**Generalizable Approach**

The collection of data about any studied algorithm inevitably involves simulation over a defined multi-agent problem space. As there is little agreement in the literature regarding fundamental choices in the design of the simulation space, it can be difficult to assure that results from an algorithm are generalizable [1]. For instance, the WHCA\* algorithm presented in [2] is tested on a randomly generated grid, with a random distribution of agents, and randomly selected task points. TPTS, on the other hand, is tested on problems which mimic the layout of a storage facility, with long corridors through which agents must avoid each other to reach their destinations [3]. Some standard test cases used in MAPF problems are offered as possible reference benchmarks, but there is little agreement on the use of and limited applicability of results from these cases [4].

As a result, those seeking to use multi-agent pathfinding algorithms in their own work must implement the algorithm and design test cases which best represent their use case. An engineer wishing to compare several algorithms may find a need for a high level of knowledge in several programming languages, the skill to modify existing code, and the ability to script the generation of test cases.

In an attempt to ease this knowledge burden and reduce the testing time requirement, this chapter presents a general approach to the implementation of algorithms. The approach presented here is a procedure which keeps the overall system coherent and easy to configure for testing. Results of the implementation of algorithms from Chapter ?? using this strategy are presented in Chapter ??? to demonstrate its feasibility.

**Common Behaviors**

By identifying a set of behaviors which must be common to all algorithms attempting to solve multi-agent problems the process can be reframed as a set of behaviors taken when certain conditions are met. The implementation of an algorithm is a problem which can then be reduced to identifying when certain behaviors occur during the lifetime of the algorithm. Decomposing the algorithm in this way provides clarity of function, modularity of implementation, and ease of adaptation for future experiments.

Separating behaviors in the algorithm in this manner requires careful consideration of what multi-agent algorithms are meant to do at the basest level:

* Agents are assigned tasks.
* Agents must take actions which work toward completion of assigned tasks.
* Agents must not collide.
* Agents must move efficiently toward their goals.

As an example, in the case of TPTS it is easy to see that there are provisions laid out for each of these desired behaviors. The algorithm efficiently finds paths for agents by using the optimal A\* search. During the search a reservation table is employed which avoids collisions. Agents are able to eventually find paths to their goals, which are assigned in an optimized fashion using proximity and task swaps.

By anchoring these behaviors and isolating the portions of the algorithm that implement them, the process of executing the algorithm can be abstracted as an implementation of a finite state machine. This well-studied concept in programming offers a concrete method for implementing the logical processes in an algorithm’s progression. It promotes the desired modularity and extensibility while presenting a simplified programming interface.

