Y$, whose sets of valuation constitute the sets $Q_x$ and $Q_y$, and augment them with two sets of clock variables $C_x$ and $C_y$ ranging over the non-negative reals and whose valuations are the elements of $Z_x$ and $Z_y$ respectively. All variable will have primed versions, $X'$, $C_x'$, $Y'$ and $C_y'$ to represent next-states in transition formulae.

The game automaton is described by the formulae $T_x(X, C_x, Y, C_y, X', C_x')$ and $T_y(Y, C_y, X, C_x, Y', C_y')$ . Such formulae specify the instantaneous transition relation, namely the transitions that can be made in zero time. These formulae should also capture the "idle" transition and thus they contain a conjunction with $X = X' \Rightarrow C_x = C_x'$ and $Y = Y' \Rightarrow C_y = C_y'$ respectively. They are the syntactic equivalents of the $T_{qq'}$ of the semantic version.

The formulae $\mathcal{T}_x(X, C_x, Y, C_y, X', C_x', C_y', t)$ and $\mathcal{T}_y(Y, C_y, X, C_x, Y', C_y', C_x', t)$ which indicate what the two automata can make by waiting $t$ and doing at most one transition, are constructed from $T_x$ and $T_y$. They are the analogues of $\delta_x$ and $\delta_y$. Note that $t$ is a free variable in these formulae:

$$
\begin{array}{l}
\mathcal{T}_x(X, C_x, Y, C_y, X', C'_x, C'_y, t) = \\
\forall t' < t \;\; T_x(X, C_x + t', Y, C_y + t', X, C_x + t') \wedge \\
\qquad\qquad T_x(X, C_x + t, Y, C_y + t, X', C'_x) \\
\mathcal{T}_y(Y, C_y, X, C_x, Y', C'_y, C'_x, t) = \\
\forall t' < t \;\; T_y(Y, C_y + t', X, C_x + t', Y, C_y + t') \wedge \\
\qquad\qquad T_y(Y, C_y + t, X, C_x + t, Y', C'_y)
\end{array}
$$

For a formula $P(X, C_x, Y, C_y)$ denoting a set of configurations we define a predecessor formula as:

$$
\pi(P)(X, C_x, Y, C_y) =
$$

$$
\begin{array}{l}
\left(\begin{array}{l}
\exists t \geq 0 \, \exists X' \neq X \, \exists C'_x \;\; \mathcal{T}_x(X, C_x, Y, C_y, X', C'_x, t) \wedge \\
\quad \forall t' \leq t \, \forall Y' \, \forall C'_y \;\; (\mathcal{T}_y(Y, C_y, X, C_x, Y', C'_y, t') \Rightarrow \\
\qquad ((t = t' \wedge P(X', C'_x, Y', C'_y)) \vee (t < t' \wedge P(X, C_x + t, Y', C'_y)))
\end{array}\right) \vee \\
\left(\begin{array}{l}
\forall t \geq 0 \;\; \mathcal{T}_x(X, C_x, Y, C_y, X, C_x + t, t) \wedge \\
\quad \forall t' \leq t \, \forall Y' \, \forall C'_y \;\; (\mathcal{T}_y(Y, C_y, X, C_x, Y', C'_y, t') \Rightarrow P(X, C_x + t, Y', C'_y)
\end{array}\right)
\end{array}
$$

As before, we can apply algorithm 1 and converge to a formula $P^*$ from which we derive the controller as a restricted transition formula $T^*_x$. Note that this formula returns revised guards and staying conditions, something which is very useful for many design problems beside synthesis.

$$
T^*_x(X, C_x, Y, C_y, X', C'_x) =
$$

$$
\begin{array}{l}
\exists t \geq 0 \;\; \mathcal{T}_x(X, C_x, Y, C_y, X', C'_x, t) \wedge \\
\; \forall t' < t \, \forall Y' \, \forall C'_y \;\; (\mathcal{T}_y(X, C_x, Y, C_y, Y', C'_y, t') \Rightarrow P^*(X', C'_x, Y', C'_y)) \wedge \\
\; (t > 0 \wedge X = X') \vee (t = 0 \wedge X \neq X')
\end{array}
$$

This concludes the symbolic synthesis for timed systems.
**Remark 3**: Logically speaking, the language we use here is some decidable fragment of the first-order logic over the reals with constants, addition and order.

