An Accelerated Algorithm for 3-Color Parity Games with an Application to Timed Games*

Luca de Alfaro[1] and Marco Faella[2]

[1] Department of Computer Engineering, University of California, Santa Cruz, USA
[2] Dipartimento di Scienze Fisiche, Università di Napoli "Federico II", Italy

Abstract. Three-color parity games capture the disjunction of a Büchi and a co-Büchi condition. The most efficient known algorithm for these games is the progress measures algorithm by Jurdziński. We present an acceleration technique that, while leaving the worst-case complexity unchanged, often leads to considerable speed-ups in games arising in practice. As an application, we consider games played in discrete real time, where players should be prevented from stopping time by always choosing moves with delay zero. The time progress condition can be encoded as a three-color parity game. Using the tool TICC as a platform, we compare the performance of a BDD-based symbolic implementation of the progress measure algorithm with acceleration, and of the symbolic implementation of the classical μ-calculus algorithm of Emerson and Jutla.

1 Introduction

The parity acceptance condition for automata and games enjoys many interesting properties. Every ω-regular language can be recognized by a deterministic parity automaton [20]. The parity accepting condition is closed under complementation, and games with parity accepting conditions admit memoryless optimal strategies for both players. Moreover, parity games have received a great deal of attention due to their equivalence to the model checking of the modal μ-calculus. The complexity of this class of games is known to be in NP $\cap$ co-NP [11], and even in UP $\cap$ co-UP [14].

We are especially interested in parity conditions with three colors. which can express the disjunction of Büchi and co-Büchi conditions. As we shall see, 3-color parity games occur in the solution of *timed* games. For 3-color parity games, the algorithm with the best worst-case complexity is the progress measure algorithm of [13]. In this paper, we present an acceleration technique that greatly improves the performance of this algorithm in many cases, while retaining its worst-case behavior. We then show how the algorithm can be implemented symbolically, and how it compares in performance with more traditional, μ-calculus based algorithms [10].

We consider parity games with colors 0, 1, 2, where the goal of Player 1 is to ensure that the minimum color visited infinitely often is even. The progress measure algorithm works by updating a function assigning an integer value to each state of the game, called the *measure* of that state. The measure of each state starts at zero; each iteration

* This research was supported in part by the NSF grant CCR-0132780. The second author was supported by a scholar mobility program from Università di Napoli "Federico II".

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 108–120, 2007.
© Springer-Verlag Berlin Heidelberg 2007

of the algorithm can either increase the measure at a state, or leave it unchanged. If the measure of a state exceeds the number n_1 of 1-color states, the state is losing for Player 1. The algorithm stops when the progress measure reaches a fixpoint. Even in the best case, the algorithm needs a number of iterations proportional to n_1.

We propose an acceleration scheme based on the following result. Suppose that, at a certain step of the algorithm, we find an integer k such that no state has measure k, but some states have measure greater than k. Call k a "gap" in the measure. We prove that all states having measure greater than the gap k are losing for Player 1; they can be immediately be assigned measure $n_1 + 1$. This enables us to solve many 3-color parity games in much fewer than n_1 iterations; as we shall see, this acceleration is especially effective for timed games.

In the second part of this paper, we show how the acceleration technique for three-color parity games can be applied to timed games, leading to efficient symbolic algorithms. Timed games are games played in such a way as to make explicit reference to the passage of time [15,3]. Generally, the players of a timed game specify in their moves both the action they want to execute and the time at which they want to execute it. Moreover, in the literature such games are usually played on timed-automata-like arenas, so that the game state is also made time-aware by the presence of clocks. As for standard games, the objective for a player is to obtain a game run belonging to a given set of desired runs, called *goal*. Common goals for timed games include reaching a given set of states (reachability) or staying forever in a given set of states (safety).

Most formulations of timed games allow players to "stop the progress of time" by proposing zero, or converging, time delays. Obviously, these physically impossible behaviors must be ruled out in order not to obtain artificial solutions to the game. Previous approaches differ in how they deal with this problem. Some papers make sure that non-physical behaviors cannot arise in the first place, by placing structural restrictions on the games they are willing to treat [3,12]. Other papers force a player to ensure time divergence [15] as a prerequisite for winning, with the result that players are precluded victory in many games where the goal can be achieved only with some delay. Still other papers ignore the issue, so that their solutions work only for sub-classes of games, such as safety [17,2] or reachability [5] games.

A technique that does not restrict the type of games that can be tackled is advocated in [9,7,1]. The approach distinguishes between the original *goal* of the game and the *winning condition*, which is a suitable modification of the goal ensuring that time-blocking strategies are not convenient for either player. The winning condition states, roughly, that in addition to achieving the goal, a player must ensure that either time diverges, or that the blame for stopping time lies with the other player [2,7]. As we shall see, for safety and reachability games, such winning condition can be captured by a 3-color deterministic parity automaton. Thus, solving safety and reachability timed games involves solving 3-color parity games.

