To appear in Computer Generated Forces and Behavioral Representation: 1994

Event Tracking in Complex Multi-agent Environments

Milind Tambe and Paul S. Rosenbloom
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
email: {tambe, rosenbloom}@isi.edu

Abstract

The Soar-IFOR project is aimed at developing intelligent automated pilots for simulated tactical air-combat. One key requirement for an automated pilot in this environment is *event tracking*: the ability to monitor or track events instigated by opponents, so as to respond to them appropriately. These events include the opponents' low level actions, which the automated pilot may directly observe, as well as opponents' high level plans and actions, which the automated pilot can not observe (but only infer). This paper analyzes the challenges that an automated pilots must face when tracking events in this environment. This analysis reveals some novel constraints on event tracking that arise from the dynamic multi-agent interactions in this environment. In previous work on event tracking, which is primarily based on single-agent environments, these constraints have not been addressed. This paper proposes one solution for event tracking that appears better suited for addressing these constraints. The solution is demonstrated via a simple re-implementation of an existing automated pilot agent for air-combat simulation[1].

1. Introduction

The Soar-IFOR project is aimed at developing intelligent automated pilots for simulated tactical air-combat environments [11, 17]. These automated pilots are intended to participate in large-scale exercises with a variety of human participants, including human fighter pilots. These exercises are to be used for training as well as for development of tactics. To participate in such exercises, the automated pilots must act in a realistic manner, i.e., like trained human pilots. Otherwise, both the training and tactics development in these environments will not be realistic.

To act in a realistic manner, an automated pilot must, among other things, be responsive to events in its environment — it must modify and adapt its own maneuvers in response to relevant events. These events may correspond to simple actions of other pilots, such as changes in heading or altitude, which the automated pilot may directly observe on its radar. Alternatively, these events may involve the execution of complex, high-level actions or plans of other pilots, which the automated pilot can not directly observe. For instance, one crucial event is an opponent's firing a missile at an automated pilot's aircraft, threatening its very survival. Yet, the automated pilot cannot directly see the missile until it is too late to evade it. Fortunately, the automated pilot can monitor the opponent's sequence of maneuvers, and infer the possibility of a missile firing based on them, as shown in Figure 1. The automated pilot is in the dark-shaded aircraft, and its opponent is in the light-shaded one.

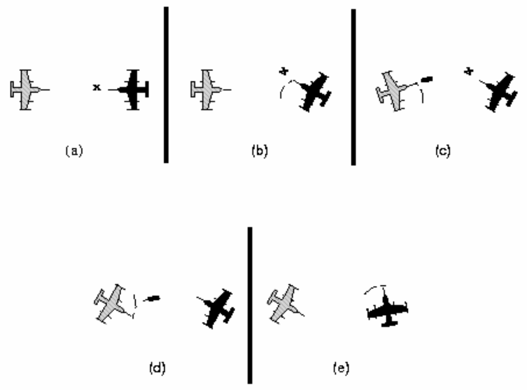

Figure 1: Manuevers of the automated pilot (in dark-shaded aircraft) and its opponent (in light-shaded one).

Suppose that initially the two aircraft are headed right toward each other as shown in Figure 1-a. The range (distance) between the two aircraft is

[1]This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan, as part of contract N00014-92-K-2015 from the Advanced Systems Technology Office (ASTO) of the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). Critical support has been provided by Dennis McBride of ARPA/ASTO; and Tom Brandt, Bob Richards, and Ed Harvey of BMH Inc.

more than 10-15 miles, so they can only see each other on radar. This range is slightly short of the range from which the opponent can fire a radar-guided missile at the automated pilot's aircraft. However, the opponent is already well-positioned to fire this missile once its range is reached. In particular, given that the two aircraft are pointing right at each other, the opponent's aircraft is at *attack heading* (a point slightly in front of the automated agent's aircraft, as shown by a small **x** in the figure). At this juncture, the automated pilot turns its aircraft as shown in Figure 1-b. Given that the opponent wants to fire a missile, she turns her aircraft in response to re-orient it to attack heading (Figure 1-c). In this situation, she reaches her missile firing range, and fires a missile (shown by -). While the automated agent cannot observe this missile, based on the opponent's turn it can infer that the opponent may be attempting to achieve attack heading as part of her missile firing behavior. Unfortunately, at this point, it cannot be certain about the opponent's missile firing, at least not to an extent where trained fighter pilots would infer a missile firing. However, if the opponent subsequently engages in an *Fpole* maneuver then that considerably increases the likelihood of a missile firing (Figure 1-d). This maneuver involves a 25-50 degree turn away from the attack heading (it is executed after firing a missile to provide radar guidance to the missile, while reducing the closure between the two aircraft). While at this point the opponent's missile firing is still not an absolute certainty, its likelihood is high enough, so that trained fighter pilots assume the worst, and react as though a missile has actually been fired. The automated pilot reacts in a similar manner, by engaging in a missile-evasion maneuver. This involves turning the aircraft roughly perpendicular to the missile-flight (Figure 1-e), which causes the aircraft to "drop-off" (become invisible to) the opponent's radar. Deprived of radar guidance, the opponent's missile is rendered harmless.

The above example illustrates that an automated pilot needs to continually monitor a variety of events in its environment, such as the opponent's turns and her (inferred) missile-firing behavior, so as to react to them appropriately. We refer to this capability as *event tracking*. Here, an event may be considered as any coherent activity over an interval of time. An event is similar to a *process* in qualitative process theory [8], as something that acts through time to change the parameters of objects in a situation. This event may be a low-level action, such as an agent's Fpole turn, or it may be a high-level behavior, such as its missile-firing behavior, which consists of a sequence of such turns. The event may be internal to an agent, such as maintaining a goal or executing a plan, or external to it, such as executing an action. The event may be instigated by any of the agents in the environment, including the agent tracking the events, or by none of them (e.g., a lightning bolt). The event may be observed by an agent, perhaps on radar, or it may be unobserved, but inferred. *Tracking* any one of these events refers to recording it in memory and monitoring its progress as long as necessary to take appropriate action in response to it. Tracking an event also includes the ability to infer the occurrence of that event from other events.