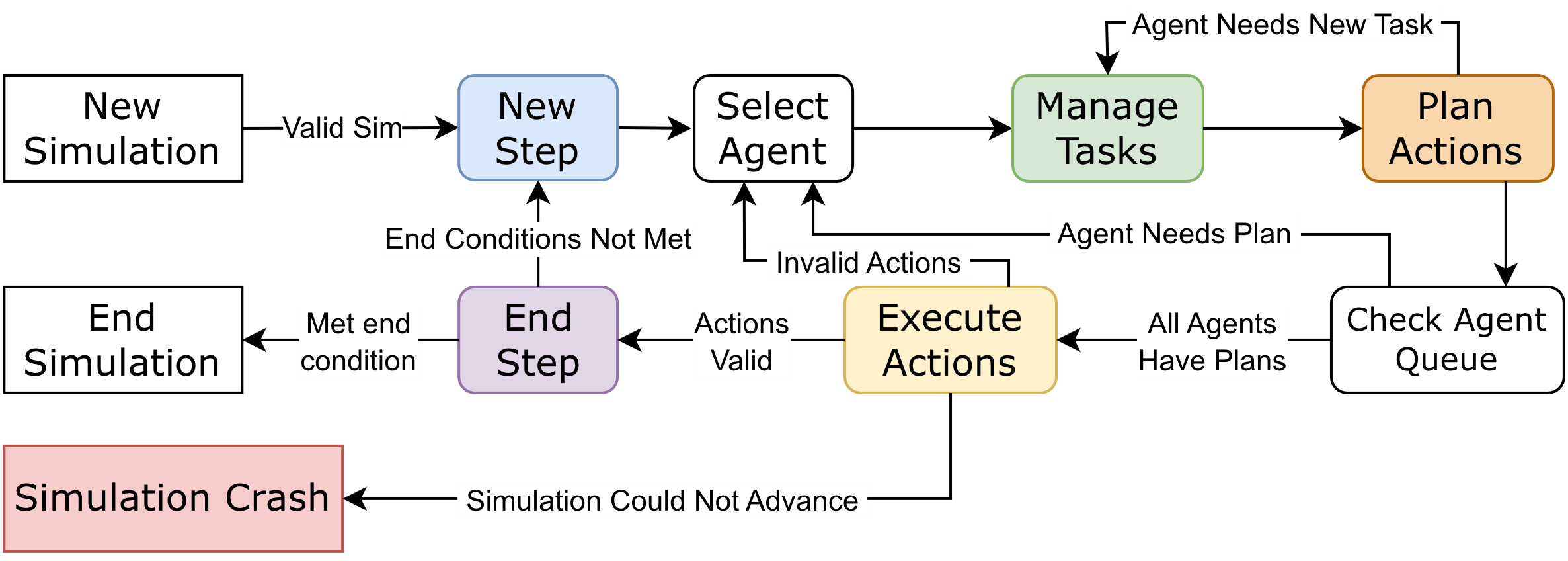


Figure 1: Graphical overview of the proposed state machine.

Actions handling collision resolution, task assignment, and pathfinding are given their own states and can, with sufficient care, be implemented in a manner which meshes well with other algorithms. For instance, the application engineer may be able to implement an optimization in pathfinding heuristics into an algorithm which contains its own optimization for task assignment.

Being only a strategy for implementation, there are no imposed requirements to enable this process in terms of the programming language or the techniques employed within the individual states. This strategy does not preclude the use of any particular algorithm which performs these functions. If the state machine is implemented with a provision for an algorithm to request a specific state, additional logical branches in the execution of the algorithm are trivial to add, further extending the functionality of the system. FleetBench is an application described in Chapter ??? which is driven by this approach in its implementation of the algorithms described in Chapter ??: WHCA\*, TPTS, and their ancestors.

The following sections provide brief overviews of the behaviors which are found in each section of the state diagram. The programming implementation of the state machine is left to discussion in the Appendix.

**Simulation Definition**

Systems which can be represented in this manner are complex and dynamic, presenting many opportunities to make impactful decisions. Before the simulation begins, there are a number of choices to be made which act as defining “rules” for the simulation. Examples include:

* How many actions can an agent take per timestep?
* Does interacting with a task point consume a timestep?
* Does rotation have a cost?
* Do agents experience faults during operation?
* How are tasks added to the system?

These types of restrictions apply globally to the simulation and must be respected throughout its lifetime. These rules must be configured during the simulation’s setup state before other operations begin. Perhaps the most important consideration is what constitutes the end of the simulation if the solver is not intended to run for an indeterminate amount of time. These configurations must be available at all times in the simulation to inform the logic of the algorithms.

This state is also an appropriate time for algorithms to execute any routines which preprocess the graph, as in the case of TP and TPTS. If any such routines fail to execute, some kind of logic must be implemented. For example, the TPTS algorithm comes with notions of what a well-formed MAPD problem is. If the simulation is run on an MAPD problem which does not meet these conditions, guarantees about completeness are revoked. In such cases it may be preferable to warn the user or abort the simulation entirely.

**New Timestep**

At each new timestep there is an opportunity for the system to be updated with new information, informing the behaviors taken during the timestep which is being simulated. Typically this new information will be composed of new tasks, either generated on the fly or as part of a predefined schedule to be released at a particular timestep. Other events such as agent breakdowns, changes in operating strategy, or the introduction of additional agents to the system could also occur here.

In a more pragmatic sense, this state is also a good place for handling various programmatic concerns.

