Using Model Checking to Find a Hidden Evader

Christopher A. Bohn[*] bohn@cis.ohio-state.edu
Paolo A.G. Sivilotti paolo@cis.ohio-state.edu
Bruce W. Weide weide@cis.ohio-state.edu

Department of Computer and Information Science
The Ohio State University
2015 Neil Avenue
Columbus, Ohio 43210-1277 USA

ABSTRACT

We present a pursuer-evader game in which the pursuer has a speed advantage over the evader but is incapable of determining the evader's location unless they both occupy the same location. By treating the players as nondeterministic finite automata, we can model the game and use it as the input for a model checker. By specifying that there is no way to guarantee the pursuer can locate the evader, the model checker will either confirm that this is the case, or it will provide a counterexample showing one search pattern for the pursuer that will guarantee the evader must eventually be caught. We show two such models and discuss variations to be investigated.

Keywords

Model checking, Unmanned aerial vehicle, Pursuer-evader game

1. INTRODUCTION

We have formulated a pursuer-evader game in which the pursuer can move faster than the evader, but the pursuer cannot ascertain the evader's location unless they occupy the same location. The pursuer's object is to occupy the same location as the evader eventually, whereas the object for the evader is to prevent collocation with the pursuer indefinitely. The game is loosely based on autonomous unmanned aerial vehicles (UAVs) with limited field-of-view attempting to locate an enemy vehicle on the ground.

In the game's simplest form, the game is played on a rectilinear grid with the one pursuer and the one evader taking turns to move. There are no obstacles to either the pursuer or evader, so each can occupy any location and can transition from any location to any adjacent location. Consider Figure 1. In Figure 1(a), the pursuer moves four spaces north and west on a 6×6 board; this is followed by the evader moving south one space in Figure 1(b). There are four basic variations on this form, based on the definition of "adjacent location". In all four variants, the players can move in the four cardinal directions (north, south, east, west); the variations are whether the pursuer, the evader, both, or neither can move diagonally (northwest, northeast, southwest, southeast). These simplifications facilitate an analysis of the necessary pursuer speed, though this analysis is beyond the scope of this paper.

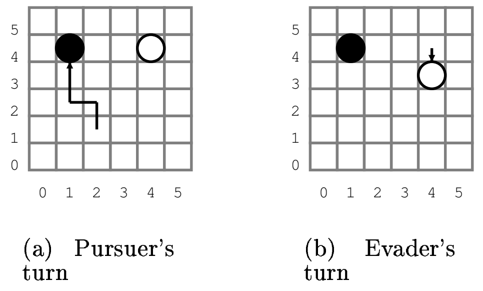


Figure 1: Examples of movements by the pursuer and the evader. Solid circle is the pursuer; hollow circle is the evader.

We also are investigating the automatic generation of search strategies for the pursuer that will guarantee that it will eventually locate the evader, as generalizing the game beyond its simple forms may not have easily-provable results, and they probably would require more time to hand-prove than is available (as a single data-point, we obtained our early empirical results well before our theoretical results were proven). We generate the search strategies using symbolic model checking. The simplifications described in the previous paragraph simplify the models we generate for the model checker, which helps us to refine the technique described in this paper.

In the remainder of this paper, we will offer an introduction to model checking in Section 2 and a description of how we use model checking to generate winning strategies for the

[*]The views expressed in this article are those of the author and do not necessarily reflect the official policy of the Air Force, the Department of Defense or the U.S. Government.

pursuer in Section 3. Then we will present an example in Section 4 and discuss the directions our research is going in Section 5.

