An Automata Theoretic Approach to Multiagent Planning

Michael Wooldridge
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, U.K.
mjw@csc.liv.ac.uk

Abstract

We present a novel approach to multiagent planning, and describe some preliminary results obtained with an implementation of this approach. The approach is novel in three respects. First, it is based on *automata theoretic model checking*, a highly efficient technique that was originally developed for the automatic verification of finite state concurrent systems. Second, it allows for goals for agents or groups of agents to be expressed using the language of Linear Temporal Logic (LTL), a succinct and expressive language for expressing not just properties of states, but of (potentially infinite) sequences of states. Third, the approach allows for plans to be developed in the context of environments that may contain multiple processes, which may independently manipulate the environment. These processes may be defined either implicitly (as non-deterministic processes that may execute actions), or explicitly, by giving the program that the execute. Following a short overview of the theory underpinning the approach, we describe its implementation in the `atmp` system, which is based upon the SPIN LTL model checker. We then give some results obtained in a series of experiments using `atmp`.

1 Introduction

We present the *Automata-Theoretic Multiagent Planning* system (`atmp`). The `atmp` system is capable of taking a representation of a planning domain that contains multiple communicating agents, a specification of the capabilities of one or more agents, and a goal expressed in the language of Linear Temporal Logic (LTL). The system will then be guaranteed to generate a sound plan that will achieve this goal, if such a plan exists and computational resources permit.

The `atmp` system is novel in both its capabilities and its implementation. First, the system is based on *automata-theoretic model checking*, a technique originally developed for the automatic verification of finite state systems [3]. In fact, `atmp` is implemented upon AT&T's SPIN model checker for LTL, which provides a powerful, highly optimized model checking engine that employs a number of techniques for efficiently managing large state spaces [10]. Secondly, the approach allows for goals for agents or groups of agents to be expressed using the language of Linear Temporal

Logic, a highly expressive language for expressing complex properties of (potentially infinite) sequences of states. Finally, `atmp` allows for plans to be developed in the context of environments that may contain multiple processes, which may dynamically operate on the environment. These processes may be defined implicitly (by defining the actions that they can perform), or explicitly (by giving the program that the agent executes).

The inspiration for the `atmp` system was the *planning as* CTL*-based model checking* paradigm of Giunchiglia, Traverso and colleagues [8]. Our approach differs from theirs in a number of respects — we discuss the relationship to this work at the end of the paper.

The remainder of this paper is structured as follows. We begin in the following section by introducing the underlying theory of automata theoretic model checking, and how this technique can be used to implement a multiagent planning system. We then describe the `atmp` system, sketch out its organisation and capabilities, and present some experimental results obtained with the system. We conclude with a brief discussion of related work and issues for future research.

2 The Theory

In this section, we give a brief overview of the theory that underpins our approach, and in particular, we illustrate how the SPIN model checker can be used as the basis of a multiagent planning system. Readers familiar with model checking and SPIN may wish to skip directly to the section on planning with SPIN.

Model Checking: *Model checking* was developed as a technique for verifying that finite state systems satisfy temporal logic specifications [3]. The name arises from the fact that verification can be viewed as a process of checking that the system is a model that validates the specification. The idea is that given a system $\mathcal{S}$, which we wish to verify satisfies some specification φ expressed in a logical language L, the possible computations of $\mathcal{S}$ can be understood as a directed graph, in which the nodes of the graph correspond to possible states of the system, and arcs in the graph correspond to state transitions. Such directed graphs are essentially *Kripke structures* — the models used to give a semantics to temporal logics. Crudely, the model checking verification process can then be understood as follows. Given a system $\mathcal{S}$, which we wish to verify satisfies some property φ, expressed in temporal logic, generate the Kripke structure $M_{\mathcal{S}}$ corresponding to $\mathcal{S}$, and then check whether $M_{\mathcal{S}} \models_L \varphi$, i.e., whether φ is valid in the Kripke structure $M_{\mathcal{S}}$. If the answer is "yes", then the system satisfies the specification; otherwise it does not.

Model checking has received considerable attention of late. There are two main reasons for this. The first is that model checking is straightforward to implement (unlike alternative verification procedures, such as deductive verification [12, 13]). The second is that, for some temporal logics, model checking algorithms are computationally tractable. In particular, for the branching temporal logic CTL [5], there is an $O(|\varphi| \cdot |M_{\mathcal{S}}|)$ model checking algorithm, where $|\varphi|$ is the size of the formula to be checked, and $|M_{\mathcal{S}}|$ is the size of the model against which it is to be checked [3, p.38]. For *linear* temporal logic, the model checking problem is ostensibly more complex (it is PSPACE complete [3, p.41]): the best LTL model checking algorithm to date has time

complexity $O(2^{|\varphi|} \cdot |M_{\mathcal{S}}|)$, i.e., exponential in the size of the specification, but linear in the size of the model [3, p.48]. However, the main problem for both branching and linear temporal logic model checking is not the time complexity of the model checking process, but the fact that the size of the model $M_{\mathcal{S}}$ that encodes the computations of $\mathcal{S}$ grows exponentially (or worse) with the size of $\mathcal{S}$. Much effort has been devoted to overcoming this *state space explosion* problem [3, p.1].

