Event Tracking for an Intelligent Automated Agent

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: In a dynamic, multi-agent environment, an automated intelligent agent is often faced with the possibility that other agents may instigate events that actually hinder or help the achievement of its own goals. To act intelligently in such an environment, an automated agent needs an *event tracking* capability to continually monitor the occurrence of such events and the temporal relationships among them. This capability enables an agent to infer the occurrence of important unobserved events as well as obtain a better understanding of interaction among events. This paper focuses on event tracking in one complex and dynamic multi-agent environment: the air-combat simulation environment. It analyzes the challenges that an automated pilot agent must face when tracking events in this environment. This analysis reveals some novel constraints on event tracking that arise from complex multi-agent interactions. The paper proposes one solution to address these constraints, and demonstrates it using a simple re-implementation of an existing automated pilot agent.

1. Introduction

An automated intelligent agent pursuing its goals in a dynamic, multi-agent environment often encounters a large number of events that significantly impact the actions it takes to achieve its goals. Some of these events may be instigated by the agent itself. Others may be instigated by other agents as they pursue their own goals, which may conflict or coincide with the goals of this agent. As time marches on, these events continue to unfold.

To act intelligently in a world that is rapidly moving by, the automated agent needs to monitor the occurrence of events in its world and monitor the temporal relationships among them (e.g., the particular sequence in which they occur). This information is essential for a variety of reasons. For instance, this information can be used to infer the occurrence of important unobserved events. Consider the following example from the simulated air-combat domain [5]. This domain involves simulated air combat, where the intelligent agents act as automated pilots for the simulated aircraft. These automated pilots will take part in exercises with human fighter pilots, where they will aid in tactics development and training. For effective performance in this domain, these automated pilots must, among other things, continually monitor events in their environment. 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 in the light-shaded one.

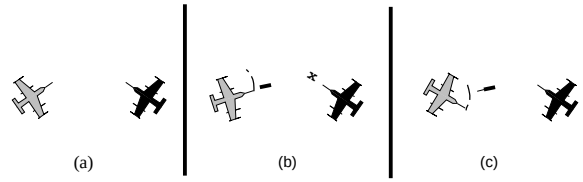

Figure 1: Inferring a missile firing event.

Suppose, initially, the two aircraft are headed as shown in Figure 1-a. After reaching her missile firing range, the opponent turns her aircraft to *attack heading* (a point slightly in front of the automated pilot's aircraft, as shown by a small **x** in Figure 1-b). In this situation, the opponent fires a missile. 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 executes an *Fpole* maneuver then that considerably increases the likelihood of a missile firing. This maneuver involves a 25-50 degree turn away from the attack heading, as shown in Figure 1-c (it is executed after firing a missile to provide radar guidance to the missile, while slowing 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 react as though a missile has actually been fired. The automated pilot must react in a similar manner. Thus, if the opponent engages in this sequence of turns and changes in heading *after* reaching her missile firing range, then the automated agent can infer a missile firing.

The above example illustrates that an automated pilot needs to continually monitor events in its world, such as the opponent's turns and her (inferred) missile firing behavior, and record information about the temporal relationships among them, so as to react to them appropriately. We refer to this capability as *event tracking*. An event here may be considered as any coherent activity over an interval of time. 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, possibly inferred from 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, or simply inferred. *Tracking* any one of these events refers to recording that event in memory, recording the temporal relationship of that event with other events, and monitoring the progress of that event.

To understand event tracking in more detail, it is useful to view it from another perspective. In particular, event tracking can be viewed as answering queries of the following form: given an event ***E*** which consists of a set of sub-events $\{E_1, E_2,...E_N\}$ and a set $\boldsymbol{R} = \{(E_i\ r_1\ E_j), (E_i\ r_2\ E_k)...\}$ of temporal relationships among the sub-events, does ***E*** occur in the world? That is, does the set $\{E_1, E_2,...E_N\}$ occur in the world so as to satisfy the relationships in ***R***? While this query type may appear limited in scope, there are at least two degrees of freedom that make it powerful, enabling it support a variety of capabilities. The first degree of freedom is that there are no restrictions on the event type ***E***: it can be any one of the variety of event types (higher/lower-level, observed/unobserved, etc.) mentioned above. This allows an automated agent to infer events related to the behaviors of other agents. Thus, the event ***E*** may be opponent's missile firing behavior, and the relevant query may check if an opponent has engaged in a particular sequence of turns and changes in heading, after reaching her missile-firing range. Alternatively, a query may involve events instigated by different agents. For example, a query may check if the automated pilot fired its missile before the opponent fired a missile at it. Responses to such queries allow an agent to better understand event interactions so as to react to them appropriately.