2. MODEL CHECKING

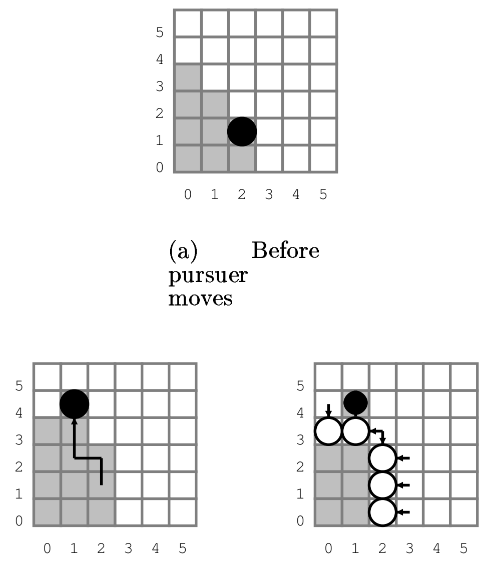
Model checking is a technique for verifying that finite state concurrent systems satisfy certain properties expressed in a temporal logic; typical examples of its use are in complex digital circuit designs and in communication protocols [1].

Symbolic model checking has three properties that are particularly desirable. First is that a model can be checked in time linear in the number of the model's states; more precisely, checking a specification f expressed as a predicate in Computation Tree Logic (CTL) requires $\mathcal{O}\left(\left(|S|+|R|\right)\cdot|f|\right)$ time, where S is the set of states and R is the set of transitions between states in the model [1]. The second is that the model's states are not represented explicitly; rather the model checker uses binary decision diagrams. Because of this, symbolic model checkers use less memory than explicit-state model checkers for a given model (or, for the same memory footprint, symbolic model checkers can check larger models). When first introduced ten years ago, symbolic model checking could verify models with up to 10^{20}(~2^{66}) states; subsequent refinements have permitted the checking of models with up to 10^{120}(~2^{398}) states [1].

The third property is that if the specification being checked is not satisfied by the model, the model checker will provide a specific counterexample showing why the property does not hold. We make use of this last capability by modeling the game as a nondeterministic finite automaton and instructing the model checker to prove that no matter how the pursuer moves, the evader can successfully avoid the pursuer. If the model checker indicates this property is true, then for the given game conditions there is no guarantee the pursuer and evader will be collocated. On the other hand, if the model checker indicates the property is false, then as a counterexample, it will indicate the model's state changes that, when properly interpreted, correspond to a sequence of moves the pursuer can make to guarantee it must locate the evader before the sequence is complete. We are pleased to report that the results of model checking the game's simple forms correspond to our analytical results, even though the sequence of pursuer moves provided by the model checker do not correspond to those used in the proofs of our analysis.

3. MODEL CHECKING THE GAME

Model checking this game is not as simple as one might hope. The naive modeler might encode the pursuer as moving nondeterministically from its current location to an adjacent location, and encode the evader similarly. The problem is that when the model checker is instructed to check whether the evader can always avoid detection, the model checker promptly offers the counterexample of the pursuer and evader approaching each other. This is not a useful result. Instead, we encode the pursuer as above, but instead of modeling the evader explicitly, we assign a predicate *cleared* to each location on the grid. A grid location is *cleared* if and only if it is not possible for the evader to occupy that location, given that the pursuer has not located the evader. Saying that all locations on the grid have been *cleared* is equivalent to saying that the pursuer has located the evader.



(a) Before pursuer moves

(b) Pursuer's turn

(c) Evader's turn

Figure 2: Examples of changes in the possible locations for the evader. Evader is known to be in unshaded region.

A location becomes *cleared* when the pursuer occupies that location, and it ceases to be *cleared* if it is possible for the evader to move into that location. Now the property to be checked is whether there is always at least one grid location that remains not *cleared*. If there model checker confirms this property is satisfied, then there is no pursuer strategy that will guarantee its victory; on the other hand, if the property is not satisfied, then the counterexample can be used as the pursuer's search strategy to clear every location.

Compare Figure 2 with Figure 1. In this hypothetical scenario, the pursuer has *cleared* a region of the southwest corner of the grid, as shown by the shaded portion of Figure 2(a)) and can conclude the evader must be outside that region. As in Figure 1(a), the pursuer moves four spaces north and west in Figure 2(b), increasing the *cleared* region by three cells. Since the pursuer does not know where the evader is located, the *cleared* region must shrink in accordance with the union of all possible moves by the evader. A move by the evader south from the northeastern-most corner would not cause the evader to enter a previously-*cleared* cell, but Figure 2(c) shows there are six ways the evader *could* move from an un*cleared* cell into a *cleared* cell, and the five *cleared* cells that could now be occupied by the evader may no longer be considered *cleared*.

