Towards Symbolic Strategy Synthesis for $\langle\langle A \rangle\rangle$-LTL

Aidan Harding, Mark Ryan
School of Computer Science
University of Birmingham
Edgbaston
Birmingham B15 2TT
UK
ath,mdr@cs.bham.ac.uk

Pierre-Yves Schobbens
Institut d'Informatique
Facultés Universitaires de Namur
Rue Grandgagnage 21
5000 Namur
Belgium
pyschobbens@info.fundp.ac.be

Abstract

We provide a symbolic algorithm for synthesizing winning strategies in Alternating Time Temporal Logic. ATL has game semantics, which makes it suitable for modelling open systems where coalitions of agents work together to achieve their goals. A typical use of the algorithm would begin with a highly non-deterministic set of agents, A, for which we wish to synthesize behaviour. These may be composed with a set of opponent agents, which provide an environment. The desired behaviour is then written in $\langle\langle A \rangle\rangle$-LTL, and a strategy for A to implement this is synthesised. If A implements this strategy, then the system will be guaranteed to satisfy the desired property.

The algorithm presented here is part of a work in progress and updates can be found at `http://www.cs.bham.ac.uk/~ath/atl_synthesis`.

1 Introduction

The aim of this work is to synthesize strategies for infinite games. These games can serve as models for reactive programs, and thus a winning strategy can be used as a prescription to automatically generate program code. The formalisms that we use to describe these games are Alternating Transition Systems (ATS) and Alternating Time Temporal Logic (ATL) [1]. The algorithm used for this synthesis is very similar to that used for model checking ATL* (which checks for the existence of winning strategies). In fact, the synthesis algorithm runs the model checking procedure at the same time as recording the strategy. Using an extra automaton to summarise the history of the computation, we generate a stateful strategy.

The implementation technology for our synthesis algorithm will use the BDD approach of symbolic model checkers [6, 8]. Symbolic model checking has allowed the size of computable examples to be increased by orders of magnitude, and to our knowledge, has not hitherto been applied to program synthesis. This encoding is more than a mere detail of the implementation – it informs how the synthesis algorithm is written. By keeping one eye on the symbolic implementation, we can not only tackle larger problems, but also hope to produce a prototype quickly by modifying/extending the symbolic model checker, MOCHA [11].

The games we consider are played by a set of agents, Σ, and proceed in rounds. At each round, all players move simultaneously and the combination of their moves decide a unique successor state for the game. The specifications (winning conditions for the game) are written in Alternating Time Temporal Logic (the specific fragment is $\langle\langle A \rangle\rangle$-LTL, see Section 2.2) – they assert a capability for some set of agents $A \subseteq \Sigma$. A strategy is a function which takes a sequence of states in the game (the history), and returns a choice for a given agent. If, by following a particular strategy, an agent will satisfy the winning condition, it is called a winning strategy. The games are not determined i.e. for a given state there may be no winning strategy for either the agents, A, or their opponents, V. In the notation of ATL, we may have $\neg\langle\langle A \rangle\rangle\phi \wedge \neg\langle\langle V \rangle\rangle\neg\phi$.

Synthesizing strategies has a number of potential benefits:

- For program synthesis: Once we have obtained a (finite state) winning strategy in a reactive system, G, it is a relatively trivial matter to encode that strategy into a new system G' which implements the strategy. The relationship between these two systems can be defined as a refinement up to the hiding of any new variables, and whilst G satisfies $\langle\langle A \rangle\rangle\phi$, G' will satisfy $\forall\phi$ i.e. if we ignore the values of new variables, any trace of G' is a possible trace of G, and all traces of G' satisfy ϕ.

Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and
Fourth International Conference on Temporal Logic (TIME-ICTL'03) 1530-1311/03 $17.00 © 2003 IEEE

- For verification: we can provide more than a yes/no answer.

  In the positive case, it may be that the specification is satisfied vacuously (e.g. a badly formed ATL* specification such as $\langle\langle A\rangle\rangle(switch = on \rightarrow \mathrm{F}light = green)$ is satisfied if the initial state satisfies $\neg(switch = on)$). Providing a winning strategy may reveal such errors in specification.

  In the negative case, model checkers usually provide a counter example in the form of a trace to show how a property is violated. This trace may not be the most useful one to help the programmer in finding a bug. A counter-strategy would encapsulate many different possible traces and thus it could provide a more intuitive explanation of why a property fails.

In the following section we provide an introduction to Alternating Time Temporal Logic; Section 3 describes the synthesis algorithm (with a running example); Section 4 discusses the corretness of the algorithm; and Section 5 contains some concluding remarks.

2 Alternating-Time Temporal Logic

Alternating-Time Temporal Logic [1] (ATL) is a temporal logic for reasoning about *reactive systems* comprised of *agents*. It contains the usual temporal operators (next, always, until) plus cooperation modalities, e.g. $\langle\langle A\rangle\rangle\phi$, where A is a set of agents. This modality quantifies over the set of behaviours of the system and means that A have a collective strategy to enforce ϕ, whatever the choices of the other players. ATL generalises CTL, and similarly ATL* generalises CTL*, μ-ATL generalises the μ-calculus. These logics can be model-checked by generalising the techniques of CTL, often with the same complexity. For our purposes we shall concentrate on the fragment that we have termed $\langle\langle A\rangle\rangle$-LTL

2.1 Alternating Transition Systems

ATL is interpreted over Alternating Transition Systems (ATS) which are Kripke structures, extended to represent the choices of agents.

