Algorithms for Real-Time Game-Tree Search for Hybrid System Control

Todd W. Neller*
Knowledge Systems Laboratory
Gates Building 2A
Stanford University
Stanford, California 94305-9020 USA
neller@ksl.stanford.edu

Abstract

This paper describes four algorithms for real-time game-tree search for hybrid system control. A hybrid system control game is a hybrid system with discretely and continuously evolving scores, and an enabled action set for each player. As computational speed increases, we can expect simulation to become more useful for informing control decisions in real-time. To this end, we seek to extend existing game-tree search techniques for real-time hybrid system control.

We introduce the notion of an n-player augmented cell-map and apply both dynamic programming and an anytime minimax algorithm with caching. For games with zero-sum scores, we generalize alpha-beta for n-players. Combining the best characteristics of these algorithms, we introduce a generalized caching alpha-beta algorithm for graphs. We discuss the benefits and limitations of each algorithm.

1 Introduction

This paper describes four algorithms for real-time game-tree search for hybrid system control. In (Neller 1998b), we formally define a *hybrid system control game* as a hybrid system with discretely and continuously evolving scores, and an enabled action set for each player. (Başar & Olsder 1995) is an excellent reference for differential game theory. However, most existing game-theoretic techniques are not suited to the more general dynamics of hybrid system. Model-based control[1] has focused on systems which evolve slowly over time. Expected increases in computational speed will enable online decision-making for much faster dynamics. To this end, we seek to extend existing game-tree search techniques for hybrid system control with an emphasis on real-time constraints.

In this paper, we introduce the notion of an n-player augmented cell-map and apply both dynamic programming and an anytime minimax algorithm with caching. For games with zero-sum scores, we generalize alpha-beta for n-players. Combining the best characteristics of these algorithms, we introduce a generalized caching alpha-beta algorithm for game-graphs. (Russell & Norvig 1995) provides a good introduction to minimax search and alpha-beta pruning.

Ideally, a game-playing agent[2] would be omniscient of all possible system behaviors and would play optimally with respect to models of the other agents' decision procedures. Simulation cannot give us such omniscience. However, we can use simulation to sample the infinitely branching game-tree and choose between alternatives.

There are two main optimizations at the core of hybrid game-tree search techniques. First, we wish to optimize our agent's choice of simulation to grow the game-tree according to its relevance to decision making. Discretization decisions concerning action and action timing alternatives should be guided by relevance to the control decision to the extent that such optimization overhead doesn't eliminate it's own utility in real-time. Second, we wish to optimize our agent's choice of action according to the game-tree. Interestingly, the two are intertwined.

The algorithms in this paper perform variants of minimax search with a predetermined discretization of the state space. They underscore the need for balance in avoiding over-use or under-use of computational resources. While many more interesting issues await in the continuation of this research, this foundational work brings into focus the computational tradeoffs which must be addressed to exploit game-tree search for real-time hybrid system control.

There are several motivations for this research. First, we wish to enable greater autonomy of real-time hybrid system control. Given models of hybrid system behavior, we wish enable agents to use them to inform action through game-play. We believe such game-theoretic deliberation with hybrid models will form an important, necessary part of future intelligent control systems.

A second motivation for this research is to provide efficient, incomplete search procedures to complement

*This work was supported by the Stanford Lieberman Fellowship, and the Defense Advanced Research Projects Agency and the National Institute of Standards and Technology under Cooperative Agreement 70NANB6H0075, "Model-Based Support of Distributed Collaborative Design".

[1] control using online models to inform control decisions

[2] e.g. an unspecified controller or adversarial control disturbance

complete verification techniques. We've seen how verification procedures proving safety of a system can be complemented by an efficient, incomplete procedure searching for an unsafe counterexample trajectory (Neller 1998a). Likewise, complete, game-theoretic hybrid system control verification techniques such as that of (Tomlin, Lygeros, & Sastry 1998) should be complemented by the incomplete, game-theoretic hybrid system control search techniques we are developing.

Finally, we're motivated to bridge a perceived gap between the AI chess researcher and the control game-theorist by extending discrete game techniques for relevance and irrelevance reasoning to the control of hybrid systems. AI discrete game-tree search research has made interesting contributions concerning relevance and irrelevance reasoning. While control can benefit from these ideas, AI research can benefit from a greater diversity of games. However, ideas and problems have not flowed easily between these two disciplines. We seek to bridge this perceived gap.