Event tracking is closely related to the problem of plan recognition [12], the process of inferring an agent's plan based on observations of the agent's actions. The term event tracking is preferred in this investigation, since it also involves events other than plans, and since it is a continuous on-going activity. However, more important than the terminology, of course, is gaining a better understanding of the nature of this capability. In particular, does the realistic multi-agent setting of air-combat simulation reveal anything new about event tracking? Given the complexity of this domain, answering this question in its entirety is beyond the scope of this single investigation. However, this paper takes a first step by focusing on events relating to the actions and behaviors of one or two opponents as they confront the automated pilot. Section 2 illustrates that even within this restricted context, the air-combat domain brings forth some novel constraints on event tracking. Following this, Section 3 presents one approach that we have been investigating to address these constraints. The key idea in this solution is a basic shift in the agent's reasoning framework: from the usual agent-centric to world-centric. Finally, Section 4 presents a summary and issues for future work.

2. Event Tracking in Air-Combat Simulation

The primary constraint on event tracking in air-combat simulation arises from the fact that this is a dynamic environment, where agents continually interact. This continuous interaction implies that the agents cannot rigidly commit to performing a fixed sequence of actions. Instead, they need high behavioral flexibility and reactivity in order to achieve their goals. For instance, in Figure 1-c, the opponent has to re-orient herself to a new attack heading in response to the automated pilot's turn in

Figure 1-b. If the automated pilot had turned in the opposite direction, so would have the opponent. A more complex interaction occurs in Figure 1-e, where the automated pilot's missile evasion maneuver is a response to the opponent's overall maneuvers in Figures 1-c and 1-d, which are identified as part of her missile firing behavior.

These types of agent interactions extend well beyond situations involving just two aircraft. For instance, consider a situation where there are two opponents attacking the automated pilot's aircraft, as shown in Figure 2-a. Again, the automated pilot is in the dark-shaded aircraft, and the opponents are in the light-shaded aircraft. These opponents may either closely co-ordinate their attack or they may attack independently. One method of close co-ordination in the opponent's attack is shown in Figure 2-b. Here, the opponent closer to the automated pilot's aircraft (the *lead*) leads the attack, while the second opponent, marked with **x** (the *wingman*) just stays close to the lead, and follows her commands. Thus, as the lead turns to gain positional advantage, the wingman needs to turn in that direction as well, so as to fly in formation with the lead, all the while making sure that she does not get in between the lead and the automated pilot's aircraft. Another method of close co-ordination is shown in Figure 2-c. Here, the opponents execute a coordinated *pincer* maneuver — as the lead turns in one direction, the wingman turns in the opposite direction, so as to confuse the automated pilot and attack it from two sides. There are other possibilities of co-ordinating the attack as well. Of course, the opponents may not co-ordinate their attack. They may instead try to gain positional advantage in the combat independently of each other, and attack independently. In all these situations, all three aircraft continually influence each other's actions and behaviors in different ways. If other aircraft are involved in the combat — for instance, if the automated pilot is coordinating its attack with a friendly aircraft — then they also interact with the other aircraft involved in the combat.

This dynamic interaction among the agents leads to the primary constraint on event tracking in this domain: an agent must be able to track highly flexible and reactive behaviors of its opponent. In so doing, the agent must take the appropriate agent interaction into account. Without an understanding of this interaction, an opponent's action may lead to unuseful or even misleading interpretation. For instance, the opponent's turn in Figure 1-c needs to be tracked as a response to the automated pilot's own turn in Figure 1-b. Otherwise, that turn may appear meaningless. Similarly, as shown in Figure

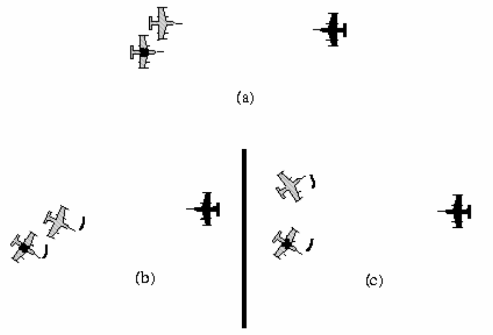

Figure 2: Agent interactions: (a) two opponents attacking the automated pilot's aircraft; (b) opponents stay close; (c) opponents stage a co-ordinated "pincer".

2, the wingman may mainly be reacting to its lead's turns, or she may be reacting to the automated pilot's aircraft independently. Understanding this interaction is important in tracking the wingman's actions.

A second related constraint here is that event-tracking must occur in real-time and must not hinder an agent from acting in real-time. For instance, in Figure 1, if the automated pilot does not track the missile firing event in real-time or does not react to it in real-time, the results could be fatal.

The third constraint on event tracking is that agents must be able to expect the occurrence of unseen, but on-going events. This constraint arises from the weakness of the sensors in this domain — an agent must sometimes track opponent's actions even though they are not visible on radar. For instance, suppose in the situation in Figure 2-c, the automated pilot concentrates its attack on the lead, and as a result the wingman (marked with **x**) drops off the automated pilot's radar. Here, given that the opponents are inferred to be executing a pincer maneuver, even though the wingman drops off the radar, some expectation about her position can be developed. Thus, the automated pilot can re-orient its radar and reset its mode to re-establish radar contact with the wingman if there is a need to do so later during the combat.

The fourth and final constraint on event tracking is that it is not a one-shot recognition task. Instead, it occurs on a continual basis, at least as long as it is relevant to the agent's achievement of its goals (such as the completion of its mission).