An ATS is a 5-tuple $\langle \Pi, \Sigma, Q, \pi, \delta\rangle$ where

- Π is a set of propositions
- Σ is a set of agents
- Q is a set of states
- $\pi : Q \rightarrow 2^{\Pi}$ maps each state to the propositions which are true in that state
- $\delta : Q \times \Sigma \rightarrow 2^{2^Q}$ is a transition function from a state, q, and an agent, a, to the set of a's choices. a's choices are sets of states, and one particular choice, Q_a, is taken by each agent in each round of the game. Successive states of the system are found by taking the intersection of one choice each for all agents $\bigcap_{a\in\Sigma} Q_a$.

  The transition function is *non-blocking* and *unique* i.e. for every state, and vector of choices, one for each agent, the intersection of these choices is singleton.

For two states q, q' and an agent a, q' is an *a-successor* of q if there exists some $Q_a \in \delta(q, a)$ such that $q' \in Q_a$. The set of a-successors of q is denoted $succ(q, a)$. For two states q and q', q' is a *successor* of q if $\forall a \in \Sigma\ q' \in succ(q, a)$. A computation, λ, is defined as an infinite sequence of states $q_0, q_1, q_2, \ldots$ such that for all $i \geq 0$, q_{i+1} is a successor of q_i.

Subsegments of a computation path $\lambda = q_0, q_1, \ldots$ are denoted by postfixing an interval in square brackets. For example, $\lambda[i, j] = q_i, \ldots, q_j$, $\lambda[i, \infty] = q_i, \ldots$ and $\lambda[i] = q_i$.

2.2 $\langle\langle A\rangle\rangle$-LTL Syntax

Let Π be a set of atomic propositions and Σ a set of agents. Formulae of $\langle\langle A\rangle\rangle$-LTL are formulae of LTL, prefixed with a cooperation modality. The grammar of $\langle\langle A\rangle\rangle$-LTL is given by:

$$\phi ::= P \mid \langle\langle A\rangle\rangle\phi$$
$$P ::= p \mid \top \mid \neg P \mid P_1 \vee P_2$$
$$\phi ::= P \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathrm{X}\phi \mid \phi_1 \mathcal{U} \phi_2 \mid \phi_1 \mathcal{R} \phi_2$$

where $p \in \Pi$ are atomic propositions, and $A \subseteq \Sigma$ is a set of agents.We use the usual abbreviations for $\rightarrow$, $\wedge$ in terms of $\neg$, $\vee$. The operator $\langle\langle\ \rangle\rangle$ is a path quantifier, and X, $\mathcal{U}$ (*until*) and $\mathcal{R}$ (*release*) are temporal operators. As in CTL, we write Fϕ for $\top\mathcal{U}\phi$, and Gϕ for $\bot\ \mathcal{R}\phi$.

2.3 $\langle\langle A\rangle\rangle$-LTL Semantics and Strategies

To define the semantics of $\langle\langle A\rangle\rangle$-LTL, the notion of *strategies* is used. A strategy for an agent a is a mapping $f_a : Q^+ \rightarrow 2^Q$ such that for all $\lambda \in Q^*$ and all $q \in Q$, we have $f_a(\lambda \cdot q) \in \delta(q, a)$. The strategies map finite prefixes of λ-computations to a choice in $\delta(q, a)$ as suggested by the strategy. We also define collaborative strategies for a set of agents, A. A collaborative strategy is a mapping $f_A : Q^+ \rightarrow 2^Q$ such that for all $\lambda \in Q^*$ and all $q \in Q$, we have $f_A(\lambda \cdot q) \in \{Q_A \mid Q_A \in \bigcap_{a\in A} Q_a, Q_a \in \delta(q, a)\}$.

The *outcomes* of a strategy must also be defined. For a state q, a set of agents A, and a family of strategies $F_A = \{f_a \mid a \in A\}$ the outcomes of F_A from q are

denoted $out(q, F_A)$. They are the q-computations that the agents in A can enforce by following their strategies. $\lambda = q_0, q_1, q_2 \ldots$ is in $out(q, F_A)$ if $q = q_0$ and for all positions $i \geq 0$ q_{i+1} is a successor of q_i satisfying $q_{i+1} \in \bigcap_{a \in A} f_a(\lambda[0, i])$.

The semantics of $\langle\langle A \rangle\rangle$-LTL are defined inductively, with propositional formulae, negation, and disjunction handled in the usual way:

- $q \vDash \langle\langle A \rangle\rangle \phi$ iff there exists a set, F_A, of strategies, one for each agent in A, such that $\forall \lambda \in out(q, F_A)$, we have $\lambda \vDash \phi$
- $\lambda \vDash \mathrm{X}\phi$ iff $\lambda[1, \infty] \vDash \phi$
- $\lambda \vDash \phi_1 \mathcal{U} \phi_2$ iff $\exists i \geq 0 . \lambda[i, \infty] \vDash \phi_2$ and $\forall 0 \leq j < i \lambda[j, \infty] \vDash \phi_1$.
- $\lambda \vDash \phi_1 \mathcal{R} \phi_2$ iff $\forall i \geq 0$, we have $\lambda[i, \infty] \vDash \phi_2$ unless there exists a position $0 \leq j < i$ such that $\lambda[j, \infty] \vDash \phi_1$.

3 The Synthesis Procedure

If $q \vDash \langle\langle A \rangle\rangle \phi$, then there is a strategy for A to obtain ϕ from q. Given a system, we aim to check for the existence of a winning strategy. When there are possible winning strategies, we will synthesize one. The synthesis procedure is summarised below, and the following subsections explain each step in detail. We first construct some automata, and then run the synthesis algorithm on the composition of these automata. An example accompanies each step of the process.

