2013 European Control Conference (ECC)
July 17-19, 2013, Zürich, Switzerland.

Radar Resource Management: Dynamic Programming and Dynamic Finite State Machines

Jinwoo Seok, Jinxin Zhao, Jhanani Selvakumar, Edwin Sanjaya,
Pierre T. Kabamba and Anouck Girard
Department of Aerospace Engineering
University of Michigan, Ann Arbor, Michigan 48109-2140
Email:{sjinu, jinxinzh, jhanani, esanjaya, kabamba, anouck}@umich.edu

Abstract— **Finite state machines are a standard tool to model event-based control logic, and dynamic programming is a staple of optimal decision-making. We combine these approaches in the context of radar resource management for Naval surface warfare. There is a friendly (Blue) force in the open sea, equipped with one multi-function radar and multiple ships. The enemy (Red) force consists of missiles that target the Blue force's radar. The mission of the Blue force is to foil the enemy's threat by careful allocation of radar resources. Dynamically composed finite state machines are used to formalize the model of the battle space and dynamic programming is applied to our dynamic state machine model to generate an optimal policy. To achieve this in near-real-time and a changing environment, we use approximate dynamic programming methods. Example scenario illustrating the model and simulation results are presented.**

I. INTRODUCTION

We develop efficient models and algorithms for a force-level resource management system that dynamically allocates available radar resources to prosecute multiple threats in the environment. By management we mean: task assignment, scheduling and motion control, and by dynamic we mean that every management decision must account for its future effect on the environment, and hence sequences of decisions must be designed.

Our modeling assumptions are: (i) our scenario is deterministic, that is, we know the output for every input. (ii) events are discrete, i.e. events happen slowly enough relative to the speed of functioning of the radar that we can eliminate monitoring of continuous variables. This is the basis for employing logic-level models (finite state machines, or FSMs).

A. Mission Overview

Consider a geographical area containing two forces: enemy (Red) and friendly (Blue). The enemy force is composed of agents that can move and bomb freely within the given area. The friendly force is composed of platforms (such as ships) and radars that can detect the enemy agents, can be deployed at various locations, and can communicate. Considering the heterogeneity of entities within the geographical area, and the possible interactions, the method of modeling should be chosen judiciously to describe the situation, be amenable to formal analysis, and be computationally tractable.

Logic-level models, such as FSMs, retain only features relevant to mission analysis, can be formally analyzed and are computationally tractable. A FSM describes the world with a set of discrete states and a set of events that denote the transitions between the states. Dynamic finite state machines (DyFSMs) allow the state space of an FSM to change dynamically as required for military missions which involve environmental changes, such as in the number of threats or number of remaining resources.

The enemy force attempts to bomb the platforms and the radars of the friendly force. The friendly force's resources (radars and ships) must be managed to maximize threat detection and to minimize the probability of losing resources. The friendly force seeks a set of optimal heterogeneous decisions to achieve the mission objectives, and such decisions should accommodate changes in the environment. Dynamic programming (DP) provides a policy leading to a sequence of decisions that are optimal with respect to some objective function based on Bellman's principle of optimality.

Although DP is powerful enough to provide optimal policies, it suffers from the so-called curse of dimensionality: it requires evaluating the optimal policy *everywhere* in the state-space. Clearly, this requirement is computationally prohibitive. To overcome this difficulty, we utilize two methods of approximate dynamic programming (ADP): value and policy iteration.

Hence, in this paper, we present an efficient modeling strategy and algorithm for dynamic force-level resource management. A system in which this algorithm is implemented will assist humans in making optimal decisions in the face of the many scenarios that battle can bring.

B. Literature Review

Various approaches to modeling, control, and decision problems in battlespaces are reviewed in this section. Time-based State Space models are used to describe a stochastic system in [1], [2] and [3]. The use of discrete logic-based modeling for adversarial situations is seen in [4] and [5] (using finite state machines), and [6] (using hybrid automata). In [7] a finite state automaton is used to represent a discrete decision process.

