AUTOMATICALLY DETERMINING CONSEQUENCES OF UNEXPECTED EVENTS

by

Brian C. Becker

A thesis submitted in partial fulfillment of the requirements
for the Honors in the Major Program
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
and in The Burnett Honors College
at the University of Central Florida
Orlando, Florida

Spring Term 2007

Thesis Chair: Avelino J. Gonzalez, Ph.D., P.E.

ABSTRACT

Planning is essential for an action-oriented, goal-driven software agent. In order to achieve a specific goal, an agent must first generate a plan. However, as the poet Robert Burns once noted, the best laid plans can often go awry. Each step of the plan is subject to the possibility of failure, a truth particularly relevant in the real-world or a realistic simulated environment. External influences not originally considered can often cause sudden, unanticipated consequences during the execution of the plan. When this happens, an intelligent software agent needs to answer the following important questions: What are the consequences of this event on its plan? How will the plan be affected? Can the plan be adjusted to accommodate the unanticipated effects? The research described in this thesis develops a model whereby intelligent agents can automatically determine consequences of unplanned events. Such a model provides agents with the ability to detect if and how events will affect the plan. This allows agents to subsequently modify the plan to mitigate unfavorable consequences or take advantage of favorable consequences.

DEDICATION

To God,

Who knows it all anyhow

ACKNOWLEDGMENTS

I wish to express my heartfelt thanks to my advisor, Dr. Avelino Gonzalez, for his extensive support throughout my undergraduate career. Dr. Keith Garfield has been indispensable with his management of the project and his keen insights through our brainstorming sessions. Many thanks go to Dr. Michael Georgiopoulos and Dr. Annie Wu for serving on my thesis committee. I am especially grateful for the financial support from the Army Research Laboratory and the Institute of Simulation & Training (IST) to fund this project.

I also thank my fellow thesis/dissertation writers Enrique G. Ortiz, Chris Moriarty, Cameron Akridge, Gary Stein, and Jimmy Secretan for words of encouragement and cracking the whip when my dedication wavered. Special thanks go to Jorge Cham and his “Piled Higher & Deeper” comic strip for keeping me morosely entertained during the sometimes all too serious and arduous process of writing a thesis.

Finally, I am greatly indebted to my parents who put up with me for these long years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations

AI	Artificial Intelligence
CBR	Case-Based Reasoning
CxBR	Context-Based Reasoning
CxG	Contextual Graphs
DoD	Department of Defense
HCI	Human Computer Interaction
SAF	Semi-Automated Forces
SIM	Situational Interpretation Metric
wff	well-formed formula

CHAPTER 1: INTRODUCTION

Planning, the act of choosing a series of actions to bring about desired changes, is an indispensable process when attempting to achieve goals. In the real world, planning varies in complexity from simple to-do lists to massive organizational strategic plans. In fact, nearly every aspect of life involves some form of planning, even the mundane trip to the grocery store. As noted in [1], a person plans "to increase his company's volume of business, or to win an election, or to write a letter, or to build a bridge…he plans all the time. By his very nature every man plans constantly." Not only do individuals benefit from planning, but studies [2] show that organizations such as businesses with written, formal plans have a higher chance of succeeding than those that operate on-the-fly. Thus, planning is a foundation upon which intelligent human and even organizational behavior is derived.

In a computer simulation, agents might range from operating purely in an instinctive, reactive manner to sifting through millions of possibilities to determine the best possible sequence of moves. Interactive computer-aided planning is especially useful in domains where "what-if" scenarios assist with the design of procedures and schedules. Automated agent-based planning includes applications in scheduling where planners manage machinery in a manufacturing environment [3], chess playing where extensive sets of possible future states are considered before committing to a move [4], and motion planning where robots calculate the best path to take in order to reach a destination [5].

