A New Algorithm for Strategy Synthesis in LTL Games

Aidan Harding[1], Mark Ryan[1], and Pierre-Yves Schobbens[2]

[1] School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK
[2] Institut d'Informatique, Facultés Universitaires de Namur, Rue Grandgagnage 21, 5000 Namur, Belgium

Abstract. The automatic synthesis of programs from their specifications has been a dream of many researchers for decades. If we restrict to open finite-state reactive systems, the specification is often presented as an ATL or LTL formula interpreted over a finite-state game. The required program is then a strategy for winning this game. A theoretically optimal solution to this problem was proposed by Pnueli and Rosner, but has never given good results in practice. This is due to the 2EXPTIME-complete complexity of the problem, and the intricate nature of Pnueli and Rosner's solution. A key difficulty in their procedure is the determinisation of Büchi automata. In this paper we look at an alternative approach which avoids determinisation, using instead a procedure that is amenable to symbolic methods. Using an implementation based on the BDD package CuDD, we demonstrate its scalability in a number of examples. Furthermore, we show a class of problems for which our algorithm is singly exponential. Our solution, however, is not complete; we prove a condition which guarantees completeness and argue by empirical evidence that examples for which it is not complete are rare enough to make our solution a useful tool.

1 Introduction

Finite-state reactive systems occur in many critical areas of computing. They can be found in places such as network communication protocols, digital circuits, and industrial control systems. Their use in systems which involve concurrency and their interaction with unpredictable or hostile environments makes reactive systems notoriously hard to write correctly. By considering such systems as games we can distinguish between events that we can control (inside the program) and events that we cannot (the environment). This gives a more realistic framework for reasoning about them than the conventional approach of "closing" the system by adding a restricted environment and treating all choices uniformly.

We take the stance that closing an open system for verification or synthesis is imprecise and that reasoning with game semantics provides a much better solution. This stance has been advocated by many other researchers [1, 9, 10, 8], but there are some verification and synthesis problems that become much

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 477–492, 2005.

harder in the game-theoretic world. In particular the problems of synthesis and verification for games with LTL winning conditions are 2EXPTIME-complete [12, 13]. This high complexity and the intricacy of the solution offered by Pnueli and Rosner have meant that despite the wealth of potential applications, there has been no implementation of synthesis for LTL games. We address this problem by providing a novel algorithm which avoids a major difficulty of the classical approach: the determinisation of Büchi automata. The best known method for this determinisation is due to Safra [14] and this method has been proven to be optimal [11] but has resisted efforts at efficient symbolic implementation [17]. Instead of trying to determinise a Büchi automaton, our algorithm uses a "shift automaton" to track the possible states that the Büchi automaton could be in and retake non-deterministic choices if they turn out to be wrong. The shift automaton is of roughly equal size to the deterministic automaton produced by Safra's algorithm, but it can be constructed symbolically. This has allowed for the construction of an efficient implementation using BDDs. In this paper we describe in detail a new algorithm for the synthesis of strategies in LTL games; we describe some small problems that can be solved by the implementation of this algorithm; and finally we give some performance data obtained by parameterising the given examples.

2 ω-Automata and Infinite Games

We quickly review the definitions and establish notations for writing about ω-automata and infinite games. Detailed information on ω-automata can be found in [18]; and information on infinite games can be found from [8] and [19].

Given an alphabet Σ, we denote the set of all finite words made from letters in Σ as Σ^*, and the set of all ω-words (infinte words) as Σ^ω. For a word $\lambda \in \Sigma^\omega$, we write $\lambda[i]$ for the i-th letter, $\lambda[i,j]$ for the finite section of the word from point i to j, and $\lambda[i,\infty]$ for the infinte suffix from point i. ω-automata provide a way of recognising sets of ω-words. An ω-automaton $\mathcal{A} = \langle Q_\mathcal{A}, i_\mathcal{A}, \delta_\mathcal{A}, Acc\rangle$ is a tuple where the component parts are as follows: $Q_\mathcal{A}$ is a finite set of states; $i_\mathcal{A}$ is an initial state; $\delta_\mathcal{A} : Q_\mathcal{A} \times \Sigma \mapsto 2^{Q_\mathcal{A}}$ is a transition function (we may define $\delta_\mathcal{A} : Q_\mathcal{A} \times \Sigma \mapsto Q_\mathcal{A}$ for deterministic automata); and Acc is an acceptance condition. A run ρ of an ω-automaton on a word λ is an infinite sequence of states such that $\rho[0] = i_\mathcal{A}$ and for all $i \geq 0$ $\rho[i+1] \in \delta_\mathcal{A}(\rho[i], \lambda[i])$. We denote the set of states that occur infintely often on a run ρ by $In(\rho)$. In this paper we are concerned with two types of ω-automata: Büchi automata and Rabin automata. We write DB for a deterministic Büchi automaton and NB for a non-deterministic Büchi automaton. The acceptance condition in a Büchi automaton is a set $F \subseteq Q_\mathcal{A}$ and a word λ is accepted if and only if there is a run ρ on λ such that $In(\rho) \cap F \neq \emptyset$. The acceptance condition on a Rabin automaton is a set of pairs $\{(E_0, F_0), \ldots, (E_n, F_n)\}$ and a word λ is accepted if and only if there is a run ρ on λ such that there exists $i \in [0, n]$ such that $In(\rho) \cap F_i \neq \emptyset$ and $In(\rho) \cap E_i = \emptyset$.