The paper is organized as follows: In Section 2, a dynamic programming cell map method for offline control design is described. For online control, a graph-based minimax algorithm with caching is described in Section 3. Aiding game-play with irrelevance reasoning, an n-player generalization of alpha-beta pruning for minimax game-tree search is described in Section 4. The benefits of these algorithms are then combined in the graph-based generalized alpha-beta method of Section 5. Section 6 summarizes applicability of the algorithms, discusses some of the issues not addressed by them, and outlines future work.

2 Dynamic Programming Cell Map Method

Cell mapping methods (Hsu 1987) have been used to perform state-space analysis of dynamical systems. In such methods the state-space is divided into cells. Each cell is mapped to another cell to which it will evolve after a fixed time interval. The resulting graph approximation of the system dynamics is then analyzed. One advantage of cell mapping is that one can form an approximation of the state space according to computational space limits, and perform an efficient, polynomial-time, global state-space analysis.

In seeking to extend such methods to n-player games, we augment the cell map with set-valued mappings from a [cell, player] pair to a set of cells, circumscribing the possible effects of a player's actions in that cell. For each player, each cell is now mapped to a set of cells to which it may evolve after a fixed time interval. Rather than performing minimax on a tree, we perform minimax on the approximating graph instead, thus reducing the exponential complexity of a minimax tree search to the polynomial complexity of a minimax graph search. Our generalization of minimax for n-players follows (Russell & Norvig 1995, ex. 5.8) where each player seeks to maximize its component of a score vector.

Algorithm 1 Procedure Iterated by Dynamic Programming Cell Map Method

```
procedure HybridDPiter(augmentedCellMap, player)
  for each cell in augmentedCellMap
    cell.newScore := negInfVector
    for each destCell in cell.playerMap[player]
      newScore :=cell.moveScore(player, destCell)+
                  destCell.score
      if newScore[player] > cell.newScore[player]
        then cell.newScore := newScore
  for each cell in augmentedCellMap
    cell.score := cell.newScore
```

Algorithm 1 is the core procedure for our dynamic programming cell map method. Following initialization, this procedure is iterated on the cell map in reverse turn order in the dynamic programming style[3]. To initialize, first zero the augmented cell map score vectors. Then initialize the individual set-valued player maps which indicate the possible actions of each player at each cell. For example, we have applied this method to the magnetic levitation (maglev) problem of (Zhao, May, & Loh 1998) in which the goal is to suspend a metal ball beneath an electromagnet. We take a game-theoretic approach for the purpose of synthesizing safe maglev control in the face of external perturbation and error introduced through modeling approximations and numerical simulation. The problem is thus described as a game where the controller may change the magnetic coil current while the adversary may perturb the behavior of the system in the period between controller actions. The controller player map maps each cell to all other cells which differ only in controller input (current). The adversary player map maps each cell to the set of cells possibly reachable during the continuous system evolution phase, taking into account perturbation and error.

Since players need not necessarily alternate turns, let us for ease of analysis define b as the effective branching factor of the player mappings as used over successive calls to Algorithm 1. Let c be the number of cells and p be the number of players. Then the time and space complexity of Algorithm 1 are $O(cb)$ and $O(cpb)$, respectively. With player maps compactly represented and/or conservatively approximated, the space complexity may be reduced to $O(cp)$.

What we have not figured into this analysis is the "curse of dimensionality" in the state-space. If we divide a state-space into a uniform grid of cells, the number of cells will grow exponentially with the dimension of the space. Thus this method is applicable to systems with low-dimensional state-spaces.

[3] from terminal states at some time horizon backwards in time through decision stages

This method also places the burden of cell-partitioning and time discretization on the user. Too coarse a cell-partition, and such computation yields little information. Too fine a partition, and we violate space constraints. Too large or too small a time-step, and approximation and numerical errors dominate. While adaptive techniques for cell-decomposition are being developed (Bradley 1995), these discretization issues are far from resolved.

We note that this method is not suited for real-time online use. While such a method could be used offline to form a control policy a priori, it is not designed to focus on an immediately relevant control decision. Rather, its computation is distributed across the entire augmented cell map. This limitation is addressed through the algorithm in the next section.