Thus, this domain poses a challenging combination of constraints for event tracking. The most novel constraint here is the first one. In previous investigations in the related areas of plan/situation recognition [12, 16, 6, 18, 3] — including one investigation focused on plan recognition in airborne tactical decision making [2] — this constraint has not been addressed. In particular, plan recognition models have not been applied in such dynamic, interactive multi-agent situations, and hence do not address strong interactions among agents and the resulting flexibility and reactivity in agent behaviors. In particular, these models assume that a single planning agent (or multiple independent planning agents) has some plans, and a recognizing agent recognizes these plans. The planning agent may be either actively cooperative (it intends for its plans to be recognized by the recognizing agent) or passive (it is unconcerned about its plans being recognized) [4]. The recognizing agent's job is to recognize these plans and possibly provide a helpful response. However, neither the recognizing agent, nor any other agents in the environment are assumed to have any influence on these plans. Consequently, these plan recognition models can rely on pre-compiled *plan libraries*, where each plan lists the sequence of events and the temporal relationships among the events [16]. However, such lists cannot be employed in tracking highly flexible and reactive agent behaviors. In particular, all possible variations on agent behaviors would need to be included in such lists, leading to a combinatorial explosion in the number of plans (unless a highly expressive plan language is developed).

Grosz and Sidner [9], in their work on discourse situations, attempt to partly address the above constraint on event tracking. They focus on what they characterize as the "master-slave" relationship between the planning agent and the recognizing agent assumed in plan-recognition models, and attempt to remedy it by using *shared plans*. Agents in their discourse situations arrive at a shared plan by establishing mutual beliefs and intentions about things such as their role in executing the plan. However, their discourse situations involve agents that are actively cooperative, while agents in air-combat simulation range from actively co-operative to passive to actively un-cooperative.

Interestingly, while plan-recognition systems have not dealt with such dynamic multi-agent situations, Distributed AI (DAI) systems, which have dealt with such situations, have not addressed the problem of plan recognition. There is some work in DAI on understanding other agents' plans [7]. However, it focuses on agents exchanging their plan data structures for active cooperation, rather than on plan recognition. Thus, the first constraint actually appears to give rise to a novel issue intersecting the areas of plan-recognition and DAI.

The remaining three constraints on event tracking — real-time performance, expectations and continuous tracking — have been addressed in previous research (e.g., in [6]). The next section presents an approach that we have been investigating for event tracking that addresses all four constraints outlined above.

3. Towards a Solution for Event Tracking

The key idea in the proposed solution for event tracking is based on the following observation. All of the agents in this environment possess similar types of knowledge, they have similar goals, and similar levels of flexibility and reactivity in their behaviors. In particular, an automated pilot agent that requires the capability to track events shares these similarities with its opponent. Thus, the key idea is that all the knowledge and implementation level mechanisms that the automated pilot agent uses in generating its own flexible behaviors may be used in service of tracking flexible behaviors of other agents.

To understand this idea in detail, it is first useful to understand how an agent generates its own flexible and reactive behaviors. Section 3.1 explains this by focusing on an automated pilot agent A_o and its flexibility and reactivity. Section 3.2 then illustrates how A_o may exploit this for tracking other agent's behaviors. Section 3.3 outlines the issues that arise in such an endeavor. Finally, Section 3.4 presents a simple re-implementation of an existing pilot agent based on the ideas presented in this section.

Note that while the solution presented here originated with the observation of similarity among agents, it is not necessarily limited to only those situations. For instance, it is possible that even though the other pilot agents may possess similar levels of flexibility and reactivity, they may be constrained in their behavior by their doctrine. To track these types of constrained behaviors, A_o would need to use similar types of doctrine-based constraints in tracking behaviors of other agents.

3.1. An Agent's Own Behavior

This section illustrates how an automated pilot agent A_o generates flexible and reactive behavior. This illustration is provided using a concrete implementation of A_o in Soar [11, 17]. Soar is an integrated problem-solving and learning architecture that is already well-reported in the literature [14, 15]. The description below abstracts away from many of the details of this implementation, and mainly focuses on Soar's problem space model of problem-solving. Very briefly, a problem space consist of states and operators. An agent solves problems in a problem space by taking steps through the problem space to reach a goal. A step in a problem space usually involves applying an operator in the problem space to a state. This operator application changes the state. If the changes are what are expected from the operator application, then that operator application is terminated, and a new operator is applied. If the operator does not change the state, or if the changes it causes do not meet the expectations, then a subgoal is created. A new problem space is installed in the subgoal to attempt to achieve the expected effects of the operator. (Note that the system uses a procedural representation for these operator expectations — a declarative representation is not necessary. In particular, a procedural representation is sufficient to determine if the expectations are achieved.)

Figure 3 illustrates the problem spaces and operators A_o employs while it is trying to get into position to fire a missile. In the figure, problem spaces are indicated with bold letters, and operators being applied in italics. In some problem spaces, alternative operators are also shown (these are not italicized). In the top-most problem space, named TOP-PS, A_o is attempting to execute its mission by applying the *execute-mission* operator. This is the only operator it has in this problem space. The expected effect of this operator is the completion of A_o's mission, which may be for example to protect its aircraft carrier. Since this expected effect is not yet achieved, a subgoal is generated to complete the application of *execute-mission*. This subgoal involves the EXECUTE-MISSION problem-space. There are various operators available in this problem space to execute A_o's mission, including *intercept* (to intercept an attacking opponent), *fly-racetrack* (to fly in a racetrack pattern searching for opponents when none is present), etc. In fact, in most of A_o's problem spaces there are always several such options available, and A_o has to select a particular operator that would allow it to make the most progress. In this case, A_o selects the *intercept* operator so as to intercept the opponent's aircraft. Given the presence of the opponent, this is the best option available.