**Task Management**

Interactions with the task set minimally consist of two operations: task generation and task assignment. Task assignment is the act of designating a particular agent as the executor of a particular task. Task generation enters a new task to the set, whether via generating a completely new task or introducing a defined task according to some task schedule supplied in the simulation definition step.

To maintain the analogy of the system to a real-world application, an external authority should manage task-defining processes. A warehouse system would require knowledge of an item’s location and destination, while an air traffic control system may need to enforce timing constraints by restricting the availability of “tasks” to certain timesteps. In these cases, tasks need to be admitted to the system in an online fashion, which is left up to the engineer. For the testing process, it is likely sufficient to implement a custom generator or use a predefined list of tasks.

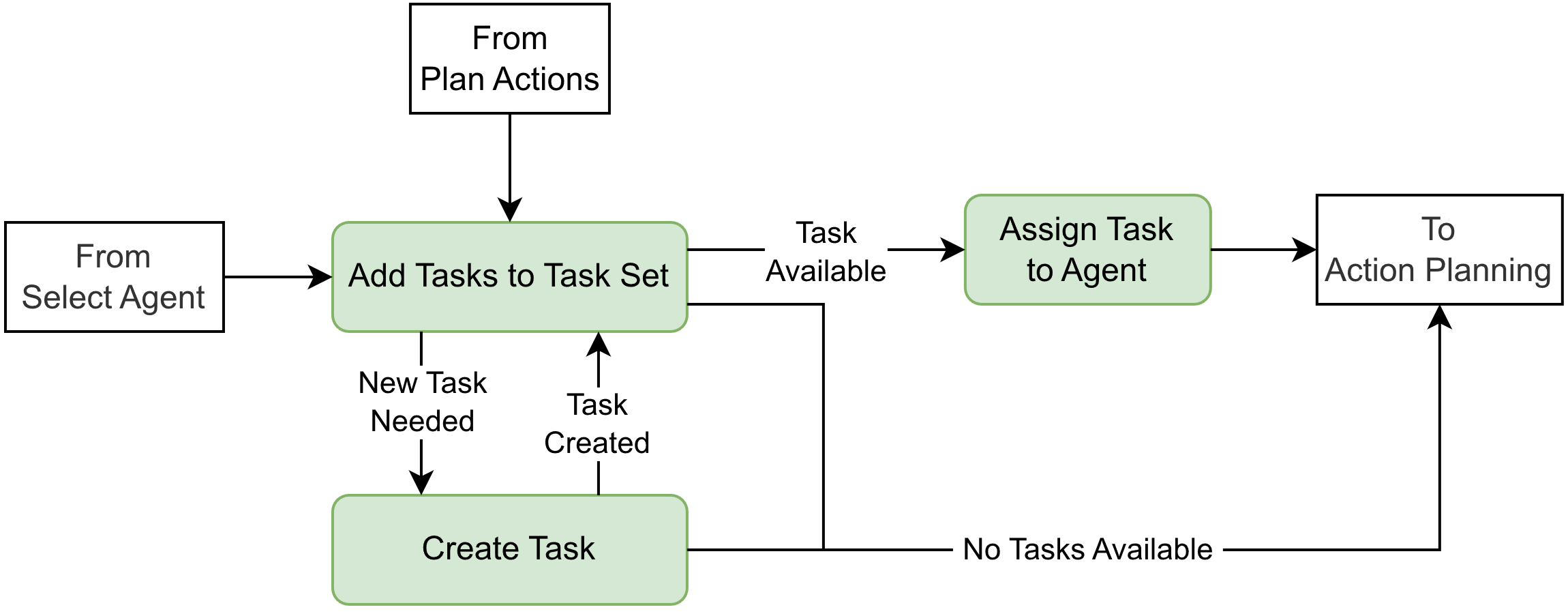


Figure 2: Graphical overview of the process undergone in the Task Management state.