In summary, the dynamic programming cell map method has polynomial time and space complexity and is applicable to offline control design for low-dimensional state spaces, assuming that a good discretization can be found. We now turn our attention to online use of minimax.

3 Graph-Based Minimax

In this section, we introduce Algorithm 2 which performs minimax on a graph and caches partial computations. This algorithm can be used in real-time to focus computation on the decision at hand. Algorithm 2 is presented for general game-graphs. There is a graph node for each cell-player pair. We also allow for terminal game nodes (no children). Keeping the same amortized computational time complexity of Algorithm 1, we trade off increased caching space requirements, bringing the computational space complexity to $O(cp(b+d))$, where d is the maximum search depth. This is a small tradeoff for such a large gain in the immediate utility of computation.

Algorithm 2 Graph-based Minimax

```
procedure GraphMinimax(node, depth)
  if ((depth = 0) or node.complete[depth])
    then return
  if empty(node.children) then
    bestScore := zeroVector
  else
    bestScore := negInfVector
    for each child in node.children
      GraphMinimax(child, depth − 1)
      newScore :=node.moveScore(child)+
                 child.score[depth − 1]
      if newScore[player] > bestScore[player] then
        bestScore := newScore
  node.score[depth] := bestScore
  node.complete[depth] := true
```

In Algorithm 2, the "complete" flags of each node signal when score information may be used by future calls. Since this algorithm keeps track of which subtree computations are complete, it may be used as part of an anytime algorithm, and total minimax computational cost can be amortized over successive calls. At time t, when a control decision is required at $t + \Delta t$, simulate forward to choose the relevant node, and perform iterative-deepening graph-based minimax on the node. That is, call GraphMinimax with successively increasing depth until just before $t + \Delta t$. The iterative-deepening procedure is then interrupted, the best next cell for the deepest complete minimax computation is chosen, and a control action leading to that cell is executed. [4]

To summarize, graph-based minimax of Algorithm 2 has the same amortized time complexity as Algorithm 1, but with a linear increase in space complexity. From this increase, we gain a focus of computation relative to a specific control decision, thus Algorithm 2 can be applied to an augmented cell map for online hybrid system control for low-dimension state spaces, assuming that a good discretization can be found. Of course, minimax computation wastes much time deep in the tree/graph, computing information provably irrelevant to the top-level decision. For zero-sum scores[5] , one approach to reasoning about irrelevance in the game tree is alpha-beta pruning. We look at an n-player generalized version of alpha-beta in the next section.

4 Generalized Hybrid Alpha-Beta Method

In minimax search, a game-tree is generated with two players MAX and MIN, alternately maximizing and minimizing the score at alternating depths of the tree. However, much of the tree need not be generated (i.e. it can be "pruned") since it is provably irrelevant given information gained during search. The idea of alpha-beta pruning for minimax search was conceived by John McCarthy in 1956, but a full description of the algorithm was not published until 1969 (Slagle & Dixon 1969). The core idea is this: If, in evaluating a node of a game tree, one can prove that a rational player will not choose the path to that node, one can avoid examination of (i.e. "prune") the subtree rooted at that node. By simple bookkeeping of the best score that each player can be guaranteed to achieve, asymptotic optimality is gained for such searches.

Our generalized extension of alpha-beta search, Algorithm 3, allows zero-sum adversarial hybrid system game-play for n players, discretizing times at which each player may make decisions. For example, we may allow one player to play every second while another is allowed to play every half second. Such discretization may correspond to the timing of a discrete controller, but will more likely be determined by the complexity of the search and the time horizon to which one wishes

[4] One should assume a default control action for incompleted depth 1 to create a true anytime algorithm.

[5] player scores constrained to sum to zero in all states

to look forward. Initially all vectors are zeroed except for the input prevGuaranteeVector which is initialized to a vector of $n\ -\infty$'s. Unlike original alpha-beta, each player in our generalized hybrid alpha-beta is seeking to maximize its individual score (utility) which is possibly affected by both discrete and continuous behaviors. Whereas the state evolved discretely with alpha-beta, generalized hybrid alpha-beta state evolves according to the simulation of the underlying hybrid system.