The second degree of freedom involves the time at which a query can be presented. In the most unrestricted form, a query about an event ***E*** may be presented at any time before, during or after ***E***'s occurrence. In this paper, we will restrict this degree of freedom: queries about ***E*** may be presented only during the occurrence of that event. We will refer to these restricted queries as *local queries*, and the unrestricted queries as *non-local queries*. Non-local queries may be used in service of planning and post-hoc explanation, and require that the system either maintain a long-term memory of past events or be capable of long-term prediction of future events.

Thus, the event tracking problem can be seen as covering a broad spectrum of problems. One aspect of this problem — related to inferring events based on other events — is closely related to plan-recognition [6, 9, 3, 4] and model tracing [2]. Another aspect of this problem — related to maintenance of temporal relationships among

events — corresponds to temporal reasoning [1]. Other aspects of the problem, corresponding to non-local queries, are related to planning and explanation.

One possible approach to the event tracking problem then is to address all its different aspects individually. An alternative approach is to treat event tracking as a single problem with a single, unified solution. We believe it is reasonable to work towards such a unified solution, since it may be able to exploit the interdependencies involved among the different aspects of this problem. Therefore, this is the approach that we take in this paper (although at present the scope of the problem is restricted to local queries).

The rest of this paper is organized as follows: Section 2 analyzes the requirements for event tracking in the air-combat simulation domain, revealing some challenging constraints. Section 3 proposes a solution for event tracking that addresses these constraints. The key idea is to exploit the similarity among the characteristics of the different agents. This solution is demonstrated using a simple re-implementation of an automated pilot agent for air-combat simulation. This automated pilot is based on a system called TacAir-Soar [5], which is developed within Soar, an integrated problem-solving and learning architecture [8]. For the purposes of this paper, we need to focus only 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 it 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 changes 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.

2. Constraints on Event Tracking

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 example, suppose an automated pilot is tracking an opponent's turns and changes in heading while the opponent is executing its maneuver to fire a missile (see Figure 1). Suppose just before the opponent fires its missile (Figure 1-b), the automated pilot suddenly turns its own aircraft. In response, the opponent may need to turn its aircraft again before firing its missile, and its Fpole (Figure 1-c) may be executed in a different manner as well. (For detailed examples of agent interactions in this domain, see [10].) 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(s). This is a challenging constraint — previous investigations in the related areas of plan/situation recognition [6, 9, 4, 3] and model tracing [2] have not addressed this constraint. In particular, plan recognition models have not been applied in such dynamic, interactive situations, and hence do not address strong interactions among agents and the resulting flexibility and reactivity in agent behaviors.

A second related constraint on event tracking here is that it 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 and final 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. In particular, an opponent may *drop off* (become invisible from) the automated pilot's aircraft

during the combat. In such situations, expectations about the opponent's location can help in quickly re-establishing radar contact with her.

These constraints rule out a variety of solutions for event tracking. In fact, as a first try, we attempted to address the event tracking problem by explicitly recording in memory all of the events and all of the temporal relationships among them. In the implementation of this solution in TacAir-Soar, if a new event E_{N+1} was seen to occur after (or at the same time as) N events $E_1,...,E_N$, then this was stored in TacAir-Soar's memory using N explicit records of the form: *after*(E_i, E_{N+1}) (or *same-time*(E_i, E_{N+1})). Unobserved events such as missile firings were inferred by using pre-compiled queries, where each query listed a sequence of events $\{E_1,..E_N\}$ and the temporal relationships among them. However, this solution could not satisfy the constraints outlined above. More specifically, the solution only recorded events that had already occurred, and did not generate expectations. It also caused a slowdown — the automated pilots were unable to function in real-time. More importantly, this solution ran into a problem in inferring unobserved events. Basically, a small number of pre-compiled queries were insufficient to capture the range and complexity of events resulting from the highly flexible and reactive agent behaviors. The next section proposes an approach for event tracking that addresses all of the constraints outlined above.

3. 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 complexity in their behaviors. In particular, consider an automated pilot agent in this environment that requires the ability to track the complex chain of events corresponding to the flexible and reactive actions and behaviors of other agents. This automated pilot itself instigates an equally complex chain of events corresponding to its own flexible and reactive behaviors. Thus, the key idea is that all the knowledge and implementation level mechanisms that the automated pilot itself uses in instigating events may be used in service of tracking events instigated by other agents.