## 5 Example: A Real-Time Scheduler

We take again two processes and a scheduler. Each process (see figure 7) has one discrete variable $S_i$ which can be in one of three states: Idle, Waiting and Busy. It has two clocks: $c_i$ which is used to enforce minimal inter-arrival time of 3, and $d_i$ which measures service time – we assume that every process must spend one time unit at $B$ before coming back to $I$.

The scheduler is as in the discrete example. We do not start the synthesis, however, with the most general scheduler but rather enforce a strong non-Zeno condition using the variable $z$. The scheduler has to spend at least 0.5 time at

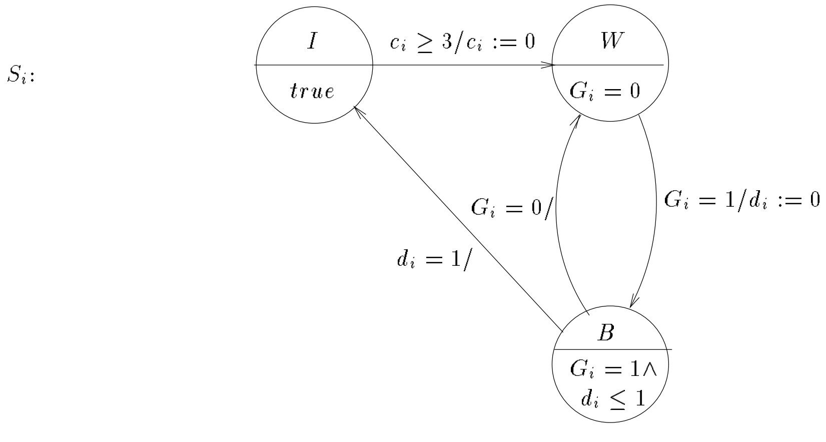


**Fig. 7.** A timed automaton for $S_i$, $c_i$ and $d_i$, $i \in \{1, 2\}$.

every state (see figure 8). For simplicity we exclude the bad state $G_1 = 1 \wedge G_2 = 1$ from the initial scheduler. The corresponding transition formulae are:

$$T_y(S_1, c_1, d_1, S_2, c_2, d_2, G_1, G_2, z, S_1', c_1', d_1', S_2', c_2', d_2') =$$

$$\begin{aligned}
&\big(S_1 = I \wedge S_1' = I \wedge c_1' = c_1 \wedge d_1' = d_1 \vee \\
&\ S_1 = I \wedge c_1 \geq 3 \wedge S_1' = W \wedge c_1' = 0 \wedge d_1' = d_1 \vee \\
&\ S_1 = W \wedge G_1 = 0 \wedge S_1' = W \wedge c_1' = c_1 \wedge d_1' = d_1 \vee \\
&\ S_1 = W \wedge G_1 = 1 \wedge S_1' = B \wedge c_1' = c_1 \wedge d_1' = 0 \vee \\
&\ S_1 = B \wedge G_1 = 1 \wedge d_1 < 1 \wedge S_1' = B \wedge c_1' = c_1 \wedge d_1' = d_1 \vee \\
&\ S_1 = B \wedge G_1 = 0 \wedge S_1' = W \wedge c_1' = c_1 \wedge d_1' = d_1 \vee \\
&\ S_1 = B \wedge d_1 = 1 \wedge S_1' = I \wedge c_1' = c_1 \wedge d_1' = d_1\big) \wedge \\
&\big(S_2 = I \wedge S_2' = I \wedge c_2' = c_2 \wedge d_2' = d_2 \vee \\
&\ S_2 = I \wedge c_2 \geq 3 \wedge S_2' = W \wedge c_2' = 0 \wedge d_2' = d_2 \vee \\
&\ S_2 = W \wedge G_2 = 0 \wedge S_2' = W \wedge c_2' = c_2 \wedge d_2' = d_2 \vee \\
&\ S_2 = W \wedge G_2 = 1 \wedge S_2' = B \wedge c_2' = c_2 \wedge d_2' = 0 \vee \\
&\ S_2 = B \wedge G_2 = 1 \wedge d_2 < 1 \wedge S_2' = B \wedge c_2' = c_2 \wedge d_2' = d_2 \vee \\
&\ S_2 = B \wedge G_2 = 0 \wedge S_2' = W \wedge c_2' = c_2 \wedge d_2' = d_2 \vee \\
&\ S_2 = B \wedge d_2 = 1 \wedge S_2' = I \wedge c_2' = c_2 \wedge d_2' = d_2\big)
\end{aligned}$$