We consider timed games played in discrete time, and we present a symbolic implementation of the progress measure algorithm, based on symbolic methods for updating the progress measure, finding the gaps, and achieving acceleration. We show that the acceleration is fundamental in achieving an efficient implementation of the progress measure algorithm: in the examples we tested, we achieved speed-up factors of several

hundreds. We also compare the performance of the resulting algorithms with the classical μ-calculus-based fixpoint algorithm of [10]. The running times of the two algorithms were, in our experiments, within a factor of two of each other, with the classical μ-calculus algorithm generally being the fastest. However, our results are not conclusive, since minor implementation details, such as the choice of variable ordering and differences in the encoding of the game transition relation, seem to have a large effect on the performance of the algorithms.

2 Algorithms for 3-Color Parity Games

For an integer $d > 0$, a *parity game* with d *colors* is a tuple (S_1, S_2, E, c), where S_1 and S_2 are the finite sets of states of Player 1 and Player 2, respectively. We require $S_1 \cap S_2 = \emptyset$ and we set $S = S_1 \cup S_2$. $E \subseteq S^2$ is the set of edges, and $c : S \to \{0, 1, \ldots, d-1\}$ is a function assigning a color to each state. For all $i = 0, \ldots, d-1$, we set $C_i = c^{-1}(i)$. Moreover, let $n = |S|$ and $m = |E|$. A *strategy* for player $i \in \{1, 2\}$ is a function $\pi : S^* \to S$ such that, for all $\sigma \in S^*$, if the last state of σ is $s \in S_i$ then $(s, \pi(\sigma)) \in E$. Let Π^1 and Π^2 denote the sets of strategies of Player 1 and Player 2, respectively. A *trace* is an infinite path in the directed graph (S, E). Given $s \in S$, $\pi^1 \in \Pi^1$ and $\pi^2 \in \Pi^2$, the *outcome* $\delta(s, \pi^1, \pi^2)$ of π^1 and π^2 from s is the unique trace $s_0 s_1 \ldots$ such that $s_0 = s$ and for all $j > 0$, $s_j = \pi^i(s_0 s_1 \ldots s_{j-1})$ if and only if $s_{j-1} \in S_i$. We say that a strategy $\pi^1 \in \Pi^1$ is *winning* from state s iff for all $\pi^2 \in \Pi^2$, the smallest color that appears infinitely often in $\delta(s, \pi^1, \pi^2)$ is even. We denote by Win^1 the set of states from which Player 1 has a winning strategy.

In the following, we examine two algorithms for solving parity games with three colors. We consider a fixed parity game (S_1, S_2, E, c) with three colors. When discussing the complexity of the algorithms, we assume an adjacency list representation for the game.

2.1 Emerson-Jutla's μ-Calculus Algorithm

From [10], parity games can be solved using a fixpoint computation involving the so-called *controllable predecessor operators*.

Definition 1 (Controllable Predecessor Operator). For a set of states $X \subseteq S$, $Cpre^1(X)$ yields all states from which Player 1 can force the game into X in one step. Formally,

$$Cpre^1(X) = \{s \in S_1 \mid \exists (s,t) \in E \,.\, t \in X\} \;\cup\; \{s \in S_2 \mid \forall (s,t) \in E \,.\, t \in X\}.$$

For parity games with three colors, the set of winning states Win^1 can be characterized using the following formula [10], written in μ-calculus notation:

$$Win^1 = \nu Z.\mu Y.\nu X.\Big[(Cpre^1(X) \cap C_2) \;\cup\; (Cpre^1(Y) \cap C_1) \;\cup\; (Cpre^1(Z) \cap C_0)\Big].$$

Such fixpoint can be computed by Picard iteration, using three nested loops; we will refer to this algorithm as the *EJ algorithm*. An enumerative implementation of this

algorithm takes time $O(m \cdot n^2)$: the inner loop can be computed in time $O(m)$ (the computation is analogous to the one used for solving safety games), while the outer loops can be performed at most n times each. On the other hand, a symbolic implementation requires time $O(m \cdot n^3)$, since the computation of $Cpre^1$ takes time $O(m)$, and it is performed $O(n^3)$ times.

2.2 Jurdziński's Progress Measure Algorithm

An alternative algorithm for computing Win^1 is the *progress measure* algorithm from [13]. For three-color parity games, this algorithm has the best worst-case complexity of all known algorithms. Let $n_1 = |C_1|$ and $M = \{0, 1, \ldots, n_1+1\}$. A *progress measure* is a function $\rho : S \to M$. The algorithm proceeds by building a monotonically increasing sequence $(\rho_i)_{i \geq 0}$ of progress measures, until a fixpoint is reached.

For $\alpha \in M$ and $j \in \{0, 1, 2\}$, we define

$$Progr(\alpha, j) = \begin{cases} 0 & \text{if } j = 0 \text{ and } \alpha < n_1 + 1, \\ \alpha + 1 & \text{if } j = 1 \text{ and } \alpha < n_1 + 1, \\ \alpha & \text{otherwise.} \end{cases} \tag{1}$$

We have $\rho_0(s) = 0$ for all $s \in S$. For all $i \geq 0$, the update from ρ_i to ρ_{i+1}, called *lift*, is dictated by the following rule, where $a \sqcup b$ denotes $\max\{a, b\}$.

$$\rho_{i+1}(s) = \rho_i(s) \sqcup \begin{cases} \min_{(s,t) \in E} Progr(\rho_i(t), c(s)) & \text{if } s \in S_1, \\ \max_{(s,t) \in E} Progr(\rho_i(t), c(s)) & \text{if } s \in S_2. \end{cases} \tag{2}$$