One caveat is true for both humans and computers: any plan assumes certain predictions about the future. A mental to-do list that just includes buying some groceries presupposes no car accident will occur on the way to the store. A plan of attack from a chess playing agent assumes the opponent will not abandon her current well-fortified position and strategy. As a plan unfolds, the pliant future may unveil events not previously expected. These interruptions may prevent the plan from executing smoothly. For instance, you notice a piece of plywood in the road ahead while driving to the grocery store, leading you to perform evasive action to prevent a possible flat tire. Or in a game of chess, the opponent melds her approach with a completely different approach, achieving a surprise capture. When unanticipated events such as these occur, a human or computer agent must be able to recognize what effects, if any, the event might have on the plan. In short, the agent should be able to answer the question "What are the consequences or implications of a given event on the plan?"

The ability to see the implications of an unanticipated event is a skill humans use almost unconsciously throughout any given day. If by chance you meet an old friend in the grocery store and engage in a lengthy conversation, you might begin to mentally tabulate how this interruption will affect the rest to your day. Perhaps that errand next on your to-do list can be done after lunch. Your lunch plans themselves might change to include your rarely-seen friend. These plan modifications come naturally in the case of a morning delay; however, in the case of a missed delivery on a complex construction project or an unexpected tactical maneuver in a battle, the implications and subsequent plan changes may not be as easy to determine. In addition to being more difficult to calculate, the consequences of these types of plans may be much more serious, as unexpected tactical maneuvers can lead to casualties. Regardless of the difficulty, humans possess techniques and heuristics that deal with determining the implications arising when unexpected events occur, while intelligent computer agents struggle with seemingly simple cause and effect scenarios in a dynamic world [6].

This research develops a model that endows computer agents the reasoning faculties to understand consequences of unexpected events. Through inspection of events occurring or known to occur in the future, agents can determine whether these events have any effect on the plan, and if so, what those effects might be. This model proposes to draw heavily from several preexisting paradigms: primarily Context-Based Reasoning (CxBR) [7], Contextual Graphs (CxG) [8], and GraphPlan [9]. While GraphPlan excels at creating plans in a static environment, CxBR and CxG provide an approach for executing a plan based on the continually changing state of the agent and the environment. Thus, an agent operating within these paradigms moves between contexts (in CxBR) or nodes on a graph (CxG), allowing the agent to intelligently adjust its behavior to a particular situation. GraphPlan provides information about the relationships between situations, aiding reasoning about events. Through adaptation and extension, a model that combines the flexible, dynamic execution flow from CxBR, the straightforward representation of CxG, and the explicit causality linkages within GraphPlan can achieve the goals of this work.

This thesis continues in Chapter 2 by reviewing background material relevant to understanding consequences. Both a historical perspective of planning and a state-of-the-art literature review of current approaches to reasoning about consequences are presented. Chapter 3 succinctly describes the problem faced, the hypothesis on which this work is based, and the potential contributions resulting from the investigation. Chapter 4 develops the formal model that provides the foundation for the prototype implemented in Chapter 5. Chapter 6 describes the tests performed using the prototype to verify the correctness and usefulness of the model. Concluding in Chapter 7 with a discussion of the testing results, this thesis summarizes the project and offers insights into future work that may further improve the subject field.

CHAPTER 2: BACKGROUND

To better appreciate the problem domain of automatically determining consequences of unplanned events in a real-world environment, a two-part background is necessary. The first part of the background introduces the topic with a historical perspective. Because agent behavior and reasoning is of particular interest to this investigation, agents, planning/replanning, and multi-agent systems are overviewed. The second part delves deeper into areas of planning by examining the current literature related to reasoning about consequences. Two approaches represented include contingency plans and dynamic analysis of event effects.