To understand this idea in detail, it is first useful to understand how an agent generates its own flexible and reactive behaviors. For this we turn to a concrete implementation of an automated pilot agent (call it A_o) in TacAir-Soar. Figure 2 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 is not yet achieved, a subgoal is generated. 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), etc. A_o selects the *intercept* operator — given the presence of the opponent, this is the best option available. Since the intercept is not yet complete, a subgoal is generated. This subgoal involves the INTERCEPT problem space, where A_o applies the *employ-missile* operator. However, the missile firing range and position is not yet reached. Therefore, A_o applies the *get-missile-lar* operator in a subgoal. (LAR stands for launch-acceptability-region, the position for A_o to fire a missile at its opponent). This subgoaling continues until the application of the *start-turn* operator in the DESIRED-MANEUVER problem space, which causes A_o to turn. Later, the *stop-turn* operator is 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 that will be terminated.

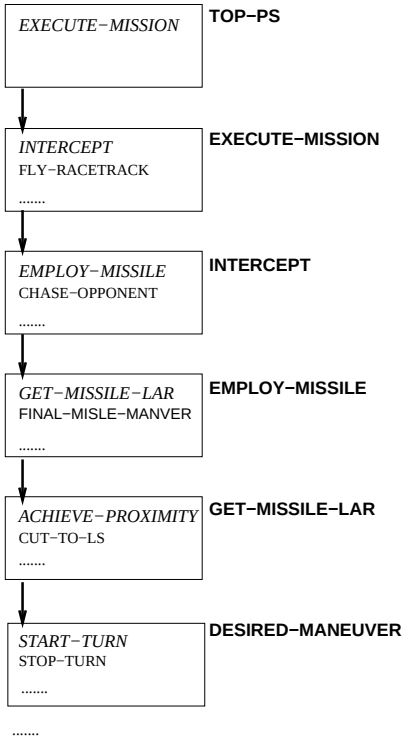

Figure 2: A_o's problem space/operator hierarchy.

Thus, by subgoaling from one operator into another a whole operator/problem-space hierarchy is generated. This organization supports reactive and flexible behaviors given appropriate operator selection and termination mechanisms [7]. For instance, there is a global state shared by all of the problem spaces. If this 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 then automatically deleted. A_o can also terminate an operator even if its expected effects are not achieved (e.g., if another operator is found to be more appropriate for a given situation).

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.

3.1. 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 operator hierarchy to track opponent's behaviors. To illustrate this idea, we begin with the simplifying assumption that A_o and its opponent are exactly identical in terms of problem spaces and operators at their disposal to engage in the air-combat simulation task. Thus, A_o can essentially use a copy of its own problem-spaces and operators to track the opponent's actions and behaviors. We will refer to these copies as *opponent-centered* problem spaces. 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*$_{opponent}$ operator is used in modeling an opponent's execution of her mission. Similarly, operator$_{opponent}$ is 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$_{opponent}$. We assume for now that A_o can easily generate an accurate state$_{opponent}$. (Note that we will make a few such assumptions in explaining event tracking in this section, and address them later in Section 3.2. Note also that while this section does not directly describe the operation of an actual implementation, it is based on an actual implementation that will be described in Section 3.3. Basically, the description presented here will be used to motivate some representational modification leading up to the implementation described in Section 3.3.)

To engage in air-combat simulation, it is possible for A_o to execute the two sets of

problem spaces — self-centered and opponent-centered — in parallel. This will allow A_o to generate its own actions with the help of self-centered problem spaces, and track opponent's behaviors with the help of opponent-centered problem spaces. With the opponent-centered problem spaces, A_o can essentially pretend to be the opponent. A_o can then track the 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 $operator_{opponent}$. $Operator_{opponent}$ is then terminated.