The interactions between opposing forces in the battlespace are modeled with the tools discussed below. Differ-

ential games are a common method used where the system is modeled with a state space. Differential games usually lead to computational intractability, that can be alleviated through heuristics. However, this may come at the cost of sub-optimality of the resulting policy [2]. A different strategy, used in discrete event systems, relies on supervisory control of the states, wherein an undesirable state is avoided by taking a preventive action [8].

Control is executed in two standard ways: optimal control and dynamic programming. In a battle space resource management problem, dynamic programming comes across as the favored tool since it produces an optimal policy everywhere in the state space ([7] and [9]). In [6], stochastic dynamic programming is used to compute optimal configurations of a defensive UAV team. In [2], [5], and [3], various approximate dynamic programming methods are used to obtain fast solutions as required in a tactically changing environment. In [2], a heuristic is developed to decouple the control commands and to reduce computational burden. Reference [5] introduces Hierarchically Accelerated Dynamic Programming (HADP), an approximation method used to reduce computation time for dynamic programming for finite state machines, at the possible cost of sub-optimality. In [3], the state space is sampled for computational tractability, and the reward function is approximated. A least square estimate and a rollout extraction method are used to solve for the optimal policy.

As the above discussion indicates, the problems of exact and approximate dynamic programming for dynamic finite state machines (i.e., state machines where the state space may change) are still open. This paper addresses these problems and illustrates their solutions through simulations. All models in this paper use only deterministic FSMs with transition costs. Specifically, the paper provides original contributions listed below.

C. Original Contributions

The original contributions of this work are :
(i) Application of Dynamic Programming to Dynamic Finite State Machines to obtain an optimal policy,
(ii) Application of Value Iteration to Dynamic Finite State Machines, and
(iii) Application of Policy Iteration to Dynamic Finite State Machines,
where (ii) and (iii) are Approximate Dynamic Programming methods.

A proof of the optimality of Dynamic Programming applied to Dynamic Finite State Machines is available in [10]. Our simulation results show that the approximate dynamic programming methods provide policies that are close to the optimal policies from dynamic programming.

D. Paper Outline

The organization of this paper is as follows. In Sec. II we present background information used in our study, relating to FSMs, DP and ADP. In Sec. III we describe how we formulate the problem, model the scenario, and compose the FSM. In Sec. IV we present our simulation and preliminary results. Finally, in Sec. V we present our conclusions and future work.

II. BACKGROUND

A. Finite State Machine

To describe the battle situation, we use an FSM that constructs the output signal one symbol at a time by observing the input signal one symbol at a time [11], [9], [12]. An FSM is a six-tuple:

$$FSM = (X, U, Y, f, g, x_0), \tag{1}$$

where X, U, and Y are sets, f and g are functions, and $x_0 \in X$. The meanings of these symbols are as follows:

1. X is the state space,
2. U is the input alphabet,
3. Y is the output alphabet,
4. $x_0 \in X$ is the initial state,
5. f: $X \times U \rightarrow X$ is the next state function,
6. g: $X \times U \rightarrow Y$ is the output function.

The interpretation of f and g is as follows: if $x(k) \in X$ is the current state at stage k, and $u(k) \in U$ is the current input signal, then the current output symbol $y(k)$ and the next state $x(k+1)$ are given by:

$$x(k+1) = f(x(k), u(k)), \tag{2}$$

$$y(k) = g(x(k), u(k)), \tag{3}$$

and $x(0)$ is x_0.

To represent an FSM, we can use sets and functions models as given above, or state transition diagrams, or state transition tables.

B. Finite State Machine Composition

We consider an FSM comprising interconnected components for our example mission: radars, platforms and bombers. Each component is itself modeled as an FSM. There may be more than one component of each type.

We obtain the composite FSM for the components using a composition operation called synchrony. In this operation, all machines in the composition react simultaneously and instantaneously. Synchrony and its composition properties are well studied in computer science; see for example [13], [14], [15], [16], [17]. Transitions in the composite machine are a set of simultaneous transitions of the component state machines. Our current model allows side-by-side and cascade synchronous compositions of FSMs. We may consider feedback compositions of state machines in the future if required by applications. Some subtleties appear in feedback compositions due to the synchronous nature of such compositions, so they will be used sparingly if needed at all.