Denoting ρ^* the fixpoint of the sequence $(\rho_i)_{i \geq 0}$, the set of winning states Win^1 is characterized by:

$$Win^1 = \{s \in S \mid \rho^*(s) < n_1 + 1\}.$$

Given ρ_i, computing ρ_{i+1} requires time $O(m)$. Since at each step the measure of at least one state increases by at least one, our formulation of the algorithm requires time $O(m \cdot n^2)$. Notice that, by applying the complexity bound cited in Theorem 11 of [13], we obtain a time complexity of $O(m \cdot n)$. The difference is due to the fact that our formulation of the algorithm updates the progress measures for all states at once, while the original algorithm only updates the progress measureone state at a time. Moreover, the $O(m \cdot n)$ complexity can only be achieved if we can somehow efficiently determine which states need to be lifted. This presumably requires bookkeeping at every state and lift propagation algorithms, that are incompatible with the symbolic implementation we discuss in Section 5.

2.3 Gap Algorithm

We present the *gap acceleration technique* for the progress measure algorithm of Jurdziński. The resulting algorithm, which we call the *gap algorithm,* is often much faster than the original progress measure algorithm, while retaining its worst case complexity.

Informally, the idea is as follows. At any step of the algorithm, let k be an integer in $\{0, 1, \ldots, n_1\}$ such that no state has progress measure k, but some states have progress

measure greater than k. We call such a value of k a "gap". We show that all states with progress measure greater than k are losing. Therefore, we can immediately set their measure to $n_1 + 1$, thus accelerating the convergence of the algorithm. In practice, after each update of the progress measure, we will seek the minimum gap k, and we will set to $n_1 + 1$ the progress measure of all states having progress measure above the gap k. The correctness of this optimization is proved by the following lemma and theorem.

Lemma 1. *For all $i \geq 0$ and $k > 0$, let $Z_i^k = \{s \in S \mid \rho_i(s) \geq k\}$. Then, for all $s \in Z_i^k$, Player 2 can enforce at least $\rho_i(s)$ visits to C_1. Moreover, only states in Z_i^k are visited before the first visit to C_1.*

Proof. Notice that, for all $i \leq j$, it holds $Z_i^k \subseteq Z_j^k$. We proceed by induction on i. For $i = 0$, the statement is trivially true, since $\rho_0(s) = 0$ for all $s \in S$. For $i > 0$, we distinguish the following cases.

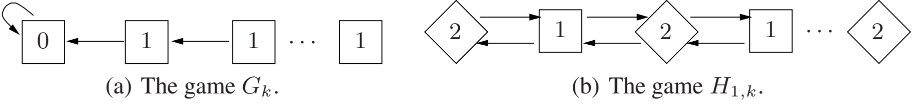
- $s \in C_2$. If $s \in S_1$ (resp. $s \in S_2$), then all (resp. at least one) of the successors t of s are such that $\rho_{i-1}(t) \geq \rho_i(s) \geq k$; thus, $t \in Z_{i-1}^k$. By inductive hypothesis, Player 2 can enforce from t at least k visits to C_1, and the first visit occurs before Z_{i-1}^k is left. Since $Z_{i-1}^k \subseteq Z_i^k$, the thesis applies to s.
- $s \in C_1$. If $s \in S_1$ (resp. $s \in S_2$), then all (resp. at least one) of the successors t of s are such that $\rho_{i-1}(t) \geq \rho_i(s) - 1 \geq k - 1$; thus, $t \in Z_{i-1}^{k-1}$. By inductive hypothesis, Player 2 can enforce from t at least $k-1$ more visits to C_1. Therefore, Player 2 can enforce from s at least k visits to C_1. The first visit to C_1 being s itself, it occurs trivially without leaving Z_i^k.
- $s \in C_0$. Then, $\rho_i(s) = 0$ or $\rho_i(s) = n_1 + 1$. If $\rho_i(s) = 0$, the result is trivial. If $\rho_i(s) = n_1 + 1$, the result follows by noticing that, if $s \in S_1$ (resp. $s \in S_2$), then all (resp. at least one) of the successors t of s are such that $\rho_i(t) = n_1 + 1$. ∎

Theorem 1. *Given $i \geq 0$ and $k > 0$, assume that $\rho_i^{-1}(k-1) = \emptyset$. Then, each state $s \in Z_i^k$ is a losing state for Player 1.*

Proof. First, we show that, starting from s, Player 2 can enforce infinitely many visits to C_1, while remaining in Z_i^k at all times. In particular, if $s \in Z_i^k \cap C_2$, by Lemma 1, Player 2 has a strategy to reach C_1 while staying in Z_i^k at all times. If instead $s \in Z_i^k \cap C_1$, Player 2 can enforce that the next state is still in Z_i^k, as the following argument shows. If $s \in S_1$, all successors t of s satisfy $\rho_i(t) \geq \rho_{i-1}(t) \geq \rho_i(s) - 1 \geq k - 1$. However, since it cannot be $\rho_i(t) = k - 1$, it must be $\rho_i(t) \geq k$, and so $t \in Z_i^k$. Finally, if $s \in S_2$, let t be the successor that maximizes $Progr(\rho_{i-1}(t), c(s))$. We have $\rho_i(t) \geq \rho_{i-1}(t) = \rho_i(s) - 1 \geq k - 1$. As before, it must be $\rho_i(t) \geq k$ and so $t \in Z_i^k$.