$$T_x(G_1, G_2, z, S_1, c_1, d_1, S_2, c_2, d_2, G_1', G_2', z') =$$

$$\begin{aligned}
&(G_1 = 0 \vee G_2 = 0) \wedge (G_1' = 0 \vee G_2' = 0) \wedge \\
&\ G_1' = G_1 \wedge G_2' = G_2 \wedge z' = z \vee \\
&\ (G_1' \neq G_1 \vee G_2' \neq G_2) \wedge z \geq 0.5 \wedge z' = 0
\end{aligned}$$

The winning condition is

$$P(S_1, c_1, d_1, S_2, c_2, d_2, G_1, G_2, z) = \neg(S_1 \neq I \wedge c_1 > 3 \vee S_2 \neq I \wedge c_2 > 3)$$

The iteration for the winning states goes as follows:

$$\begin{array}{ll}
P_0 = & \neg(S_1 \neq I \wedge c_1 > 3 \vee S_2 \neq I \wedge c_2 > 3) \\
P_1 = P_0 \wedge & \neg(S_1 = B \wedge z < 0.5 \wedge c_1 - d_1 > 2 \wedge c_1 - z > 2.5 \vee \\
& \quad S_2 = B \wedge z < 0.5 \wedge c_2 - d_2 > 2 \wedge c_2 - z > 2.5 \vee \\
& \quad S_1 = W_1 \wedge z < 0.5 \wedge c_1 - z > 2.5 \vee \\
& \quad S_2 = W_2 \wedge z < 0.5 \wedge c_2 - z > 2.5) \\
P_2 = P_1 \wedge & \neg(S_1 = W \wedge c_1 > 2.5 \vee \\
& \quad S_2 = W \wedge c_2 > 2.5 \vee \\
& \quad S_1 = B \wedge c_1 - d_1 > 2 \wedge c_1 > 2.5 \vee \\
& \quad S_2 = B \wedge c_2 - d_2 > 2 \wedge c_2 > 2.5) \\
P_3 = P_2 \wedge & \neg(S_1 = W \wedge z < 0.5 \wedge c_1 - z > 2 \vee \\
& \quad S_2 = W \wedge z < 0.5 \wedge c_2 - z > 2 \vee \\
& \quad S_1 = B \wedge z < 0.5 \wedge c_1 - d_1 > 2 \wedge c_1 - z > 2 \vee \\
& \quad S_2 = B \wedge z < 0.5 \wedge c_2 - d_2 > 2 \wedge c_2 - z > 2) \\
P_4 = P_3 \wedge & \neg(S_1 = W \wedge c_1 > 2 \vee \\
& \quad S_2 = W \wedge c_2 > 2 \vee \\
& \quad S_1 = B \wedge c_1 - d_1 > 2 \vee \\
& \quad S_2 = B \wedge c_2 - d_2 > 2) \\
P_5 = P_4 \wedge & \neg(S_1 = W \wedge z < 0.5 \wedge c_1 - z > 1.5 \vee \\
& \quad S_2 = W \wedge z < 0.5 \wedge c_2 - z > 1.5) \\
P_6 = P_5 \wedge & \neg(S_1 = S_2 = W \wedge c_1 > 1.5 \wedge c_2 > 1.5 \vee \\
& \quad S_1 = W \wedge S_2 = B \wedge c_2 > 1.5 \wedge c_1 - d_2 > 1 \vee \\
& \quad S_1 = B \wedge S_2 = W \wedge c_1 > 1.5 \wedge c_2 - d_1 > 1) \\
P_7 = P_6 \wedge & \neg(S_1 = S_2 = W \wedge z < 0.5 \wedge c_1 - z > 1 \wedge c_2 - z > 1 \vee \\
& \quad S_1 = W \wedge S_2 = B \wedge z < 0.5 \wedge c_2 - z > 1 \wedge c_1 - d_2 > 1 \vee \\
& \quad S_1 = B \wedge S_2 = W \wedge z < 0.5 \wedge c_1 - z > 1 \wedge c_2 - d_1 > 1) \\
P_8 = P_7 \wedge & \neg(S_1 = S_2 = W \wedge c_1 > 1 \wedge c_2 > 1) \\
P_9 = P_8 \wedge & \neg(S_1 = S_2 = W \wedge z < 0.5 \wedge c_1 - z > 0.5 \wedge c_2 - z > 0.5)
\end{array}$$