The inputs to the algorithm are an Alternating Transition System, G, representing the program (game) and a specification, $\langle\langle A \rangle\rangle \phi$, in $\langle\langle A \rangle\rangle$-LTL. The outputs are a set of states in G for which $\langle\langle A \rangle\rangle \phi$ holds, and a strategy which ensures ϕ from those good states. At an abstract level, the synthesis runs as follows:

1. Generate a Büchi automaton for ϕ. This is done in the standard way [10], with some care taken to ensure that it can be done symbolically. We call it the *specification automaton* and write is as B. Let N be its set of accepting states.
2. Create a *memory automaton*, M, for ϕ. This is a partially-determinised version of B, where each state is a set of states from B. For efficiency, we do not complete the determinisation by taking into account accepting states. Rather, we will use M and B together. M has the advantage that its transitions are uniquely determined by the propositions of G, so it will be deterministic when they are run together.
3. Use a modified version of the Emerson-Lei algorithm [4] on all three automata G, B, and M. The algorithm will find the states in G from where A can force paths that are accepted by B, i.e. they reach N infinitely often. During this calculation, the necessary steps for A are recorded into a strategy which relies on the current state of M to summarise the history of the game.

3.1 Generating the Specification Automaton

We are presented with an ATS, $G = \langle \Pi, \Sigma, Q, \pi_s, \delta \rangle$, and a specification $\langle\langle A \rangle\rangle \phi$. The parts of the ATS are defined in Section 2.1 above, and ϕ is simply an LTL formula over Π.

The first step of the algorithm is to construct a symbolic representation of the tableau automaton, B, for ϕ. This construction is similar to that given by [3]. The state space of this automaton is $Q_B = Q \times Q_T$ where Q_T is the set of states over a new set of propositions, T, introduced to characterise each temporal operator in ϕ. T is made up of three sets: T_U, T_R, and T_X. For each occurrence of a temporal operator in ϕ, we introduce a new proposition. For $\phi_1 \mathcal{U} \phi_2$, we introduce $p_{\mathrm{X}(\phi_1 \mathcal{U} \phi_2)} \in T_U$. For $\phi_1 \mathcal{R} \phi_2$, we introduce $p_{\mathrm{X}(\phi_1 \mathcal{R} \phi_2)} \in T_R$. For $\mathrm{X}\phi$, we introduce $p_{\mathrm{X}\phi} \in T_X$. The original formula can now be re-written in terms of $\Pi \cup T$ by recursively applying the following rules:

$$(\phi_1 \mathcal{U} \phi_2) \Leftrightarrow \phi_2 \vee (\phi_1 \wedge p_{X(\phi_1 \mathcal{U} \phi_2)}) \qquad (E_U)$$

$$(\phi_1 \mathcal{R} \phi_2) \Leftrightarrow \phi_2 \wedge (\phi_1 \vee \wedge p_{X(\phi_1 \mathcal{R} \phi_2)}) \qquad (E_R)$$

$$\mathrm{X}\phi \Leftrightarrow p_{\mathrm{X}\phi} \qquad (E_X)$$

By performing this translation once on ϕ, we obtain ϕ_0 which is in terms of $\Pi \cup T$. The set of initial states of B is $\{q_0 \mid q_0 \in Q \times Q_T) \wedge q_0 \vDash \phi_0\}$ i.e. the set of states which are consistent with ϕ_0. Of course, this set is easily represented by the BDD for ϕ_0. The rule E_T, below, allows us to define the transitions.

$$p_{X\phi_1} \Leftrightarrow \phi_1' \qquad (E_T)$$

To find the successors of a state, $q \in Q_B$, we forget the Π part of the state and apply E_T to each proposition in T (these are the propostitions which describe the future). We write this application as $E_T(q)$, to mean $\bigwedge_{p \in (T \cap \pi(q))} E_T(p)$. This gives a new formula over $\Pi' \cup T'$, which describes the next state. We simply strip the primes, and start the procedure again by replacing temporal operators using E_U, E_R, and E_X.

Clearly, all of this can be done symbolically. The sets of states are defined by formulae over their propositions, so these provide the functions that the BDDs will represent. Each of the subsequent operations on the sets of states can be performed by substitutions.

```
module Server
  external  request : bool
  interface send    : bool

  atom controls send
  reads request, send
  update
    [] true -> send' := nondet
  endatom
endmodule

System := Client || Server


module Client
  interface request : bool

  atom controls request
  update
    [] true -> request' := nondet
  endatom
endmodule
```

Figure 1. MOCHA **code for the example system**

Example To demonstrate the algorithm we shall introduce a running example. The MOCHA code for the input system is given in Figure 1. A Client can make requests as often as it likes, and the Server's behaviour is to nondeterministically send responses (ignoring the input provided). The (fairly trivial) specification formula is $\phi = \langle\langle S \rangle\rangle \mathrm{G}(request \rightarrow \mathrm{F}send)$ (we will abbreviate $request$ as r and $send$ as s, thus $Q = \{r, s\}$). The synthesised system should insure that any request from the Client is eventually followed by a response from the server.

Constructing the Büchi automaton proceeds as follows:

1. Calculate the set of new variables: $T = \{p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}, p_{\mathrm{XF}s}\}$
2. Rewrite the LTL part of ϕ in terms of Π and T to obtain $\phi_0 = (r \rightarrow (s \vee p_{\mathrm{XF}s})) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$
3. The set of initial states are those that satisfy ϕ_0. Explicitly,
   $\{\{p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}\}, \{r, s, p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}\},$
   $\{r, p_{\mathrm{XF}s}, p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}\}, \{s, p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}\},$
   $\{p_{\mathrm{XF}s}, p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}\}\}$
4. Conditions on succesor states are calculated by applying E_T to the inital states, then stripping primes and making substitutions for temporal operators with E_U and E_R.