LTL Model Checking: The predominant approach to LTL model checking is based on the close relationship between LTL formulae and Büchi automata — a type of finite automata on infinite words [15]. Suppose we want to verify that finite state system $\mathcal{S}$ satisfies LTL formula φ. The first step is to model $\mathcal{S}$ as a Büchi automaton $\mathcal{A}_{\mathcal{S}}$; the language $\mathcal{L}(\mathcal{A}_{\mathcal{S}})$ recognised by $\mathcal{A}_{\mathcal{S}}$ will represent the set of possible computations of $\mathcal{S}$. Given an LTL formula φ, it is possible to construct a Büchi automaton $\mathcal{A}_{\varphi}$ such that $\mathcal{L}(\mathcal{A}_{\varphi})$ is exactly the set of models of φ. It should then be clear that $\mathcal{S}$ satisfies φ iff

$$\mathcal{L}(\mathcal{A}_{\mathcal{S}}) \subseteq \mathcal{L}(\mathcal{A}_{\varphi}) \tag{1}$$

i.e., if the set of possible computations of $\mathcal{S}$ are a subset of the computations that satisfy φ. Since Büchi automata are closed under complementation, we can rewrite (1) as

$$\mathcal{L}(\mathcal{A}_{\mathcal{S}}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset \tag{2}$$

i.e., if the intersection of the set of computations of $\mathcal{S}$ and the set of computations *disallowed* by φ is empty. As Büchi automata are closed under intersection, we can further rewrite (2) as

$$\mathcal{L}(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\neg\varphi}) = \emptyset \tag{3}$$

So, checking that $\mathcal{S}$ satisfies φ involves first generating $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_{\neg\varphi}$, then checking whether $\mathcal{L}(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\neg\varphi}) = \emptyset$. The state explosion problem can be alleviated in automata theoretic model checking by the use of two additional techniques. First, the automaton $\mathcal{A}_{\mathcal{S}}$, representing the system to be modelled, need only be generated *on-the-fly*: states are only generated as they are needed, and so only in the worst case must the entire automaton $\mathcal{A}_{\mathcal{S}}$ be generated. Second, the set of possible transitions in $\mathcal{A}_{\mathcal{S}}$ can be reduced by only considering *representatives* from equivalence classes of computations [3, pp.141–170]: if a number of operations commute, (in the sense that they generate the same result whatever order they are executed), then it is only necessary to consider a single representative ordering of the operations, rather than all possible orderings. (For historical reasons, this technique is known as *partial order reduction*.)