The first 5 iterations deal with failures not related to interaction of the two processes. At each step a part of "size" 0.5 is subtracted from the set of winning configurations (this is due to the anti-Zeno constant 0.5 of the initial controller). The condition $\neg P_4$ means that it is too late to serve a process. The condition $P_5$ and the first line of $P_6$ exclude a situation when we are unable to start a service in due time because of switching delays. The last 3 iterations correspond to failures related to interaction of the processes. This is done in steps of "size" 0.5. The synthesized controller is expressed by the formula:[7]

[7] We leave the drawing of the controller to the reader.

$$
\begin{aligned}
&T_x^*(G_1, G_2, z, S_1, c_1, d_1, S_2, c_2, d_2, G_1', G_2', z') = \\
&\\
&P^*(S_1, c_1, d_1, S_2, c_2, d_2, G_1, G_2, z) \wedge \\
&T_x(G_1, G_2, z, S_1, c_1, d_1, S_2, c_2, d_2, G_1', G_2', z') \wedge \\
&(S_1 \neq I \wedge S_2 \neq I \wedge c_1 > 0.5 \wedge c_2 > 0.5 \wedge (G_1 = 1 \vee G_2 = 1)) \Rightarrow (G_1' = 1 \vee G_2' = 1) \wedge \\
&(S_1 = B \wedge S_2 = W \wedge c_1 > 1 \wedge c_2 > 1) \Rightarrow G_1' = 1 \wedge \\
&(S_1 = W \wedge S_2 = B \wedge c_1 > 1 \wedge c_2 > 1) \Rightarrow G_2' = 1 \wedge \\
&(S_1 = S_2 = W \wedge c_1 = c_2 = 1) \Rightarrow (G_1' = 1 \vee G_2' = 1) \wedge \\
&(S_1 = S_2 = W \wedge c_1 > 1 \wedge c_2 = 1) \Rightarrow G_1' = 1 \wedge \\
&(S_1 = W = S_2 = W \wedge c_1 = 1 \wedge c_2 > 1) \Rightarrow G_2' = 1 \wedge \\
&(S_1 = B \wedge C_1 > 1.5) \Rightarrow G_1' = 1 \wedge \\
&(S_2 = B \wedge C_2 > 1.5) \Rightarrow G_2' = 1 \wedge \\
&(S_1 = W \wedge G_2 = 1 \wedge c_1 > 1.5) \Rightarrow (G_1' = 1 \vee G_2' = 1) \wedge \\
&(S_2 = W \wedge G_1 = 1 \wedge c_2 > 1.5) \Rightarrow (G_1' = 1 \vee G_2' = 1) \wedge \\
&(S_1 = W \wedge G_1 = G_2 = 0 \wedge c_1 > 1.5) \Rightarrow G_2' = 0 \wedge \\
&(S_2 = W \wedge G_1 = G_2 = 0 \wedge c_2 > 1.5) \Rightarrow G_1' = 0 \wedge \\
&(S_1 \neq I \wedge c_1 \geq 2) \Rightarrow G_1' = 1 \wedge \\
&(S_2 \neq I \wedge c_2 \geq 2) \Rightarrow G_2' = 1
\end{aligned}
$$