**Action Planning**

During each timestep an agent must take some kind of action, even if the action is to wait in place. The act of determining which action best advances the system toward a solution state will account for the majority of an algorithm’s procedure and is therefore the most involved aspect of the implementation process. Following a similar decomposition process makes it clear that an algorithm must be able to make decisions regarding certain functions: fulfilling task requirements, finding paths, and acting on planned paths. These are grouped together as logical branches of the action planning state.

Generally, an agent which is in its goal location should attempt to fulfill its objective by performing its tasked behavior. In simplistic implementations, this could be merely being in the goal node at some point in time, but the approach makes no assertions that this must be true. For example, the execution of a task could be sufficiently complicated and time-consuming that it requires multiple timesteps. In such cases, the planning algorithm must compensate.

Agents which are not where they need to be should be driven closer to their goals while adhering to other system requirements (chiefly, no collisions along their paths). If an agent already has a plan and there are no immediate problems in continuing to execute the plan, the default case should be that it advances along its plan. This framework does not prevent the implementing engineer from altering plans in an online fashion, remaining flexible in the case where an auxiliary goal should be achieved, such as avoiding future congestion. Alternatively, if an agent has no plan at all—as may be the case immediately after the assignment of a new task—it should attempt to find a valid plan.

An implementation concern arises. Path planning operations may fail in certain cases. For example, an agent may not be able to take any action without colliding with the intent of another agent. A complete path may also not exist under current system conditions. In such cases it is possible a bounded path search could find a partial path as in the case of WHCA\*. Once again, the state machine approach does not offer restrictions on how such cases should be handled, although the implementation in this work uses a collision resolution system which takes effect once all agents have declared their intents.

The agent queue should be exhausted by the end of the action planning phase, with all agents having declared some intent to take a particular action. All that remains from this point on is the validation and execution of these planned actions.