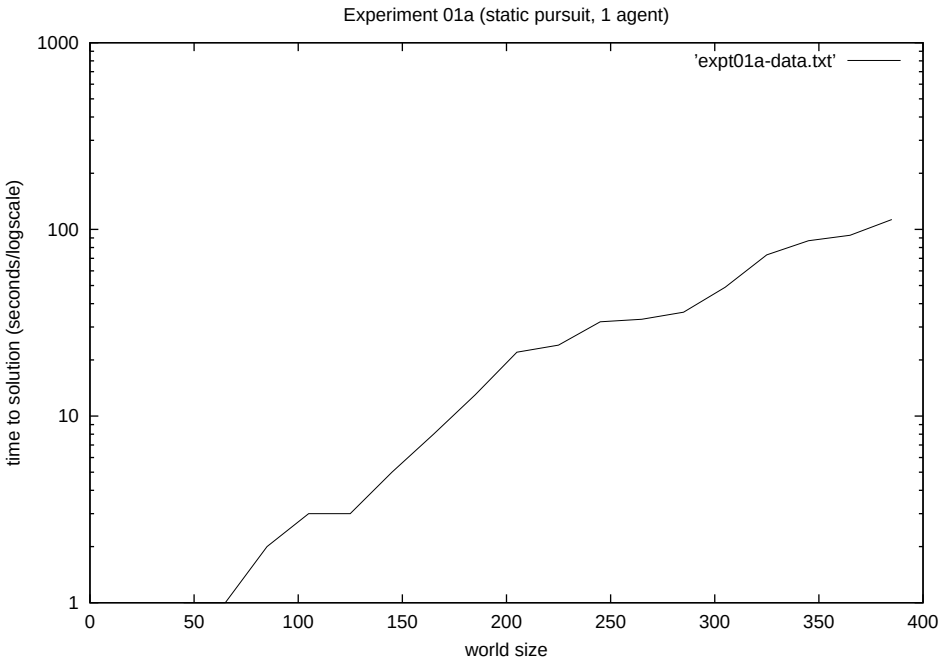
The SPIN LTL Model Checker: The SPIN system, upon which our multiagent planning approach has been implemented, takes as input a description of a system containing n concurrent processes, $\mathcal{S} = P_1 \parallel \cdots \parallel P_n$. SPIN then generates the Büchi automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ corresponding to these processes, and then generates the system automaton $\mathcal{A}_{\mathcal{S}}$ from $\mathcal{A}_1, \ldots, \mathcal{A}_n$; to be precise, $\mathcal{A}_{\mathcal{S}}$ is actually generated on the fly, using partial order reduction as described above. The system processes P_i are encoded in the input using the PROMELA language, a guarded command language somewhat resembling C, which includes communication based on Hoare's CSP formalism [9]. As well as taking as input these processes, SPIN takes an LTL formula, and

```
// variable declarations
vars{

// the size of the world
#define     SIZE ...

// initial location of agent
int         a1x = ...;
int         a1y = ...;

// initial location of target
int         targetX = ...;
int         targetY = ...;

// predicate used in the goal formula
#define     c1 ((a1x == targetX) && (a1y == targetY))

}

// the actions available to the agent
action      a1MoveNorth
    pre {   a1y < SIZE }
    post{   a1y = a1y + 1 }

action      a1MoveSouth
    pre {   a1y > 0 }
    post{   a1y = a1y - 1 }

action      a1MoveEast
    pre {   a1x < SIZE }
    post{   a1x = a1x + 1 }

action      a1MoveWest
    pre {   a1x > 1 }
    post{   a1x = a1x - 1 }

// definition of the agent via its capabilities
agent       a1 a1MoveNorth, a1MoveEast,
                a1MoveSouth, a1MoveWest

// LTL formula defining the goal
goal{       <>c1    }
```

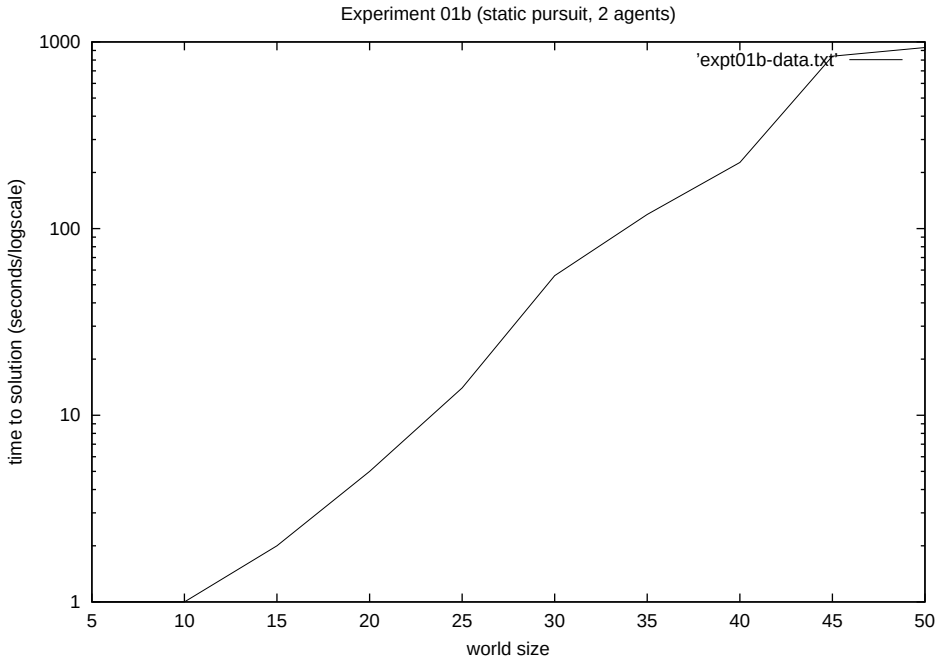
Figure 1: The `atmp` implementation of the static pursuit problem (Experiment 01a).

from it generates a Büchi automaton $\mathcal{A}_{\neg\varphi}$, as decribed above; it then checks whether $\mathcal{L}(\mathcal{A}_{\mathcal{S}} \cap \mathcal{A}_{\neg\varphi}) = \emptyset$, and if not, produces a witness to this effect, in the form of a run.

SPIN is particularly appropriate for the verification of multi-process communicat-



(a)



(b)

Figure 2: Experiment 1: The pursuit problem with static target, with one agent (a), and two agents (b).

ing systems, as PROMELA provides high-level communication constructs to support the description of such systems. As a consequence, SPIN has been widely used in the

verification of protocols [10].