An infinite game is a tuple $G = \langle Q_G, i_G, Q_P, Q_A, \delta_G \rangle$ representing a two-player game between the protagonist P and the antagonist A. Q_G is a set of states from which i_G is the initial one; Q_P and Q_A partition Q_G into turns for P and A, respectively; $\delta_G : Q_G \mapsto 2^{Q_G}$ is a transition function such that $\forall q \in Q_P\ \delta_G(q) \subseteq Q_A$ and $\forall q \in Q_A\ \delta_G(q) \subseteq Q_P$ i.e. the players alternate turns. A play of the game is an infinite sequence λ of states from Q_G such that $\lambda[0] = i_G$ and for all $i \geq 0$ $\lambda[i+1] \in \delta_G(\lambda[i])$. We formalise the capabilites of players in the game with the notion of strategies. A strategy $f : Q_G^+ \mapsto 2^{Q_G}$ for P restricts the choices of P by prescribing how he should play his moves. We require that for any play λ and all $i \geq 0$ $f(\lambda[0,i]) \subseteq \delta_G(\lambda[i])$. The set of outcomes $out(f, q_0)$ of playing a strategy f from a state q_0 is defined as

$$out(f, q_0) = \{q_0, q_1, \ldots \mid \forall i \geq 0 \text{ if } (q_i \in Q_P)\ q_{i+1} \in f(q_0, \ldots, q_i) \text{ else } q_{i+1} \in \delta_G(q_i)\}$$

We also use partial strategies which are partial functions with the same type as a normal strategy. The set of outcomes of a partial strategy is defined as:

$$out(f, q_0) = \{q_0, q_1, \ldots \mid \forall i \geq 0 \text{ if } (q_i \in Q_P \wedge f(q_0, \ldots, q_i) \text{ defined}) \; q_{i+1} \in f(q_0, \ldots, q_i) \text{ else } q_{i+1} \in \delta_G(q_i)\}$$

When a game is provided with a winning condition W, we say that P can win the game if and only if there is a strategy (partial or complete) for P such that all outcomes of the strategy satisfy W.

3 Synthesis for NB Games

The main algorithm in our synthesis procedure takes as input a game $G = \langle Q_G, i_G, Q_P, Q_A, \delta_G \rangle$ and a winning condition in the form of a Büchi automaton, $B = \langle Q_B, i_B, \delta_B, F_B \rangle$. The algorithm identifies a set of states that are winning for P and produces a partial strategy (partial because it may be undefined from states which are not winning for P) such that the set of outcomes of the strategy from any winning state are accepted by B.

To see why the conventional approach uses determinisation, let us consider an approach which uses an obvious extension of the algorithm used for games with Büchi winning conditions [19]. A Büchi winning condition (as opposed to a winning condition specified by a NB) specifies a set $F_G \subseteq Q_G$ such that plays are winning for P if and only if they visit F_G infinitely often. The solution for these games offered in [19] is a game-theoretic extension of the algorithm proposed by Emerson and Lei for finding fair strongly connected components [7]. This algorithm is attractive as it has shown to be quite efficient when compared against other symbolic fair cycle detection algorithms [16]. It works by finding the set of states from where P has a strategy to reach F_G, then the states where P has a strategy to reach F_G and from there has a strategy to reach F_G again etc. An increasing number of visits to F_G are required until a fixed-point is achieved whereupon we know that P has a strategy to visit F_G infinitely often.

(a) Game (b) Winning Condition

Fig. 1. A game where the winning condition "needs" determinisation. Square states are P's moves, circles are A's

The game-theoretic aspect of the algorithm in [19] works by adapting the notion of predecessor to enforce that a state in Q_P is winning if there exists a winning successor, and a state in Q_A is winning if all successors are winning.

We could try to perform the same computation over $G \times B$, always evaluating the B component existentially. However, there is an assumption in this algorithm that winning is transitive i.e. the states along a winning path are winning themselves. Whilst this may seem like a natural thing to expect, it is not actually true. The game and specification in Figure 1 give an example of how winning can fail to be transitive in this sense. P has only one choice which comes at state 3; he can win this game by always choosing $3 \rightarrow 4$ if he gets that choice. Although there is a winning strategy from $(1, t_0)$, if we follow that strategy and, at the same time, try to construct a winning run from the B part, we cannot be sure to reach a state from where there is another winning strategy. The opponent can stay in $\{(1, t_?), (2, t_?)\}$ as long as he likes and we must choose what to do with the Büchi component. We cannot allow the Büchi automaton to visit t_1 in case the opponent later chooses 3. So we either have a losing run in the Büchi component, $((1, t_0)(2, t_0))^\omega$ or reach $\{(1, t_1), (2, t_1)\}$ from where there is no winning strategy (A chooses 3 and B is stuck). On this basis, $(1, t_0)$ would not be identified as winning because P cannot be sure to reach a state in $Q_G \times F_B$ from where he has a winning strategy.

Our solution to this is to allow some "shifting" between Büchi states. If the Büchi automaton is in a dead-end state, we now allow the transition relation to be overridden by making a shift i.e make a transition as if the Büchi automaton were in a different state. We maintain the set of reachable Büchi states at all times, and this provides the justification for shifts – whenever a shift is made, it is made to some reachable state and, thus, is equivalent to retaking some earlier non-deterministic choices. The set of reachable states is provided by the shift automaton, $S = \langle Q_S, i_S, \delta_S \rangle$, a deterministic automaton derived from B with the subset construction where each state in Q_S represents a set of states from Q_B:

$$Q_S = \mathcal{P}(Q_B) \qquad i_S = \{i_B\} \qquad \delta_S(\phi, q) = \{t' \mid \exists t \in \phi . t' \in \delta_B(t, q)\} \qquad (1)$$