The astute reader will note that the state space in this new model is considerably greater than in the naive model. Neglecting overheads such as variables used to synchronize the pursuer and evader, if the grid has n locations, then the state space of the first model has n^2 states – n possible locations for the pursuer and n possible locations for the evader. In the second model, while the pursuer still has n

Board Dimensions	Number of Locations	Pursuer Speed (moves/turn)	Number of States	Reachable States	Time	Memory
2×2	4	2	$2^{8.6}$	$2^{6.6}$	$0.01s$	$1.2MB$
2×4	8	2	$2^{13.6}$	$2^{11.1}$	$0.04s$	$1.3MB$
3×3	9	3	$2^{15.2}$	$2^{13.3}$	$0.08s$	$1.3MB$
3×5	15	3	$2^{21.9}$	$2^{17.6}$	$0.66s$	$2.6MB$
4×4	16	4	$2^{23.3}$	$2^{19.4}$	$5.39s$	$7.3MB$
4×6	24	4	$2^{31.9}$	$2^{23.3}$	$7m\ 26s$	$52.1MB$
5×5	25	5	$2^{33.2}$	$2^{25.7}$	$2h\ 4m\ 3s$	$180.4MB$
5×7	35	5	$2^{43.7}$	??	$> 24h$	??
6×6	36	6	$2^{45.0}$	??	$> 24h$	??

Table 1: Time and memory requirements to obtain a pursuer search strategy.

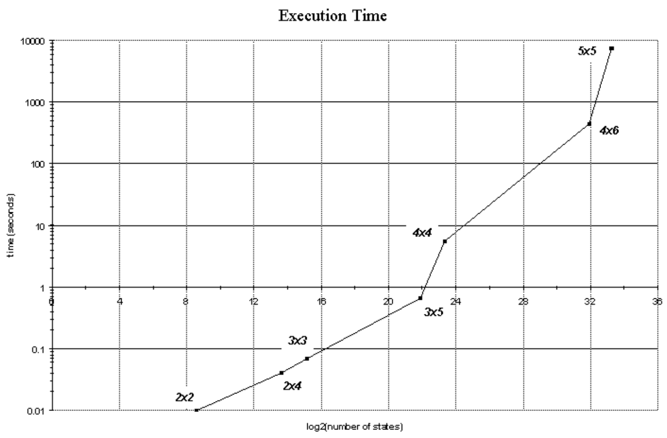


Figure 3: Time required by model checker to complete, by the the size of the game model.

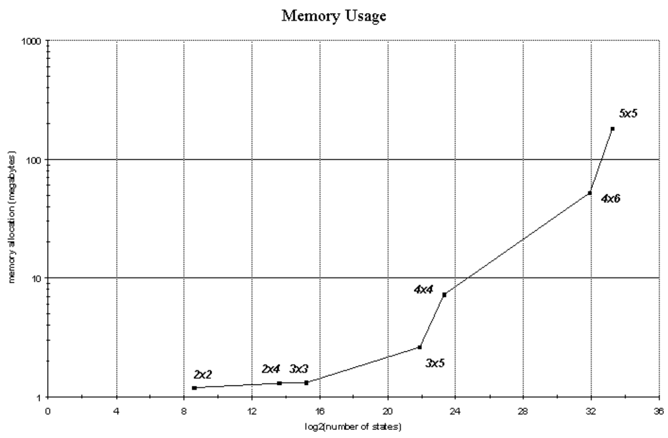


Figure 4: Memory displacement of model checker, by the size of the game model.

possible locations, there is not an evader; instead, there are n boolean variables. This means the state space in the new model has $n \cdot 2^n$ states. When the number of states is exponential in the size of the grid, being able to check properties in time that is linear in the number of states still requires time that is exponential in the size of the grid. Table 1 and Figures 3 and 4 shows the execution time and memory displacement to run the model checker SMV [2] on a 933MHz Pentium III system for different-sized boards with movement only in the four cardinal directions. The jagged appearance of the graphs is the effect of including data from $n \times n$ boards and $n \times (n + 2)$ boards on the same graph.