It remains to be proved that, while visiting C_1 infinitely often, C_0 is not visited infinitely often. Notice that for all $s \in Z_i^k \cap C_0$, it holds $\rho_i(s) = n_1 + 1$. Therefore, if a state in C_0 is ever visited, it is a losing state for Player 1. ∎

It is not hard to devise an example where the gap acceleration does not decrease the total number of iterations. For all $k > 0$, consider the game G_k in Figure 1(a). States drawn as "◇" belong to S_1 while those drawn as "□" belong to S_2. The numbers in the states represent their color. The game G_k is a chain of k states of color one, leading to a sink state of color zero. The lock-step algorithm requires k global lifts to reach the



(a) The game G_k.

(b) The game $H_{1,k}$.

Fig. 1. Two game families illustrating different performance gains offered by the gap acceleration

fixpoint. During the process, the progress measure exhibits no gaps, thus neutralizing the proposed acceleration technique.

On the other hand, the gap acceleration technique can be responsible for an unbounded speed-up compared to both the original algorithm and our lock-step formulation of it. For all $k > 0$, consider the game $H_{1,k}$ from [13], depicted in Figure 1(b). The game is essentially a bi-directional chain made of k states of color one, alternating with $k + 1$ states of color 2. As proven in [13], the original algorithm has to lift each state k times before acknowledging that all states are losing, thus reaching a complexity of $O(k^2)$. Similarly, the lock-step formulation of the algorithm requires k global lifts, leading to a complexity of $O(k^2)$. However, after two global lifts all states have progress measure greater than zero. Therefore, if the gap acceleration is enabled, three lifts are enough to reach the fixpoint, for a total time complexity of $O(k)$.

3 Timed Interfaces with Variables

In this section, we present a model of real-time interfaces which is obtained from the *sociable interfaces* of [6], by adding discrete clocks in the spirit of [9].

The state space of our timed interfaces is represented by variables, interpreted over a given domain $\mathcal{D}$. Given a set of variables V, a *state* over V is a mapping $s : V \to \mathcal{D}$ that associates with each $x \in V$ a value $s(x) \in \mathcal{D}$. We denote by $[\![V]\!]$ the set of all states over V. For a set of variables $V' \subseteq V$, and a state $s \in [\![V]\!]$, the restriction of s to V' is a state $s' \in [\![V']\!]$ denoted by $s|_{V'}$. For two disjoint sets of variables V' and $V \setminus V'$, and two states $s' \in [\![V']\!]$ and $s'' \in [\![V \setminus V']\!]$, the operation $(s' \cdot s'')$ concatenates the two states resulting in a new state $s \in [\![V]\!]$. For two sets A and B, we write $f : A \rightrightarrows B$ to indicate that f is a function with domain A and codomain 2^B.

Definition 2 (Timed Interface). A *timed interface* is a tuple $M = (\Sigma_M, V_M^G, V_M^L, C_M, \tau_M^I, \tau_M^O, \varphi_M^I, \varphi_M^O)$, where:

- Σ_M is a set of *actions*.
- V_M^G is a set of *global variables*, V_M^L is a set of *local variables*, and C_M is the set of *clock variables*. Clock variables are interpreted over the set $\mathbb{N}_0$ of natural numbers including zero. We require $C_M \subseteq V_M^L$ and $V_M^G \cap V_M^L = \emptyset$. We set $V_M = V_M^G \cup V_M^L$.
- For all actions $a \in \Sigma_M$, $\tau_M^I(a) : [\![V_M]\!] \rightrightarrows [\![V_M]\!]$ is the *input transition relation* of a. We require this transition relation to be *deterministic* w.r.t. variables in V_M^L, that is,

$$\forall a \in \Sigma_M, s \in [\![V_M]\!], \forall s_1, s_2 \in \tau_M^I(a)(s).\ (s_1|_{V_M^L} = s_2|_{V_M^L}).$$

- For all actions $a \in \Sigma_M$, $\tau_M^O(a) : [\![V_M]\!] \rightrightarrows [\![V_M]\!]$ is the *output transition relation* of a.

- $\varphi_M^I \subseteq [\![V_M]\!]$ is the *input invariant*.
- $\varphi_M^O \subseteq [\![V_M]\!]$ is the *output invariant*.

The set of states $[\![V_M]\!]$ of a timed interface M is denoted by S_M. For $s \in S_M$, we denote by $s+1$ the state which coincides with s, except that the clock variables have been incremented by one. Formally, $(s+1)(v) = s(v)+1$ for all $v \in C_M$, and $(s+1)(v) = s(v)$ for all $v \in V_M \setminus C_M$.

The semantics of a timed interface is a game between players Input and Output. At each step, both players propose a move and the state of the interface evolves according to the following definitions. Each move can be *(i)* a state reachable from the current one by taking an action, *(ii)* the request to let time advance (move Δ_1), or *(iii)* the null move Δ_0. Each player can only play moves that maintain the player's invariant. In the following, we consider a fixed interface M.