2.1 HISTORICAL PERSPECTIVE

Simulating agent behaviors in general can be approached from several different levels. At the lowest cognitive level, agents act solely on instincts (reflexive agents) [10]. Agents behaving on instincts operate completely in the “now,” without thought to the past or future. They observe the environment and react with solutions designed to satisfy a perceived want or need without regard for the future consequences of such actions. On a higher level, reactive agents [10] may be programmed to “feel” emotions or motivations that provide contextual information to limit the available actions, weighting an agent’s action selection mechanism toward certain tendencies. For example, feeling fear places constraints on the actions the agent can perform. Even if an attractive option becomes openly available, the feeling of fear may cancel any distracting thought of taking advantage of the attractive option by keeping the agent concentrated on “flee or fight behaviors” [11]. At the highest level, agents utilize cognitive abilities, considering the past, present, and future to determine what behaviors to adapt (reflective agents) [10]. This is the level at which agents possess the cognitive abilities to project the consequences of actions into the future and formulate plans. Behavior based on the first two levels (reflexive and reactive) do have a place in software agent behavior [12] and can be used as the basis for other Artificial Intelligence (AI) algorithms. However, it is not suitable for general purpose planning [13]. For effective planning, agents must operate primarily at the highest level of cognitive reasoning.

2.1.1 WHAT IS PLANNING?

Wezel, Journa, and Meystel [1] define a plan as a goal state and the methods of achieving that goal state. This corresponds to the “what & where” and the “how & when” of the plan [14]. They go on to postulate that the act of planning requires three components:

1. An entity to develop a plan
2. An entity to execute the plan
3. The plan itself as represented as the communication between the two entities

It is not necessary for the two entities to be separate; the creator and executer of the plan can be the same entity. However, when implemented in a software agent, the module that develops the plan is often separate from the module that executes the plan. Thus, the separation of the creation and execution of the plan is an important distinction.

2.1.1.1 Model of the Future

The process of planning can be thought of selecting a sequence of particular actions out of a large number of possible actions in order to achieve some specified goal [1]. Competent selection and successful execution of actions should, in an ideal world, eventually yield the goal state. One of the major problems with planning is the fact that any projection of actions into the future requires a model of the future [1]. Because the future does not yet exist and is subject to uncertainty, the success of a plan depends on the quality of the model of the future. How a plan is affected by the model of the future is easily seen in two extremes of human outlook:

- A pessimist, generally described as a gloomy person, always expects the worst possible occurrence
- An optimist views the world through rose-colored glasses, often ignoring the harsh reality

Typically, neither of these two extremes represents an accurate model of the future. Consider a plan that includes two alternative activities: an outing to the beach or a shopping trip to the mall. When developing the plan, the model of the future is very important. If you ask the pessimist, she would say it is probably going to rain and thus recommend the shopping trip. If you ask the optimist, she would be much more inclined to disregard the possibility of it raining and recommend the beach trip. If you were in charge of the day's plan, which model would you use? Perhaps a better model of the future might be the weather forecast for the day. It is interesting to note that the most useful model of the future may differ depending on the type of planning. Military planning may benefit from a more pessimistic model in an attempt to always anticipate and forestall the worst possible scenario. Regardless of the application domain, planning is highly influenced by the agent's perception of the future.

2.1.1.2 Hierarchical Planning

Planning, by its very nature, is a hierarchical process [1]. Initially, the plan is conceived using a very coarse granularity. If planning to do some errands around town, the initial plan may only consist of a list places to visit. For example, one goal of the plan might be to buy food for supper. Thus, the place "grocery store" may be part of the initial plan, yet simply visiting the grocery store will not accomplish the goal. As the plan is developed, it must be further expanded to a finer granularity. Each place to visit may be expanded to include a sub-list of tasks to perform. In the case of the grocery store example, the tasks may include selecting a recipe and creating list of ingredients to buy. In general, a single action may represent a group of sub-actions; this grouping is referred to as *aggregation of actions*. As the plan is developed, each aggregation of actions is dis-aggregated, giving the plan a finer granularity of details. This enhances the efficiency of the planning process. Instead of initially attempting to manage myriads of unrelated pieces of information, details are expounded only after the structure of the plan has been developed. In fact, some small details of the plan may be disaggregated only moments before execution, such the exact aisle and path to take in the grocery store.