Accordingly, we define the synthesis algorithm over $G \times B \times S$. We call this product the *composite*, C, and define some shorthands to ease the burden of notation:

$$Q_C = Q_G \times Q_B \times Q_S \tag{2}$$

$$F_C = \{(q, t, \phi) \in Q_C \mid t \in F_B\} \tag{3}$$

$$Q_{CP} = \{(q, t, \phi) \in Q_C \mid q \in Q_{GP}\} \tag{4}$$

$$Q_{CA} = \{(q, t, \phi) \in Q_C \mid q \in Q_{GA}\} \tag{5}$$

Now, if we are in a situation such as the one in Figure 1, we can make winning transitive by allowing some shifts. The informal argument is as follows: From $(1, t_0, \{t_0\})$ we always go to $(2, t_1, \{t_0, t_1\})$ because we are optimistic about getting an accepting run in the B component. If the opponent always chooses $2 \rightarrow 1$, then we have an accepting run made up of $(1, t_0, \{t_0\})$ $((2, t_1, \{t_0, t_1\})$ $(1, t_0, \{t_0, t_1\}))^\omega$. If the opponent eventually chooses $2 \rightarrow 3$, then we take a shift: In state $(2, t_1, \{t_0, t_1\})$ the B component could have been in t_0, so when the opponent chooses 3 we make a shift and take the $t_0 \rightarrow^3 t_2$ transition i.e. we go to $(3, t_2, \{t_0, t_2\})$. From here, P can win and we end up with an overall winning path made up by $(1, t_0, \{t_0\})((2, t_1, \{t_0, t_1\})(1, t_0, \{t_0, t_1\}))^*(2, t_1, \{t_0, t_1\})$ $((3, t_2, \{t_0, t_2\})$ $(4, t_2, \{t_0, t_2\}))^\omega$.

Shifting helps the issue of completeness, but allowing an infinite number of shifts would be unsound. However, if shifting is only allowed finitely often, the language is not changed. Informally we justify this on the basis that acceptance is evaluated over infinite paths, and although shifting may allow finitely many extra visits to F_C, paths must eventually have no more shifts and thus would be accepting without any shifting. The soundness of finite shifting is implied by Theorem 2.

To write down the main synthesis algorithm with finite shifting, we first define two predecessor functions pre_P and pre_A. These are evaluated over the triple state-space of $G \times B \times S$, respecting the alternation of the game and allowing for shifting. Unlike a conventional predecessor function, two arguments are supplied. The second argument is a set that we allow shifts into. $pre_P(X, W)$ is the set of transitions which obey the game and shift automaton, and either have a transition in B to reach X ($t' \in \delta_B(t, q') \wedge (q', t', \phi') \in X$ in Equation 6) or have a shift justified by the shift automaton to reach W ($t' \in \phi' \wedge (q', t', \phi') \in W$ in Equation 6). $pre_A(X, W)$ simply uses $pre_P(X, W)$ as an approximation, and then makes sure that there is a good transition for every possible game-successor.

$$\begin{aligned} pre_P(X, W) = \{\langle (q, t, \phi), (q', t', \phi') \rangle \mid\ & q' \in \delta_G(q), \phi' = \delta_S(\phi, q'), \\ & (t' \in \delta_B(t, q') \wedge (q', t', \phi') \in X) \vee (t' \in \phi' \wedge (q', t', \phi') \in W)\} \end{aligned} \tag{6}$$

$$\begin{aligned} pre_A(X, W) = \{\langle (q, t, \phi), (q', t', \phi') \rangle \in pre_P(X, W) \mid\ & \forall q'_2 \in \delta_G(q) \\ & \exists \langle (q, t, \phi), (q'_2, t'_2, \phi'_2) \rangle \in pre_P(X, W)\} \end{aligned} \tag{7}$$

Using these definitions, we write the main algorithm in Figure 2[1]. To understand how the synthesis algorithm works, consider each of the variables in turn:

[1] We denote the k-th projection of a tuple T by $\pi_k(T)$

1 $w_0 := \emptyset;$
2 $\delta_{\mathcal{F},0,\infty,\infty} := \emptyset;$
3 repeat counted by $j = 1 \dots$
4 $\quad z_{j,0} := Q_C;$
5 $\quad$ repeat counted by $k = 1 \dots$
6 $\quad\quad \tau_{j,k} := z_{j,k-1} \cap F_C;$
7 $\quad\quad s_{j,k,0} := \emptyset;$
8 $\quad\quad \delta_{\mathcal{F}j,k,0} := \delta_{\mathcal{F}j-1,\infty,\infty};$
9 $\quad\quad$ repeat counted by $l = 1 \dots$
10 $\quad\quad\quad u_A := pre_A(\tau_{j,k} \cup s_{j,k,l-1}, w_{j-1}) \cap (Q_{CA} \times Q_{CV});$
11 $\quad\quad\quad u_V := pre_V(\tau_{j,k} \cup s_{j,k,l-1}, w_{j-1}) \cap (Q_{CV} \times Q_{CA});$
12 $\quad\quad\quad \delta_{\mathcal{F}j,k,l} := \delta_{\mathcal{F}j,k,l-1} \cup \{\langle (q,t,\phi),(q',t',\phi')\rangle \in u_A \cup u_V$
$\quad\quad\quad\quad\quad\quad \mid (q,t,\phi) \notin \pi_1(\delta_{\mathcal{F}j,k,l-1})\};$
13 $\quad\quad\quad s_{j,k,l} := s_{j,k,l-1} \cup \pi_1(u_A) \cup \pi_1(u_V);$
14 $\quad\quad$ until $s_{j,k,l} = s_{j,k,l-1}$
15 $\quad\quad z_{j,k} := z_{j,k-1} \cap s_{j,k,\infty};$
16 $\quad$ until $z_{j,k} = z_{j,k-1}$
17 $\quad w_j := w_{j-1} \cup z_{j,\infty};$
18 until $w_j = w_{j-1}$

Fig. 2. Synthesis algorithm with finite shifting

- w_j: At the end of the algorithm, this will contain the set of winning states. The j subscript is the maximum number of shifts required to win from a state in w_j.
- $z_{j,k}$: At the end of the middle loop, this is the set of states from where every outcome reaches $z_{j,k} \cap F_C$ infinitely often with no shifting or just reaches w_{j-1} (possibly by shifting). During the middle loop, every outcome reaches $z_{j,k-1} \cap F_C$ with no shifting or w_{j-1} (possibly by shifting).
- $\tau_{j,k}$: The "target" for the innermost loop. This variable could be substituted for its definition at each use, it is clearer (and more efficient in implementation) to write separately.
- $s_{j,k,l}$: The set of states from where P can be sure to reach $\tau_{j,k}$ in l steps with no shifting or w_{j-1} in l steps with a shift.
- $\delta_{\mathcal{F},j,k,l}$: The partial strategy as it is synthesised. On the first j-loop it will be a strategy to win with no shifting. On line 12 we must be careful not to overwrite old moves. On iteration l of the inner loop, when a transition is first added to the strategy it must go into $s_{j,k,l-1} \cup \tau_{j,k}$ or w_{j-1}. However, this state will be rediscovered on later iteration of the inner loop and u_P/u_A may contain transitions which do not make progress towards an accepting state and we must therefore keep the transition from the first discovery. Having built a strategy with no shifts, we carry this forward to the next iteration of the j-loop. Here another strategy is built up, but this time it allows the possibility of a shift to w_{j-1} since we already have a winning strategy from there. New moves are written for states in $w_j - w_{j-1}$, but as soon as the strategy reaches w_{j-1} the old strategy takes over.