To understand this in concrete terms, consider the example in Figure 1. Specifically, consider the situation just at the beginning of Figure 1-a. Given our assumptions above, A_o can use an opponent-centered copy of its set of self-centered problem spaces from Figure 2 to track its opponent's behaviors. Here, the $execute\text{-}mission_{opponent}$ operator models the opponent's execution of her mission. Since the opponent's mission is not completed, a subgoal is generated, where the operator $intercept_{opponent}$ is applied. This operator subgoals into $employ\text{-}missile_{opponent}$ and so on, until $start\text{-}turn_{opponent}$ is applied to $state_{opponent}$. If the opponent actually starts turning, then the operator $start\text{-}turn_{opponent}$ is corroborated and terminated. The next operator in this problem space is $stop\text{-}turn_{opponent}$. When the opponent actually stops turning after reaching collision course, then $stop\text{-}turn_{opponent}$ is corroborated. This is the situation in Figure 1-a. Right at the beginning of Figure 1-b, the opponent actually reaches the missile LAR, and $get\text{-}missile\text{-}lar_{opponent}$ is corroborated and terminated. A_o now applies $final\text{-}missile\text{-}maneuver_{opponent}$ in the EMPLOY-MISSILE problem space. This subgoals into $achieve\text{-}attack\text{-}heading_{opponent}$. This subgoals into $start\text{-}turn_{opponent}$. When the opponent actually turns to attack heading as shown in Figure 1-b, $achieve\text{-}attack\text{-}heading_{opponent}$ is corroborated and terminated. A new operator from the FINAL-MISSILE-MANEUVERS problem space — $push\text{-}fire\text{-}button_{opponent}$ — is now applied. This operator predicts a missile firing, but it is known that that cannot be observed. Hence, $push\text{-}fire\text{-}button_{opponent}$ is terminated even though there is no direct observation to support that termination. (This corroboration without observation requires the addition of some knowledge that is specific to opponent-centered problem spaces, i.e., not copied from self-centered problem spaces.) This also corroborates and terminates $final\text{-}missile\text{-}maneuvers_{opponent}$. However, the resulting missile firing is marked as not being highly likely. Following that, an $Fpole_{opponent}$ operator in the EMPLOY-MISSILE problem space predicts an Fpole turn. When the opponent executes her Fpole turn in Figure 1-c, the $Fpole_{opponent}$ operator is corroborated and terminated. With this Fpole turn, the missile firing is now considered as being highly likely. A_o may now attempt to evade the missile.

Viewing the above in terms of event tracking, essentially, each $operator_{opponent}$ is an event $\boldsymbol{E}$. The suboperators of $operator_{opponent}$ correspond to the set $\{E_1, E_2..E_N\}$, and their sequence of execution corresponds to the temporal relationships $\boldsymbol{R}$ among the events. Thus, the execution of $operator_{opponent}$ corresponds to the dynamic generation and execution of the query regarding the event $\boldsymbol{E}$. This method allows A_o to track events related to behaviors of other agents without pre-compiling all possible queries related to those behaviors. However, this method does face an interesting challenge when responding to queries involving events

instigated by different agents. For instance, consider a query that checks if the automated pilot fired its missile before the opponent fired a missile at it. Responding to such a query requires the execution of an operator in both self-centered and opponent-centered problem spaces, which this method does not directly support. The next section outlines a solution that facilitates responding to such queries.

3.2. Addressing Constraints

The previous section described a mechanism for A_o to generate its own behaviors while tracking opponent's behaviors: A_o executes two sets of problem spaces — self-centered and opponent-centered — in parallel. This section analyzes the weaknesses in this mechanism particularly given the constraints on event tracking identified in Section 2.

The first constraint on event tracking was for an agent to track highly flexible and reactive behaviors of its opponent. The use of opponent-centered problem spaces with $operator_{opponent}$ and $state_{opponent}$ helps in partly addressing this constraint (this was the motivation behind this approach to begin with). In particular, $operator_{opponent}$ can be activated and terminated in the same flexible manner as $operator_{own}$. There is complete uniformity in the treatment of the two types of operators.

However, one assumption in our description of event tracking with opponent-centered problem spaces is that A_o can generate an accurate $state_{opponent}$. While this seems like a highly problematical assumption at first glance, there are several ways in which this problem is simplified. First, there are some *strong assumptions* that A_o can make about its opponent's state based on the "intelligence" information. This information may include things such as the approximate range of the opponent's radar, the range and types of missiles the opponent's aircraft can carry, and so on. Based on these strong assumptions, A_o can make further *weak assumptions* about the opponent's state. For instance, based on the opponent's radar range, A_o can assume that its aircraft would be visible to the opponent's radar at a particular range. This is a weak assumption because information about the opponent's radar range is approximate, and more importantly, the opponent's radar may not be pointed in A_o's direction. So should A_o assume that it is visible on opponent's radar? If A_o assumes that it becomes visible on opponent's radar as soon as the range is reached, A_o may not maneuver to gain positional advantage. On the contrary, as A_o moves closer and closer to the opponent, the chances of the opponent seeing it continually increase, and A_o can commit a serious mistake if it continues to assume that the opponent cannot see it. Thus, the general problem here is understanding when to make (nor not make) such weak assumptions, without committing serious mistakes.