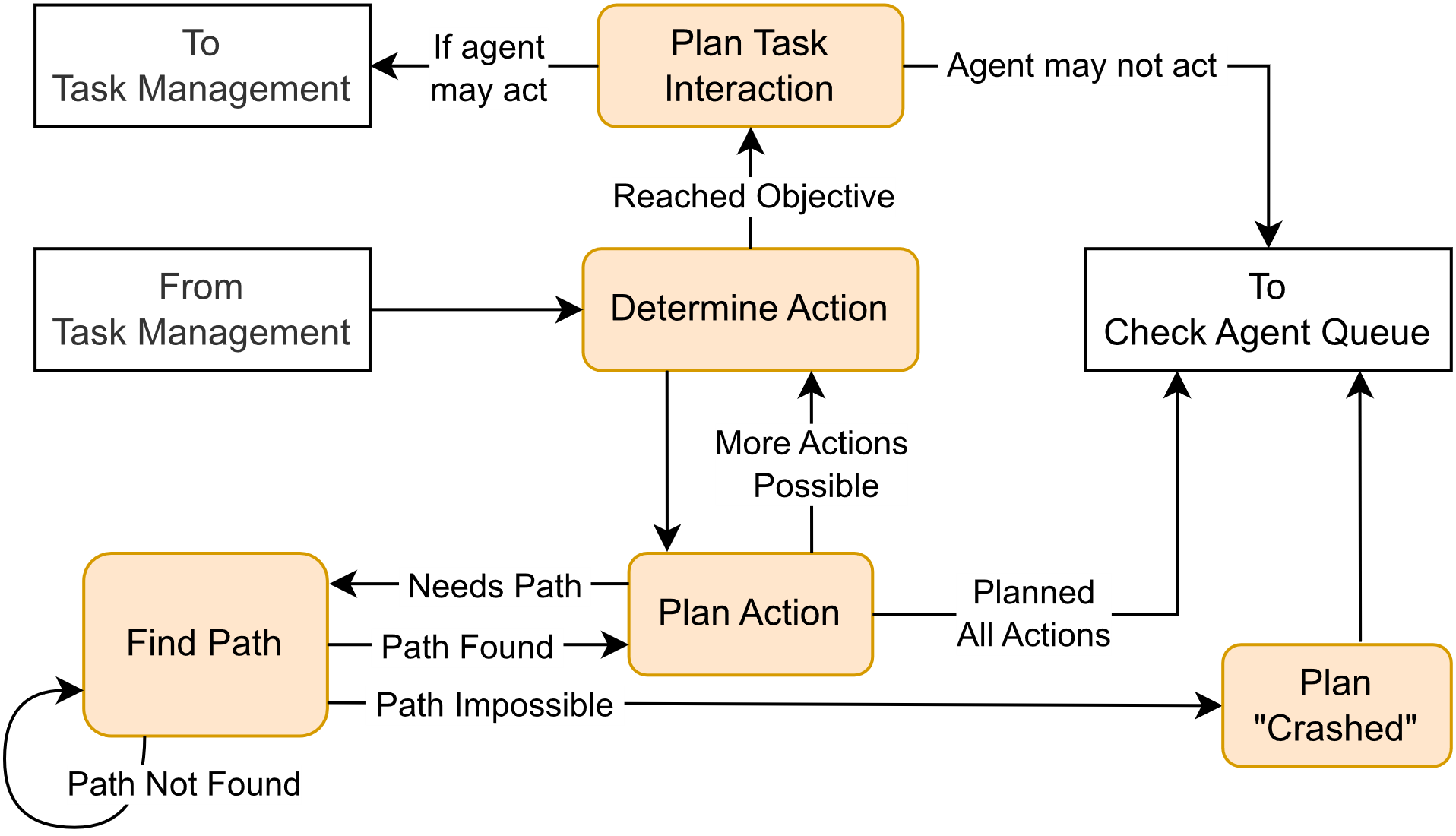


Figure 3: Graphical overview of the process undergone during the Action Planning state.

**Action Execution**

With a set of actions found for each agent in the system, the algorithm should be able to advance by executing all agent actions simultaneously. Successfully doing so represents the passage of a timestep.

As mentioned in the Action Planning section, it is possible that generated agent plans are insufficient in some way that causes collisions or other errors. As both a pragmatic and performance concern, such occurrences must be handled. Simulations which crash on the first instance of incompatible plans provide little to no data about the functionality of the algorithm in other cases. Further, not every algorithm is well-adapted to the MAPD challenges, as will be shown with the implementation of WHCA\* in Chapter ???. Collecting data on the occurrence rate of failures throughout the simulation lifetime is probably useful and so the approach includes handling these cases.

If no disallowed collisions exist, then the implementation of the algorithm must be able to mutate the state of the simulation in a manner corresponding to the planned agent actions, at which point the timestep is considered completed.

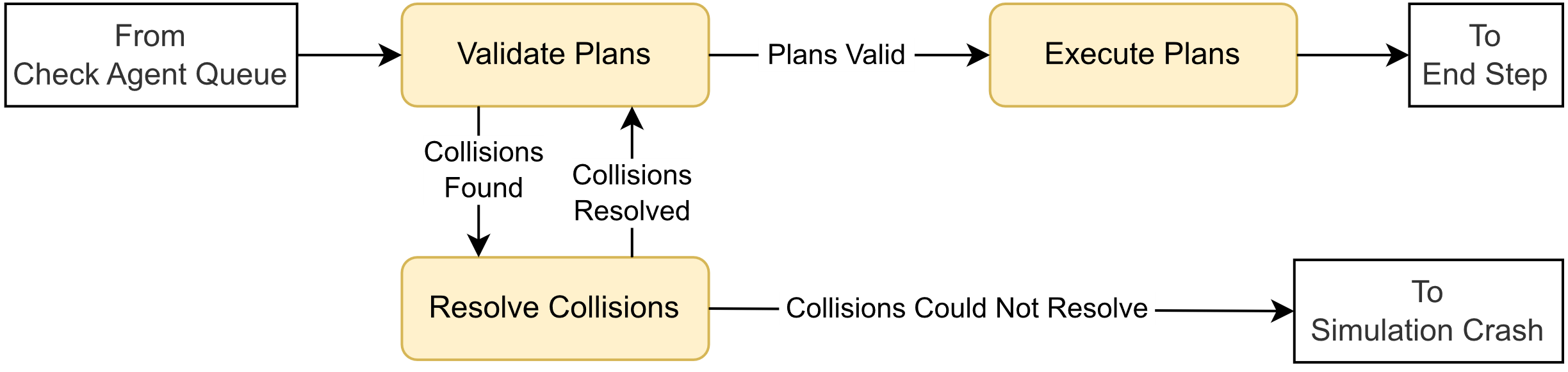


Figure 4: Graphical overview of the process undergone in the Action Execution state.