The algorithm in Figure 2 is computing nested fixed-points which can be characterised as $\mu w.\nu z.\mu s.\pi_1((pre_P(s \vee (z \wedge F_C), w) \wedge (Q_{CP} \times Q_{CA})) \vee (pre_A(s \vee (z \wedge F_C), w) \wedge (Q_{CA} \times Q_{CP}))$. We note that on the first iteration of the outer loop in this algorithm, the computation performed is the naïve extension of the solution for Büchi games i.e. $\nu z.\mu s.\pi_1((pre_P(s \vee (z \wedge F_C), \bot) \wedge (Q_{CP} \times Q_{CA})) \vee (pre_A(s \vee (z \wedge F_C), \bot) \wedge (Q_{CA} \times Q_{CP}))$. This calculation does not depend on the shift automaton and, in this way, we are sometimes able to perform the strategy synthesis without having to generate the shift automaton. We can give a precise condition which assures this by first defining trivially determinisable Büchi automata.

Definition 1. *A Büchi automaton is trivially determinisable if and only if it can be made deterministic by removing 0 or more transitions without changing its language.*

Using this definition it is possible to prove the following theorem:

Theorem 1. *For any game G, with a winning condition specified by a Büchi automaton B, if B is trivially determinisable, then all winning states for P in Q_C satisfy $\nu z.\mu s.\pi_1((pre_P(s \vee (z \wedge F_C), \bot) \wedge (Q_{CP} \times Q_{CA})) \vee (pre_A(s \vee (z \wedge F_C), \bot) \wedge (Q_{CA} \times Q_{CP}))$.*

Whilst this definition includes the shift automaton, it is clear that the predecessor functions do not depend on the shift automaton when W is empty, so this proves that if B is trivially determinisable, the algorithm can succeed without generating the shift automaton. Intuitively, this theorem holds because the naïve algorithm is complete for deterministic Büchi automata and since the transitions of B are evaluated existentially, a trivially determinisable Büchi automaton is as good as a deterministic one. In the long version of this paper, this theorem and all the other theorems that follow are proven in the appendix.

Since pre_P and pre_A are monotonic functions and the state-spaces involved in the algorithm are finite, it follows that the algorithm terminates. The algorithm's soundness is asserted by the following theorem:

Theorem 2. *Once the algorithm has terminated, for all $(q, t, \phi) \in w_\infty$, $\delta_{\mathcal{F}_\infty,\infty,\infty}$ is a partial strategy such that $\forall \lambda \in out(\delta_{\mathcal{F}_\infty,\infty,\infty}, (q, t, \phi))$ $\pi_1(\lambda[1, \infty]) \in \mathcal{L}(B, \phi)$.*

In much the same way as the completeness condition in Theorem 1, we can give a condition for the algorithm in Figure 2. To do this, we introduce the concept of the *generalised Rabin expansion* of a Büchi automaton. Intuitively, this automaton encodes the idea of finite shifting by its structure and winning condition.

Definition 2. *Let B be a Büchi automaton, and S be the corresponding shift automaton for B. The* generalised Rabin expansion*, $R = \langle Q_R, i_R, \delta_R, F_R, E_R \rangle$, of B and S is defined as*

$$Q_R = Q_B \times Q_S$$
$$i_R = i_B \times i_S$$
$$\delta_R = \{((t,\phi), a, (t',\phi')) \in Q_R \times \Sigma \times Q_R \mid \phi' \in \delta_S(\phi, a), t' \in \phi'\}$$
$$F_R = \{(t,\phi) \in Q_R \mid t \in F_B\}$$
$$E_R = \{((t,\phi), a, (t',\phi')) \in \delta_R \mid t' \notin \delta_B(t, a)\}$$

where, in the usual way, Q_R, i_R, and δ_R are the state-space, initial state, and transition function, respectively. F_R and E_R are used to define the winning condition of R: A run ρ on a word λ is winning if and only if $\exists^\infty i \geq 0.\rho[i] \in F_R$ and $\exists j \geq 0.\forall k \geq j\ (\rho[k], \lambda[k], \rho[k+1]) \notin E_R$.

It is convenient to specify a winning condition on transitions rather than states, but it is easy to translate such an automaton into a conventional Rabin automaton. The translation could be done as follows: create a second copy of every state; make every transition in E_R go instead to the copy; make every transition in the copy go back into the original; finally, set the Rabin condition to have infinitely many visits to F_R in the original and only finitely many visits to the copied states. We also note that for any reachable state (t, ϕ) in R, the invariant is maintained that $t \in \phi$.

With this definition in place, it is possible to prove the following theorem about completeness for the synthesis algorithm.

Theorem 3. *For any game G, with a winning condition specified by a Büchi automaton B, if B's generalised Rabin expansion is trivially determinisable then all winning states for P in Q_C satisfy $\mu w.\nu z.\mu s.\pi_1((pre_P(s \vee (z \wedge F_C), w) \wedge (Q_{CP} \times Q_{CA})) \vee (pre_A(s \vee (z \wedge F_C), w) \wedge (Q_{CA} \times Q_{CP}))$.*

We note that this is a safe approximation of the class of problems for which the algorithm will be complete. In fact, the structure of the game is also cruical to completeness. Providing a characterisation which uses the structure of both the game and the specification would be an interesting avenue for future research.