2.1.2 What Is an Agent?

The definition of an agent is of particular interest when dealing with agent-based systems. What entities can be classified as agents? By what criteria is intelligence judged? One definition postulates that it is the possession of several characteristics such as autonomy, social ability, reactivity, and pro-activeness that differentiates an agent from a software entity [15]. This definition may be expanded to a more general one: an entity can be considered an intelligent agent only if it exhibits the mental or emotional characteristic normally attributed to humans [15]. For the purposes of this discussion, an agent will be defined as an intelligent software entity with a purpose (i.e. it has some goal to achieve) and the ability to change the state of the environment through the intelligent execution of actions.

2.1.3 What Is Replanning?

Because planning deals with future expectations in an uncertain future, the actual unfolding of the future may deviate from the agent's expectations, or model of the future. These deviations from the agent's model of the future result from the non-deterministic nature of a real-world environment or from the interference of another agent's actions. In short, the real world deals with probabilities – nothing is certain. Entire branches of mathematics such as Bayesian statistics deal with the likelihood that a particular situation will occur [16].

Oftentimes, these unexpected events will interrupt the plan in some fashion. The severity of the interruption may vary from a slight inconvenience to a complete plan failure if the action sequence is continued without modification. This is not to say that all unexpected events bring unfavorable interruptions; in fact, unexpected events may aid the agent in achieving the goal. Nonetheless, agents are typically more concerned with obstacles to the plan. Because of these uncertainties, any planning agent operating in the real world or a simulated environment that accurately mimics the real world must account for error between what the agent expects to happen and what actually does happen. Once this error has been detected, the agent must re-analyze the plan and make corrections, including the possibility of generating a completely new plan. This adjustment of the plan, or "course correction," is referred to as *replanning*.

2.1.4 Multi-Agent Systems

Currently, research into agent based systems is focused primarily on multi-agent systems. While single agents can accomplish complex objectives, more comprehensive objectives often require the teamwork of several agents. Additionally, the interactions and cooperation among multiple agents are fascinating not only to computer scientists, but also those in the fields of psychology and sociology. Teams consisting of humans and software agents pique the interest of Human-Computer Interaction (HCI) experts in addition to organizations such as Department of Defense (DoD) as they investigate integrating semi-autonomous forces (SAFs) into the military [17]. Furthermore, multi-agent systems facilitate the development of simulations where more than a single entity must be represented, such as is the case when training soldiers with teams of intelligent agents acting as the opposing force.

In spite of multi-agent system's popularity, single agent systems are still useful and are particularly beneficial when developing and testing a new reasoning model. During the construction of any new software, test cases must be built up gradually and incrementally to verify each part of the software. Thus, the scope of this thesis is limited to developing a reasoning model for a single agent. The goal of the model presented in this thesis is to develop an initial version of the model and verify that the model is sound, both of which can be done satisfactorily with a single agent system. Because of the usefulness of multi-agent systems, one cannot simply ignore them. Thus, the eventual goal of the model is to support multi-agent systems. In order to develop a model that can be easily extended into the domain of multi-agent systems, the investigation reported in this thesis keeps the requirements and opportunities available in multi-agent systems in the background of all choices and considerations. As a result, when developing the model, preference was given to cases where initial model features or structures could easily be extended to facilitate future multi-agent expansions.

2.1.5 Classical Planning Approaches

In 1969, McCarthy and Hayes [18] introduced *situation calculus*, a conceptual view of planning that serves as the foundation for may subsequent planning approaches. Situational calculus represents the planning environment through situations, goals, and the effects of actions. It provides a clear methodology for planning entities to project a plan into the future by manipulating the world state through the execution of actions. This concept is used widely in *state-based* planning systems where a planner executes actions that cause a series of world state changes as a means of achieving a specific goal [19].