A_o attempts to apply the *intercept* operator. However, the expected effect of this operator — the opponent is either destroyed or chased away —

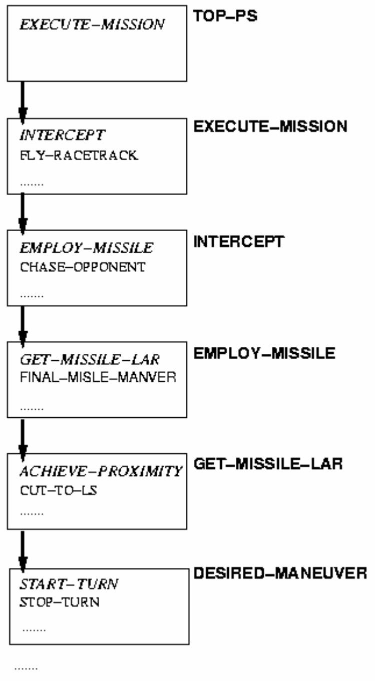

Figure 3: A_o's problem space/operator hierarchy. Boxes indicate problem spaces. Text in italics indicates currently active operator within a problem space.

is not directly achieved. This leads to a subgoal into the *intercept* problem space, where A_o attempts to apply the *employ-missile* operator. However, the missile firing range and position is not yet reached. Therefore, A_o subgoals into the EMPLOY-MISSILE problem space, and applies the *get-missile-lar* operator. (LAR stands for launch-acceptability-region, the position for A_o to fire a missile at its opponent). The *get-missile-lar* operator results in the application of the *achieve-proximity* operator in a subgoal. Finally, this leads to a subgoal into the *start-turn* operator in the DESIRED-MANEUVER problem space. The application of this *start-turn* operator causes A_o to turn. Another operator — *stop-turn* — will be applied to stop the aircraft's turn when it reaches a particular heading (called collision-course). This heading will be maintained until missile firing position is reached. At that time, the expected effect of A_o's *get-missile-lar* operator will be achieved, and hence it will be terminated. A_o can then apply the *final-missile-maneuver* operator from the EMPLOY-WEAPONS problem space. The *final-missile-maneuver* operator may lead to subgoals in other problem spaces, not shown in the figure.

Thus, by subgoaling from one operator into another a whole operator/problem-space hierarchy is generated. The state in each of these problem spaces consists of a global portion shared by all of the problem spaces and a local portion that is local to that particular problem space. This organization supports reactive and flexible behaviors given appropriate pre-conditions (or conditions) for the operators, and the appropriate operator selection and termination mechanisms, as outlined in [13]. In particular, if the global state changes so that the expected effects of any of the operators in the operator hierarchy is achieved, then that operator can be terminated. All of the subgoals generated due to that operator are automatically deleted. Note that A_o may also terminate an operator even if its expected effects are not achieved. This may be achieved if another operator is found to be more appropriate for the changed situation. For instance, suppose the opponent suddenly abandons the combat and turns to return to it base while A_o is attempting to fire a missile at the opponent as shown above. In this case, the *chase-opponent* operator may be more appropriate than the *employ-missile* operator in the *intercept* problem space. Hence, A_o terminates the *employ-missile* operator (all its subgoals get eliminated as well), and instead, A_o applies the *chase-opponent* operator.

Since all of the above operators are used in generation of A_o's own actions, they will be henceforth denoted using the subscript *own*. For instance, *employ-missile*$_{own}$ will denote the operator A_o uses in employing a missile. Operator$_{own}$ will be used to denote a generic operator that A_o uses to generate its own actions. The global state in these problem spaces will be denoted by state$_{own}$. Problem-spaces that consist of state$_{own}$ and operator$_{own}$ will be referred to as *self-centered* problem spaces. The motivation for using this method for denoting states operators and problem spaces will become clearer below.

3.2. Tracking Other Agent's Behaviors

Given the similarities between A_o and its opponent, the key idea in our approach to event tracking is to use A_o's problem space and operator hierarchy to track opponent's behaviors. We will first illustrate this idea in some detail using some simplifying assumptions. The detailed issues involved in operationalizing this idea will be discussed in Section 3.3.

To begin with, let us assume that A_o and its opponent are exactly identical in terms of the knowledge they have of this domain, and all their other characteristics related to this domain. That is, A_o and its opponent have identical problem spaces and operators at their disposal to engage in the air-combat simulation task. This simplifies A_o's event tracking task, since it can essentially use a copy of its own problem-spaces and operators to track the opponent's actions and behaviors. Operators in these problem spaces represent A_o's model of its opponent's operators. These operators are denoted using the subscript *opponent*. Thus, the *execute-mission* operator used in modeling an opponent's execution of her mission is denoted by *execute-mission*$_{opponent}$. Similarly, operator$_{opponent}$ will be used to denote a generic operator used by the opponent.

The global state in these problem-spaces represents A_o's model of the state of its opponent, and is denoted by state$_{opponent}$. Generating state$_{opponent}$ requires A_o to model features such as the opponent's sensor input. Based on information such as the range of opponent's sensors, at least a portion of this state can be generated. However, other portions of state$_{opponent}$ may require fairly complex computation, essentially mirroring the computation that A_o requires to generate all of the information in state$_{own}$. For instance, one important piece of information that is computed in state$_{own}$ is the "angle off" (the angle between the A_o's flight path and opponent's position). Mirroring this computation in state$_{opponent}$ will mean the computation of this "angle off" from the opponent's perspective (the angle between the opponent's flight path and A_o's position). For now, we make another simplifying assumption — that A_o generates a detailed and accurate state$_{opponent}$ — and revisit this issue in Section 3.3.

The problem spaces consisting of state$_{opponent}$ and operator$_{opponent}$ discussed above are referred to as *opponent-centered* problem spaces. With the opponent-centered problem spaces, A_o can essentially pretend to be the opponent. A_o then tracks opponent's behaviors and actions by pretending to engage in the same behaviors and actions as the opponent. In particular, A_o applies operator$_{opponent}$ to state$_{opponent}$, thus modeling the opponent's actual application of her operator to her actual state. Since A_o is modeling the opponent's action, operator$_{opponent}$ does not change state$_{opponent}$. Instead, if the opponent takes some action in the real-world, then that change is modeled as a change in state$_{opponent}$. If this change matches the expected effects of operator$_{opponent}$, then that effectively corroborates A_o's modeling of operator$_{\text{opponent}}$. (Note that as with A$_o$'s operator$_{\text{own}}$, these expectations of operator$_{\text{opponent}}$ may also only be represented procedurally. This procedural representation is sufficient to match the expectations.) If these expectations are successfully matched, operator$_{\text{opponent}}$ is then terminated. As an example, consider *start-turn*$_{\text{opponent}}$ being applied to state$_{\text{opponent}}$. If the opponent actually starts turning, then the operator *start-turn*$_{\text{opponent}}$ is corroborated and terminated. Of course, low-level operators such as *start-turn*$_{\text{opponent}}$ are easy to corroborate in this manner, since the actions they model are directly observable. Others, however, may not generate low-level actions that are directly observable. One category of such operators are the higher level operators like *employ-missile*$_{\text{opponent}}$, which consists of a number of low-level actions. This issue will be discussed below.