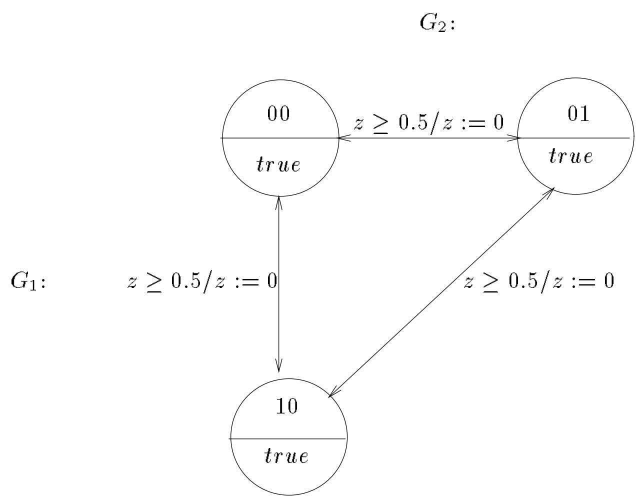


**Fig. 8.** The initial scheduler.

## 6 Discussion

We have demonstrated how the synthesis problem for real-time systems can be formulated and solved symbolically. As one potential application let us mention the delay analysis of asynchronous circuits. It has been shown [MP95] that every such circuit can be modeled as a timed automaton such that every wire requires one Boolean variable and one clock variable. The guards and staying conditions for the automaton are derived from the delay characteristics of the gates and from constraints on the variability of the input signals. Using the method described in this paper, we can formulate a game by considering the inputs as environmental variables ($Y$) and the outputs of the gates as controllable variables. By solving the synthesis problem for some winning condition (specification), we obtain the minimal timing requirements the gates need to satisfy in order to meet the specification. If we switch the roles and treat the gate variables as fixed and the inputs as controllable, we solve the opposite problem: what is the largest class of input signals against which the circuit will behave properly.

Without a convincing implementation, all this remains, of course, wishful thinking. For the discrete part of the system, there is no a-priori reason to believe that the practical hardness of synthesis is much bigger than that of verification. The main challenge is to find more efficient data-structures and algorithms for treating $k$-polyhedral sets, and combining them with the discrete ones.

**Acknowledgment**: This work grew out of discussions with J. Sifakis. We thank A. Bouajjani and Y. Lakhneche for commenting on previous drafts of the paper and A. Nerode for his help in stopping Time.

## References

[AD94] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 183–235, 1994.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill, Model Checking in Dense Real Time, Information and Computation 104, 2–34, 1993.

[ALW89] M. Abadi, L. Lamport, and P. Wolper, Realizable and Unrealizable Concurrent Program Specifications. In *Proc. 16th ICALP*, volume 372 of Lect. Notes in Comp. Sci., pages 1–17. Springer-Verlag, 1989.

[BHG+93] S. Balemi, G.J. Hoffmann, P. Gyugyi, H. Wong-Toi and G.F. Franklin, Supervisory Control of a Rapid Thermal Multiprocessor, *IEEE Trans. on Automatic Control* 38, 1040–1059, 1993.

[Bry86] R.E. Bryant, Graph-based Algorithms for Boolean Function Manipulation, *IEEE Trans. on Computers* C-35, 677-691, 1986.

[BCM+93] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, Symbolic Model-Checking: $10^{20}$ States and Beyond, *Proc. LICS'90*, Philadelphia, 1990.

[BW93] B.A. Brandin and W.M. Wonham, Supervisory Control of Timed Discrete-event Systems, *IEEE Transactions on Automatic Control*, 39, 329-342, 1994.

[BL69] J.R. Büchi and L.H. Landweber, Solving Sequential Conditions by Finite-state Operators, *Trans. of the AMS* 138, 295-311, 1969.

[Chu63] A. Church, Logic, Arithmetic and Automata, in *Proc. of the Int. Cong. of Mathematicians 1962*, 23-35, 1963.