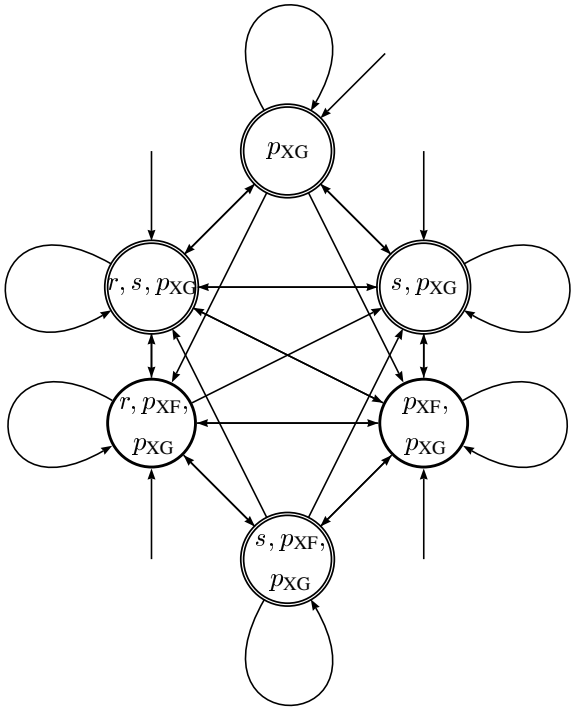

Figure 2. The nondeterministic Büchi automaton for $\mathrm{G}(request \rightarrow \mathrm{F}send)$**. (**$p_{\mathrm{XF}}$ **is used to abbreviate** $p_{\mathrm{XF}s}$**, and** p_{XG} **for** $p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$**). The propositions in** Π **are drawn on states rather than arrows for brevity. If in state** $q_B \in B$**, the automaton reads the input** $q \in Q$**, it may transition to any succesor which contains** q **in its label. Double circles denote accepting states.**

$$
\begin{aligned}
E_T(p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}) &= (G(r \rightarrow \mathrm{F}s))' \\
&\rightsquigarrow (r \rightarrow (s \vee p_{\mathrm{XF}s})) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)} \\
E_T(r \wedge p_{\mathrm{XF}s} \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}) &= (\mathrm{F}s \wedge G(r \rightarrow \mathrm{F}s))' \\
&\rightsquigarrow (s \vee p_{\mathrm{XF}s}) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)} \\
&\vdots
\end{aligned}
$$

5. This gives us a new set of states from which to search for successors. The process is repeated until no new successors are found. The automaton is drawn out explicitly in Figure 2.

3.1.1 Adding Fairness

So far, B does not force liveness formulae to be satisfied. We will add fairness constraints to address this, and thus enforce the semantics of LTL. For each subformula $u_i = \phi_1 \mathcal{U} \phi_2$ of ϕ, we introduce a fairness constraint $c_i = \neg u_i \vee \phi_2$. In terms of $\Pi \cup T$, $c_i = \neg p_{\mathrm{X}u_i} \vee \neg \phi_1 \vee \phi_2$. This allows for loops where u_i is always false, but as soon as it becomes

true, E_U and E_T ensure that c_i can only be satisfied by ϕ_2 being encountered.

A Büchi automaton accepts paths which visit the set of accepting states, N, infinitely often. When we have a set, F, of constraints where $|F| > 1$, B must be augmented by extra variables in order to construct N. These variables will monitor the satisfaction of the fairness constraints without affecting the existing transitions. The set of accepting states will then be those where the monitors bear witness to all fairness constraints having been satisfied.

We use one bit, n_i for each constraint, c_i. n_i values are checked off as the constraints that they observe are satisfied. When they are all satisfied, the bits are reset. We distinguish the states where all n_i are true as the set of necessary states, N; we distinguish the initial state of the fairness constraints, where all n_i are false, as N_0. The transitions of n_i are characterised by E_F, below.

$$\forall i \geq 0\ init(n_i) = \bot$$
$$\forall i \geq 0\ n_i' = \left(\bigvee_{j\geq 0} \neg n_j \vee \bigvee_{j\geq 0} \neg c_j\right) \wedge (n_i \vee c_i) \qquad (E_F)$$

So we extend the state space of B to include a new set of propositions, N, whose values are defined by E_F. We require that N is true infinitely often. This completes the construction of the specification automaton.

Example Continuing with the example from above, there is one fairness constraint $u_0 = \neg p_{\mathrm{XF}s} \vee s$. So there is no need to involve E_F. The Büchi condition is that an accepting path must go through one of the following states infinitely often (drawn with double circles in Figure 2):
$\{\{p_{\mathrm{XG}(r\rightarrow \mathrm{F}s)}\}, \quad \{r, s, p_{\mathrm{XG}(r\rightarrow \mathrm{F}s)}\}, \quad \{s, p_{\mathrm{XG}(r\rightarrow \mathrm{F}s)}\}, \{s, p_{\mathrm{XF}s}, p_{\mathrm{XG}(r\rightarrow \mathrm{F}s)}\}\}$

3.2 Constraints on the Specification Automaton

In order for the synthesis algorithm to work correctly, we need to ensure that B satisfies certain criteria. More detailed reasoning can be found in Section 4, but here we introduce the terms and show how they apply to our construction.

For the interaction between the specification and the memory to be correct, we would like B to be *globally reverse deterministic* [5] ([2] calls such automata unambiguous). This means that each accepted word has exactly one path in the automaton, or to put it another way, the languages accepted by any two states in B are disjoint. In practice, this is too strong a restriction, and we define a relation, $\sim$, on the states so that we can have B *globally reverse deterministic up to* $\sim$. We define $\sim$ is such a way as any two states in B either have disjoint languages or they accept the same language and are related by $\sim$ i.e.