This technique of event tracking, where an agent models another by pretending to be in that agent's position, has been previously used in automated tutoring systems [1, 19]. These tutoring systems need the ability to model the actions of the students being tutored. For this, these systems use student-centered problem spaces where states and operators model the students under scrutiny. This technique of modeling the student is referred to as *model tracing*. The approach proposed here for event tracking is thus based on this model tracing work. However, there are some significant differences. For instance, previous work has primarily focused on static, single-agent environments, where the agent being modeled is the only one causing changes in the environment [10]. There are some other differences as well. However, before exploring the impact of these differences, it is useful to first understand in detail how A$_o$ can perform event tracking using its opponent-centered problem spaces. This is explained below using the example from Figure 1. While this explanation does not directly describe the operation of an actual implementation, it is based on an actual implementation that will be described in Section 3.4. Basically, the description presented here will be used to motivate some representational modification leading up to the implementation described in Section 3.4.

Consider the situation in Figure 1-a. In this case, A$_o$ models the opponent's operator hierarchy as shown in Figure 4-a. A$_o$ is seen to accurately model this goal hierarchy, and in particular without any ambiguity about what actions the opponent is exactly engaged in. This is again a simplifying assumption, and we will return to it in Section 3.3. Figure 4-b shows A$_o$'s own operator hierarchy corresponding to the situation in Figure 1-a. We assume that A$_o$ dovetails the execution of these operator hierarchies, communicating important relevant information from one to the other.

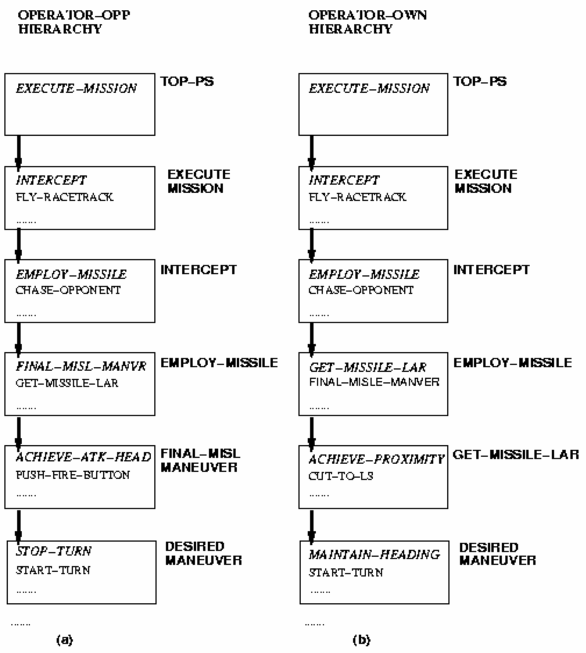

Figure 4: (a) A model of opponent's operator hierarchy, and (b) A$_o$'s own operator hierarchy.

Consider the model of the opponent's operator hierarchy from Figure 4-a. One of the operators in this hierarchy is *final-missile-maneuvers*$_{\text{opponent}}$, which models the opponent's final missile-launching behavior. This is a high-level operator, and its expectations cannot be directly corroborated by observation. This operator is seen to generate a subgoal, where the first operator is *achieve-attack-heading*$_{\text{opponent}}$. This would require a *start-turn*$_{\text{opponent}}$ operator to turn to attack-heading. In Figure 1-a, attack heading is achieved, and state$_{\text{opponent}}$ encodes that fact. Hence, *stop-turn*$_{\text{opponent}}$ is being modeled as the current operator, to model the opponent's stopping her turn at attack-heading.

Now consider A$_o$'s own operator hierarchy in Figure 4-b. A$_o$ is attempting to get into position to fire its own missile using the *achieve-proximity*$_{\text{own}}$ operator in the GET-MISSILE-LAR problem space. When the situation changes from Figure 1-a to Figure 1-b, A$_o$ selects the *cut-to-ls*$_{\text{own}}$ operator in place of the *achieve-proximity*$_{\text{own}}$ operator in the GET-MISSILE-LAR problem space. This operator is intended to increase the lateral separation between the two aircraft.[2] The *cut-to-ls*$_{\text{own}}$ operator causes A_o to turn its own aircraft as shown in Figure 1-b. As the aircraft turns to a particular heading, this new heading is modeled in state$_{\text{own}}$. Thus the *cut-to-ls*$_{\text{own}}$ operator leads to indirect modification of state$_{\text{own}}$.

This change in state$_{\text{own}}$ has to be communicated to state$_{\text{opponent}}$, to update A_o's heading in state$_{\text{opponent}}$. This leads to further modification in state$_{\text{opponent}}$, indicating that the opponent's attack heading is no longer achieved. Based on this modification, *achieve-attack-heading*$_{\text{opponent}}$ is re-activated (or re-applied). This operator again subgoals into the DESIRED-MANUEVER problem space where the *start-turn*$_{\text{opponent}}$ operator is reapplied. When the opponent starts turning, this operator is corroborated and terminated. The next operator in this problem space is *stop-turning*$_{\text{opponent}}$. When the opponent actually stops turning after reaching attack heading, as shown in Figure 1-c, state$_{\text{opponent}}$ is modified to indicate that opponent's attack-heading is achieved, and hence *stop-turning*$_{\text{opponent}}$ operator is corroborated. The change in heading in state$_{\text{opponent}}$ needs to be communicated back to state$_{\text{own}}$, so that A_o may readjust its heading in *cut-to-ls*$_{\text{own}}$ if required.