C. Dynamic Programming

Given the state transition mapping,

$$x(k+1) = f(x(k), u(k)), \tag{4}$$

let $x \in X$ be the state, k be the stage, $u \in U$ be the decision at stage k and state x, and let f be the next state function.
The objective function is:

$$J(x(0), u(0), ..., u(K-1)) = \sum_{k=0}^{K-1} \Phi(x(k), u(k)), \tag{5}$$

where Φ is the transition cost and K is the total number of stages. Dynamic Programming is based on solving the stationary Hamilton-Jacobi Bellman (HJB) Equation:

$$J^*(x(k)) = min_{u(k)\in U}\{\Phi(x(k), u(k)) + J^*(f(x(k), u(k)))\}, \tag{6}$$

where J^* is the optimal cost to-go from stage k and state x. The optimal decision is then,

$$u^*(x(k)) = argmin_{u(k)\in U}\{\Phi(x(k), u(k)) + J^*(x(k+1))\}, \tag{7}$$

From (4) and (6),

$$J^*(x(k)) = min_{u(k)\in U}\{\Phi(x(k), u(k)) + J^*(x(k+1))\}, \tag{8}$$

Equation (8) illustrates that Dynamic Programming proceeds backwards with respect to stages, i.e., the optimal cost and decision at stage k depend on the optimal cost and decision at stage $(k+1)$. By solving DP for every stage k, we obtain a sequence of optimal decisions (u^*) according to Φ, and this is the optimal policy (π^*) [18] [19]. Note that increase in cardinality of the state space increases the computational time, leading to the so-called 'Curse of Dimensionality'. This calls for approximations to be performed on large-scale models.

D. Approximation

The exact DP algorithm suffers from the curse of dimensionality: as the cardinality of the state space increases, the computational time increases non-linearly. This is a serious drawback when solving for optimal policies in near real time (NRT). Hence approximations of the actual DP algorithm are needed [20].

1) Value Iteration (VI): The solution to the Bellman equation is obtained by iteration, starting from an initial guess for the optimal cost-to-go (J^*). We start with an arbitrary initial function of a state variable J, and successively compute $T(J)$, $T^2(J)$ and so on where

$$(T(J))(x) = min_{\pi\in\Pi}[\Phi(x, \pi(x)) + \gamma J(f(x, \pi(x)))], \tag{9}$$

where $\pi \in \Pi$ is a policy, and Π is the policy space.

Theoretically, the left hand side values converge to $J^*(x)$. The number of iterations required for the corresponding policy to become stationary is however, finite. It is also observed that VI does not yield convergence of optimal values (in fact the values increase linearly) while the policy itself quickly settles to a near-optimal policy.

2) Policy Iteration (PI): The PI method is similar to the VI method. The policy (π) is initialized and then $T(J)$, $T^2(J)$ and so on are successively found with the policy being improved in each complete iteration. This method generates a sequence of stationary policies each with improved cost over the previous one. In this method there are two steps involved:

- At the mth iteration, evaluate the function $J_m = T^m(J)$ with the current policy π_m
- Based on the evaluated cost function, find the improved policy for this iteration

$$\pi_{m+1} = argmin_{\pi\in\Pi}\{\Phi(x, \pi(x)) + J_m(f(x, \pi(x)))\}. \tag{10}$$

III. TECHNICAL APPROACH

A. Problem Formulation

There are Blue (friendly) forces in the open sea, equipped with one multi-function radar. The Red forces (enemy) consist of missiles that target the Blue forces radar. The mission of the Blue force is to foil the enemies threat by careful allocation of radar resources.

The problem is formulated as follows. One blue force ship is located on the center of a Manhattan grid. The ship has z units length of radar range, v units length of interception range, and does not move. Thus, the number of states will be $q \times p \times r^t$ where, q is the number of ship's states, p is the number of radar's states, r is the number of target's states, and t is the number of detected targets. There are d red force missiles approaching the ship from various ranges and directions, and with different speeds. After the missiles get into the radar range, the ship can predict their trajectory exactly. Given this information, we want to find the policy that engages the incoming missiles the earliest. We use DP, VI, and PI, to find and approximate the optimal policy for the blue ship.