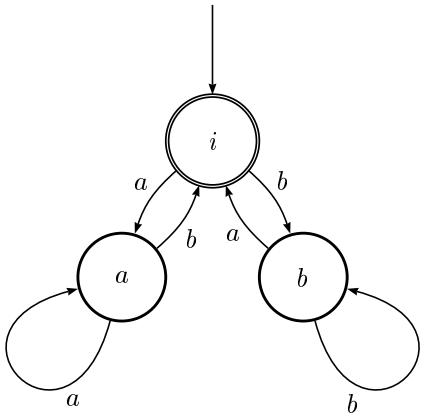

Figure 3. A Büchi automaton to accept a language with infinite number of as and bs

$$\mathcal{L}(s) = \mathcal{L}(t) \implies s \sim t \qquad (1)$$
$$s \not\sim t \implies \mathcal{L}(s) \cap \mathcal{L}(t) = \emptyset \qquad (2)$$

This relation is required because our automaton deals with fairness. Consider Figure 3, it shows an automaton to accept the language with an infinite number of as and bs. Since the as and bs could occur in any order, by necessity, all of the states accept the same language. When we impose fairness constraints on B, much the same thing happens: We get distinct states that accept the same language.

By the construction of B we know that it will be globally reverse deterministic upto $\sim$. We further need it to be *locally reverse deterministic up to* $\sim$. This means that any two states which transition into the same state, with the same label are related by $\sim$.

$$\forall s, t \in Q_B \frac{\delta_B(s, a) \cap \delta_B(t, a) \neq \emptyset}{s \sim t} \qquad (3)$$

If the language of a state is empy, we call it a *dead state*. By removing dead states, we can ensure that our globally reverse deterministic automata is also locally reverse deterministic; therefore we remove the dead states. To do this, we use the algorithm in Figure 4. This finds the set of states which can reach N infinitely often, thus those that have a non-empty language. It works with a nested fixpoint computation, accumulating its result in the variable z. Initially, our target, τ, is just N and we perform reachability to find the set of states, s which can reach N. This is the new value for z; and the new target, τ, is $z \cap N$ i.e. the states in N that can reach N again once. We then do reachability on τ to find the states which can reach N twice. The process is repeated until z reaches a fixpoint, at which time z is the set of states that can reach N infinitely often.

$z := \top$
repeat counted by k
    $s := \emptyset$
    $\tau := z \cap N$
    repeat counted by l
        $u := pre(s \cup \tau)$
        $s := s \cup u$
    until s stabilises
    $z := z \cap s$;
until z stabilises

Figure 4. The Emerson-Lei algorithm. NB $pre(X)$ returns the set of states which can transition to X in one step

3.3 Creating the Memory Automaton

As the name suggests, the memory automaton, M, will provide the memory for the strategy. Intuitively, the specification tells us exactly what we need to remember. We could start off with the specification formula, and modify it as requirements are added or met. In fact, this is exactly what B does. Looking at Figure 2, we see that whenever an r is read, unless it is matched with an s, the $p_{\mathrm{XF}s}$ remains until an s is seen. However, we cannot use B as the memory because it is non-deterministic. When the strategy is to be used, we would have to record not just a move in the game, but also a move in the memory. Instead, we partially determinise B with the subset construction to give an automaton which has deterministic transitions, and enough states to characterise every run of B. Although B is (forwards) non-deterministic, it is reverse deterministic and will only be used in the backwards computation of the synthesis algorithm. This combination of forwards and reverse determinism are enough to generate a strategy (this claim is justified, in Section 4).

Each state in M will correspond to a set of states from B, or equivalently a formula over $\Pi \cup T$. So, each state of M represents the goal that we are currently trying to achieve. Although we have a symbolic construction below, the states of M are as those that would be found by applying the usual subset construction to B. The set of accepting states, however, is not defined. This is to avoid the complexities of determinising the winning condition of a Büchi automaton [9]. Instead, we will run M and B together, using M in the strategy generation and B to evaluate the winning condition.

To construct M symbolically, we set the initial state to be the inital goal of B, $\phi_0 \wedge N_0$. The successor states will be defined by E_T and E_F. The choice of variables from T that are admitted to E_T is dependendent on the variables in Π. We look at the possible valuations for the variables in Π to find distinct cases in T. This can easily be done with BDDs by ordering the Π variables before the T variables and then identifying unique subtrees in the T layers. The Π root of each subtree in the T layer defines one case, C_i. Each case gives rise to a goal, ϕ_i, over T. For each case, we apply $E_T(\phi_i)$, replace any temporal operators with their propositional identities (using E_U, E_R, and E_X), and strip primes to obtain a formula representing the next state (this part is the same process as used in section 3.1, above). The N part of the successor state is simply derived from E_F. We repeat this process of finding successors for each of the resulting states, continuing until M stabilises.

Example

1. The initial state in the memory automaton for our running example is represented by $(r \rightarrow (s \vee p_{\mathrm{XF}s})) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$.
2. To illustrate the choosing of cases, its BDD is given at the top of Figure 5. There are two cases (left and right in the figure): $(r \wedge \neg s) \wedge p_{\mathrm{XF}s} \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$ and $(\neg r \vee (r \wedge s)) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$.
3. Take the first case, we apply $E_T(p_{\mathrm{XF}s} \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)})$, to obtain $(\mathrm{F}s)' \wedge (\mathrm{G}(r \rightarrow \mathrm{F}s))'$. Applying E_U, and E_R, and simplifying, we get $(s \vee p_{\mathrm{XF}s})' \wedge p'_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$. So, one next state is $(s \vee p_{\mathrm{XF}s}) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$. Since the model is highly non-deterministic, it does not restrict the transitions of G_ϕ.
4. The second case corresponds to returning to the initial state.
5. Taking cases from $(s \vee p_{\mathrm{XF}s}) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$, and extending them we find no more new states, but two new transitions. The complete memory automaton is given in Figure 6.