Algorithm 3 Generalized hybrid alpha-beta search

```
function HybridAB(node, prevGuaranteeVector)
  scoreGuaranteeVector := prevGuaranteeVector
  bestMove := NULL_MOVE
  if (LeafNode(node) or CutoffTest(node.state)) then
    node.abScoreVector := node.scoreVector
    return bestMove
  m := GenerateMove(node)
  prune := false
  while not (m = NULL_MOVE or prune) do
    child := SimulateMove(m, node)
    HybridAB(child, scoreGuaranteeVector)
    s := child.abScoreVector[node.player]
    if (bestMove = NULL_MOVE or
        s > bestScoreVector[node.player]) then
      bestScoreVector := child.abScoreVector
      bestMove := m
      for i := 1 to players
        if (not i = node.player) and
           (bestScoreVector[i] < scoreGuaranteeVector[i])
          then prune := true
      if (s > scoreGuaranteeVector[node.player])
        then scoreGuaranteeVector[node.player] := s
    if not prune then m := GenerateMove(node)
  node.abScoreVector := bestScoreVector
  return bestMove
```

Algorithm 3 is briefly described as follows: After initialization, check if the node is a leaf-node and, if so, assign the node score vector to the node's alpha-beta score vector, and return. Otherwise, for each allowable player move in succession, do the following: Generate a child node through hybrid simulation of the move. This also evolves the child node score. Recursively perform alpha-beta search on the child node and take note of the score the current player would gain from making that move. If it is the new best move from the node, check if the subtree can be pruned and update the node score guarantees. Discontinue this iteration if the pruning condition is met. Finally, update the node's alpha-beta score vector.

The zero-sum algebraic constraint over the scores provides the rational basis for this pruning, but what if the game is not zero-sum? Interestingly, knowledge of one's problem domain may provide even more useful constraints. If it can be proved that one player will choose a move in a state which is guaranteed to cause another player to preclude the possibility of reaching that state out of preference for another line of play, all search beyond that state may be pruned. For instance, consider a cooperative form of the aircraft collision avoidance problem of (Tomlin, Lygeros, & Sastry 1998), where all scores are identically the minimum distance between any two aircraft over time. Once all aircraft are receding from one another, we may obviously conclude that the scores will remain fixed. This is an example of a constraint on future scores which enables pruning without ever reaching cutoff states. Pruning constraints may take on other forms as well. If, for instance, it can be proved that the best adversarial maglev perturbation is a maximal perturbation, we reduce the dimensionality of relevant adversary actions. In broadening the constraints one considers, one may introduce far more significant forms of pruning to minimax search.

For real-time control, such an algorithm could be used within an iterative-deepening, or iterative-refinement anytime algorithm. By iterative-refinement, we mean that we start with a coarse discretization of player decision points and compute an approximate solution (recommended control action) with our hybrid alpha-beta algorithm. We store the action, refine our discretization (i.e. allow more frequent turns), and iterate, computing successively better approximate solutions until the algorithm is halted and the stored action is returned.

Although this approach does not require discretization of the state-space, the user still has to supply discretizations of continuous ranges of actions and decision times. However, there are possibilities for dynamically choosing and refining such discretizations which will be discussed in Section 6.

One limitation of this approach is one shared by all tree-based methods: High branching factors quickly force shallow search. Since we are dealing with a minimax search on a tree rather than a graph, the time complexity is $O(b^d)$, where b is the effective branching factor and d is the maximum search depth. However, the space complexity is $O(d)$, so we've significantly traded off time for space. We have not only under-utilized computational space resources, but we've saved no information for future use and cannot expect its performance to improve over time. Given the infinite state-space of the search, and the approximate nature of simulation, it would make sense to use approximation and/or abstraction in order to achieve better performance over time. One possible step in this direction is to use generalized alpha-beta with iterative-deepening on a cell-map, caching results of partial alpha-beta computations in order to speed-up future minimax searches and allow greater depth of search over time. We introduce this new synthesis of techniques in the next section.

5 Graph-based Generalized Alpha-Beta

In this section, we propose a new synthesis of the previous techniques. This graph-based, generalized alpha-beta algorithm (1) represents an n-player game as a graph with zero-sum node and cost scores, (2) caches results of partial computations to reduce amortized cost of successive alpha-beta searches, and (3) is suited for real-time use in an anytime algorithm. For brevity, we ignore many details and present the algorithm for use on n-player discrete game-graphs. For application to hybrid control systems, the game-graph is an augmented cell map.