The solution we are experimenting with is to inject the weak assumption into $state_{opponent}$ at the point where the opponent indicates (by turning her aircraft) that there is likely some change in her state. The motivation here is to avoid making the weak assumption too soon, by triggering it with at least some indication of a change of state from the opponent. The injected assumption is verified by corroborating the resulting $operator_{opponent}$ with the opponent's actual actions. For instance, if the weak assumption of A_o being visible to the opponent is injected into $state_{opponent}$, the resulting $operator_{opponent}$ indicates that the opponent is likely to turn to collision course. If the opponent does indeed turn to collision course, the weak assumption is considered validated.

Besides the weak assumptions, the second major issue in $state_{opponent}$ is the overhead of computing and maintaining derived information in $state_{opponent}$. For instance, assuming for now that A_o is indeed visible to the opponent's radar, $state_{opponent}$ needs to be elaborated with the information that the opponent is likely to obtain from her radar. This includes A_o's heading, altitude, the range between the two aircraft, target aspect from the opponent's perspective (the angle between the A_o's flight path and the opponent's position) the angle off from the opponent's perspective (the angle between the opponent's flight path

and A_o's position) and so on. Calculating angles such as target aspect, angle off etc from the opponent's perspective can be very expensive. Additionally, given the dynamic nature of the environment, there is a need to continuously update all of this information to keep it consistent. For instance, as A_o turns, $state_{opponent}$ has to be modified to change all of the information on it regarding A_o's heading, target aspect and so on.

The problem A_o faces here is that opponent-centered and self-centered problem spaces are compartmentalized operator hierarchies. This leads to problems in modeling the strong agent interactions present in this domain. As one entity changes in one compartment, there is a substantial overhead of keeping all the information consistent with it in (all of the) other compartments. The solution we are investigating here is to merge the different operator hierarchies into a single compartment, which we will refer to as a 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, with a single world state. This single world state can now allow information sharing between $state_{own}$ and $state_{opponent}$, thus helping to keep it consistent. For instance, the A_o's range to its opponent is identical to the opponent's range to A_o. Thus, the range information, which A_o has available on $state_{own}$ from its radar, can be directly shared with $state_{opponent}$. As this range changes in $state_{own}$, it is automatically updated in $state_{opponent}$. The angle off (target aspect) from the opponent's perspective is also shared, since that turns out to be the target aspect (angle off) from A_o's perspective. WCPS encourages such sharing of information, and thus dramatically reduces the burden of modeling $state_{opponent}$.

WCPS also facilitates responding to queries such as the one discussed at the end of Section 3.1, which checks if the automated pilot fired its missile before the opponent. In WCPS this query can be executed by directly executing a *multi-agent* operator, i.e., $operator_{self\text{-}and\text{-}opponent}$, which can operate on both $state_{own}$ and $state_{opponent}$.

The second constraint on event tracking relates to A_o's ability to track events in real-time. One key impact of this constraint is on the generation of an accurate $operator_{opponent}$ hierarchy. In particular, this constrains the amout of time A_o can spend in generating an accurate operator hierarchy: an exhaustive search is definitely ruled out. This issues is on the top of our list of items 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.

Finally, the key assumption in the previous section was that the automated pilot agent A_o and its opponent are identical. The main implication of this assumption is that A_o can create a copy of its own operator and problem space hierarchy to model the opponent. If A_o does have some additional knowledge about how some of the opponent's operators differ from its own, then A_o could use those operators in modeling the opponent, instead of using copies of its own operators. 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.

3.3. 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 [5], which is being developed using Soar. TacAir-Soar is a non-trivial system that includes about 800 rules. 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 (by hand) a copy of these operators and problem spaces to model the automated pilot's opponent within a single WCPS. 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 that the current TacAir-Soar implementation. The implementation tracks opponent's action and behavior as described in Section 3.1. Simultaneously, by using WCPS it reduces the overheads as outlined in Section 3.2. The implementation currently only works in simple single opponent situations. Work on extending the implementation to more complex situations is currently in progress.

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 raises some novel issues for event tracking. The second contribution is the idea of world-centered problem spaces (WCPS). WCPS is independent of problem spaces as such — the key ideas are that an agent treats the generation of its own behavior and tracking of others uniformly and that it shares as much information as possible to avoid computational overheads. 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, learning from observation of opponent's actions, 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. Allen, J. "Maintaining knowledge about temporal intervals". *Communications of the ACM 26*, 11 (November 1983).

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

3. 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.

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

5. 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.

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

7. 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.

8. 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.

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

10. Tambe M., and Rosenbloom, P. S. Event tracking in complex multi-agent environments. Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation, May, 1994.

Table of Contents

List of Figures