   The two states reflect the important parts of the specification. The upper state is for when no requests have been received, or one has just been granted. The lower is for when a request is pending. As we perform the synthesis, these roles will decide the actions of the strategy.

3.4 The Synthesis Algorithm

The synthesis algorithm (Figure 7) runs over all three automata: the game, G, the specification, B, and the memory M. States in this statespace are written as triples (q, t, ϕ), where $q \in Q, t \in Q_B, \phi \in Q_M$. It is very similar to the

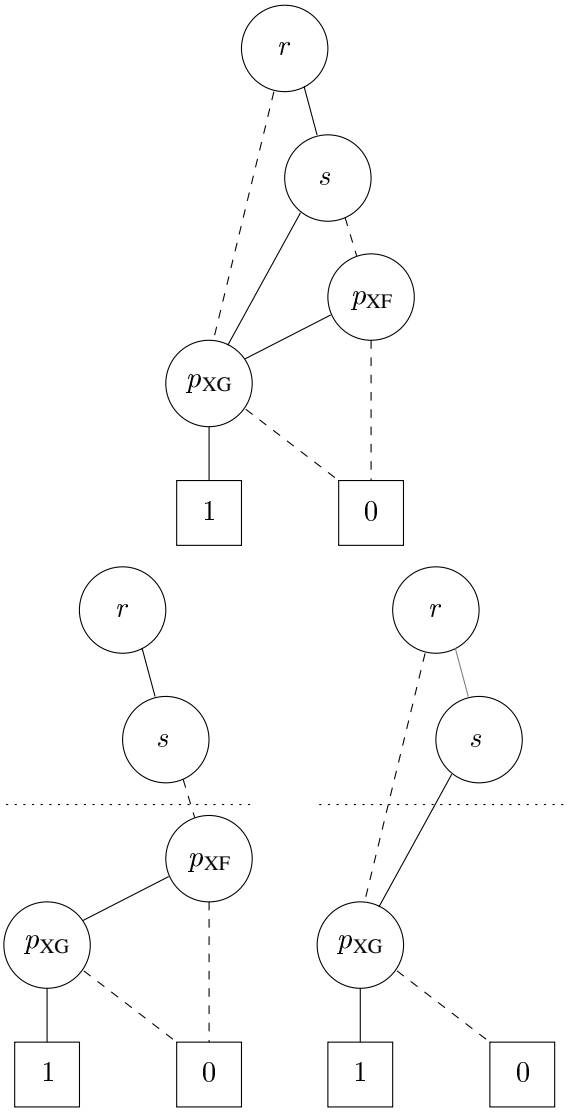

Figure 5. The BDD for $(r \rightarrow (s \vee p_{\mathrm{XF}s})) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$ (top) with the unique subtrees in the T layer shown below. p_{XF} is used to abbreviate $p_{\mathrm{XF}s}$, and p_{XG} for $p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$.

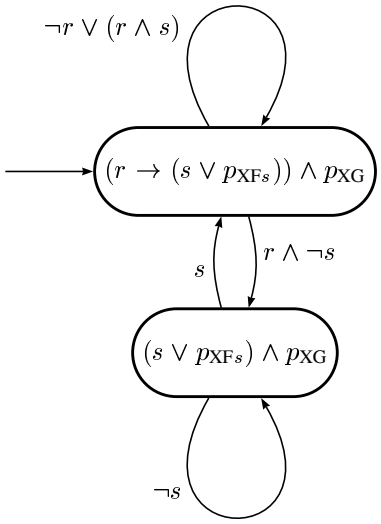

Figure 6. The memory automaton for $(r \rightarrow (s \vee p_{\mathrm{XF}s})) \wedge p_{\mathrm{XG}(r \rightarrow \mathrm{F}s)}$

algorithm used in the removal of dead states (Figure 4) because one part of its job is to do just that: find the set of states in the combined statespace from where A can force plays which are accepted by the specification component i.e. the states that satisfy $\langle\langle A \rangle\rangle \phi$. Of course, the synthesis algorithm also generates a strategy for A to acheive ϕ. We begin our explanation of the algorithm with a list of the variables used, and their purpose:

- z_k – The set of states from where A can force k visits to the set of states in the combined statespace where $t \in N$. z_0 begins by including everything, and states are removed as k increases. When z_k stabilises, it is the greatest fixed point and A can force an infinite number of visits to $\{(q, t, \phi) \mid t \in N\}$.

- τ_k – The set of states, (q, t, ϕ), where $t \in N$, and A can force $k-1$ visits to $\{(q, t, \phi) \mid t \in N\}$. NB these later visits may be to states outside τ_k, but still satisfying $t \in N$.

- $s_{k,l}$ – The set of states from where A can force play into τ_k in l steps or less. As l increases, each $s_{k,l}$ is larger than the last until it stabilises at a least fixed point. The states in a stable $s_{k,l}$ are all of those that can force play into τ_k in a finite number of steps.

- u – The set of states from which A can force play into $\tau \cup s_{k,l-1}$ in one step.

- $g_{k,l}(a, \phi)$ – The strategy by which there exists $t \in \phi$ such that A can force play from (a, t, ϕ) into some state in τ_k within l steps.