Planning with SPIN: We are now in a position to describe the theory that underpins our approach. First, note that, by default, verifying that $\mathcal{S}$ satisfies LTL specification φ involves showing that (with a slight abuse of notation):

$$\forall c \in comp(\mathcal{S}) \text{ we have } c \models \varphi \tag{4}$$

where $comp(\mathcal{S})$ denotes the set of computations of $\mathcal{S}$. Now, for reasons discussed in the preceding section, the SPIN model checker is in fact optimized to check whether, given an LTL formula φ and system $\mathcal{S}$:

$$\exists c \in comp(\mathcal{S}) \text{ such that } c \models \neg\varphi \tag{5}$$

If the answer to (5) is no, then (4) is true. If the answer is no, then SPIN will actually produce a witness to this, in the form of a computation of $\mathcal{S}$ that fails to satisfy φ.

Now, consider a multiagent planning domain $\mathcal{S}$ and a goal φ, expressed as an LTL formula. Showing that there is a plan that satisfies φ in $\mathcal{S}$ amounts to showing that there is a computation c of $\mathcal{S}$ such that $c \models \varphi$. But this implies that we can make use of (5) to use SPIN to solve such problems, by simply plugging in the negation of the goal φ. This is the basic idea that informs `atmp`: we will now describe the actual implementation of `atmp`.

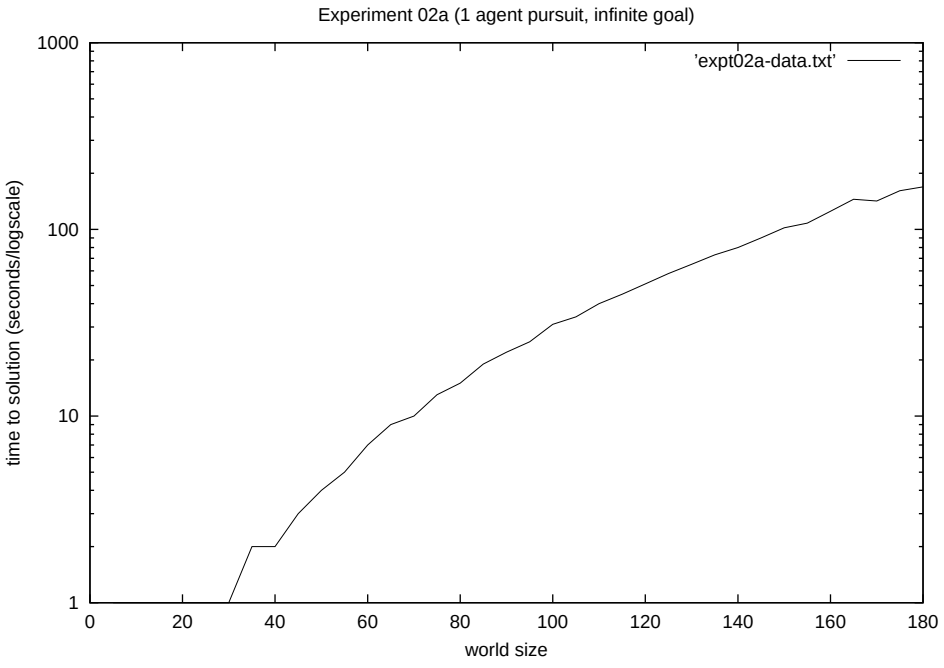
3 The ATMP System

The `atmp` system allows plans to be developed for any number of agents (computational resources permitting). In ATMP, we specify the agents for which we wish to develop plans by defining their capabilities, in terms of the actions that they can perform. Actions are defined using a STRIPS-style pre-/post-condition notation [6]. The environment in which a plan is to be developed is specified by defining a number of agents that may modify the environment. These agents may be defined either *implicitly* (by defining the actions that these agents may perform) or *explicitly* (by giving PROMELA code that constitutes their program). Note that explicitly defined environmental agents may make full use of PROMELA's facilities, and in particular they may communicate with one-another — this makes it possible to define extremely sophisticated environments.