Definition 3 (Moves). For all states $s \in S_M$ and $i \in \{I, O\}$, let $D^i(s) = \{\Delta_1\}$ if $s+1 \in \varphi_M^i$, and $D^i(s) = \emptyset$ otherwise. The set of possible moves for player i at s is:

$$\Gamma_M^i(s) = \left(\bigcup_{a \in \Sigma_M} \tau_M^i(a)(s) \cap \varphi_M^i\right) \cup \{\Delta_0\} \cup D^i(s).$$

We also define $\Gamma_M^i = \bigcup_{s \in S_M} \Gamma_M^i(s)$.

Two Boolean variables bl^I and bl^O are used for specifying whether a player lets time elapse or not (i.e. proposes a Δ_1 action). bl^I (bl^O) is true if and only if the action proposed by the input (output) player is not Δ_1. An *extended state* $\hat{s}$ is a state $s \in S_M$ augmented with the truth values for the Boolean variables bl^O and bl^I. The set of all extended states of M is $\hat{S}_M = S_M \times \{T, F\}^2$.

Definition 4 (Moves Outcome). For all states $s \in S_M$ and moves $m^I \in \Gamma_M^I(s)$ and $m^O \in \Gamma_M^O(s)$, the *outcome* $\delta_M(s, m^I, m^O)$ of m^I and m^O at s is the set of extended states defined by the following table, where rows represent choices for m^I and columns represent choices for m^O.

	Δ_0	Δ_1	s''
Δ_0	$\{(s, \mathrm{bl}^I, \mathrm{bl}^O)\}$	$\{(s, \mathrm{bl}^I, \neg\mathrm{bl}^O)\}$	$\{(s'', \neg\mathrm{bl}^I, \mathrm{bl}^O)\}$
Δ_1	$\{(s, \neg\mathrm{bl}^I, \mathrm{bl}^O)\}$	$\{(s+1, \neg\mathrm{bl}^I, \neg\mathrm{bl}^O)\}$	$\{(s'', \neg\mathrm{bl}^I, \mathrm{bl}^O)\}$
s'	$\{(s', \mathrm{bl}^I, \neg\mathrm{bl}^O)\}$	$\{(s', \mathrm{bl}^I, \neg\mathrm{bl}^O)\}$	$\{(s', \mathrm{bl}^I, \neg\mathrm{bl}^O), (s'', \neg\mathrm{bl}^I, \mathrm{bl}^O)\}$

Definition 5 (Strategy). A *strategy* for player $i \in \{I, O\}$ in M is a function $\pi^i : \hat{S}_M^* \to \Gamma_M^i$ that associates, with every finite sequence of extended states σ whose last state is $\hat{s} = (s, \mathrm{bl}^I, \mathrm{bl}^O)$, a move $\pi^i(\sigma) \in \Gamma_M^i(s)$. We denote by Π_M^I and Π_M^O the set of input and output strategies in M, respectively.

Definition 6 (Strategy Outcomes). Given a state $s \in S_M$, an input strategy $\pi^I \in \Pi_M^I$ and an output strategy $\pi^O \in \Pi_M^O$, the set of *outcomes* $\hat{\delta}_M(s, \pi^I, \pi^O)$ of π^I and π^O from s consists of all infinite sequences over extended states $\sigma = (s_0, \mathrm{bl}_0^I, \mathrm{bl}_0^O), \ldots, (s_i, \mathrm{bl}_i^I, \mathrm{bl}_i^O), \ldots$ such that $s_0 = s$, and for all $i \geq 0$ $(s_{i+1}, \mathrm{bl}_{i+1}^I, \mathrm{bl}_{i+1}^O) \in \delta_M(s_i, \pi^I(\sigma_{\leq i}), \pi^O(\sigma_{\leq i}))$ where $\sigma_{\leq i}$ denotes the prefix of σ up to the i-th extended state. Notice that bl_0^I and bl_0^O are arbitrarily defined.

In the following, we use *tick* as a shorthand for $\neg \text{bl}^O \wedge \neg \text{bl}^I$, which means that both players propose a time elapse step. Furthermore, we use the LTL notation [16] to denote sets of traces.

As discussed in [7], in order to take into proper account illegal behaviors that would lead to an artificial stopping of time, if player $i \in \{I, O\}$ has a certain goal $goal$, he should actually enforce the winning condition $WC^i(goal)$, defined as follows:

$$WC^I(goal) = (goal \wedge \Box\Diamond \text{tick}) \vee \Diamond\Box \text{bl}^O$$
$$WC^O(goal) = (goal \wedge \Box\Diamond \text{tick}) \vee \Diamond\Box\neg \text{bl}^O.$$

Intuitively, these conditions require a player to ensure that if time diverges, the goal is realized, and if time fails to diverge, the blame lies with the adversary. The conditions are asymmetrical, reflecting the fact that Input and Output do not behave in fully symmetrical ways during composition [9]. Given $s \in S_M$, a strategy $\pi^I \in \Pi_M^I$ is *I-winning* from s w.r.t. the goal $goal$, iff $\forall \pi^O \in \Pi_M^O \,.\, \hat{\delta}(s, \pi^I, \pi^O) \subseteq WC^I(goal)$. Similarly, a strategy $\pi^O \in \Pi_M^O$ is *O-winning* from s w.r.t. $goal$, iff $\forall \pi^I \in \Pi_M^I \,.\, \hat{\delta}(s, \pi^I, \pi^O) \subseteq WC^O(goal)$. A state $s \in S_M$ is *I-winning* (resp. O-winning) iff there exists an Input strategy that is I-winning (resp. O-winning) from s. The set of all I-winning (resp. O-winning) states is denoted by $Win_M^I(goal)$ (resp. $Win_M^O(goal)$).