2.1.5.1 STRIPS

In 1971, Fikes and Nilsson [20] borrowed concepts from situation calculus and proposed a new approach to planning and problem solving by introducing STRIPS, the STanford Research Institute Problem Solver. Through a world representation that has since become commonly referred as a "STRIPS domain," the problem solver describes the environment through a "composition of operators that transforms a given initial world model into one that satisfies some stated goal condition" [20].

The world model is represented through facts and conditions, or well-formed formulas (wffs) and first-order predicated calculus, respectively. Agents or the global problem solver can use operators, which correspond to specific actions that have certain, pre-defined effects on the world model. These effects fall into two categories: effects that add or remove wffs. Changing a condition in the environment would thus be represented via two effects: one that removes a pre-existing wff and a subsequent one that adds the same wff with a new value. Because operators can execute only under the presence of predefined wffs, operators are said to be executable only when certain preconditions in the world model exist. Given an objective, or final world conditions that must be achieved, an agent searches through the problem space for sequences of operations that lead towards the goal.

2.1.5.2 GraphPlan

Building on the work of Fikes and Nilsson, Blum and Furst [9] extended STRIPS with an improved planner called GraphPlan. Due to the increases in computer memory and computational power, GraphPlan utilizes a Planning Graph, which concisely organizes operators and their preconditions and effects in a graph. Each node in the graph holds an action (operator) and each edge defines the preconditions and effects associated with the connected actions. As in STRIPS, effects are defined as add-effects and delete-effects. The graph is built in layers, beginning with the initial state of the world where the agent exists in the current time slice. As compatible operators are selected and added to the graph, layers are built, with each layer representing all the possible actions for a particular step in the plan. Once the graph is built, the planner begins with the end goal and works backwards in the graph to trace what actions must be executed to bring about the goal state. During this procedure, mutually exclusive actions are eliminated to ensure plan validity. The path of actions necessary to bring about the goal state is selected as the plan to execute.

The advantages to this approach are several-fold. First, graphs can be built quickly in polynomial time. Second, the graph organizes the problem space in a logical fashion, linking actions, their prerequisite conditions, and the effects. However, most importantly, by representing the problem space in a graph, pre-existing graph algorithms can be used to efficiently analyze and rank various plans. In terms of the determining consequences, by explicitly defining causality between an agent's action and the effects that occur in the world, reasoning about unexpected events becomes quite intuitive. However, GraphPlan has no built-in method to handle such reasoning. Typical implementations of GraphPlan simply respond to events by taking a snapshot of the current world and using that as the initial conditions when executing the planner again. Furthermore, GraphPlan lacks the ability to reason about the time and resources required to carry out actions.

2.1.5.3 GraphPlan Variants

Since the introduction of GraphPlan, numerous variants have been developed to address limitations within GraphPlan. Krogt [21] addresses the lack of intelligent replanning by proposing the Action and Resource Planning Formalism that attempts to incrementally modify the plan so as to reduce replanning time and prevent radical shifts in an agent's behavior. To facilitate planning situations where achieving the objective quickly is desirable, Dinh [22] replaces the instantaneous execution of actions in GraphPlan with action durations and incorporates optimizing features into his system, called CPPlanner, to find plans that require the least amount of execution time. Because the original GraphPlan did not deal with real-world uncertainties, Blum [23] extends actions to include multiple probabilistic effects and improves the plan evaluation to search for plans that have the highest likelihood of succeeding. In effect, approaches such as these represent a shift away from classical planning where knowledge is perfectly known, outcomes are purely deterministic, and no unexpected events occur.

2.2 STATE-OF-THE-ART REVIEW

As classical planning and replanning paradigms proved insufficient to solve real-world problems, newer approaches began to assimilate more realistic models of the real world. One aspect of particular interest that models have incorporated into the planning phase is intelligently responding to unexpected events that occur during the execution of the plan. The next sections delve deeper into the general problem of planning and replanning to review how current state-of-the art literature has approached this specific problem of reasoning about unexpected events and determining their consequences. Currently, two general approaches to handling unexpected events exist: activating pre-programmed contingency plans or dynamically selecting new actions based on analysis of event effects. Each will be examined in subsequent sections.