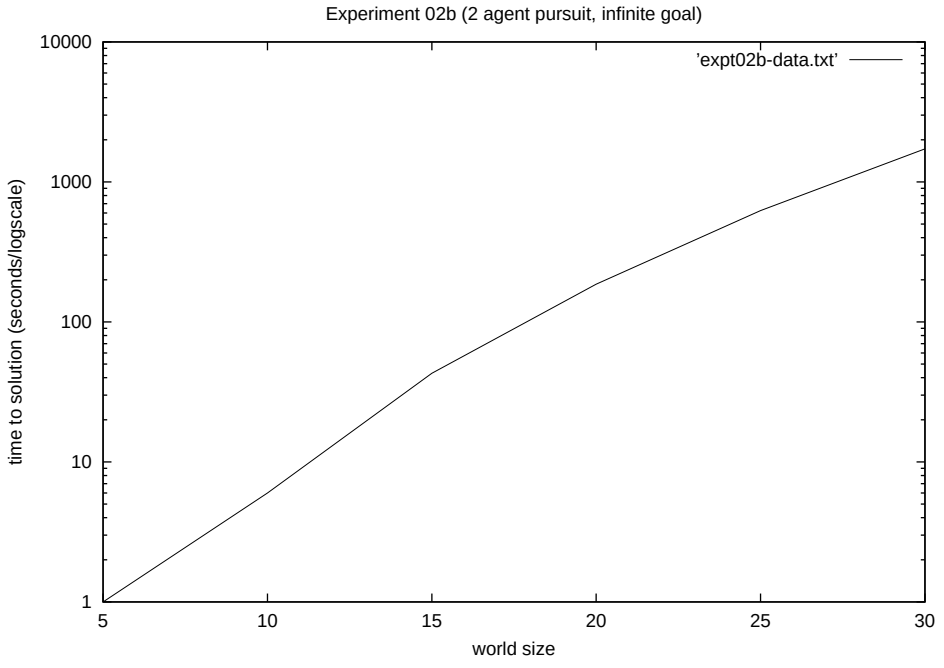
Actions in `atmp` are defined using a STRIPS-style pre-/post-condition formalism. Here is an example of an `atmp` action definition:

```
action north
  pre  { y < SIZE   }
  post { y = y + 1  }
```

This action represents an agent moving North in a two-dimensional grid world: the pre-condition is that the *y* location of the agent is not currently at the Northern-most grid edge, while the post-condition is that the *y* location is incremented by one. Note that in this example, `y` is a global variable in the environment. Such variables are defined in a `vars` section of the `atmp` system.



(a)



(b)

Figure 3: Experiment 2: The pursuit problem with an infinite goal, with one agent (a), and two agents (b).

An implicit agent definition involves naming an agent and the actions that it can perform, where these actions are defined using the above notation. For example, this

definition says that agent named `a1` may perform actions `north`, `east`, and so on:

```
agent a1 north, east, south, west
```

Goals in `atmp` are expressed as formulae of LTL [12, 13]. The language of LTL extends that of classical logic with a number of temporal modal connectives. Those of interest to us are: "$\Box$" (henceforth), "$\Diamond$" (at some time in the future), and "$\mathcal{U}$" (until). Thus a formula $\Box\varphi$ expresses the fact that φ is true now and forever more; $\Diamond\varphi$ means that φ is eventually true; and $\varphi\,\mathcal{U}\,\psi$ means that eventually ψ is true, and until then, φ is true. Temporal connectives can be combined to succinctly express complex properties of state sequences. For example, $\Box\Diamond\varphi$ means that φ happens infinitely often in the future, while $\Diamond\Box\varphi$ means that eventually, φ becomes true and remains true thereafter. Here is an example `atmp` goal definition, which expresses the fact that the goal is that eventually agent `a1` gets to a target location (`X`, `Y`) and stays there:

```
goal { <>[] ((x == X) && (y == Y)) }
```

The propositions that appear in goal specifications must be defined over the global environment variables, as declared in the `vars` section of the system definition, by making use of PROMELA constructs [9, 10].

To summarize, an `atmp` system definition contains: a variable declaration section, in which the global variables defining the properties of the environment are defined; a number of pre-/post-condition action definitions; a number of implicit agent definitions, in which agents are defined by listing the actions that they may perform; a number of explicit agent definitions, in which agents are defined by explicitly giving the PROMELA code that they execute; and finally, a goal specification, expressed as an LTL formula.

Given a system specification, as described above, `atmp` works as follows. Variable declarations and explicit agent definitions are translated directly into PROMELA variables and processes respectively. Goals are negated and translated into the LTL form required by the SPIN system. Implicit agent definitions are handled as follows. Suppose we have an agent *Ag*, which can perform actions $\alpha_1, \ldots, \alpha_n$, where each action α_i has pre-condition *pre*(α_i) and post-condition *post*(α_i). We translate *Ag* into a non-deterministic process (in fact, a Büchi automaton) with the following properties. For each action α_i, we create a guarded state transition in the process. The guard for the transition is *pre*(α_i). A transition is enabled if its guard is true: at any given time, there may be a number of transitions enabled. In any given computation of such a system, only one enabled transition will be selected for execution. If a particular enabled transition associated with action α_i is selected, then the post condition *post*(α_i) associated with the transition is made true: this can in general be done by *executing* the post condition. (One technical aside: when translating actions to transitions, we force PROMELA to consider them as being *atomic*; that is, between an action being selected and the post-condition being made true, no other process can intervene.)