1) Manhattan Grid: An object in a Manhattan grid only can move along the horizontal and vertical paths between two points in the grid. We restrict the movement of the red missiles to the Manhattan grid to simplify the problem.

2) Radar and Interception range: The radar range is the maximum range in which an object can be detected by the radar. The interception range is the maximum range in which the ship can intercept the missile. The blue ship can detect the red missiles only when they are in the radar range, and intercept them only in the interception range.

B. Modeling

The battle state machine is obtained by composing FSMs for ships, radars and targets. There are nine possible types of inputs to the battle state machine, of which four are uncontrollable, and five are controllable. The four uncontrollable inputs, labeled 1 through 4 are: 1) $no\ alert$, 2) $alert1$,

3) $alert2$, and 4) $bombed$. The five controllable inputs, labeled 5 through 9, are: 5) $scanning$, 6) $get\ information$, 7) $tracking$, 8) $engage$, and 9) $hit/kill$. Both the radar and the target sub-components are sources of uncontrollable inputs. $alert1$, $alert2$, and $no\ alert$ are from the radar, and $bombed$ comes from targets. $scanning$ searches the area within the radar range to detect enemies. $get\ information$ collects information on detected targets. The radar collects information on the targets such as speed, type of target, and predicted trajectory. $tracking$ is the input that tracks the target after sufficient information has been obtained. $engage$ sets the target for interception and has to be done before $hit/kill$. $hit/kill$ is the input that starts interception. $alert1$ indicates that the radar has detected a target. $alert2$ indicates that the target has entered the interception range. $no\ alert$ is the input that indicates no target is detected from the radar. Finally, $bombed$ indicates that the ship is bombed. Note that there is one incidence of $alert2$ and $hit/kill$ inputs for each detected enemy, so the total number of inputs can be greater than nine.

There are nine possible types of outputs, that are: $no\ alert$, $alert1$, $alert2$, $information$ $(info)$, $number\ of\ target$ (num), $engage$ (eng), $kill$, $dead$, and $empty$. $no\ alert$, $alert1$, and $alert2$ can be both input and output for the radar FSM. $info$ is the output that denotes that information on the target has been acquired. num tells how many targets have been detected in the radar range. eng indicates which target has been engaged. $kill$ is issued a target is killed and $dead$ indicates that the ship is dead. Finally, $empty$ means there is no output with respect to the inputs.

1) Ship: The ship FSM has three states: $SAFE$, $UNSAFE$, and $DEAD$. Figure 1 shows the FSM state transition diagram. The six-tuple description of the FSM for the ship is as follows:

1. X = $\{SAFE, UNSAFE, DEAD\}$
2. U = $\{alert1, alert2, no\ alert, bombed\}$
3. Y = $\{dead, empty\}$
4. $x_0 \in States$ = $SAFE$
5. f: $X \times U \to X$
6. g: $X \times U \to Y$

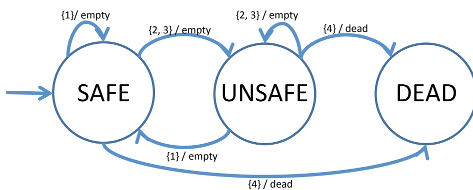

Fig. 1. Modeling of Ship.

2) Radar: The radar FSM has five states: $IDLE$, $DETECTED$, $TRACKING$, $ENGAGING$, and $BATTLE$. Figure 2 shows the FSM state transition diagram.

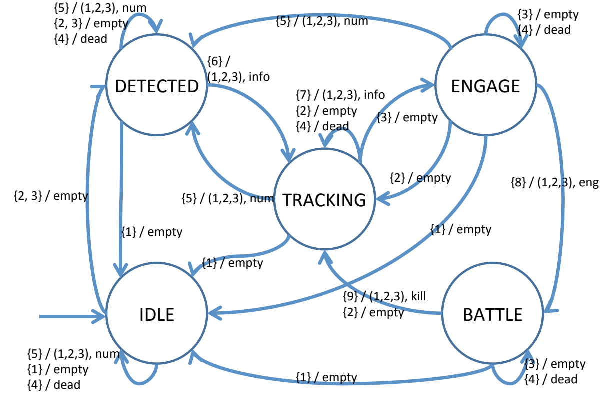

Fig. 2. Modeling of Radar.