2.2.1 CONTINGENCY PLANNING

One approach to dealing with unexpected events is to specify pre-determined *contingency plans* that handle events. If actions or situations commonly fail for well known reasons stemming from an event occurring, an agent can be programmed to recognize these events and avoid them by executing an alternative, or contingency, plan. A situation where such a contingency plan might be helpful in the real world is in a checkout line at a store. If the credit card won't work or an item won't scan properly for the person in front of you, you would recognize that this event will have certain effects on the plan; namely, your checkout procedure will have an unspecified delay. When designing a planning system, a programmer can hardcode modules that detect events such as these, calculate the consequences, and respond with pre-defined contingency plan. For instance, the planning system may include the contingency plan to leave the delayed checkout line and join another one.

While adding predefined contingency plans may be advantageous to the agent, the primary problem with this approach is that it masks the underlying problem. The true problem is that agents cannot automatically determine the consequences of unexpected events. By manually adding contingency plans, a human is performing all the cognitive work of programming the planning system to recognize and respond to events. In this sense, unexpected events as seen in these systems are no longer really all that unexpected because the planning system realizes they will inevitably occur. The problem is further exacerbated by the fact when a truly unexpected event occurs, the system has no methodology do deal with it. As a result, if no contingency plan has been hard-coded into the system for a particular event, the planner cannot reason about any of the potential consequences of that event.

While it is true that implementing pre-determined contingency plans may not be the best or latest solution to the problem of automatically determining consequences, it lays the foundation for more effective solutions and must therefore be carefully investigated.

2.2.1.1 Contextual Graphs (CxG)

While not designed to serve as a planning paradigm, the approach of recognizing consequences of events and responding to them in a pre-determined manner can be implemented through the use of the AI paradigm Contextual Graphs (CxG). Originally designed to specify human reactions to incidents that can take place in particular situations, CxG represents this knowledge using contextual cues. Brezillon [8] defines a contextual graph as “an acyclic directed graph with a unique input, a unique output, and a serial-parallel organization of nodes connected by oriented arcs”. The CxG reasoning process begins with a unique input node, traces a single path through the contextual graph, and ends when a unique output node is reached. Each type of node in the graph represents an action, a sub-graph, a decision point, or a recombination of a previous decision. By combining these contextual elements, a plan or process resembling a flowchart is built by a contextual graph designer to handle a particular situation.

One of the strengths of CxG is its adaptability to incremental knowledge acquisition. The designer of the contextual graph can easily extend the graph to include contingency plans. Once the graph is redeployed, the graph can handle pre-determined events. As a result, contextual graphs are well-suited for facilitating the construction of a plan and then reasoning about what to do when a plan interruption occurs – as long as a contingency plan exists to handle any situation the agent may encounter. However, CxG suffers from several drawbacks. While CxG’s can reason about the effects of events occurring in the immediate future, it has no capability to project the effects of events into the far future and avoid potential trouble spots. Another major disadvantage of CxG is its inflexible nature and its inability to respond to any events not preprogrammed. This greatly reduces the usability of Contextual Graphs in an automated system. Finally, because contingency plans are encoded into decision nodes, they can only be activated at particular points in the plan.