When the translation process is complete, `atmp` runs SPIN over the system, together with the (negated) goal. If there exists some computation of the system that satisfies the goal, then SPIN reports this sequence of actions as a witness. This sequence of actions is in fact a (multiagent) plan. If SPIN reports a sequence of actions

$\alpha_1, \ldots, \alpha_n$, then from construction of the translated system, we can be assured that this plan, if executed from the initial state, will achieve the goal. In fact, by construction, the planning process is both sound and complete: it guarantees to find a plan if such a plan exists (modulo the availability of sufficient computational resources), and if a plan is announced, then this plan is guaranteed to be correct.

One issue that arises with this approach is that of *scheduling* different agents. By default, SPIN will simply report *any* sequence of actions that accomplishes the goal. Such a sequence may not constitute a "reasonable" plan, as it need not contain the actions of one or more agents in the system. We have addressed this issue by allowing the user to enforce "round robin" scheduling in the system, where agents are forced to take it in turns to execute actions. Another issue is that SPIN will report the first plan that it finds that satisfies the goal, which may not be the shortest plan to accomplish the goal. SPIN can be forced to find the shortest solution by first finding any solution, and then iteratively reducing the maximum search depth; however, for the purposes of this paper, we were not concerned with this problem. We were rather concerned with simply finding *sound* plans.

4 Experimental Results

We now present some experimental results obtained with the `atmp` system on a number of problems.

4.1 Experiment 1: The Pursuit Problem

The first set of experiments we carried out were on the *static pursuit problem* (called the hunter-prey problem in [2]). In this problem, we have a number of agents inhabiting a grid world of dimensions $n \times n$. Initially, the agents are located at randomly allocated grid locations. At some other randomly allocated grid location is a target. The agents are each able to move around the grid in directions *N*, *S*, *E*, and *W*; in this first version of the pursuit problem, the target is static. The agents are not allowed to move outside the grid world. Naturally enough, the goal is for the agents to converge on the target. Within this general setup, we carried out two experiments, as follows.

Experiment 1a

In experiment 1a, we had just one agent in the system. The `atmp` implementation of this system is presented in Figure 1. For this experiment, we systematically varied the size of the world. For each size of grid world, we ran 50 trials. The trials were generated on a desktop PC running RedHat LINUX 7.3, with a 1.3 GHz AMD Athlon processor and 512MB of RAM; we used SPIN version 3.4.16. For each trial, we kept track of the amount of time taken to find a plan, to the nearest second (the experiments were automated in a C program, and timing was done via GNU implementation of the standard UNIX C `time(...)` system call). For completeness, we note the setting of the various search parameters that may be set within SPIN: the `-m` parameter, which is used to set the maximum search depth in SPIN's double depth-first search algorithm was set to $10 \times S$, where *S* is the size of the world; this is in effect telling SPIN to stop searching for unnecessarily long plans. The `-a` flag was passed to the PAN search

program generated by SPIN, which forces it to look for acceptance cycles (we elaborate on this issue below). Finally, the REACH symbolic variable was set when compiling PAN, ensuring that a full search was undertaken.

The results for experiment 1a are shown in Figure 2(a), which plots the size of the world against the average time taken for atmp to find a plan (the average is taken over the 50 trials, where in each trial the target and agent are randomly located). Note that the *y* axis is plotted on a log scale, as is the case for all graphs in this paper.

The main observation that can be made is that the time taken to find a solution appears to grow exponentially with the size of the search space. Notice that the total time to find a solution includes the time taken to create a Büchi automaton from the goal LTL formula; in this experiment, (and in fact for all experiments we ran), the time taken to generate this automaton is included in the total time to solution on the *y* axis, and was strongly dominated by the search time, except for very small values of world size.

Experiment 1b

The basic structure of Experiment 1b is as Experiment 1a, but this time we had two agents. The goal was to have both agents converge on the target simultaneously. The LTL formula defining the goal was thus <>c1, where this time the predicate c1 was defined as

```
#define c1 \
   (((a1x == targetX) && (a1y == targetY))  && \
    ((a2x == targetX) && (a2y == targetY)))
```

where, as might be expected, a2x and a2y define the *x* and *y* coordinates of agent 2.

The basic results are as Experiment 1a: the time taken to find a solution is exponential in the size of the world, as might be expected. However, the presence of multiple agents means that the exponent seems to be larger: whereas for a world size of 25 in the single agent case a solution was found in about one second of real time, it took about 10 seconds on average for the two agent case of Experiment 1b.