4 Synthesis for LTL Games

In the previous section we provided an algorithm for solving games with NB winning conditions that was complete under a condition on the form of the NB. We can perform synthesis for LTL games by using the tableau method to translate an LTL specification into a NB and then using the algorithm in Figure 2. With the restriction of Theorem 3 and our goal of symbolic implementation in mind, our choice of translation from LTL to NB must be made wisely. The method that we use is based on the symbolic construction of [4], with three changes: First, we deal with formulae in negation normal form rather than using a minimal set of temporal operators – as noted by [15], this provides us with a slight efficiency improvement as safety formulae do not have to be treated as negated liveness formulae. Secondly, we define a weaker transition formula than [4] – this allows

some types of formulae to be translated into trivially determinisable Büchi automata and allows for the last change. Finally, we split variables into "required" and "optional" forms meaning that the fairness constraints of optional formulae do not necessarily have to be met as long as some other fairness constraints are whilst the constraints of required formulae must always be met.

First we recall the syntax and semantics of LTL, a more thorough review can be found in [6]. Syntactically, we consider an LTL formula ψ in negation normal form to obey the following grammar

$$\Pi ::= p \mid \neg\Pi \mid \Pi \vee \Pi \qquad \psi ::= \Pi \mid \psi \wedge \psi \mid \psi \vee \psi \mid X\psi \mid \psi \,\mathcal{U}\, \psi \mid \psi \,\mathcal{R}\, \psi$$

where p is a member of the set of atomic propositions. The semantics of LTL are defined inductively over infinite paths λ.

- $\lambda \vDash p$ iff $p \in \lambda[0]$.
- $\lambda \vDash \neg\psi$ iff $\lambda \nvDash \psi$.
- $\lambda \vDash \psi_1 \vee \psi_2$ iff $\lambda \vDash \psi_1$ or $\lambda \vDash \psi_2$.
- $\lambda \vDash \psi_1 \,\mathcal{U}\, \psi_2$ iff $\exists i \geq 0.\lambda[i,\infty] \vDash \psi_2$ and $\forall j \in [0, i-1]\ \lambda[j,\infty] \vDash \psi_1$.
- $\lambda \vDash \psi_1 \,\mathcal{R}\, \psi_2$ iff $\forall i \geq 0$ either $\lambda[i,\infty] \vDash \psi_2$ or there exists $j \in [0,i]$ such that $\lambda[j,\infty] \vDash \psi_1 \wedge \psi_2$.

Like [4], we define a function el() to return the set of elementary sub-formulae of an LTL formula. The set $\mathrm{el}(\psi)$ forms the set of propositions in the tableau for ψ.

- $\mathrm{el}(p) = \{p\}$
- $\mathrm{el}(\neg\psi) = \mathrm{el}(\psi)$
- $\mathrm{el}(\psi_1 \wedge \psi_2) = \mathrm{el}(\psi_1) \cup \mathrm{el}(\psi_2)$
- $\mathrm{el}(\mathrm{X}\psi_1) = \{(\mathrm{X}\psi_1)^r\} \cup \mathrm{el}(\psi_1)$
- $\mathrm{el}(\psi_1 \vee \psi_2) = \mathrm{el}(\psi_1) \cup \mathrm{el}(\psi_2) \cup \{x^o \mid x^r \in \mathrm{el}(\psi_1) \cup \mathrm{el}(\psi_2)\}$
- $\mathrm{el}(\psi_1 \,\mathcal{U}\, \psi_2) = \{(\mathrm{X}(\psi_1 \,\mathcal{U}\, \psi_2))^r\} \cup \mathrm{el}(\psi_1) \cup \mathrm{el}(\psi_2)$
- $\mathrm{el}(\psi_1 \,\mathcal{R}\, \psi_2) = \{(\mathrm{X}(\psi_1 \,\mathcal{R}\, \psi_2))^r\} \cup \mathrm{el}(\psi_1) \cup \mathrm{el}(\psi_2)$.

We see from this that the propositions arising from formulae under an $\vee$ appear in optional and required forms. This is how the optional formation is used – in a formula $\psi_1 \vee \psi_2$ the conventional tableau construction would generate an automaton which chooses between three covering formulae: $\psi_1 \wedge \neg\psi_2$, $\neg\psi_1 \wedge \psi_2$, and $\psi_1 \wedge \psi_2$. In a game where the opponent can infinitely often choose between satisfying the fairness constraints of ψ_1 or ψ_2, this splitting can necessitate infinite shifting for the algorithm in Figure 2. By making the fairness constraints optional, we allow the tableau to follow the $\psi_1 \wedge \psi_2$ path as long as the play is consistent with the safety requirements of $\psi_1 \wedge \psi_2$ and consider the play to be accepted if it satisfies either the fairness constraints of ψ_1 or the fairness constraints of ψ_2.

Again like [4], we define a function sat() which takes an LTL formula and returns a formula representing the set of states in the tableau for which outgoing fair paths are labelled by plays which satisfy the LTL formula. It works uniformly for all $t \in \{r, o\}$. The only change from the standard definition is in

$\mathrm{sat}((\psi_1 \vee \psi_2)^r)$; here we always allow the possibility of the optional versions being taken instead of the required ones. Since the clause for the optional variables is $(\mathrm{sat}(\psi_1^o) \wedge \mathrm{sat}(\psi_2^o))$, the structure of the tableau ensures that paths are consistent with both formulae (i.e. they satisfy the non-fairness part of the formulae). Our new definition of the fairness constraints will allow one or the other to be satisfied. For all $t \in \{r, o\}$, $\mathrm{sat}(\psi^t)$ is defined as:

- $\mathrm{sat}(\Pi^t) = \Pi$
- $\mathrm{sat}((\psi_1 \wedge \psi_2)^t) = \mathrm{sat}(\psi_1^t) \wedge \mathrm{sat}(\psi_2^t)$
- $\mathrm{sat}((\psi_1 \vee \psi_2)^t) = \mathrm{sat}(\psi_1^t) \vee \mathrm{sat}(\psi_2^t) \vee (\mathrm{sat}(\psi_1^o) \wedge \mathrm{sat}(\psi_2^o))$
- $\mathrm{sat}((X\psi_1)^t) = (X\psi_1)^t$
- $\mathrm{sat}((\psi_1 \,\mathcal{U}\, \psi_2)^t) = \mathrm{sat}(\psi_2^t) \vee (\mathrm{sat}(\psi_1^t) \wedge X(\psi_1 \,\mathcal{U}\, \psi_2)^t)$
- $\mathrm{sat}((\psi_1 \,\mathcal{R}\, \psi_2)^t) = (\mathrm{sat}(\psi_1^t) \wedge \mathrm{sat}(\psi_2^t)) \vee (\mathrm{sat}(\psi_2^t) \wedge X(\psi_1 \,\mathcal{R}\, \psi_2)^t)$.

The transition relation ensures that if $X\psi$ occurs in a state, all fair paths from all successors of that state satisfy ψ.

$$\bigwedge_{t \in \{r,o\}} \bigwedge_{(X\psi_1)^t \in \mathrm{el}(\psi^r)} (X\psi_1)^t \Rightarrow \mathrm{sat}(\psi_1^t)'. \tag{8}$$

This differs from [4] by the inclusions of optional/required tags and by using $\Rightarrow$ instead of $\Leftrightarrow$. This relaxation is possible because the input formulae are in negation normal form and its soundness is implied by Theorem 4.

The fairness constraints on the tableau are defined by another new function, fsat(). Conventionally, the fairness constraints for a formula ψ would require that for each sub-formula of the form $\psi_1 \,\mathcal{U}\, \psi_2$ a fair path infinitely often has either $\neg X(\psi_1 \,\mathcal{U}\, \psi_2)$ or $\mathrm{sat}(\psi_2)$ i.e. at any point, either $\psi_1 \,\mathcal{U}\, \psi_2$ is not required or it is eventually satisfied. Our definition of fsat is based on this notion, but allows for the special case of optional variables. Fairness on a path π is defined by the following function:

- $\mathrm{fsat}(\Pi^t) = \top$
- $\mathrm{fsat}((\psi_1 \vee \psi_2)^t) = \mathrm{fsat}(\psi_1^t) \wedge \mathrm{fsat}(\psi_2^t) \wedge (\mathrm{fsat}(\psi_1^o) \vee \mathrm{fsat}(\psi_2^o))$
- $\mathrm{fsat}((\psi_1 \wedge \psi_2)^t) = \mathrm{fsat}(\psi_1^t) \wedge \mathrm{fsat}(\psi_2^t)$
- $\mathrm{fsat}((X\psi_1)^t) = \mathrm{fsat}(\psi_1^t)$
- $\mathrm{fsat}((\psi_1 \,\mathcal{U}\, \psi_2)^t) = \mathrm{fsat}(\psi_1^t) \wedge \mathrm{fsat}(\psi_2^t)$
  $\wedge\, \exists^\infty i \geq 0.\pi[i] \in \mathrm{sat}(\psi_2^t) \vee \neg(X(\psi_1 \,\mathcal{U}\, \psi_2))^t$
- $\mathrm{fsat}((\psi_1 \,\mathcal{R}\, \psi_2)^t) = \mathrm{fsat}(\psi_1^t) \wedge \mathrm{fsat}(\psi_2^t)$.

We see that the only departure from the conventional usage (a convention observed by [4]) is in allowing one or the other of a pair of optional variables to be satisfied.

Since our construction is a relaxation of the one given by [4], we do not prove its completeness. However, its soundness is asserted by the following theorem:

Theorem 4. *Let ψ be an LTL formula over a set, $\mathcal{P}$, of atomic propositions. Let T be the symbolic tableau automaton for ψ constructed as above. For any $\psi_1 \in \mathrm{sub}(\psi)$, any $t \in \{r, o\}$ and any path, π, in T, for all $i \geq 0$ if $\pi[i] \in \mathrm{sat}(\psi_1^t)$ and $\pi[i, \infty]$ satisfies $\mathrm{fsat}(\psi_1^t)$ then $\pi[i, \infty] \vDash \psi_1$.*

Using this tableau construction we can perform the entire synthesis procedure symbolically. First the LTL formula is translated in the manner described above. This tableau is then used as the Büchi automaton in the algorithm from Figure 2 which can also be computed symbolically. Despite the doubly exponential worst-case complexity of this procedure we shall see in the following section that useful results can be computed. First, we note that for request-response specifications [20], strategies can be synthesised without generating a shift automaton i.e. the tableau for such specifications is trivially determinisable. Request-response specifications can be written in LTL as $G(p \wedge (r_0 \Rightarrow Fs_0) \wedge \ldots \wedge (r_n \Rightarrow Fs_n))$ and we prove the following proposition in the appendix.

Proposition 1. *The tableau for LTL formulae of the form* $G(p \wedge (r_0 \Rightarrow Fs_0) \wedge \ldots \wedge (r_n \Rightarrow Fs_n))$ *(where* p*,* r_i*, and* s_i *are propositional formulae) is always trivially determinisable.*

5 Implementation

The synthesis algorithm has been implemented in Java using native calls to the CuDD [5] library to handle BDDs. As input, the program takes an XML file containing a symbolic description of the game and an LTL specification. After successful synthesis, a number of output options are possible: the program can print the set of winning states; produce an explicit graph of the winning strategy with dot; show an interactive, expandable tree of the strategy so that the user can play it out; and convert the strategy into a program in the language of the Cadence SMV model checker [3]. The output to SMV can be used to check the correctness of the implementation by checking against the original LTL specification (once a winning strategy has been synthesised, it is possible to view the strategy as a closed system and check for correctness on all paths).