**End of Step**

To account for simulations or tests which have a defined endpoint, a branch in logic must be introduced which evaluates the current simulation state against the set requirements for completion. Simple conditions upon which a simulation should end include elapsed timesteps, completed task counts, and unresolvable collisions. Equalizing end conditions ensures a level playing field for comparison of multiple algorithms. In the case of simulations which should run forever, it is sufficient for the ending criteria to always evaluate to false, thus keeping the system looping through new timesteps for an arbitrary amount of time.

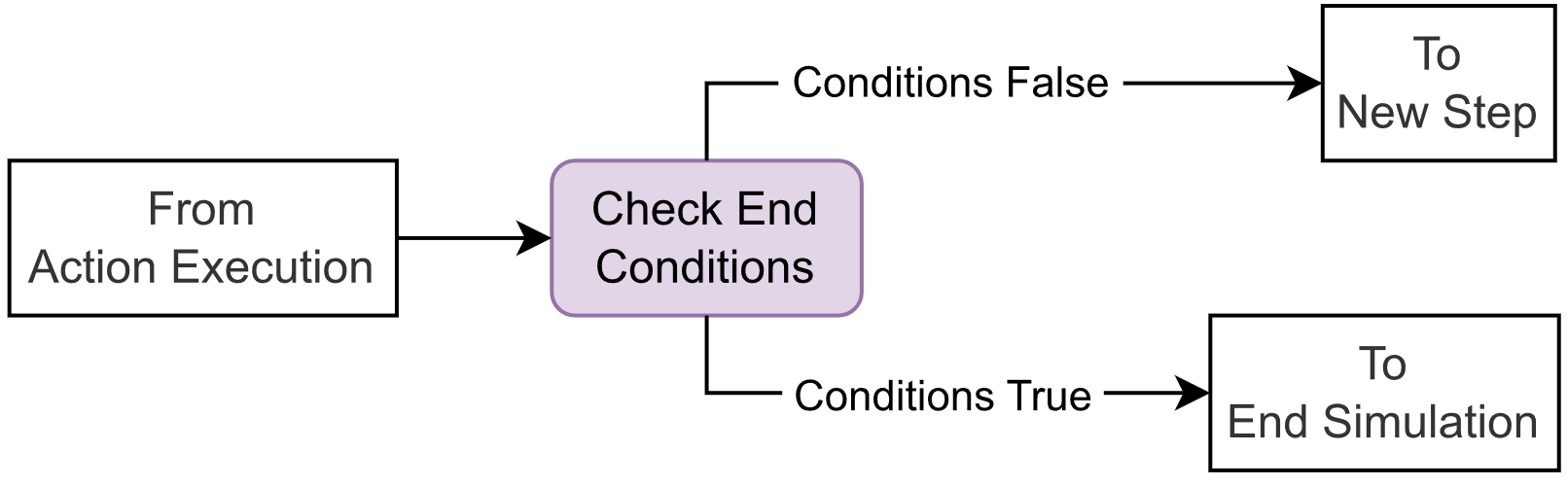


Figure 5: Graphical Overview of the process undergone during the End Step.

[1] O. Salzman and R. Stern, “Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems,” *N. Z.*, 2020.

[2] D. Silver, “Cooperative Pathfinding,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 1, no. 1, pp. 117–122, Sep. 2021, doi: 10.1609/aiide.v1i1.18726.

[3] 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

[4] R. Stern *et al.*, “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks,” *Proc. Int. Symp. Comb. Search*, vol. 10, no. 1, pp. 151–158, Sep. 2021, doi: 10.1609/socs.v10i1.18510.