4.2 Experiment 2: An Infinite Goal

The basic setup for Experiment 2 is as Experiment 1: we have agents inhabiting a grid world, around which they can move in *N*, *S*, *E*, and *W* directions. This time, however, there are *two* targets placed randomly in the world, and we want our agent to *visit both infinitely often*. In other words, our goal is not simply of the form "achieve this state of affairs". We want a plan that contains an infinite loop; we want to generate a *cyclic* plan. To understand how such a plan may be generated, it is necessary to understand a little more of the automata theoretic foundations of temporal logic. We noted earlier that for any LTL formula φ, there is a Büchi automaton — an ω-regular automaton — that accepts just the (infinite) computations that are models of φ. In fact, if a formula φ is satisfiable, then there is an ω-regular expression

$$e = \alpha \cdot (\beta^{\omega})$$

such that all the words that may be generated from *e* are models of φ (where α and β are finite words and ω is the infinite repetition operator). In other words, if LTL formula φ is satisfiable — has any models — then these models must take the form of a finite sequence of events, *followed by a cycle, or loop*. (As an aside, it is known that the length of α is $O(2^{|\varphi|})$, where $|\varphi|$ is the size of the formula [14, p.743].) SPIN caters for this state of affairs by looking for what are known as *acceptance cycles*, which may be understood simply as infinite loops.

Experiment 2a

For Experiment 2a, we had a single agent, but two targets, randomly located in the world. We had two predicates `c1` and `c2`, defining when an agent reached these targets, as follows.

```
#define c1 ((a1x == targetX1) && (a1y == targetY1))
#define c2 ((a1x == targetX2) && (a1y == targetY2))
```

The goal may be understood as follows: we want a cyclic plan such that, if the agent follows it, then there will always be some point in the future at which it will be at target 1, and some later point at which it will be at target 2. In temporal logic, this goal is:

$$\Box\Diamond(c_1 \wedge \Diamond c_2)$$

which in `atmp` notation is:

```
goal{ []<>(c1 && <>c2) }
```

The results for Experiment 2a are shown in Figure 3(a). Again, we see exponential, as expected; but the results do not differ greatly from Experiment 1a.

Experiment 2b

For Experiment 2b, we had the same basic setup as 2a, but with two agents that were required to visit both targets infinitely often, alternating between them; thus while agent *i* was at target 1, agent *j* (where $j \neq i$) was required to be at target 2; and so on

The predicates used in defining the goal were as follows.

```
#define c1 ((a1x == targetX1) && (a1y == targetY1))
#define c2 ((a1x == targetX2) && (a1y == targetY2))
#define c3 ((a2x == targetX1) && (a2y == targetY1))
#define c4 ((a2x == targetX2) && (a2y == targetY2))
```

In conventional temporal logic notation, the goal was as follows:

$$\Box\Diamond((c_1 \wedge c_4) \wedge \Diamond(c_2 \wedge c_3))$$

which in `atmp` format becomes:

```
goal{ []<>((c1 && c4) && <>(c2 && c3)) }
```

The results for this experiment are given in Figure 3(b). Here, we see a much more dramatic increase in the time to solution than in Experiment 1b. For a world size of 30, the time to solution is on average nearly 2000 seconds, as compared to about 50 seconds in Experiment 1b.

4.3 Experiment 3: A Moving Target

For Experiment 3, we used the same basic setup again, but this time we had a *moving target*; in other words, the environment the agents had to inhabit was *dynamic*. The target was thus itself encoded as an agent, given as an explicit agent definition in PROMELA: see Figure 4 for the definition. (The lexemes `%{` and `%}` are used to delimit the agent definition; `NEXT` is an `atmp` macro definition that is used in explicit agent definitions to facilitate round robin scheduling.) Again, the goal was simply to converge on the target.

Our results for this experiment are given in Figure 5 (we only ran this experiment with one agent). In the case of a single agent pursuing the target, the results are comparable to Experiment 2a (the pursuit problem with an infinite goal).

5 Related Work

Although there is much work in the literature on multiagent planning, we are not aware of any that directly uses LTL model checking; see [4] for an overview of work on multiagent planning. Our approach was inspired by the work of Giunchiglia, Traverso, and colleagues, who have used symbolic CTL model checking for single agent planning [8]. The basic idea in this approach was that a classical planning domain D could be encoded as a Kripke structure M_D, and the goal as a state formula φ of CTL; the behaviour of actions in the domain is captured in the transitions of M_D. To determine whether there exists a plan to achieve φ, simply check whether the CTL formula $\mathsf{E}\Diamond\varphi$ (on some path, φ eventually holds) is true in M_D: if it does, then the witness to this will be a path through M_D encoding the actions that must be performed to achieve φ. The work of several other authors is worth mentioning: Bacchus and Kabanza have looked at the synthesis of "rule $\rightarrow$ action" pairs from temporal logic specifications [1], and have also investigated the synchronisation of multiagent plans using temporal logic [11]. However, these approaches are not based on model checking. Finally, Fisher's METATEM paradigm is based on the direct execution of temporal logic formula, although the execution algorithms are based on checking the satisfiability of the input formula, rather than on model checking [7].