$z_0 := \{(q,t,\phi) \mid q \in Q, t \in Q_B, \phi \in Q_M, t \in [\phi]_\sim\}$;
repeat counted by $k = 1 \ldots$
  $g_{k,0} := \emptyset$;
  $s_{k,0} := \emptyset$;
  $\tau_k := \{(q,t,\phi) \in z_k \mid t \in N\}$;
  repeat counted by $l = 1 \ldots$
    $g_{k,l} := g_{k,l-1}$;
    $u := Pre_A(\tau \cup s_{k,l-1})$;
    forall $\langle (q,t,\phi), Q' \rangle \in u$
      if($g_{k,l}(q,\phi)$undefined) $g_{k,l}(q,\phi) := Q'$;
    endfor
    $s_{k,l} := s_{k,l-1} \cup \pi_1(u)$;
  until $s_{k,l}$ stabilises
  $z_k := z_{k-1} \cap s_l$
until z_k stabilises

$$Pre_A(X) = \{\langle (q,t,\phi), Q' \rangle \mid Q' \in \delta_G(q,A) \cap succ(q), \forall q' \in Q' \exists \phi' \in \delta_M(\phi,q'), t' \in \delta_B(t,q') \text{ s.t. } (q',t',\phi') \in X\}$$

Figure 7. The synthesis algorithm over three automata, G, B, and M (top). The redefinition of pre (bottom)

Consider the first part of the task. We want to find the set of states from which the transitions of G provide words accepted by B i.e. in the combined automaton, B goes through N infinitely often. The code for this is almost identical to that in Figure 4. The difference occurs in the definition of Pre. Normally, $pre(X)$ returns the set of states from which *there exists* a transition into X. For our purposes, we need a mix of quantifiers: We want A to be able to ensure that for any next state in the game, it is possible for the specification and memory to transition such that the combined successor must be in X. We might call such a predecessor function pre_A. As a matter of convenience in building the strategy, we return not just the set of states that can reach X, but with each state we return a choice for A that will reach X. This redefinition of pre_A, we call Pre_A and it is given in Figure 7. When we write $\pi_1(u)$ in the algorithm, this returns the set of elements of u projected to their first element i.e. what would have been obtained by pre_A.

The second part of the task is to generate a strategy for A. In general, this strategy will require memory. In our case, this is provided by the memory automaton, M. We leave the question over whether there are enough states in M until Section 4, for now we just assume that there are. For each iteration of the inner loop, every element of u is checked for whether we need to update the strategy. An update is made, if the particular (a,ϕ) combination has not previously been defined on this iteration of the k-loop. Clearly, when (a,ϕ) is updated, it is doing the right thing - it picks a choice which (by being in u) is known to be either in τ_k, or closer to it. If a combined state has two different choices which may reach $\tau_k \cup s_{k,l-1}$, this corresponds to two elements in u. Which one gets recorded in the strategy is arbitrary, and either would work. If a combined state is discovered on two different iterations of the l-loop, then only the first instance is recorded in the strategy. This is correct, because the first find takes a more direct route to τ_k. If two combined states (a,t_1,ϕ), and (a,t_2,ϕ) where $t_1 \neq t_2$ are discovered on different iterations of the l-loop, then only the first one is recorded into the strategy. Say that t_1 is found first, we can show that any path which reaches t_2 could also reach t_1, thus the use of t_1's strategy is sound.

Example In order to run the synthesis algorithm on our example, we have write the statespace in triples over the system, the specification, and the memory. The system states are $\{\emptyset, r, s, rs\}$, where r and s indicate the truth value of $receive$ and $send$, respectively. $\emptyset$ is when both are false. The specification states are $\{\emptyset, s, p_{\mathrm{XF}s}, rs, rp_{\mathrm{XF}s}, p_{\mathrm{XF}s}s\}$ where r, s are as before and f stands for $p_{\mathrm{XF}s}$. We omit the $p_{\mathrm{XG}(r \to \mathrm{F}s)}$ part in our notation because it is required in every state. The memory states are $\{(r \to (s \vee p_{\mathrm{XF}s})), (s \vee p_{\mathrm{XF}s})\}$, again ommiting the $p_{\mathrm{XG}(r \to \mathrm{F}s)}$. We write the triples as in the algorithm e.g. $(r, rf, (r \to (s \vee p_{\mathrm{XF}s}))$

1. We begin with τ_1, which are the reachable states where $t \in \phi$ and $t \in N$. We do not need to worry about $\sim$, because the specification has only one fairness constraint.
   $\tau_1 = \{(\emptyset,\emptyset,(r \to (s \vee p_{\mathrm{XF}s}))), (s,s,(r \to (s \vee p_{\mathrm{XF}s}))), (rs,rs,(r \to (s \vee p_{\mathrm{XF}s}))), (s,sf,(r \to (s \vee p_{\mathrm{XF}s})))\}$

2. We then find $Pre_A(\tau_1 \cup \emptyset)$:
   $u = \{\langle(\emptyset,\emptyset,(r \to (s \vee p_{\mathrm{XF}s}))), \{s, rs\}\rangle,$
   $\langle(s,s,(r \to (s \vee p_{\mathrm{XF}s}))), \{s, rs\}\rangle,$
   $\langle(rs,rs,(r \to (s \vee p_{\mathrm{XF}s}))), \{s, rs\}\rangle,$
   $\langle(s,sf,(r \to (s \vee p_{\mathrm{XF}s}))), \{s, rs\}\rangle,$
   $\langle(r,rf,(s \vee p_{\mathrm{XF}s})), \{s, rs\}\rangle,$
   $\langle(\emptyset,f,(s \vee p_{\mathrm{XF}s})), \{s, rs\}\rangle,$
   $\langle(\emptyset,f,(r \to (s \vee p_{\mathrm{XF}s}))), \{s, rs\}\rangle\}$

3. Having found a set u, we store good moves into the strategy $g_{1,1}$. We just take the first definition for each (q,ϕ) pairing, because any given one from u will satisfy the specification.