Coping with this complexity is critical if we are to improve the fidelity of the models relative to their real-world inspiration. After looking into alternative logics, including Alternating-Time Temporal Logic (ATL) [3] and a proposed variant, Blinded ATL, we have concluded there is no escaping this complexity. Instead we are investigating ways to reduce the state space by constraining the forms solutions might have.

4. EXAMPLE

The SMV model checker's input language consists of modules organized as a collection of variables and their update actions. The distinguished `main` module is used to instantiate the other modules into a concurrent system. All assignment statements in `main` and all instantiated modules are executed in parallel [4]. We shall now present and describe two models used to generate winning pursuer strategies.

4.1 Full model for 2×2 grid

The first model is that for a 2×2 grid with the pursuer and evader moving only in the four cardinal directions. Though this is a small grid size, the structure of the models is sufficiently regular that the reader should be able to infer what a larger model would look like.

We begin with the outline of the `main` module and the variable it's responsible for, the *clock*:

```
MODULE main
VAR
  clock : 0..2;
  ...
DEFINE
  speed := 2;
  ticks := speed + 1;
ASSIGN
  init( clock ) := 0;
  next( clock ) := clock + 1 mod ticks;
SPEC
  ...
```

Here we define the pursuer's *speed* as 2, and so the clock has four *ticks*, 0..3. Initially, *clock* is assigned the value 0, and in each round it is updated by 1 modulo *ticks*. By convention, the pursuer moves when $0 \leq clock < speed$, and the evader moves when $clock = speed$.

Now we consider the module for each `cell` on the grid:

```
MODULE cell( x , y , n , w , e , s ,
             hunter , timer )
VAR
  inferClear : boolean;
DEFINE
  occupied := ( x=hunter.x &
                y=hunter.y );
  neighborsClear := ( n.cleared &
                      w.cleared &
                      e.cleared &
                      s.cleared );
  cleared := ( inferClear | occupied );
ASSIGN
  init( inferClear ) := 0;
  next( inferClear ) :=
    case
      timer = 2    : occupied |
          ( inferClear & neighborsClear );
      !(timer = 2) : cleared;
    esac;
```

A `cell` is defined by its location on the grid (x, y) and its neighbors (n, w, e, s). The instantiation parameters *hunter* and *timer* are used to create aliases to the pursuer and the clock, respectively, so the assignment statements can query their states. We define *occupied* as the predicate that the *hunter*'s x and y values are identical to those of the `cell`, and we define the predicate *neighborsClear* as the conjunct of whether the cells to the north, west, east, and south are *cleared*. The predicate *cleared* is the disjunction of whether the `cell` is *occupied* (in which case it is trivially *cleared*) and of whether we can infer the `cell` to be *cleared*. The variable *inferClear* is initially FALSE. If it is the pursuer's turn to move, *inferClear* assumes the value of the *cleared* predicate from the previous clock tick. If it is the evader's turn to move, then as with Figure 2(c), *inferClear* can remain TRUE only if it's *occupied* or if it was already *cleared* and all its neighbors are *cleared*.

Since the evader cannot leave the grid, we define `nullCells` to serve as neighbors to `cells` along the grid boundary; `nullCells` are always *cleared*:

```
MODULE nullCell
DEFINE
  cleared := 1;
```

The only module remaining is that for the `pursuer`:

```
MODULE pursuer( timer )
VAR
  x : 0..1;
  y : 0..1;
  moveDirection : { xx , yy };
ASSIGN
  init( x ) := 0;
  init( y ) := 0;
  init( moveDirection ) := { xx , yy };
  next( moveDirection ) := { xx , yy };
  next( x ) :=
    case
      moveDirection = xx & !(timer = 2) :
      case
        x = 0 : {     0 , 1 };
        x = 1 : { 0 , 1     };
      esac;
      moveDirection = yy | timer = 2 : x;
    esac;
  next( y ) :=
    case
      moveDirection = yy & !(timer = 2) :
      case
        y = 0 : {     0 , 1 };
        y = 1 : { 0 , 1     };
      esac;
      moveDirection = xx | timer = 2 : y;
    esac;
```

