**Framework Application**

In Chapter ?, two decoupled multi-agent problem algorithms were introduced and their inner workings explained. In Chapter ??, a flexible and general framework for the implementation of algorithms was presented. This chapter combines the concepts of the preceding chapters, applying the process such that two different algorithms are adapted to function on equivalent playing fields through the use of the state machine approach.

To begin, the algorithms are decomposed into their underlying behaviors such that they cohere with the state machine diagrams presented in Chapter ?. With increasing granularity, routines present in the algorithms are assigned to states in the governing state machine until the algorithm is fully represented. From this position it becomes trivial to extract a strategy for practical implementation, as is done in the design of FleetBench. The use of this process, demonstrated in broad strokes in this chapter and at the implementation level in the Appendix, demonstrates the advantages of approaching MAPF and MAPD solution implementation in this manner.

This chapter assumes that the Simulation Definition state is already completed by a user configuring certain options. The actual implementation of the definition state experiences a great degree of freedom in terms of what options are available in the simulation and must be developed as needed. As an example, Table ? lists all the configuration options for the implementation used in FleetBench at time. As mentioned, these rules may be expanded significantly to cover unplanned agent breakdowns, kinematic concerns such as rotation, or limitations on agent charge or fuel, among real-world behaviors.

|  |  |  |
| --- | --- | --- |
| **Relevance** | **Option Name** | **Choices** |
| Pathfinding | Solver Algorithm | Single-Agent A\* |
| Multi-Agent A\* (LRA\*) |
| Cooperative A\* (CA\*) |
| Hierarchical CA\* (HCA\*) |
| Windowed HCA\* (WHCA\*) |
| Token Passing (TP) |
| TP with Task Swaps (TPTS) |
| Heuristic Function | Dijkstra |
| Manhattan |
| Euclidean |
| Heuristic Relaxation Coefficient |  |
| Pathfinder Search Window Depth | , if applicable |
| Agent Behavior | Agent Collision Handling | Respected |
| Ignored |
| Task Interaction Cost | Instantaneous |
| Timestep |
| Agent Count |  |
| Task Generation | Task Generation Technique | Scheduled |
| On Agent Availability |
| Task Location Configuration | Node weights |
| Include/Exclude node |
| End Conditions | Simulation ends on task completions |  |
| Simulation ends on timesteps elapsed |  |
| Simulation ends on schedule completion | True or False |

**WHCA\***

To begin fitting the decoupled MAPF algorithm WHCA\* into the state machine model it is worth noting that WHCA\* offers no particular strategy for task assignment, as the MAPF problem assumes that all agents have a task assigned before the problem should be solved. As a result, any implementation of the WHCA\* algorithm in an MAPD context will need a generic strategy for task assignment. Here, as in the implementation given in Appendix ?, the generic routine is named GenerateTask. It simply selects the first available task from the task set or creates a new task if the simulation definition allows.

Critically, WHCA\* is also incomplete in the MAPD case. It fails to consistently avoid collisions during its runtime. This problem arises when agents finish their current plans, and thus reservations, while another agent is attempting to reach the same goal location. If the first agent to arrive finds itself trapped, it will be unable to move away while simultaneously not being able to remain in place. In order to avoid an immediate end of the simulation via the crashed state, it is necessary to develop a generic collision resolver. Even in the MAPF case, a one-agent width corridor of sufficient depth (exceeding the window size) will prevent progress from being made as neither agent will find an escape from its current position, resulting in an infinite stall. FleetBench approaches this problem using a collision resolver which prioritizes trapped agents and forces a replanning of agent motions until the problem is resolved. The resolver is presented in the Appendix, as it is not central to the work done here.

The bulk of the logic employed in WHCA\* is for pathfinding which makes it relatively trivial to fit into the state machine model. For completeness, the algorithm is expanded to include the routines used in FleetBench for task selection and collision resolution, explained in Appendix ?.