Continuing with Figure 1-c, the opponent's achievement of attack-heading also corroborates the *achieve-attack-heading*$_{\text{opponent}}$ operator, which is now terminated. A new operator from the FINAL-MISSILE-MANEUVERS problem space — *push-fire-button*$_{\text{opponent}}$ — is now applied. This operator predicts a missile firing, but it is known that that cannot be observed. Hence, *push-fire-button*$_{\text{opponent}}$ is terminated even though there is no direct observation to support that termination. However, the resulting missile firing is marked as not being highly likely. Nonetheless, this missile launch, even with its low likelihood, is communicated to state$_{\text{own}}$, so that A_o may react to it (for instance if A_o's mission forbids it from taking any risks at all). At this point, given the termination of the *push-fire-button*$_{\text{opponent}}$ operator, opponent's *final-missile-maneuvers*$_{\text{opponent}}$ operators is corroborated and terminated. Following that, an Fpole$_{\text{opponent}}$ operator in the EMPLOY-MISSILE problem space predicts an Fpole turn. This again generates a subgoal, back into the DESIRED-MANEUVER problem space and the *start-turn*$_{\text{opponent}}$ operator is reapplied. When the opponent executes her Fpole turn in Figure 1-d, the Fpole$_{\text{opponent}}$ operator is corroborated and terminated. At this point, all of the expectations for the high-level employ-missile$_{\text{opponent}}$ operator are corroborated; and hence state$_{\text{opponent}}$ is modified to indicate that a missile launch is highly likely. These changes in state$_{\text{opponent}}$ — the change in the opponent's heading and the highly likely status of the missile launch — are once again communicated to state$_{\text{own}}$. Based on the high likelihood of the missile launch, A_o activates the operator missile-evasion$_{\text{own}}$ to evade the incoming missile (Figure 1-e). This change in A_o's heading is once again communicated back to state$_{\text{opponent}}$.

Thus, A_o executes its own operators, and tracks opponent's actions and behaviors using the operator$_{\text{opponent}}$ and state$_{\text{opponent}}$. This can help A_o to track its opponent's behaviors, and address all of the constraints on event tracking outlined in Section 2. However, there are some important issues involved in addressing our earlier constraints with this approach. There are also some simplifying assumptions that we made in illustrating event tracking: (i) A_o and its opponent are identical; (ii) A_o performs all of the complex computation that is necessary to accurately model opponent's state; and (iii) A_o can accurately model opponent's operator hierarchy without any ambiguity. Relaxing these assumptions leads to some additional issues, which also relate to the constraints on event tracking. These issues are all discussed in the next Section.

3.3. Addressing Constraints on Event Tracking

The first constraint on event tracking was for an agent to track highly flexible and reactive behaviors of its opponent, while taking appropriate agent interactions into account. The use of opponent-centered problem spaces with operator$_{\text{opponent}}$ and state$_{\text{opponent}}$ helps in partly addressing this constraint (this was the motivation behind this approach to begin with). In particular, operator$_{\text{opponent}}$ can be activated and terminated in the same flexible manner as operator$_{\text{own}}$. There is complete uniformity in the treatment of the two

[2]Lateral separation is defined as the perpendicular distance between the line of flight of A_o's aircraft and the position of its opponent. When the two aircraft are pointing right at each other as in Figure 1-a, there is no lateral separation between the two aircraft. Increasing lateral separation provides a positional advantage.

types of operators.

However, these opponent-centered problem spaces by themselves do not address the issue of modeling the interactions among the different agents. In particular, the method outlined in Section 3.2 requires building one operator hierarchy for A_o, and one for each opponent, with their own global states. This leads to a situation where multiple compartmentalized operator hierarchies with their own global states are generated. Modeling the strong agent interactions present in this domain requires passing messages from one compartment to another. For instance, as described above, when A_o changes heading, that information needs to be propagated from $state_{own}$ to $state_{opponent}$. Similarly, when the opponent fires a missile that information has to be communicated to $state_{own}$ from $state_{opponent}$. Similarly, if A_o is to take some action depending on whether the *intercept*$_{opponent}$ operator is being executed, then that information would need to be propagated to A_o's compartment.

Given the level of interactions among A_o and its opponents, this message passing can be a substantial overhead. Furthermore, there can be many aircraft involved in the combat, leading to an increase in the message passing overheads. This is particularly problematical given the second constraint on event tracking (of real-time performance) and the fourth constraint (which implies continuous agent interactions). Additionally, the communication among the different compartments essentially duplicates the information of one compartment in another. For instance, when a missile is fired, this information is duplicated in different compartments. Such duplication is problematical in terms of maintaining its consistency. If a missile is removed from one compartment, it must be removed from all of the others.

The solution we are investigating to alleviate the problem with this compartmentalization is to merge the different operator hierarchies for the different agents into a single compartment, which we will refer to as world-centered problem space (WCPS for short). WCPS eliminates the boundaries between different self-centered and opponent-centered problem spaces. Instead, the different operator hierarchies are maintained within the context of a single WCPS. There is also a single world state. This state includes A_o's own problem-solving state ($state_{own}$), A_o's model of the state of its opponent ($state_{opponent}$), as well as A_o's model of the states of other entities, including other opponents or friendlies in the world.

WCPS eliminates the need for passing messages to model interactions. Instead, interactions get modeled in terms of changes to the single global state. $Operator_{own}$ and $operator_{opponent}$ are directly able to reference this global state as well as other operators. Furthermore, the problem of duplication of information is avoided. For instance, a missile fired by the opponent gets modeled within this single global state as a single missile. Operator hierarchies modeling all of the different agents can directly react to this missile.

An additional benefit of the single global state in WCPS also relates to one of the assumptions mentioned in Section 3.2. In particular, A_o need not perform all of the complex computation required in modeling opponent's state, but instead it may "re-use" some of the computation. Consider the example of the computation of "angle off" from the opponent's perspective, as mentioned in Section 3.2. With the global state in WCPS, A_o does not need to recompute this "angle off". Instead, this is automatically computed in A_o's $state_{own}$, and this can simply be reused. In particular, A_o's $state_{own}$ already maintains the computation of "target aspect" from its own perspective (the angle between the opponent's flight path and A_o's position). This is precisely the definition of "angle off" the opponent's perspective. Thus, instead of computing the "angle off" from the opponent's perspective and "target aspect" from A_o's perspective separately, a single computation can be performed and used for both purposes. Of course, not all of the complex computation involved in generating the opponent's state can be avoided in this manner. The interesting research question then is determining what portion can be re-used in this manner, and how much extra computation is really necessary.