3) Target: The target FSM has three states: $ALIVE$, $ENGAGEABLE$ and $KILLED$. Figure 3 shows the FSM state transition diagram.

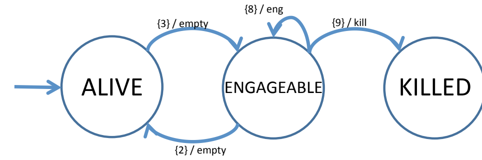

Fig. 3. Modeling of Target.

C. Composition

We compose the individual FSMs using cascade composition as described in Sec. II-B. For the example of a composition of one ship and one radar, there are 15 states for the composed FSM, which is the number of states of the ship FSM times the number of states of the radar FSM (3×5).

D. Dynamic Programming for DyFSM

DyFSMs allow the state space of an FSM to change dynamically. For example, in our scenario, when the blue radar detects a new enemy, a new target FSM is created and composed with the current battle FSM. Each transition of the battle FSM is assigned a cost by a heuristic. DP is then be applied to the FSM to find the optimal policy for each state using equation (7) in Sec. II-C.

IV. SIMULATIONS & RESULTS

We set one specific situation for the simulation to explain the behavior of the policy, the DyFSM method and the difference in computation times between DP and ADP. We then run 200 simulations to verify the results.

We use a 111x111 Manhattan grid with values of $z = 50$, $v = 35$, $q = 3$, $p = 5$, $r = 2$, and $d = 16$, as defined in Sec. III-A. Initially, the 16 Red missiles are randomly distributed outside the radar detection range, and are assigned different, but constant speeds.

All missiles head towards the ship along the Manhattan grid once the simulation begins. Figure 4 shows the initial configuration of the simulation.

Every simulation is performed on the following environment: A 2.66 GHz intel core2 Duo® processor, and 4GB 1067 MHz DDR3 memory.

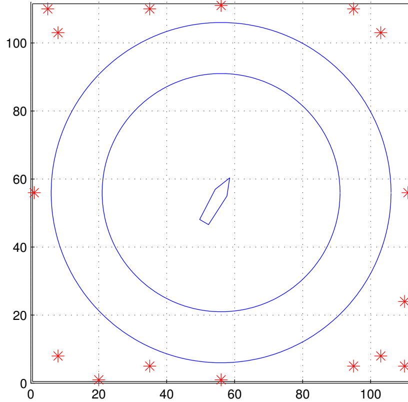

Fig. 4. Scenario Configuration: Blue ship located at center of grid. Inner circle indicates interception range, outer circle indicates radar range, and red stars indicate missiles.

A total of three cases are studied: DP, VI, and PI. As explained in Sec. III-B, the total number of inputs increases with the total number of detected targets, and this affects computation time.

For DP, we vary the number of stages, K, based on the total number of states. We use a larger K for a larger number of states because, if the number of states increases, more stages are needed to obtain a stationary optimal policy.

The three methods produce exactly the same policy and in all three cases, all sixteen targets are successfully destroyed using this policy.

(i) a controllable input determined by the policy,
(ii) an uncontrollable input (alert) which is the result of enemy behavior,
(iii) an output that is either num, $info$, eng, $kill$, or $dead$.

Table I shows the computation time at seven different stages in the course of the simulation. The first column contains the stages at which re-composition and policy calculation occur, and indicates the total number of states and number of detected targets in each stage. The second column indicates the operation - composition or policy calculation. The three remaining columns show the time taken for composition and calculation in each of the three cases (DP, VI and PI). When a new target arrives in radar range, the state space expands to incorporate it. This necessitates

TABLE I

SIMULATION RESULTS COMPARISON: numS is number of states, numT is number of detected targets, Comp is composition, Cal is calculation and K is number of stages for DP.