Algorithm ? shows the translated version of WHCA\* in the context of the whole system loop, where lines of the original algorithm have been replaced with named routines for ease of reference. The resulting information is directly transferrable to the state machine diagram. Extra care should be taken to ensure that the additional loops which are possible due to implementation choices. For instance, how collisions are resolved—by nullifying plans and re-entering the agent selection state or by forcing new plans within the action execution state? Implementing the former would require an additional loop which begins with agent selection, moving through task management and action planning until all agents have created valid paths. For the latter, plans are simply re-cemented during the action execution state.

|  |  |  |
| --- | --- | --- |
| **Algorithm 1** Solving MAPD problems using WHCA\* | | |
|  | Input is an MAPD problem , defined by the user | **State** |
|  | PreProcess | **New Simulation** |
|  |  |
|  | **while** **not** endConditions | **End Step** |
|  | Update | **New Step** |
|  | **for** | **Select Agent** |
|  | **if** allows task creation: | **Manage Tasks** |
|  |  |
|  | **if** has no assignment: |
|  |  |
|  | **if** |
|  | Assign to |
|  | **while** mayAct: | **Plan Actions** |
|  | **if** Position Goal |
|  |  |
|  | **else**: |
|  | **if** has : |
|  |  |
|  | **else**: |
|  | WHCAStar(PositionGoal |
|  |  |
|  | Store in system memory |
|  | validatePlans | **Execute Actions** |
|  | **while not** |
|  | resolveCollisions |
|  | Validate all stored plans |
|  | **if**  is impossible: | **Sim. Crash** |
|  | System Crashes |
|  | Execute all actions | **Execute Actions** |
|  | System time increments | **End Step** |
|  | Solution found for , for endConditions | **End Simulation** |

**TPTS**

TPTS, being designed for the MAPD problem, better analogizes real-world applications in continuously active cooperative robotics applications and is more involved. The algorithm has a strategy for both task selection and task exchanging, offering an improvement in efficiency by minimizing unnecessary travel time. It also offers conditional guarantees regarding completeness of the algorithm. If the system map conforms to the definitions presented in [1], then it should be impossible for a collision to occur. Verifying whether those conditions are met is a task for the Simulation Definition state. In the case that a system map is not well-formed and is simulated anyway a conflict resolution system must be in place to prevent the simulation from entering the crashed state.

Once again, a line-by-line process of re-composing the algorithm into named routines is presented in Algorithm 2. However, a problem arises. Because the assignment optimization requires comparison of path lengths (line 16), the ***GetTask*** routine of TPTS interleaves the search for a task assignment with the planning of the path, as an optimization to avoid recalculating paths a second time. Without extra effort, this approach will not fit neatly into the state machine model, requiring the implementing engineer to jump through hoops and introduce additional logic. This issue is demonstrated in Algorithm 3.

|  |  |  |
| --- | --- | --- |
| **Algorithm 2** Solving MAPD problems using TPTS | | |
|  | Input is an MAPD problem , defined by the user | **State** |
|  | PreProcess | **New Simulation** |
|  | checkWellformed |
|  | **if not** |
|  | Handle badly formed problems, simulation may abort |
|  |  |
|  | **while** **not** endConditions | **End Step** |
|  | Update | **New Step** |
|  | **for** : | **Select Agent** |
|  | **if** allows task creation: | **Manage Tasks** |
|  |  |
|  | **if** has no assignment: |
|  | GetTask |  |
|  | **while** mayAct: | **Plan Actions** |
|  | GetTask |  |
|  | validatePlans | **Execute Actions** |
|  | **while not** |
|  | resolveCollisions |
|  | Validate all stored plans |
|  | **if**  is impossible: | **Sim. Crash** |
|  | System Crashes |
|  | Execute all actions | **Execute Actions** |
|  | System time increments | **End Step** |
|  | Solution found for , for endConditions | **End Simulation** |

The state machine design pattern allows for two methods of fixing this problem. First, the state machine can be adjusted with a logical branch uncritically allowing this behavior, shown in Figure ?. This is akin to direct manipulation of the data intended to be managed within the action planning state. Because the underlying routines may be called from anywhere, as is done in the direct implementation of TPTS, this is not really a problem.