As with a `cell`, a `pursuer` occupies some location on the grid(x, y), which range over the dimensions of the grid; the variable *moveDirection* is a modeling artifact to ensure the pursuer does not move diagonally. As with the `cell` module, the `pursuer` module maintains an alias to the clock in the *timer* parameter. Initially, a `pursuer` is positioned in the southwest corner $(0, 0)$, and its initial *moveDirection* is a nondeterministic choice. In each subsequent round, the choice of *moveDirection* remains nondeterministic. To update the x variable, the `pursuer` must be moving in the xx direction, and it must be the pursuer's turn to move. If x can be updated, then in general it can be decremented, maintained, or incremented – if x is either its least or greatest value, then care must be taken not to update x to a value outside its range. The update for y is analogous.

Now that we've defined all the necessary modules, we can instantiate them in the `main` module:

```
MODULE main
VAR
  clock : 0..2;
  UAV   : pursuer( clock );
  cNul : nullCell;
  c0_0 : cell( 0 , 0 , c0_1 , cNul ,
               c1_0 , cNul , UAV , clock );
  c0_1 : cell( 0 , 1 , cNul , cNul ,
               c1_1 , c0_0 , UAV , clock );
  c1_0 : cell( 1 , 0 , c1_1 , c0_0 ,
               cNul , cNul , UAV , clock );
  c1_1 : cell( 1 , 1 , cNul , c0_1 ,
               cNul , c1_0 , UAV , clock );
DEFINE
  speed := 2;
  ticks := speed + 1;
ASSIGN
  init( clock ) := 0;
  next( clock ) := clock + 1 mod ticks;
SPEC
  ...
```

All that remains is the specification, expressed as a CTL formula. Typically, this would be some property of the model that the modeler would want to be satisfied. What is the property that we want? That all `cells` are *cleared*:

$$c0\_0.\mathit{cleared} \wedge c0\_1.\mathit{cleared} \wedge$$
$$c1\_0.\mathit{cleared} \wedge c1\_1.\mathit{cleared}$$

We want this state predicate to eventually be TRUE, and we express this by creating a path formula with the state formula preceded by an `F` (eventually, or "in the **f**uture on this path"). We only require that this path formula be satisfied in at least one possible computation, and this would be the computation the pursuer would follow; we express this by creating a state formula with the path formula preceded by an `E` (existentially, or "a path **e**xists a path"). This creates the following formula that we require to be satisfied in the initial state:

$$\mathtt{E\ F}\quad c0\_0.\mathit{cleared} \wedge c0\_1.\mathit{cleared} \wedge$$
$$c1\_0.\mathit{cleared} \wedge c1\_1.\mathit{cleared}$$

If we were to use this specification, then SMV would return with the answer that, yes, the specification is satisfied – there does exist a computation path along which there is some future state in which all `cells` are *cleared.* But we desire to know *how* to get to that future state. Recalling that SMV provides a counterexample whenever the specification is not satisfied, we negate the specification we just formulated – we instruct the model checker to prove that no computation exists:

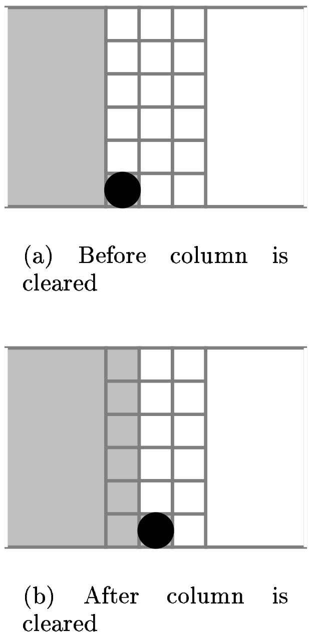
```
SPEC
  !EF (
        c0_0.cleared & c0_1.cleared &
        c1_0.cleared & c1_1.cleared
      )
```

It is useful to observe that this formula is equivalent to expressing that invariantly, at least one `cell` is not *cleared.* "Invariantly" is described with `A G`, where `A` is the universal quantifier ("**a**ll paths") and `G` is the always operator ("**g**lobally on this path):

```
SPEC
  AG !(
        c0_0.cleared & c0_1.cleared &
        c1_0.cleared & c1_1.cleared
      )
```

This observation is useful for more than as an exercise in quantifier/operator manipulation; with appropriate command-line arguments, SMV can make use of certain optimizations to reduce the execution time when checking such "AG" specifications.