5.1 Examples

Mutual Exclusion. In this example we synthesise a controller to enforce mutual exclusion. We solve this problem for various numbers of processes in order to get a measure of the scalibility of the implementation, using n as the parameter for size.

The game is modelled with the following boolean variables: u indicates the current turn and alternates between moves. When u is true, it is the user processes turn, when it is false it is the system's turn. $r1, r2, \ldots, rn$ indicate that a process requesting access to its critical section. When a request is made, it cannot be withdrawn until the critical section is reached. Furthermore, when it is the system's turn, the request variables must keep their old values. $c1, c2, \ldots, cn$ indicate that a process is in its critical section. ci can only become true if it is the system's turn and ri is true. When it is true, ci will become false in the next turn.

The specification of mutual exclusion and liveness can easily be written in in LTL as

$$G(\bigwedge_{i \in [1,n]} ((ri \Rightarrow Fci) \wedge \bigwedge_{j \in [i+1,n]} \neg(ci \wedge cj)))$$

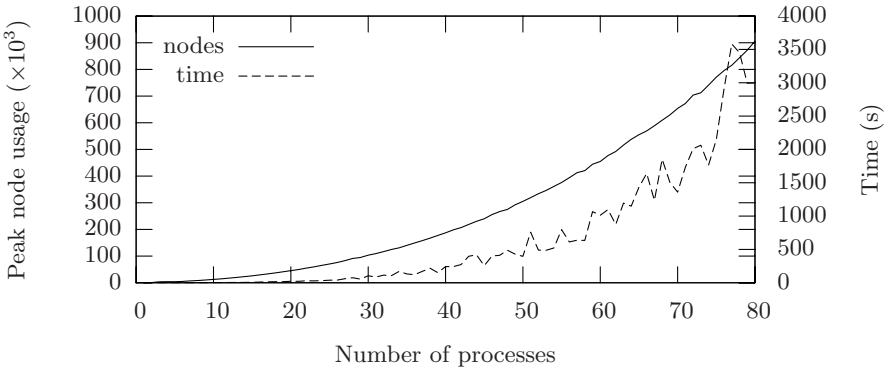

Fig. 3. Mutual exclusion performance data

For two processes, the synthesis implementation ran in about 30ms and produced a strategy which plays out Peterson's algorithm for mutual exclusion. Whenever both processes request at once, one process (say process 1) is chosen non-deterministically to proceed. From then until process 2 gets to enter its critical section, if both processes request at once then process 2 will be favoured. This behaviour is symmetric for both processes, guaranteeing that neither process starves. Strategies were found for problems with between 2 and 80 processes, recording the "peak live node usage" and time for each problem. The peak live node usage is a statistic gathered by the CuDD library denoting the maximum number of BDD nodes that have been used during a computation. The tests were run on a 1.5GHz Pentium 4 system with 256MB of RAM running Linux. The results are shown in Figure 3. We note that even with 80 processes, which gives a state-space of $2^{(80\times 2)+1} \approx 10^{48}$ and a formula with 80 liveness sub-formulae, the time taken was about 50 minutes. This is quite reasonable for a model checking tool and the overall growth in the two plots shows the feasibility of the approach.

Lift System. Here we synthesise a controller for a lift (elevator) system. The game describes some general behaviour regarding the physical situation of a lift (there are user-controlled buttons, movement between floors is consecutive etc.), and the LTL specification puts requirements on the actual controller. We model the lift system with various numbers of floors, using n as a parameter for the size; the variables used to describe the system are: u which indicates the current turn and works as in mutual exclusion. f which indicates the current floor. This is modelled with $\lceil log(n) \rceil$ variables which we treat as a single integer variable. We write $f[0], f[1], \ldots f[n]$ to denote floors. Initially, the floor is 0; we require that transitions between floors are consecutive and that the floor does not change on users turns. $b[0], b[1], \ldots, b[n]$ are the button variables. These boolean variables are controlled by the users to simulate requests for the lift. Initially, all buttons

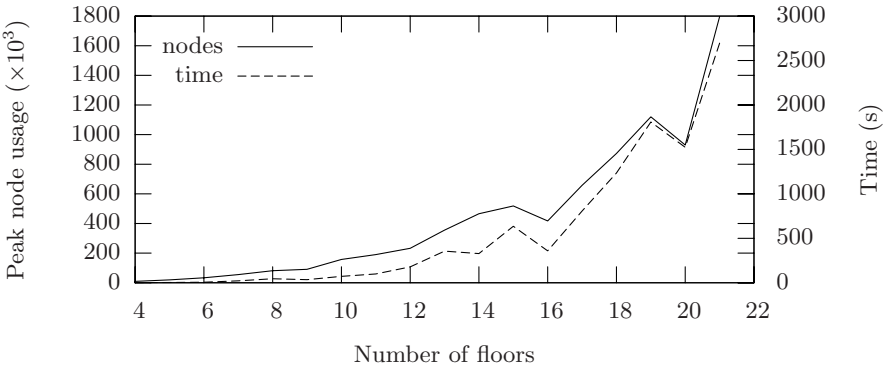

Fig. 4. Lift performance data

are off; once lit, a button stays on until the lift arrives and the buttons do not change on the system's turns. *up* is a boolean variable which observes the transition between floors and is true if the lift is going up. Initially *up* is false. The specification that we use for the lift system is as follows:

$$G\Big(\big(\bigwedge_{i\in[0,n]} b[i] \Rightarrow Ff[i]\big) \wedge \bigwedge_{i\in[0,n]} f[i] \wedge \neg sb \Rightarrow f[i]\,\mathcal{U}\,(sb\,\mathcal{R}\,(F(f[0] \vee sb) \wedge \neg up))\Big)$$

Where *sb* is an abbreviation for $\bigvee_{i\in[1,n]} b[i]$ to mean "some button is lit". The first conjunct says that every request is eventually answered, the second demands that the lift should park when it is idle. If we synthesise a strategy without the parking specification, we find that the lift does answer all calls, but it does so by moving up and down continuously regardless of what calls are made (this was apparent from playing out the strategy and verified formally using SMV). In the second conjunct, the release formula is the actual parking action: the lift should go to $f[0]$ by going continuously down, unless some button is pressed. The rest of the formula can be read as: if the lift is on floor i and no button (other than 0) is pressed, then remain at floor i until parking can commence. Synthesising a strategy for the entire specification, we find that the lift now behaves as expected and, once again, we can verify that the strategy implements the specification by using SMV. The results for a range of sizes are plotted in Figure 4 – we see the time taken and the number of nodes used rising dramatically. This is due to the size and complexity of the specification resulting in an extremely large tableau. The fluctuations in time taken and nodes used are hard to explain. One possible cause could be the heuristic nature of the re-ordering algorithms in CuDD. Similar fluctuations are seen in the shifting example below, but they seem to be magnified here by the size of the state-space increase as each extra floor is added to the lift problem.