|  |  |  |
| --- | --- | --- |
| **Algorithm 3** GetTask from TPTS | | |
|  | Demonstrates that planning paths occurs between management of tasks, requiring a different strategy. | **State** |
|  | **function** GetTask | **Manage Tasks** |
|  |  |
|  | **while** |
|  |  |
|  |  |
|  | **if** no agent assigned to |
|  | Assign to |
|  | Path1 | **Plan Actions** |
|  | **return** | **Manage Tasks** |
|  | **else**: |
|  |  |
|  | agent assigned to |
|  | Path |
|  | Unassign from ; Remove from |
|  | Path1 | **Plan Actions** |
|  | **if** : | **Manage Tasks** |
|  | GetTask |
|  | **if** : |
|  | **return** |
|  | **else**: |
|  |  |
|  | **if** Position |
|  | Path2 | **Plan Actions** |
|  | **if** | **Manage Tasks** |
|  | **return** |
|  | **else**: |
|  | **if** |
|  | Stay | **Plan Actions** |
|  | **else**: | **Manage Tasks** |
|  | Path2 | **Plan Actions** |
|  | **return** | **Manage Tasks** |
|  | **return** |

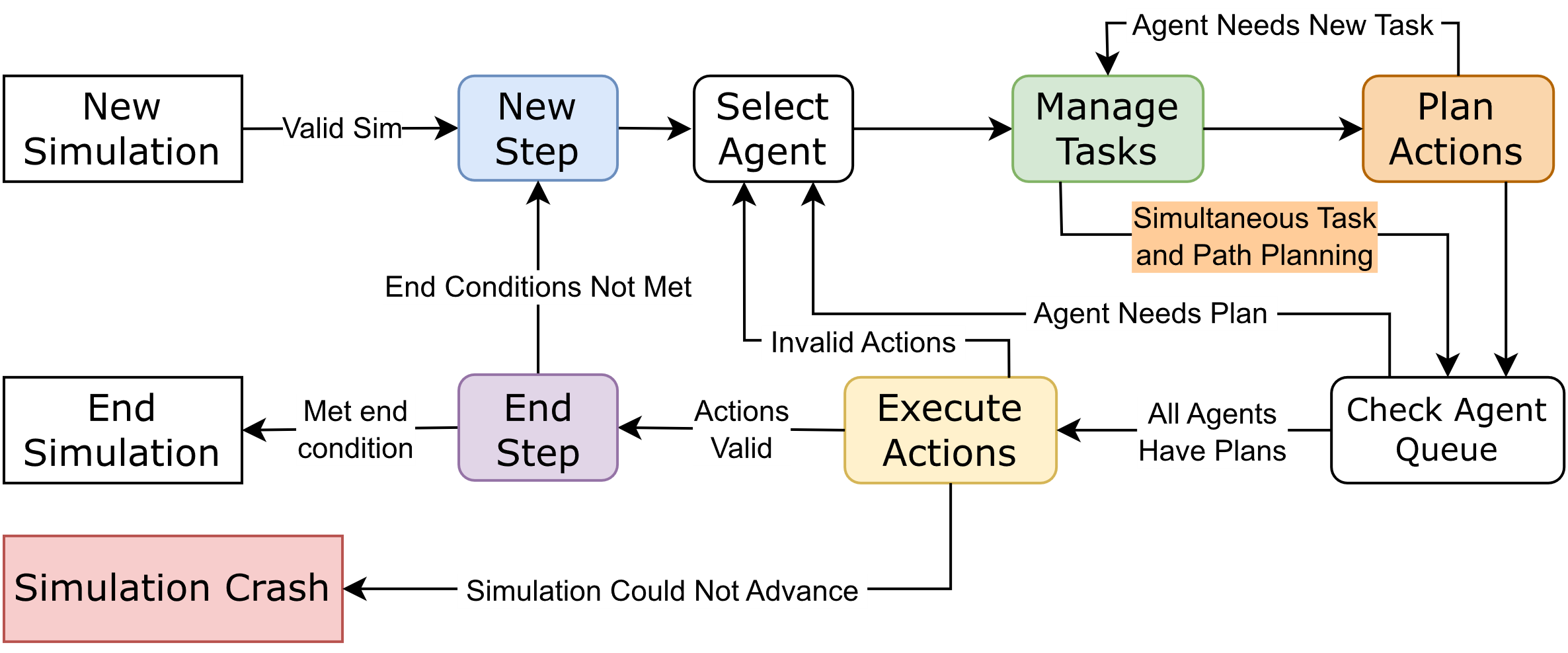


Figure 1: Modified state machine diagram, showing a new branch which accounts for simultaneous task and path planning during the manage tasks state.