The counterexample produced by the model checker when it finds the specification is not satisfied shall be the very path we want the pursuer to use. By causing all cells to be *cleared,* the pursuer is guaranteed to have found the evader at some point during the game. For this small 2×2 problem, the counterexample is the pursuer moving from $(0,0)$ to $(0,1)$ and to $(1,1)$. The evader then moves, and $(0,0)$ is no longer *cleared.* Now the pursuer moves from $(1,1)$ to $(1,0)$ and to $(0,0)$, and all cells have been *cleared.*


(a) Before column is cleared

(b) After column is cleared

Figure 5: Abstraction of grid unbounded in the x direction.

4.2 Abstracted model for $\infty \times 6$ grid

We noted at the end of Section 3 that if we constrained the form of the pursuer's search strategy, then we can keep the state space relatively small. For the simple game variant that we have presented here, one such restriction is that the pursuer clears column of the grid while preventing the evader from moving into previously-*cleared* column, and then the pursuer positions itself to clear the next column. If we obtain a computation path that describes this, and we repeatedly apply it, then the pursuer will "sweep" across the grid, eventually clearing all locations.

Consider Figure 5. We may consider this to be a grid with an arbitrary number of *cleared* columns to the west and an arbitrary number of un*cleared* columns to the east. If it's ever possible for the evader to enter the westernmost region, then the technique of clearing columns will not compose, and so while maintaining the possibility that the "cell" at the western edge of the grid can become un*cleared,* we require that it remain *cleared.* Contrariwise, the "cell" at the eastern edge will never be *cleared* since we will not permit the pursuer to move too far from the column it must clear. By limited the pursuer to the current column and two more to the east, we keep the state space small; our model has 2^{28} states, with $2^{21.1}$ of them reachable – compare this to the state spaces in Table 1.

The building-block modules for this new model are the same as in the previous model, with two exceptions. The x variables range over $1..3$ instead of $0..x_{max}$, and there is one addition, a `borderCell` module that will be used at the westernmost and easternmost positions of the grid:

```
MODULE borderCell( b0 , b1 , b2 ,
                   b3 , b4 , b5 ,
                   initClear , timer )
VAR
  inferClear : boolean;
DEFINE
  occupied := 0;
  neighborsClear := ( b0.cleared &
                          ...    &
                      b5.cleared );
  cleared := ( inferClear | occupied );
ASSIGN
  init( inferClear ) := initClear;
  next( inferClear ) :=
    case
      timer = 6    : occupied |
         ( inferClear & neighborsClear );
      !(timer = 6) : cleared;
    esac;
```

Not surprisingly, this module is similar to the `cell` module. The first difference is that the `borderCell` doesn't have an alias to the pursuer and isn't instantiated with its coordinates, since we are constraining the pursuer such that it cannot occupy a `borderCell`. Instead, the initial value of *inferClear* is passed as a parameter – the western `borderCell` must always be *cleared*, and the eastern `borderCell` is never *cleared*.