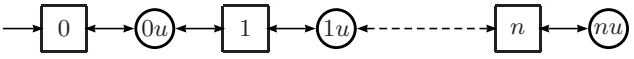

Fig. 5. Game for shifting example

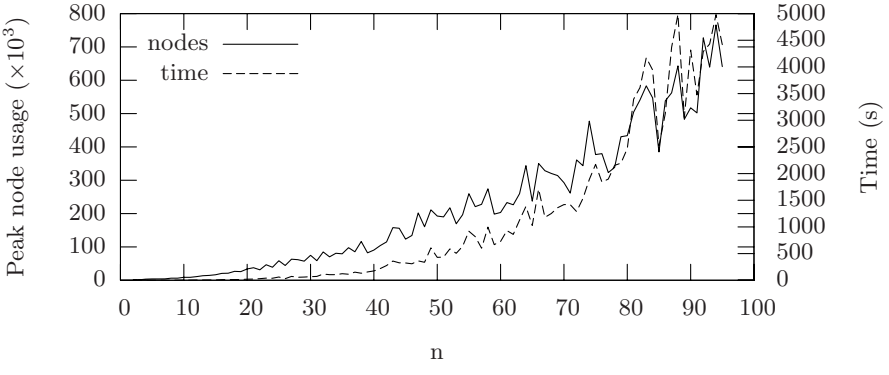

Fig. 6. Shift performance data

A Shifting Problem. The previous two examples were computed without having to use any shifting. Although a number of specifications were tried in the context of these systems, shifting was never required. It is clear that shifting is necessary for some specifications, though, so we use an abstract example based on the one from Figure 1 to measure performance in such cases. The game is shown in Figure 5.

The specification used is simply $FG0 \vee FG1 \vee \ldots \vee FGn$. For $n = 2$, this produces a tableau with the same property as the Büchi automaton in Figure 1 – it has two disjoint regions, one for $FG0$ and one for $FG1$. In order to solve this game, the algorithm needs to consider shifting. As expected, the synthesis algorithm generates a shift automaton and terminates using one shift. The winning strategy simply keeps play in 1 if the opponent ever chooses it, otherwise it has no choice but to stay in 0. Figure 6 shows the performance over a range of sizes. Even with a length of 95, and having to generate the doubly exponential shift automaton, the implementation ran in 1 hour 14 minutes.

6 Conclusion

We have provided a new algorithm and corresponding symbolic implementation to solve the problem of strategy synthesis in LTL games. Whilst this algorithm is not complete, it has performed well in the test-cases that were given to it and holds enough promise to warrant use on larger, real-world problems in the

future. The separation between the partial solution to NB games and the symbolic tableau method itself offers the prospect of future improvements to the completeness result by changes to the tableau method. It would be interesting to see whether the completeness result given in terms of automata could be related back to a fragment of LTL so that it might be compared with such work as [2]. At present, there are formulae for which we know that the algorithm here does not perform optimally and there are formulae which cannot be dealt with in fragments solved by [2], but can be dealt with by our work.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.
2. R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Transactions on Computational Logic*, 5(1):1–25, January 2004.
3. SMV 10-11-02p1. `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`, November 2002.
4. E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *CAV94*, volume 818, pages 415–427. Springer-Verlag, 1994.
5. CuDD: Colorado university decision diagram package, release 2.30. `http://vlsi.colorado.edu/~fabio/CUDD/`, February 2001.
6. E. A. Emerson. *Handbook of Theoretical Computer Science Volume B*, chapter Temporal and Modal Logic, pages 995–1072. Elsevier, 1990.
7. E. A. Emerson and C. Lei. Efficient model checking in fragments of the propositional model mu-calculus. In *IEEE Symposium on Logic in Computer Science*, pages 267–278, June 1986.
8. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logic, and Infinite Games*. Number 2500 in Lecture Notes In Computer Science. Springer, 2002.
9. S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. *Journal Of Computer Security*, 11(3):399–429, 2003.
10. O. Kupferman and M. Y. Vardi. Module checking. In *8th Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 75–86, 1996.
11. C. Löding. Optimal bounds for the transformation of ω-automata. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1738 in LNCS, pages 97–109. Springer, 1999.
12. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings Of 16th ACM Symposium On Principles Of Programming Languages*, pages 179–190, 1989.
13. R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.
14. S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, March 1989.
15. K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. *Lecture Notes in Computer Science*, 2250, 2001.
16. F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic scc hull algorithms. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 88–105. Springer-Verlag, 2002.

17. S. Tasiran, R. Hojati, and R. K. Brayton. Language containment using non-deterministic omega-automata. In *Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME'95*, number 987 in Lecture Notes In Computer Science, pages 261–277, 1995.
18. W. Thomas. *Handbook of Theoretical Computer Science Volume B*, chapter Automata on Infinite Objects, pages 133–192. Elsevier, 1990.
19. W. Thomas. On the synthesis of strategies in infinite games. In *Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995.
20. N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proceedings of the 8th International Conference on the Implementation and Application of Automata, CIAA*, volume 2759 of *Lecture Notes in Computer Science*, pages 11–22, 2003.