A particularly important game is the *well-formedness* game, where the goals of the players are simply $\mathtt{T}$, so that their winning conditions are $Win_M^I(\mathtt{T})$ and $Win_M^O(\mathtt{T})$, respectively. Intuitively, if a player can win the well-formedness game, it means that it can “keep the system going”, without entering dead-end states from which time cannot progress [9,7].

4 Example: Scheduling as a Timed Game

We present an example of a periodical scheduling problem encoded as a timed interface. In the timed interface, the actions of Input represent scheduler decisions, such as the decision of starting a task. The actions of Output represent task nondeterminism, such as the variability in task execution times. The goal of Input is to ensure that no deadline is missed. If Input can win the game, the scheduler has a strategy that correctly schedules the tasks, ensuring that no deadline is missed regardless of task nondeterminism. Technically, the goal of not missing deadlines is a safety condition, stating that, while the tasks' execution has not completed, certain clocks should have values not exceeding the deadlines. We take this safety condition as the Input invariant, thus saddling the Input player, representing the scheduler, with the goal of meeting deadlines. We will see that taking into account for time progress in the winning condition is essential, if we wish to encode scheduling problems as timed games. Indeed, if the requirement for time progress is disregarded, the easiest way to ensure deadlines are met is to block the progress of time: as time cannot progress, deadlines cannot be missed!

The timed interface in Figure 2 encodes a periodical, non-preemptive scheduling problem involving two tasks, A and B. Task A has a period of 5s (measured by clock `cA`), and lasts up to 3s (measured by clock `dA`); task B has period 9s, and lasts up to 4s. The output invariant enforces the fact that neither task can be active for longer than

```
module Scheduling:
    var cpu, activeA, activeB, doneA, doneB: bool
    var cA, dA, cB, dB: clock

    oinv: (activeA -> dA <= 3) & (activeB -> dB <= 5)
    iinv: (cA <= 4) & (cB <= 9)

    input startA : { local: ~doneA & ~activeA & ~cpu ==>
                            activeA' := true, cpu' := true, dA' := 0 }
    input startB : { local: ~doneB & ~activeB & ~cpu ==>
                            activeB' := true, cpu' := true, dB' := 0 }

    output stopA : { activeA ==> ~activeA' & ~cpu' & doneA' }
    output stopB : { activeB ==> ~activeB' & ~cpu' & doneB' }

    input periodA: {local: doneA & cA = 4 ==> cA' := 0, doneA' := false}
    input periodB: {local: doneA & cB = 9 ==> cB' := 0, doneB' := false}
endmodule
```

Fig. 2. Timed interface representing the periodic scheduling problem of two non-preemptable tasks

its specified maximal duration. The input invariant states that the values of the clocks `cA` and `cB` cannot grow larger than the period lengths, namely, 5 and 9. This forces the scheduler to reset these clocks, via actions `periodA` and `periodB`, before they go beyond values 5 and 9. The action `periodA` signals the start of a new period for task A; its guard `doneA` specifies that `periodA` can be taken only once the execution of task A has completed. The situation for task B is similar. Therefore, to avoid violating the input invariant, Input (the scheduler) must issue actions `startA`, `startB`, `periodA`, `periodB` with a timing ensuring that jobs A and B terminate no later than the end of their respective periods. An Input strategy for doing this corresponds to a scheduling strategy for the task set.

This example illustrates why the winning condition needs to account for time divergence. Had we taken T as winning condition for Input, rather than $Win^I(\mathrm{T}) = \Box\Diamond\mathrm{tick} \vee \Diamond\Box\mathrm{bl}^O$, Input could have won simply by stopping time progress, for instance, by playing always move Δ_0.

5 Symbolic Solution of the Well-Formedness Game

Consider the winning condition for the input player in the well-formedness game.

$$WC^I(\mathrm{T}) = \Box\Diamond\mathrm{tick} \vee \Diamond\Box\mathrm{bl}^O.$$

Being the disjunction of a Büchi and a co-Büchi condition, it can be expressed as a parity condition with three colors, assigned as follows:

$$C_0 = \neg\mathrm{bl}^I \wedge \neg\mathrm{bl}^O; \qquad C_1 = \mathrm{bl}^I \wedge \neg\mathrm{bl}^O; \qquad C_2 = \mathrm{bl}^O.$$

If ϕ is a safety, reachability, Büchi, or co-Büchi formula, it is similarly possible to obtain 3-color deterministic parity automata encoding $WC^I(\phi)$ and $WC^O(\phi)$.

We note that C_1 consists of the states where Input is forced to play either an action, or the 0-delay move Δ_0. Thus, in a timed game, the gap is related to the maximal number of times for which Input can be forced to play without letting time advance. This number is generally much smaller than the number of C_1 states, as these chains of forced 0-time transitions tend, in practical examples, to be fairly short (it is unusual for them to be longer than a dozen transitions). This explains the very large speedup provided by the gap acceleration in the analysis of timed games.