The modules are instantiated in `main`:

```
MODULE main
VAR
  clock : 0..6;
  UAV   : pursuer( clock );
  cNul  : nullCell;
  c0    : borderCell( c1_0 , c1_1 , c1_2 ,
                      c1_3 , c1_4 , c1_5 ,
                      1 , clock );
  c1_0 : cell( 1 , 0 , c1_1 , c0   ,
               c2_0 , cNul , UAV , clock );
  c1_1 : cell( 1 , 1 , c1_2 , c0   ,
               c2_1 , c1_0 , UAV , clock );
  ...  : ...
  c3_4 : cell( 3 , 4 , c3_5 , c2_4 ,
               c4   , c3_3 , UAV , clock );
  c3_5 : cell( 3 , 5 , cNul , c2_5 ,
               c4   , c3_4 , UAV , clock );
  c4   : borderCell( c3_0 , c3_1 , c3_2 ,
                     c3_3 , c3_4 , c3_5 ,
                     0 , clock );
...
```

And now we must formulate an appropriate specification. We shall make use of CTL's "until" operator (`U`). Specifically, *c*0.*cleared* must be TRUE *until* the pursuer has cleared the current column and is positioned to clear the next column. When the pursuer has cleared the current column, *c*0.*cleared* must still be TRUE, as must

$$\bigwedge_{0 \le y < 6} c1\_y.cleared$$

When the pursuer is in position for the next column, it must be in cell $(2,0)$ or $(2,5)$, and it must be the beginning of the pursuer's next turn ($clock = 0$). As with the specification for the previous model, we only require this to occur along at least one computation path, and to obtain the states along that path, we negate the formula:

```
SPEC
  !E [ c0.cleared U (
       clock=0 & c0.cleared &
       UAV.x=2 & ( UAV.y=0 | UAV.y=5 ) &
       c1_0.cleared & c1_1.cleared &
       c1_2.cleared & c1_3.cleared &
       c1_4.cleared & c1_5.cleared ) ]
```

Obtaining the counterexample (a computation path of 50 states) for the specification of this model requires 76 seconds and 12.25 MB.

5. FUTURE WORK

The other paths along which our research is progressing are variations on the game. This includes hexagonal grids, multiple cooperative pursuers, concurrent movement of the pursuer and evader, transitions which the pursuer and/or evader cannot make, grid locations which the pursuer and/or evader cannot occupy[1], different fields of view for the pursuer, limits on the pursuer's turn radius, a known or bounded initial position for the evader, and combinations thereof.

There are some simple ways to take advantage of having multiple pursuers, such as dividing the grid and assigning portions to each pursuer and using the single-pursuer solutions – though even in this simple approach, some care is needed or it might be possible for the evader, occupying a section of the grid assigned to one pursuer, to move into a section of the grid assigned to another pursuer that was believed to be inaccessible to the evader. This care is obviously needed if the pursuers travel at different speeds, but it is fact needed even when the pursuers are identical. Of greater interest is whether there are faster solutions than simply treating the multiple-pursuer variant as a collection of smaller single-pursuer games. We believe this will become even more interesting when applied to variations other than a turn-based game with a simple rectilinear grid and no obstacles. Another variant would be dealing with pursuers that have different speeds and different capabilities, such as a "hunter" UAV that locates targets and then leaves the target's location to seek other targets while a "killer" UAV uses the target's known-initial location to re-locate and destroy the target.

6. CONCLUSION

We have described a means to generate search strategies that will guarantee a pursuer can locate an evader, or to establish that no such strategy exists. The state space grows exponentially and generating search strategies becomes unwieldy with even relatively small problems. As we investigate more interesting game variants, this problem will be exasperated. This has led us to investigate ways to constrain the form the search strategies may take. One such example in which we can compose the winning strategies of smaller problems has been presented here, and we have shown that a model checker can generate the composable strategies in considerably less time than would be required to generate a monolithic strategy.

[1]Of course, if there is some location the pursuer cannot occupy but the evader can, then there can be no guaranteed win for the pursuer unless there is a fairness requirement that the evader must infinitely-often not occupy any given cell.

Acknowledgment

The authors gratefully thank the US Air Force and the Air Force Institute of Technology for their direct support of the primary author. This work was supported by the AFRL/VA and AFOSR Collaborative Center of Control Science (Grant F33615-01-2-3154).

7. REFERENCES

[1] Jr. Edmund M. Clark, Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge MA: The MIT Press, 1999.

[2] The SMV system. http://www.cs.cmu.edu/˜modelcheck/smv.html.

[3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, September 2002.

[4] K.L. McMillan. *The SMV System for SMV version 2.5.4*, November 2000. User's Manual.