2.2.1.2 Situational Objects

Working towards the goal of developing an approach that allows contingency plans to be activated when the need arises, Interrante [24] introduces the concept of *situational objects* in her model of selective attention to sensors. In this model, Interrante aims to limit sensory overload by analyzing only those sensors relevant to the current situation. To accomplish this, she develops a set of situational objects that control the agent in particular situations. Deriving information from relevant sensors, an agent can develop a set of expectations about the future state of the world on a global timeline. Using current sensor information and the expectations of the future, transitions between situational objects occur to keep the agent's behavior optimal for each type of situation it encounters. If at any time the agent predicts that a critical failure will occur soon, an object from a separate bank of situational objects reserved for emergencies can be activated. These situational objects represent contingency plans and respond to pre-determined events sensed through expectations on the global timeline. The timeline functions as the method for determining consequences and identifying events. For instance, an airplane might initiate the contingency plan for an emergency landing when the expectation timeline indicates a critical fuel shortage as detected through sensor information. As with other approaches to contingency planning, if an unrecognizable situation develops, no functionality exists to reason about the situation.

2.2.1.3 Bayesian Belief Network

To dynamically consider uncertainty in a real-world environment, Blythe [25] proposes a system that models action and event uncertainty through a Bayesian Belief Network. In this model, events have enabling conditions that allow them to occur with a certain probability. For instance, in the grocery store example used earlier, the event *bad credit card* has enabling conditions of agent being in the store checkout line and occurs with a frequency of 0.01. Furthermore, actions are modified so they become non-deterministic, meaning that the execution of an action may result in completely different effects depending on probabilistic functions. For example, the action *walk* typically results in physical agent movement; however, an infrequent, but possible alternate outcome may result in bruising if the agent slips on a banana peel. All the possible alternative outcomes are added to the belief network to create plans that include uncertainty. As alternatives are tabulated and the probabilities are assigned to paths through the belief network, plans that have a high likelihood of success can be chosen. Because causality about uncertain actions and events are explicitly considered, consequences are considered throughout the planning process. It is of interest that contingency plans are considered as the agent plans and may be utilized even if a plan interruption has not yet occurred. This is a byproduct of the attempted maximization of successful plans; if an action results in high probability of failure, that action may be avoided altogether and an alternative contingency plan might be selected. Ideally, this approach leads to more robust plans by considering unexpected events a-priori and selecting actions and paths that lead to plans that maximize the likelihood of success. However, as before, all unexpected events and their associated probabilities must be pre-programmed into the planning system.

2.2.1.4 Situational Interpretation Metrics

In the field of real-time tactical human behavior representation, Gonzalez and Ahlers [7] proposed a paradigm called Context-Based Reasoning (CxBR). Resembling an augmented finite state machine, CxBR adds the concept of contexts where each context can be likened to a state that addresses the needs, actions, and actions necessary to act intelligently in a particular situation. Organized hierarchically, context types include Mission Contexts, Major Contexts, and Sub-Contexts (with Sub-Sub-Contexts, ad infinitum) to successively control finer grained behaviors. Thus, while a Mission Context might specify the overall goals, constraints, and motives of the agent, a Sub-Context may control low-level physical movement. At any given time, only one Mission Context, one Major Context, and an optional Sub-Context may be active. Although CxBR does not explicitly construct plans, Grama et al. [26] describe how CxBR can be used to construct a plan consisting of the sequence of contexts that an agent anticipates using to achieve the objective.

One of the weaknesses of CxBR is the selection and transition of contexts [17]. Historically, this has been accomplished with pre-defined sentinel rules; however, this approach results in rather inflexible transitions, especially in large systems. One solution to this problem, advocated by Gonzalez and Saeki [27], introduces the concept of Situation Interpretation Metrics (SIMs). SIMs quantify the situation in a tangible manner, revealing the difference between the current situation and what the agent's needs. Under their Competing Contexts Concept, this difference between reality and the agent's perceived goals (needs) drive the transition of contexts. By allowing contexts to compete amongst each other for the chance to handle the situation, it is significantly more likely that the context best suited will become active. The context that most effectively addresses the current agent's needs by achieving or partially achieving the goal state is deemed the best context and wins the competition. Through this approach, selecting and transitioning to a context is no longer fixed through pre-programmed sentinel rules, but changes dynamically as the situation and agent needs change.