If we restrict the variable domain $\mathcal{D}$ to be finite, and we manage to let clock variables also range over a finite set, we can apply the EJ and gap algorithms to the problem of checking well-formedness of an interface. The tool TICC [8] allows the user to specify timed interfaces using a convenient syntax based on guarded commands. The tool is in the process of being extended to discrete real-time. In TICC, clock variables can only be compared to (or assigned from) constants. Under this assumption, it is well known that, for each clock x, it is sufficient to consider the range of values going from zero to the maximum constant to which x is ever compared (or assigned from), plus one.

We implemented in TICC both the EJ and the gap algorithms; we experimented with both algorithms for solving well-formedness games. In the tool, interfaces are internally represented using Multi-valued Decision Diagrams [19] (MDDs) as implemented by the CUDD library [18]. Therefore, in the following we discuss the issues regarding the symbolic implementation of both algorithms.

5.1 Gap Algorithm

Since the progress measure algorithm is tailored to turn-based games, we have to emulate the turns by providing separate transition relations for Input and Output. Input moves from the original (or *regular*) states of the concurrent game, while Output moves from intermediate *virtual* states. Notice that, if from a regular state s Input chooses to reach state s' via action a, Output in the next virtual state can decide to let a happen (by picking move Δ_0), or rather take an alternative action b from s. Thus, we have to store in the virtual state both the start state s and the proposed destination s'. Therefore, we end up having three copies of the state variables V_M, which we call V, V', and V''. The transition relation of Input in the turn-based game is represented by the predicate τ^I, of type $V \to V', V'', \mathrm{bl}^I$. The transition relation of Output is represented by the predicate τ^O, of type $V', V'', \mathrm{bl}^I \to V, \mathrm{bl}^O$. We need an extra variable ρ to represent the progress measure.

Next, we need to represent the function *Progr* from (1), used to update the progress measure. For states of color one, *Progr* has to increment the value of the progress measure by one. Consider the general problem of having a predicate α over the set of variables Z, and wanting to increment by one the variable $z \in Z$, unless the value of z is already equal to its maximum value z_{max}. Using standard MDD operators, this can be achieved by having an extra variable z' and performing the following computation:

$$incr(\alpha, z) = (\exists z(\alpha \wedge z' = z + 1))[z/z'] \vee (\alpha \wedge z = z_{max}).$$

However, the above computation leads to very poor performance: since ρ can have a very high maximum value, the computation of the predicate $\rho' = \rho + 1$ alone requires a very large amount of time. Thus, in place of the above computation, we developed a specific increment operator, as follows. Let $z_0, z_1, \ldots, z_k$ be the binary variables encoding variable z, ordered from the least significant (z_0) to the most. For $c \in \{0, \ldots, k\}$, and $\sim \in \{<, \leq, >, \geq\}$, let $z_{\sim c} = \{z_j \mid j \sim c\}$.

function Increment(α, z)
vars: $r, \overline{\alpha}, \underline{\alpha}$, pos, neg : MDD
r := **false**
$\overline{\alpha} := \alpha \wedge (z = z_{max})$
$\underline{\alpha} := \alpha \wedge (z \neq z_{max})$
for $i := 0$ **to** k **do**
neg := $\neg z_0 \wedge \neg z_1 \wedge \ldots \wedge \neg z_{i-1}$
pos := $z_0 \wedge z_1 \wedge \ldots \wedge z_{i-1}$
$r := r \vee \big(\text{neg} \wedge z_i \wedge \exists z_{\leq i} \,.\, (\underline{\alpha} \wedge \text{pos} \wedge \neg z_i)\big)$
done
return $r \vee \overline{\alpha}$

Then, in order to implement the measure update step described by (2), we need the following symbolic operation. Let α be a predicate over the set of variables Z and let $z \in Z$. For each assignment to the variables in $Z \setminus \{z\}$, α may contain several different assignments to z. We want to preserve the minimum value of z only. We call this predicate $\min_z \alpha$. In set notation, we have:

$$\min_z \alpha = \big\{s \in \alpha \mid s(z) = \min\{s'(z) \mid s' \in \alpha\}\big\}.$$

No efficient implementation of $\min$ exists using standard MDD operators. We thus developed a new "min" operator according to the following algorithm.

function Min(α, z)
vars: r : MDD
$r := \alpha$
for $i := k$ **down to** 0 **do**
$r := r \wedge \big((\neg z_i \wedge \exists z_{>i} \,.\, r) \vee (z_i \wedge \forall z_{\geq i} \,.\, (\neg z_i \implies \neg r))\big)$
done
return r

The "min" operator is also useful to determine the minimum gap in a measure. If α is the predicate over variables $(V, \mathrm{bl}^I, \mathrm{bl}^O, \rho)$ representing the measure of each regular state in the game, the equation $\min_\rho \forall V \forall \mathrm{bl}^I \forall \mathrm{bl}^O \,.\, \neg\alpha$ yields "false" if the measure has no unused values, or otherwise a predicate of the type $\rho = c$, where c is the minimum unused value of the measure (and thus a good candidate to be a gap). Such predicate can then be used to implement the acceleration technique presented in Section 2.3.

5.2 Emerson-Jutla's μ-Calculus Algorithm

To apply the EJ algorithm of Section 2.1, we do not need to consider the turn-based version of the game. Rather, we simply use as controllable predecessor operator the following.