Algorithm 4 Graph-based generalized alpha-beta search

```
function GraphGenAB(node, prevGuaranteeVector, depth)
  if (depth = 0) or LeafNode(node) then
    return {NULL_MOVE, worstScoreVector}
  if node.complete[depth] then
    return {node.bestMove[depth], worstScoreVector}
  if (prevGuaranteeVector ≥ node.pruneCondVector[depth])
    then return {node.bestMove[depth],
                 node.pruneCondVector[depth]}
  scoreGuaranteeVector := prevGuaranteeVector
  newPruneCondVector := worstScoreVector
  newBestMove := NULL_MOVE
  m := GenerateMove(node)
  prune := false
  while not (m = NULL_MOVE or prune) do
    child := MakeMove(m, node, depth)
    {childBestMove, childPruneCondVector} :=
      GraphGenAB(child, scoreGuaranteeVector, depth−1)
    newPruneCondVector :=
      Max(newPruneCondVector, childPruneCondVector)
    scoreVector := child.abScoreVector[depth]
    s := scoreVector[node.player]
    if (newBestMove = NULL_MOVE or
       s > newBestScoreVector[node.player]) then
      newBestMove := m
      newBestScoreVector := scoreVector
      for i := 1 to players
        if (not i = node.player) and
           (bestScoreVector[i] < scoreGuaranteeVector[i])
          then prune := true
      if (s > scoreGuaranteeVector[node.player])
        then scoreGuaranteeVector[node.player] := s
    if not prune then m := GenerateMove(node)
  newPruneCondVector[node.player] :=
    Min(newPruneCondVector[node.player],
        prevGuaranteeVector[node.player])
  isComplete := (newPruneCondVector = WorstScoreVector)
  atomic:
    node.abScoreVector[depth] := newBestScoreVector
    node.bestMove[depth] := newBestMove
    node.pruneCondVector[depth] := newPruneCondVector
    node.complete[depth] := isComplete
  return {newBestMove, newPruneCondVector}
```

Since Algorithm 4 is based on Algorithm 3 and shares much the same structure, what follows is a description of their differences: Almost all differences are concerned with caching information to avoid redundant computation in future calls to the algorithm. In addition to the best move at the node, the function also returns the weakest score guarantees necessary to achieve the same evaluation. Initially, in addition to checking if the node is a leaf node, check if the minimax evaluation is complete for the given depth, or if the input score guarantees from the calling node are subsumed by those of the previous call to the node. If so, cached information is simply returned. Otherwise, perform recursive alpha-beta search on the subtree, keeping track of weakest pruning conditions needed to avoid repeating computation. Note the variables which must now be indexed by depth of the alpha-beta call. The atomic block of assignments at the end of the function must either all be performed or not at all. This is important for the use of the function in an anytime algorithm in order to avoid leaving the graph data in an inconsistent state when the algorithm is halted before completion.

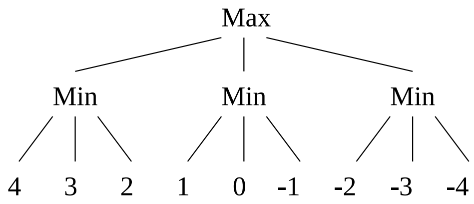

Figure 1: The importance of alpha-beta node ordering

Given that storing all information gained from all pruning conditions of all calls for a given node and depth would be prohibitive, we opt to store the most recently used weakest necessary pruning conditions. This does not pose a problem, as completely evaluated subtrees do not get re-evaluated.

Node ordering is important for alpha-beta pruning optimization. Consider the standard two-player minimax tree of Figure 1. If nodes in this figure are searched left-to-right, maximal pruning occurs with four of nine leaf nodes left unevaluated. If the nodes are instead searched right-to-left, minimal pruning occurs and all nodes are evaluated.

This example is significant to our problem domain because we're searching continuous spaces where scores may vary continuously over portions of the trajectory-sampling search tree. Therefore node-ordering may be very significant to pruning maximization. We may happily take advantage of this fact in our algorithm by having the GenerateMove function use cached information to heuristically order moves in order to maximize pruning. For instance, one might first generate the best move from previous alpha-beta searches in order to first maximize the current player score guarantee. We get this benefit without increased computational complexity.