To adhere to the state flow diagram more strictly (and enjoy its benefits) it is necessary to decouple the task swapping from path planning. This re-introduces the duplication of path searches in the case a task swap occurs, which is intuitively undesirable. However, because the intention of this work is to provide a generalizable method for testing algorithms, it is expected that a significant amount of work will be done by implementors of a particular algorithm to optimize its function in the real-world application space. This is clearly beyond the scope of this work, whose primary aim is to evaluate the *system performance* under a multitude of conditions rather than its *program runtime*, while still adhering to the logic of the algorithms in use.

To make this possible, a few pseudo-task singletons are introduced to represent an agent’s intent: ***IntendTask***, ***IntendRest***, ***IntendStay***. These will be evaluated in the determine action state, shown in Figure ?? from Chapter ??, to determine whether ***Path1***, ***Path2***, or ***Stay*** should be called, respectively. Instead, during the task management state the calls to the pathfinders ***Path1ps***, ***Path2ps***, and ***Stayps*** are used to find paths without requesting space in the reservation table, returning the singletons alongside the paths. Paths which remain valid in the future do not need to be recalculated, so can be stored in the agent’s memory. Agents which have planned a move before a task swap renders the plan useless will have their intended action for the timestep revoked, thus being caught by the check agent queue state and made to seek alternative actions. The new procedure is given in Algorithm 4, showing that the task determination and the action planning are decoupled and solved independently.

|  |  |  |
| --- | --- | --- |
| **Algorithm 4** Solving MAPD problems using TPTS | | |
|  | Input is an MAPD problem , defined by the user | **State** |
|  | PreProcess | **New Simulation** |
|  | checkWellformed |
|  | **if not** |
|  | Handle badly formed problems, simulation may abort |
|  |  |
|  | **while** **not** endConditions | **End Step** |
|  | Update | **New Step** |
|  | **for** : | **Select Agent** |
|  | **if** allows task creation: | **Manage Tasks** |
|  |  |
|  | **if** has no assignment: |
|  |  |
|  | **while** |
|  | t |
|  |  |
|  | **if** no agent assigned to |
|  | Assign to t |
|  | **else**: |
|  | Store assignments, reservations in memory |
|  | agent assigned to |
|  | Path |
|  | Unassign from |
|  | Remove from , |
|  | Path1ps |
|  | **if** : |
|  | GetTask |
|  | **if** : |
|  | **break** |
|  | **else**: |
|  | Restore assignments, reservations |
|  | **if** is not “intendTask”: |
|  | **if** Position |
|  | Path2ps |
|  | **else**: |
|  | **if** |
|  | Stayps |
|  | **else**: |
|  | Path2ps |
|  | **while** mayAct: | **Plan Actions** |
|  | If is still valid use it, otherwise: |
|  | **if**  is “intendTask”: |
|  | Path1 |
|  | **else if** is “intendRest”: |
|  | Path2 |
|  | **else if** is “intendStay”: |
|  | Stay |
|  | validatePlans | **Execute Actions** |
|  | **while not** |
|  | resolveCollisions |
|  | Validate all stored plans |
|  | **if**  is impossible: | **Sim. Crash** |
|  | System Crashes |
|  | Execute all actions | **Execute Actions** |
|  | System time increments | **End Step** |
|  | Solution found for , for endConditions | **End Simulation** |

This implementation could therefore be used without adjustment to the top-level state machine diagram, minimizing the amount of “hard-coding” that need be done during implementation of a new algorithm to an existing simulator based on the principles laid out in Chapter ?. In the appendices, it will be shown that FleetBench is capable of implementing the first solution without compromising state flow, proving that this is generally not a significant concern and may be left up to the discretion and preference of the user.

[1] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, “Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks.” arXiv, May 30, 2017. Accessed: Nov. 05, 2023. [Online]. Available: http://arxiv.org/abs/1705.10868