Stage (numS / numT)	Type (K)	Time [ms]		
		DP	**VI**	**PI**
1	Comp	19	18	19
(15 / 0)	Cal(20)	1	21	9
6	Comp	81	63	64
(60 / 2)	Cal(100)	3	3	3
7	Comp	725	725	729
(7,680 / 9)	Cal(100)	766	504	661
9	Comp	1594	1620	1616
(15,360 / 10)	Cal(200)	3331	1048	1389
14	Comp	3843	3518	3558
(30,720 / 11)	Cal(200)	7295	2169	2869
15	Comp	38090	38546	38417
(245,760 / 14)	Cal(300)	248561	19374	25797
44	Comp	6	9	10
(15 / 0)	Cal(20)	180	28	24

composition of a new FSM and calculation of a new optimal policy. During the time between detection of two targets, the radar implements the most recent policy.

For example, in stage one the FSM has 15 states and no target has been detected. At stage five, the radar detects two targets. At stage six, a new battle FSM is composed comprising 60 states and two detected targets. For the DP case, it takes 81 [ms] to compose the FSM and 3 [ms] to find the optimal policy. The next composition and calculation occur immediately at stage seven, as seen in table I. There are nine detected targets in the radar range, and a total of 7,680 states. It takes 725 [ms] to compose the FSM and DP takes 766 [ms] to find the optimal policy.

The composition time is nearly the same for all three cases. The differences reflect the accuracy of measurement. Comparing the calculation times, we can observe that with a small number of states, the difference between DP and ADP cases is less that 0.1 seconds, which is an inconclusive result. However, when the number of states is larger, the ADP methods are faster than DP. For example, with 245,760 states, DP is by far the slowest, with a computation time of 248,561 [ms], compared to VI, which takes 19,374 [ms], and PI, which takes 25,797 [ms]. Since all three methods generate the same optimal policy based on the same objective function, VI is the best method to solve this problem, because it is the fastest.

For the scenario described above, there exist two trivial failure cases. First, if a target has infinite speed, the radar cannot handle it. Specifically, if the target's speed is greater than 16, the radar will be bombed before it kills the target. Second, if there are infinitely many targets each with unit speed, the radar cannot handle them. Specifically, if the number of targets is greater than 18, the radar also will be bombed before it kills all the targets. Considering these two trivial cases, to verify that PI, VI and DP always yield the same policy, and to compare performance with other methods, we run 200 simulations. For these simulations, the

scenario geometry is the same, but the targets differ in initial positions and speeds. For each simulation, the number of targets is seven. Target initial positions are set randomly somewhere outside of the radar range and target speeds are also set randomly between one and six. For all 200 simulations we obtain exactly the same policy from the PI, VI, and DP methods. We compare the policy with two greedy methods; one is a 'fastest first' method and the other is a 'nearest first' method. The performance in comparison with other methods is shown in table II.

TABLE II
PERFORMANCE COMPARISON

	Complete Mission (%)	Total Kill (%)
DyFSM	152 / 200 (70%)	1176 / 1400 (84%)
Fastest first	129 / 200 (64.5%)	1072 / 1400 (76.57%)
Nearest first	125 / 200 (62.5%)	1093 / 1400 (78.07%)

Our method using DyFSM has better performance than both greedy methods. First, of a total of 200 missions, DyFSM completed 152 missions (70%), while 'fastest first' and 'nearest first' completed 129 missions (64.5%) and 125 missions (62.5%) respectively. A complete mission is one where all targets are killed and the radar is not bombed by the targets. In addition, DyFSM kills the most targets: a total of 1,400 (seven targets for each of the 200 simulations). Thus, we can conclude that our DyFSM method yields a well-behaved policy for the radar.