This shift from small self-centered and opponent-centered problem-spaces to WCPS is related to the *objective* framework used in simulation and analysis of DAI systems [5], which describes the essential, "real" situation in the world. However, the focus of our work is on an individual agent using its world-centered model for event-tracking. While this model introduces a shift towards an objective point of view, by definition, it is an agent's subjective view of its environment, and may contain approximations in $operator_{opponent}$ and $state_{opponent}$.[3]

[3]Note that if the agents do not interact, then a single WCPS may not be appropriate, and separate problem spaces may be the right choice for modeling them.

The second constraint on event tracking relates to A_o's ability to track events in real-time. The key impact of this decision is on generating an accurate and unambiguous operator$_{opponent}$ hierarchy — one of the assumptions made in the previous section. In particular, this constrains the methods A_o can employ in attempting to generate an accurate and unambiguous operator hierarchy. For instance, Ward [19] presents one general method for generating an unambiguous operator hierarchy. This method involves an exhaustive search over all possible operator applications until the one that creates the right expectations, i.e., one that matches the opponent's current actions, is created. If there is more than one such operator application, then one is chosen randomly. A wrong choice can be made in such situations. However, as soon as that is discovered, another exhaustive search can be performed. Given the real-time constraint on event tracking, this type of exhaustive search strategy can not be applied. While Ward suggests some heuristics to constrain the search, this remains a difficult problem. The WCPS approach at least provides a partial answer here. In particular, given the uniformity among operator$_{own}$ and operator$_{opponent}$ in WCPS, the mechanism employed in resolving ambiguity in operator$_{own}$ operators — search control rules — can also be used in resolving ambiguity in operator$_{opponent}$. Besides search control rules, another possibility for resolving ambiguity in WCPS is to generate the goal hierarchy bottom-up rather than top-down. While both of these are powerful tools in WCPS, their advantages and disadvantages in this context are not yet well understood.

The real-time constraint also raises the issue of abstractions in event tracking. In particular, Hill and Johnson [10] have recently argued that tracking an individual agent's actions in detail in a dynamic environment may prove computationally intractable. They advocate detailed tracking only where necessary, and reliance on abstractions elsewhere. In WCPS, abstractions in modeling an operator would imply that detailed subgoals for modeling that operator need not be generated. For instance, A_o may not model the detailed operators used in accomplishing *get-missile-lar*$_{opponent}$. Thus, when *get-missile-lar*$_{opponent}$ is activated, it may not lead to any subgoals. However, when the opponent actually reaches the LAR (missile firing position), *get-missile-lar*$_{opponent}$ can be considered as corroborated and terminated. Unfortunately, this method of abstract modeling may not be appropriate for corroborating an operator such as *employ-missile*$_{opponent}$, which involves multiple maneuvers. In this case, the intermediate headings of opponent's aircraft may be important and just testing the terminating position may be an inappropriate test for corroboration. Automatic generation of the right levels of abstraction is an interesting issue for future work.

The third constraint on event tracking was the generation of expectations for an unseen, but on-going event. In WCPS, the application of an operator$_{opponent}$ in essence is the expectation for the opponent to execute a certain plan or action. Thus, this constraint can be addressed in a straightforward manner. However, since the event is unseen, there can be no corroboration of it. One possibility to deal with this situation is to terminate operator$_{opponent}$ if the relevant action is known to be unobservable (for instance, since the opponent's aircraft is not observable on radar).

The fourth constraint is related to the continuous nature of event tracking. The main implication of this constraint is the continuous interaction among agents, which as discussed above, leads to the move towards WCPS.

There were also three assumptions made in the previous section to simplify event tracking. The second and the third assumption, related to modeling of the opponent's state and operator hierarchy have been discussed above. However, the first one of the assumptions has not been discussed. This assumption is that the automated pilot agent A_o and its opponent are identical. The key implication of this assumption is that A_o can create a copy of its own operator and problem space hierarchy to model the opponent. (This creation of a copy by itself may not be straightforward if all of A_o's knowledge is essentially procedural.) This assumption essentially substitutes for another assumption in the plan recognition literature: the agent that is recognizing a plan is assumed to have full knowledge of all of the plans that the planning agent can execute [12]. If A_o has such additional knowledge about how its opponent's plans or operators, and how those differ differ from its own, then A_o need the ability to interleave those with its own copy of operators while tracking opponent's behaviors. If A_o does not have this additional knowledge, then A_o will need to model its opponent with incomplete information, or to learn that information from observation of the opponent's actions or by some other means.

3.4. A Prototype WCPS-based Agent

An important test of the WCPS model is its actual application in a dynamic, multi-agent environment. The task of developing an automated pilot for the air-combat simulation domain is tailor-made for this test. The development of automated pilots in this domain is currently based on a system called TacAir-Soar [11, 17], which as mentioned earlier, is developed using the Soar integrated problem-solving and learning architecture. TacAir-Soar is a "non-trivial" system that includes about 800 rules.[4] Its original self-centered problem space design worked against an initial inactive opponent. However, it very quickly failed against an active opponent — there was a need for tracking events related to actions of the other agents.

To survive in this real-time environment, the system was forced to employ world-centered problem spaces. However, these world-centered problem-spaces are created based on an incomplete and ad-hoc mechanism, that suffers from three problems. First, event tracking is not robust, meaning the automated pilot agent can and does generate unuseful or misleading interpretations for key opponent actions, such as the opponent's turn in Figure 1-c. This lack of robustness also implies that the automated pilot is unable to deal with sensor limitations effectively. Thus, sometimes if radar contact is momentarily lost, the agent may not track the opponent's actions. A second problem with the existing world-centered problem spaces is that event tracking does not generate expectations. A third problem is that the agent's real-time response can suffer due to sequential operator execution.