Definition 7 (Concurrent Controllable Predecessor Operator). $Cpre^I : 2^{\hat{S}_M} \rightarrow 2^{S_M}$ assigns to each set of extended states X, the set of states from which Input can force the game into X in one step. Formally,

$$Cpre^I(X) = \{s \in S_M \mid \exists m^I \in \Gamma^I_M(s) \,.\, \forall m^O \in \Gamma^O_M(s) \,.\, \delta_M(s, m^I, m^O) \subseteq X\}.$$

The transition predicates τ^I and τ^O developed in the previous section can also be used to obtain a symbolic implementation of $Cpre^I$. Given a predicate α over variables $(V, \mathrm{bl}^I, \mathrm{bl}^O)$, we have:

$$Cpre^I(\alpha) = \exists V' \exists V'' \exists \mathrm{bl}^I \,.\, \tau^I \wedge \big(\forall V \forall \mathrm{bl}^O \,.\, \tau^O \implies \alpha\big).$$

Given the symbolic implementation of $Cpre^I$, the EJ algorithm can be implemented in a straightforward manner, using three nested loops that compute the fixpoint by Picard iteration.

5.3 Experimental Results

On the basis of the implementation discussed above, we compared the performance of the EJ and gap algorithms. Our results indicate that the performance improvement afforded by the gap acceleration is essential: for the scheduling example, for instance, the acceleration reduces the number of iterations from at least 73,920 in the original Jurdziński progress measure algorithm to 163 in the gap algorithm — a speed-up of over 450. Without acceleration, we believe that the progress measure algorithm is highly impractical for solving 3-color parity games.

Our results indicate that there is no clear winner between the EJ algorithm and the gap algorithm. The running times of the two algorithms were, in our experiments, within a factor of two of each other, with the EJ algorithm generally being the fastest. We suspect that the BDD variable ordering, and other details of the symbolic implementation, have a large influence on the results, so that we do not believe that our experiments are conclusive. For the scheduling example of Section 4, the input well-formedness of the interface can be computed in 144s with the EJ algorithm, and 302s with the gap algorithm, on an AMD Athlon 64 4400+ CPU running 32-bit linux. In the gap algorithm, the main expense occurs in the "lift" operation; we are investigating more efficient symbolic implementations of this operation.

In the same paper [13] that introduced the progress measure algorithm, the following acceleration is mentioned: in place of n_1, it suffices to take the maximum number n'_1 of C_1-states belonging to a strongly-connected component (SCC) of the game graph (S, E); clearly, $n'_1 \leq n_1$. In an enumerative setting, both this SCC-based acceleration, and our gap-based acceleration, are of interest, and each provides greater speed-ups on some games. In a symbolic setting, the time required to compute SCCs must be taken into account; the straightforward symbolic algorithm may require a quadratic number of iterations. In contrast, our gap-based acceleration can be performed at negligible cost.

References

1. Adler, B., de Alfaro, L., Faella, M.: Average reward timed games. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 65–80. Springer, Heidelberg (2005)
2. Alur, R., Henzinger, T.A.: Modularity for timed and hybrid systems. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 74–88. Springer, Heidelberg (1997)
3. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Proc. I.F.A.C. (ed.) Proc. IFAC Symposium on System Structure and Control, pp. 469–474. Elsevier, Amsterdam (1998)
4. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal strategies in priced timed game automata. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 148–160. Springer, Heidelberg (2004)
5. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
6. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: Gramlich, B. (ed.) Frontiers of Combining Systems. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
7. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: The element of surprise in timed games. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 144–158. Springer, Heidelberg (2003)
8. de Alfaro, L., Faella, M., Legay, A.: An introduction to the tool TICC. In: Proc. of Workshop on Trustworthy Software, IBFI, Schloss Dagstuhl, Germany (2006)
9. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
10. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS 91: Proc. 32nd IEEE Symp. Found. of Comp. Sci. pp. 368–377. IEEE Computer Society Press, Los Alamitos (1991)
11. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model checking for fragments of μ-calculus. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 385–396. Springer, Heidelberg (1993)
12. Faella, M., La Torre, S., Murano, A.: Automata-theoretic decision of timed games. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 94–108. Springer, Heidelberg (2002)
13. Jurdziński, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
14. Jurdziński, M.: Deciding the winner in parity games is in UP $\cap$ co-UP. Information Processing Letters 68(3), 119–124 (1998)
15. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Mayr, E.W., Puech, C. (eds.) STACS 95. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
16. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1991)
17. Segala, R., Gawlick, G., Søgaard-Andersen, J., Lynch, N.: Liveness in timed and untimed systems. Information and Computation 141(2), 119–171 (1998)
18. F. Somenzi: CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~{}fabio/CUDD/cuddIntro.html`
19. Srinivasan, A., Kam, T., Malik, S., Brayton, R.: Algorithms for discrete function manipulation. In: ICCAD 90: Proc. of IEEE Int. Conf. on Computer-Aided Design, pp. 92–95. IEEE Computer Society Press, Los Alamitos (1990)
20. Thomas, W.: Automata on Infinite Objects. In: Handbook of Theoretical Computer Science, Elsevier, Amsterdam (1990)