V. CONCLUSIONS & FUTURE WORK

In this paper, we have developed efficient models and algorithms for a force-level resource management system that dynamically allocates available radar resources to prosecute multiple threats in the environment. We used dynamic finite state machines to model components in a battlespace including ships, radars, and targets. Dynamic finite state machines are important in modeling battlespaces, since they allow the state space of the world's finite state machine to change dynamically as required for military missions. The control algorithms are based on approximate dynamic programming methods (value iteration and policy iteration), since dynamic programming suffers from the curse of dimensionality. Value iteration and policy iteration give sub-optimal policies in near real-time for large scale problems, which is required in combat situations. To illustrate the performance of dynamic finite state machines, dynamic programming, and approximate dynamic programming, we simulated a naval battle scenario.

We plan to incorporate a variety of improvements in the future. First, we will use a more sophisticated radar model. Stochastic extensions, such as modeling with Markov Decision Process or Partially Observable Markov Decision Process are possible, assuming appropriate data becomes available. Because there is also significant uncertainty in the transition costs, Robust Dynamic Programming may also be considered. Second, we will conduct a sensitivity analysis of optimal policies with respect to cost functions. Third, we will allow a more sophisticated enemy model. Fourth, we will increase the number of ships in the friendly Blue force and determine how their communication and cooperation affect the optimal policies. If needed, more efficient approximate dynamic programming heuristics may be considered in this step.

VI. ACKNOWLEDGMENTS

The authors thank Philippe Kirschen for constructive help with editing.

This research is supported by the Office of Naval Research, U.S Naval Surface Warfare Center, under grant number BAA N00178-12-C-2003.

REFERENCES

[1] J. B. Cruz, Jr. M. A. Simaan, A. Gacic, H. Jiang, B. Letellier, M. Li, and Y. Liu. Modeling and control of military operations against adversarial control. In *in Proceedings of the 39th IEEE Conference on Decision and Control*, volume 3, pages 2581–2586, 2000.

[2] M. Faied, I. Assanein, and A. Girard. Uavs dynamic mission management in adversarial environments. *International Journal of Aerospace Engineering*, pages 1–10, 2009.

[3] J. S. McGrew, J. P. How, L. A. Bush, B. Williams, and N. Roy. Air-combat strategy using approximate dynamic programming. *Journal of Guidance, Control, and Dynamics*, 33:1641–1654, 2010.

[4] G. A. McIntyre R. RHill, J. O. Miller. Application of discrete event simulation to modeling military problems. In *in Proceedings of the Winter Simulation Conference*, volume 1, pages 780–788, 2001.

[5] G. Shen and P. E. Caines. Hierarchically accelerated dynammic programming for finite-state machines. *IEEE Transaction on Automatic Control*, 47(2):271–283, Feb 2002.

[6] M. Faied and A. Girard. Modeling and optimization of military air operations. In *in Proceedings of the 48th IEEE Conference on Decision and Control*, pages 6274–6279, Dec 2009.

[7] R. M. Karp and M. Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, May 1967.

[8] P. J. Ramadge and W. H. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 1:206–230, 1987.

[9] C.G Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2007.

[10] J. Selvakumar. Dynamic programming for dynamic finite state machines. *ARCLab Technical Report No.2, University of Michigan*, 2013.

[11] S. S. Epp. *Discrete Mathematics with Applications*. Cengage Learning, 4th edition, 2010.

[12] N. A. Lynch and M. R. Tuttled. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[13] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, pages 433–440, 2003.

[14] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, pages 389–448, Jul 1984.

[15] S. A. Edwards, N. Halbwachs, R. v. Hanxleden, and T. Stauner. Synchronous programming. In *Synchronous Programming*, Novl 2005.

[16] F. Maraninchi and N. Halbwachs. Compositional semantics of non-deterministic synchronous languages. In *Programming Languages and System*, pages 235–249, Apr 1996.

[17] F. Maraninchi and Yann Remond. Argos: an automaton-basedsynchronous language. *Computer Languages*, 27:61–92, Apr 2001.

[18] R. R. Weber. Optimization and control. University of Cambridge, 1999.

[19] D. P. Bertsekas. *Dynamic Programming and Optimal Control Volume I*. Athena Scientific, 3rd edition, 2005.

[20] D. P. Bertsekas. *Dynamic Programming and Optimal Control Volume II*. Athena Scientific, 4th edition, 2012.