We have implemented a variant of TacAir-Soar that is fully based on WCPS. To create this variant, we started with the operators and problem spaces that are used by a TacAir-Soar-based automated pilot in generating its flexible actions and behaviors. We then generated a copy of these operators and problem spaces to model the automated pilot's opponent within a single WCPS. This copy was hand generated (since most of TacAir-Soar's knowledge is procedural, automatic generation of such a copy is an interesting research question that is left for future work). In generating this copy, some of TacAir-Soar's operators and problem spaces were abstracted away — these opponent actions were not modeled in detail. The result is an implementation that is able to track events while generating expectations. It is also promising in terms of being more robust in tracking events. The implementation tracks opponent's action and behavior as described provided in Section 3.2. Simultaneously, as discussed in Section 3.3, it avoids the communication overheads and duplication of information. The implementation currently only works in single opponent situation. Work on extending the implementation to multiple opponent situations is currently in progress.

[4]Since the completion of the experiment described in this section, the size of the TacAir-Soar system has grown to about 1500 rules.

4. Summary

This paper makes two contributions. First, it presents a detailed analysis of event tracking in the "real-world", dynamic, multi-agent environment of air-combat simulation. This analysis reveals interesting issues that represent a novel intersection of the areas of plan recognition and DAI. Tools and techniques that have emerged from single-agent environments are inadequate to address these issues. The second contribution of the paper is the idea of world-centered problem spaces (WCPS), for use in general multi-agent situations. WCPS is independent of problem spaces as such — the key idea is that an agent treats the generation of its own behavior and tracking of others uniformly. WCPS was used in (re)implementing automated pilots for air-combat simulation.

The paper also outlined several unresolved issues in WCPS. Among them, resolving ambiguity in opponent's actions, generating approximations, learning about the opponent from observation, and so on. We hope that addressing these issues will help in allowing WCPS to perform event tracking in a more robust fashion.

References

1. Anderson, J. R., Boyle, C. F., Corbett, A. T., and Lewis, M. W. "Cognitive modeling and intelligent tutoring". *Artificial Intelligence 42* (1990), 7-49.

2. Azarewicz, J., Fala, G., Fink, R., and Heithecker, C. Plan recognition for airborne tactical decision making. National Conference on Artificial Intelligence, 1986, pp. 805-811.

3. Carberry, S. Incorporating default inferences into Plan Recognition. Proceedings of National Conference on Artificial Intelligence, 1990, pp. 471-478.

4. Carberry, S. *Plan Recognition in Natural Language Dialogue.* MIT Press, Cambridge, MA, 1990.

5. Decker, K., and Lesser, V. Quantitative modeling of complex computational task environments. Proceedings of the National Conference on Artificial Intelligenence, 1993.

6. Dousson, C., Gaborit, P., and Ghallab, M. Situation Recognition: Representation and Algorithms. International Joint Conference on Artificial Intelligence, 1993, pp. 166-172.

7. Durfee, E. H., and Lesser, V. R. Using Partial Global Plans to Coordinate Distributed Problem Solvers. In Bond, A. H., and Gasser, L., Ed., *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Palo Alto, CA, 1988.

8. Forbus, K. "Qualitative Process Theory". *Artificial Intelligence 24* (1984), 85-168.

9. Grosz, B. J., and Sidner, C. L. Plans for Discourse. In *Intentions in Communication*, MIT Press, Cambridge, MA, 1990, pp. 417-445.

10. Hill, R., and Johnson, W. L. Impasse-driven tutoring for reactive skill acquisition. Proceedings of the Conference on Intelligent Computer-aided Training and Virtual Environment Technology, 1993.

11. Jones, R. M., Tambe, M., Laird, J. E., and Rosenbloom, P. Intelligent automated agents for flight training simulators. Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation, March, 1993.

12. Kautz, A., and Allen J. F. Generalized plan recognition. National Conference on Artificial Intelligence, 1986, pp. 32-37.

13. Laird, J.E. and Rosenbloom, P.S. Integrating execution, planning, and learning in Soar for external environments. Proceedings of the National Conference on Artificial Intelligence, July, 1990.

14. Laird, J. E., Newell, A. and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

15. Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R. "A preliminary analysis of the Soar architecture as a basis for general intelligence". *Artificial Intelligence 47*, 1-3 (1991), 289-325.

16. Song, F. and Cohen, R. Temporal reasoning during plan recognition. National Conference on Artificial Intelligence, 1991, pp. 247-252.

17. Tambe, M., Jones, R., Laird, J. E., Rosenbloom, P. S., and Schwamb, K. Building Believable Agents for Simulation Environments. Proceedings of the AAAI Spring Symposium on Believable Agents, 1994. (to appear).

18. Van Beek, P., and Cohen, R. Resolving Plan Ambiguity for Cooperative Response Generation. Proceedings of International Joint Conference on Artificial Intelligence, 1993, pp. 938-944.

19. Ward, B. *ET-Soar: Toward an ITS for Theory-Based Representations.* Ph.D. Th., School of Computer Science, Carnegie Mellon University, May 1991.

Milind Tambe is a computer scientist at the Information Sciences Institute, University of Southern California (USC) and a research assistant professor with the computer science department at USC. He completed his undergraduate education in computer science from the Birla Institute of Technology and Science, Pilani, India in 1986. He received his Ph.D. in 1991 from the School of Computer Science at Carnegie Mellon University, where he continued as a research associate until 1993. His interests are in the areas of integrated AI systems, and efficiency and scalability of AI programs, especially rule-based systems.

Paul S. Rosenbloom is an associate professor of computer science at the University of Southern California and the acting deputy director of the Intelligent Systems Division at the Information Sciences Institute. He received his B.S. degree in mathematical sciences from Stanford University in 1976 and his M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University in 1978 and 1983, respectively. His research centers on integrated intelligent systems (in particular, Soar), but also covers other areas such as machine learning, production systems, planning, and cognitive modeling. He is a Councillor of the AAAI and a past Chair of ACM SIGART.

Table of Contents

List of Figures