[DOY94] C. Daws, A. Olivero and S. Yovine, Verifying ET-LOTOS Programs with KRONOS, *Proc. FORTE'94*, Bern, 1994.

[DV94] A. Deshpande and P. Varaiya, Control of Discrete Event Systems in Temporal Logic, Unpublished manuscript, 1994.

[Dil89] D.L. Dill, Timing Assumptions and Verification of Finite-State Concurrent Systems, in J. Sifakis (Ed.), *Automatic Verification Methods for Finite State Systems*, volume 407 of Lect. Notes in Comp. Sci., Springer, 1989.

[EC82] E.A. Emerson and E.M. Clarke, Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons, *Science of Computer Programming* 2, 241-266, 1982.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic Model-checking for Real-time Systems, *Information and Computation* 111, 193–244, 1994.

[HW92-a] G. Hoffmann and H. Wong-Toi, Symbolic synthesis of supervisory controllers, *Proc. of the 1992 American Control Conference*, 2789-2793, 1992.

[HW92-b] G. Hoffmann and H. Wong-Toi, Symbolic Supervisory Synthesis for the Animal Maze, *Proc. of Workshop on Discrete Event Systems*, 189-197, Birkhauser Verlag, 1992.

[MPS95] O. Maler, A. Pnueli and J. Sifakis, On the Synthesis of Discrete Controllers for Timed Systems, In E.W. Mayr and C. Puech (Eds.), Proc. STACS '95, volume 900 of Lect. Notes in Comp. Sci., 229-242, Springer-Verlag, 1995.

[MP95] O. Maler and A. Pnueli, Timing Analysis of Asynchronous Circuits using Timed Automata, *Proc. Charme'95*, to appear, 1995.

[MWa80] Z. Manna and R.J. Waldinger, A Deductive Approach to Program Synthesis, *ACM Trans. of Prog. Lang. and Sys.* 2, 90-121, 1980.

[MWo84] Z. Manna and P. Wolper, Synthesis of Communication Processes from Temporal Logic Specifications, *ACM Trans. of Prog. Lang. and Sys.* 6, 68-93, 1984.

[McM93] K.L. McMillan, *Symbolic Model-Checking: an Approach to the State-Explosion problem*, Kluwer, 1993.

[NYY92] A. Nerode, A. Yakhnis and V. Yakhnis, Concurrent Programs as Strategies in Games, in Y. Moschovakis (Ed.), *Logic From Computer Science*, Springer, 1992.

[NM44] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, 1944.

[OW90] J.S. Ostroff and W.M. Wonham, A Framework for Real-time Discrete Event Control, *IEEE Trans. on Automatic Control* 35, 386-397, 1990.

[PR89-a] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module, In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[PR89-b] A. Pnueli and R. Rosner. On the Synthesis of an Asynchronous Reactive Module, In *Proc. 16th ICALP*, volume 372 of Lect. Notes in Comp. Sci., 653–671, 1989.

[RW89] P.J. Ramadge and W.M. Wonham, The Control of Discrete Event Systems, *Proc. of the IEEE* 77, 81-98, 1989.

[TW94] J.G. Thistle and W.M. Wonham, Control of Infinite Behavior of Finite Automata, *SIAM J. of Control and Optimization* 32, 1075-1097, 1994.

[Tho94] W. Thomas, On the Synthesis of Strategies in Infinite Games, In E.W. Mayr and C. Puech (Eds.), *Proc. STACS '95*, volume 900 of Lect. Notes in Comp. Sci., 1-13, Springer-Verlag, 1995.

[TB73] B.A. Trakhtenbrot and Y.M. Barzdin, *Finite Automata: Behavior and Synthesis*, North-Holland, Amsterdam, 1973.

[WD91] H. Wong-Toi and D.L. Dill, Synthesizing Processes and Schedulers from Temporal Specifications, in E.M. Clarke and R.P. Kurshan (Eds.), *Computer-Aided Verification '90*, DIMACS Series, AMS, 177-186, 1991.

[WH92] H. Wong-Toi and G. Hoffmann, The Control of Dense Real-Time Discrete Event Systems, Technical report STAN-CS-92-1411, Stanford University, 1992.