$g_{1,1}(\emptyset, (r \rightarrow (s \vee p_{\mathrm{XF}s}))) := \{s, rs\}$;
$g_{1,1}(s, (r \rightarrow (s \vee p_{\mathrm{XF}s}))) := \{s, rs\}$;
$g_{1,1}(rs, (r \rightarrow (s \vee p_{\mathrm{XF}s}))) := \{s, rs\}$;
$g_{1,1}(r, (s \vee p_{\mathrm{XF}s})) := \{s, rs\}$;
$g_{1,1}(\emptyset, (s \vee p_{\mathrm{XF}s})) := \{s, rs\}$;

4. Now, $s_{1,1}$ becomes $\pi_1(u)$, $g_{1,2}$ becomes $g_{1,1}$ and we find $Pre_A(\tau_1 \cup s_{1,1})$. The first part of the set is the same as before, but the following extra elements are included:
$\{\langle(\emptyset, \emptyset, (r \rightarrow (s \vee p_{\mathrm{XF}s}))), \{\emptyset, r\}\rangle,$
$\langle(s, s, (r \rightarrow (s \vee p_{\mathrm{XF}s}))), \{\emptyset, r\}\rangle,$
$\langle(rs, rs, (r \rightarrow (s \vee p_{\mathrm{XF}s}))), \{\emptyset, r\}\rangle,$
$\langle(s, sf, (r \rightarrow (s \vee p_{\mathrm{XF}s}))), \{\emptyset, r\}\rangle,$
$\langle(r, rf, (s \vee p_{\mathrm{XF}s})), \{\emptyset, r\}\rangle,$
$\langle(\emptyset, f, (s \vee p_{\mathrm{XF}s})), \{\emptyset, r\}\rangle,$
$\langle(\emptyset, f, (r \rightarrow (s \vee p_{\mathrm{XF}s}))), \{\emptyset, r\}\rangle\}$

5. In updating the strategy, we find that $g_{1,2}$ is defined for every value found in this u. This is not surprising, because no state is more than one step away from τ_1. The strategy is not changed, because the new choices we have discovered would just postpone reaching τ_1.

6. We find that $s_{1,2}$ is stable, and so is z_1. The strategy, is simply for the server to aways be sending, regardless of whether it has received a request. A trivial strategy, but a successful one.

4 Correctness

At the time of writing this paper, the proof of correctness is not yet complete. When the proof is finished, it will be available from `http://www.cs.bham.ac.uk/~ath/atl_synthesis`. In the meantime, we provide less formal explanations for why the construction works.

The first question is whether the algorithm correctly identifies the set of winning states. We would certainly expect that to be true, since our algorithm is based on the one given by Emerson and Lei in [4]. The only change is the redefinition of Pre. Our definition differs from the norm by quantifying over the choices enforcable by agents and by having the memory running alongside the game and specification. The agent part is well understood from [1]. We can prove that the memory automaton is not constraining the system by using reverse determinism and its construction. Take a state (a, t, ϕ) where $t \in [\phi]_\sim$. For each state in B, distinct paths back to the set of initial states are labelled by distinct words. When a word is put into the memory automaton, its subset construction ensures that the path will lead to to the same t. So, when the algorithm looks backwards from τ_k, the memory automaton does not prevent it from finding states that would be discovered without it.

We use similar reasoning to argue that there are enough states in M for strategies to remember all that they need. Since the definition of a strategy is given when we know that there is a chance to make progress, if there is enough memory then the strategy will be winning.

5 Conclusions

We have presented an algorithm for synthesizing strategies in agent-based systems that can be implemented with symbolic methods. Although we do not yet have an implementation to demonstrate the feasability of our approach, the results of symbolic model checkers give us reason to be optimistic for our synthesis algorithm. The novel use of partial determinisation has avoided the non-trivial problem of determinising a non-deterministic Büchi automaton, and the use of a game-oriented temporal logic has provided the ideal setting for the synthesis of open systems.

The most obvious line of future work is implementation. MOCHA [11] is a symbolic model checker for verifying ATL specifications against programs written in reactive modules. It is available with source, and contains a scripting language, so a prototype should be straightforward to build.

An interesting theoretical extension of the synthesis algorithm would be to consider incomplete information i.e. the notion that agents do not have the ability to read the global state, but just some subset of it. Kupferman and Vardi [7] considered this problem for two agents in a CTL/CTL* setting, but there are many situations where three agents with different views would be needed. For example, in modelling security protocols where two agents, S and R, try to communicate without leaking information to an intruder I, each agent has a different view. S and R can only see local information and are forced to communicate over some public channel. I can only see what is put on the channel. So, even though S and R are cooperating, we cannot amalgamate their views of the system.

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.

[2] O. Carton and M. Michel. Unambiguous Büchi automata. *Theoretical Computer Science*, 297:37–81, 2003.

[3] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-*

Aided Verification CAV, volume 818, pages 415–427, Standford, California, USA, 1994. Springer-Verlag.

[4] E. A. Emerson and C. Lei. Efficient model checking in fragments of the mu-calculus. In *IEEE Symposium on Logic in Computer Science*, pages 267–278, June 1986.

[5] E. A. Emerson and A. P. Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[7] O. Kupferman and M. Vardi. Synthesis with incomplete informatio (sic). In *2nd International Conference on Temporal Logic*, pages 91–96. Kluwer Academic Publishers, July 1997.

[8] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993.

[9] S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, March 1989.

[10] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Scien ce*, pages 322–331, 1986.

[11] Mocha. `http://www-cad.eecs.berkeley.edu/~tah/mocha/`.