6 Summary and Future Work

With a goal of enabling real-time hybrid system control, we developed and discussed extensions of discrete game-playing techniques. Algorithm 1, our dynamic programming method for augmented cell maps, has polynomial time and space complexity and is applicable to offline control design for low-dimensional state spaces, assuming that a good discretization can be found. Algorithm 2, graph-based minimax, has the same amortized time complexity as Algorithm 1, but with a linear increase in space complexity for caching subtree evaluations. From this increase, we gain a focus of computation which makes it suitable for online hybrid system control. Alpha-beta pruning is a form of irrelevance reasoning which increases efficiency of minimax search. Algorithm 3 is an n-player generalization of minimax search with alpha-beta pruning, which increases the depth of search but lacks the caching and approximation advantages of Algorithm 2. We finally synthesized these approaches and introduced Algorithm 4, graph-based n-player alpha-beta search with caching. This provides a more efficient means of online hybrid system control for low-dimensional state spaces, assuming that a good discretization can be found.

Two issues concerning minimax and alpha-beta motivate future research in reasoning about uncertainty and relevance in game-tree search. First, minimax search assumes no uncertainty in node evaluations, so small errors in node-evaluations may significantly misinform decisions. Second, alpha-beta pruning is concerned entirely with provable irrelevance given such an assumption. Without the ability to focus search direction according to probable *relevance* to the root decision, alpha-beta search is ill-equipped to handle large branching factors, forcing an arbitrary, pre-determined pruning or discretization (for continuous ranges of actions). Automatically choosing state-space or action-space discretizations according to the task of real-time reasoning about control is an open problem. Even given a good discretization of a hybrid system control game, a large branching factor can force an impractically shallow search and yield poor decisions.

Probabilistic game-playing methods (Russell & Wefald 1991) have been developed to handle uncertainty and to direct search with relevance to maximizing expected utility of the decision. We are currently extending these approaches for use with hybrid system control games. This still leaves overarching discretization questions concerning continuous state-spaces, ranges of actions, and decision points in intervals of time. We expect that previous work on information-based optimization (Neller 1998a) will be relevant in addressing such questions. Briefly, information-based optimization is concerned with using the information from previously sampled points to inform the choice of future sample points. Using such optimization to dynamically choose the sampling of actions and decision points should prove very interesting.

As algorithms employ increasingly computationally complex meta-level reasoning, computational overhead will grow to the point of diminishing returns in overall utility. Over time, we expect to develop a suite of methods which lie along a spectrum of computational complexities of meta-level reasoning, and describe their applicability to different classes of hybrid system control games. We hope that these will contribute to development of algorithms for real-time control and bounded rationality.

References

Başar, T., and Olsder, G. J. 1995. *Dynamic Noncooperative Game Theory, 2nd Ed.* London: Academic Press.

Bradley, E. 1995. Autonomous exploration and control of chaotic systems. *Cybernetics and Systems* 26(5):499–519.

Henzinger, T. A., and Sastry, S., eds. 1998. *LNCS 1386: Hybrid Systems: computation and control, First International Workshop, HSCC'98, Proceedings.* Berlin: Springer.

Hsu, C. 1987. *Cell to Cell Mapping; A Method of Global Analysis for Nonlinear Systems.* Springer-Verlag.

Neller, T. W. 1998a. Information-based optimization approaches to dynamical system safety verification. In Henzinger and Sastry (1998). 346–359.

Neller, T. W. 1998b. Preliminary thoughts on the application of real-time ai game-tree search to control. In *Proceedings of the IFAC Symposium on Artificial Intelligence in Real-Time Control, October 5–8, 1998, Grand Canyon National Park, Arizona, USA*. Oxford, UK: Elsevier Science.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: a modern approach.* Upper Saddle River, NJ, USA: Prentice Hall.

Russell, S., and Wefald, E. 1991. *Do the Right Thing: studies in limited rationality.* Cambridge, MA, USA: MIT Press.

Slagle, J., and Dixon, J. 1969. Experiments with some programs that search game trees. *Journal of the Association of Computing Machinery* 16(2):189–207.

Tomlin, C.; Lygeros, J.; and Sastry, S. 1998. Synthesizing controllers for nonlinear hybrid systems. In Henzinger and Sastry (1998). 360–373.

Zhao, F.; May, J. A.; and Loh, S. C. 1998. Controller synthesis and verification for nonlinear systems: a computational approach using phase-space geometric models. Submitted to IEEE Control Systems Magazine.