6 Conclusions

We have described a novel approach to multiagent planning, and reported some results with an implementation of this approach — the `atmp` system. In `atmp`, multiagent planning is treated as a problem of Linear Temporal Logic model checking, as implemented in the SPIN system. Our results thus far are encouraging; the fact that we

```
#define EAST    0
#define WEST    1
int direction = WEST;

agent target %{
do
::  (direction == EAST) && (targetX == SIZE - 1) ->
    atomic {
        printf("target changing direction to WEST");
        direction = WEST;
        NEXT;
    }
::  (direction == WEST) && (targetX == 1) ->
    atomic {
        printf("target changing direction to EAST");
        direction = EAST;
        NEXT;
    }
::  (direction == WEST) && (targetX > 1) ->
    atomic {
        printf("target moves WEST");
        targetX = targetX - 1;
        NEXT;
    }
::  (direction == EAST) && (targetX < SIZE - 1) ->
    atomic {
        printf("target moves EAST");
        targetX = targetX + 1;
        NEXT;
    }
od
%}
```

Figure 4: The moving target as an agent (Experiment 3).

were able to get results with more than one agent, for moderately complex LTL goals suggests that the approach is certainly worth further investigation. There are many avenues for future research. An example is the ongoing development of more complex environments, with larger numbers of agents: on the basis of our experiments, we hypothesise that the number of agents/processes in the environment is the dominant factor in determining the time to solution, rather than merely the number of bits in the state vector. It would be interesting to investigate whether this hypothesis is borne out.
Acknowledgements: Thanks to Rafael Bordini for proof reading & sanity checking.

References

[1] F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1215–1222, Portland, OR, 1996.

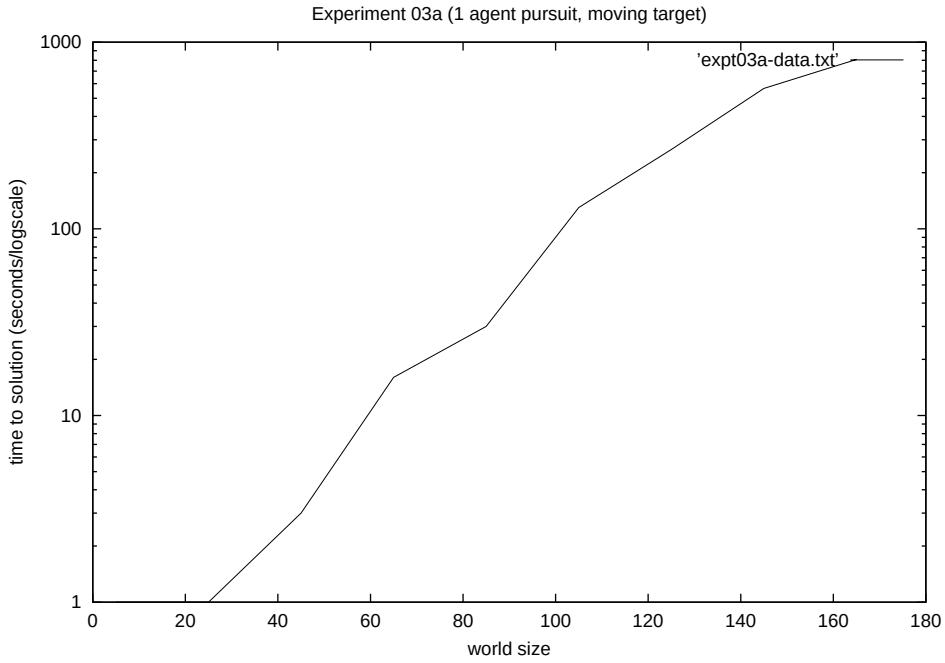


Figure 5: Experiment 3: The pursuit problem with a moving target.

[2] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *Journal of AI Research*, 13:305–338, 2000.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[4] E. H. Durfee. Distributed problem solving and planning. In G. Weiß, editor, *Multiagent Systems*, pages 121–164. The MIT Press: Cambridge, MA, 1999.

[5] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[6] R. E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[7] M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Berlin, Germany, July 1994.

[8] F. Giunchiglia and P. Traverso. Planning as model checking. In S. Biundo and M. Fox, editors, *Recent Advances in AI Planning (LNAI Volume 1809)*, pages 1–20. Springer-Verlag: Berlin, Germany, 1999.

[9] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall International: Hemel Hempstead, England, 1991.

[10] G. Holzmann. The Spin model checker. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997.

[11] F. Kabanza. Synchronizing multiagent plans using temporal logic specifications. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 217–224, San Francisco, CA, June 1995.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.

[13] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.

[14] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[15] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 133–192. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.