Salva [17] extends this work in context transitions with the concept of enabling processes and interruptions. Enabling processes can be viewed as the pre-requisites for an agent to transition and remain in a context. Interruptions represent events that disrupt the enabling processes, thus putting the whole plan at risk of incompletion. Any interruption requires a change in the plan. Depending on the implications of the interruption, an alternative context may be available which will still allow the agent to complete its goals. Not only does this approach simplify replanning, but the concept of interruptions carrying specific implications provides an easy method for determining the consequences of events that occur. The limitation of the approach is that interruptions and alternative contexts must be specified a-priori, which is impractical in a dynamic real-world environment.

2.2.1.5 Multi-Agent Architecture

While most contingency planning approaches operate in single agent systems, Micacchi [28] extended these ideas into soft real-time environments with multiple agents. In his a multi-agent system, the different types of agents include a central *coordinator* agent that tasks *worker* agents with goals to achieve. The worker agent is responsible for achieving the goal and reporting back to the coordinator. To operate within the constraints of a soft real-time environment, unexpected events are separated into three categories: opportunities, potential causes of failure, and barriers. While opportunities have no detrimental impact on the plan, barriers prevent the plan from achieving the goal. Potential causes of failures may become barriers in the future if the current plan is continued without any modifications. A list of hard-coded responses, or contingency plans, is associated with each unexpected event. When a worker agent encounters an unexpected event, it first determines whether one of the response contingency plans is appropriate, meaning it will lead to the successful completion of its task. If so, the response that resolves the effects of the event the least amount of time is selected. If no appropriate response exists, the worker agent sends a panic message back to the coordinator and awaits a new task or command, thus passing off the burden of determining consequences. Because each contingency plan is built into the system at construction time, reasoning about the consequences is rather limited.

2.2.2 DYNAMIC ANALYSIS

Dynamic analysis, in contrast to pre-installed contingency plans, attempts to analyze the effects of the unexpected event on the plan and determine how the plan can be salvaged without resorting to pre-defined contingency plans. In short, dynamic analysis extracts more information from the effects of events and processes this information more intensively to develop a new plan that depends only on normal, available actions. If it is possible to repair the plan, the process of dynamic analysis should yield an alternative sequence and/or selection of actions for the agent to execute that will successfully complete the plan goals. While a computationally and cognitively more difficult endeavor, this has the distinct advantage of providing an agent with the flexibility of reasoning about any unexpected event, not just those programmed by the developer. In the previous example of waiting in a checkout line, an event consisted of the person at the front of the line having problems with the credit card machine. In this situation, an agent using dynamic analysis will use the effects of the event (delayed checkout) to determine the consequences (missed movie show, etc). Once the agent realizes the current plan is no longer tenable because it will result in unachievable objectives, the agent can begin the process of trying to adjust the plan by rearranging the order of actions or adding new actions to mitigate the effect of the event. For example, the agent may find that getting out of the line and walking over to another line is an appropriate solution which will allow the agent to achieve its goal. Several approaches can be seen in related literature.

2.2.2.1 External Transitions

Nareyek and Sandholm [29] present the concept of *external transitions* to develop a planner that can discover indirect consequences and exploit the combination of parallel actions and events. Instead of representing an event as a single set of effects that take place at a certain point in time, they define external transitions as the ongoing changes in the world that result from the occurrence of an event. From this, they develop an enhanced rule-based system that can reason about indirect consequences. Indirect consequences form when the effects of simultaneous actions executed by multiple agents and/or events interact to produce results not originally intended. As a plan consisting of actions from multiple agents or events working in parallel is built, the plan is dynamically analyzed for unanticipated results, which are recorded and stored as indirect consequences. Avoiding the consequences of unexpected combinations of actions or events allows the replanning process to achieve the goals of the agent while taking into account unexpected events. Because events are defined similarly to actions in that they have specific preconditions, this method works well to determine side effects, but has some limitations when it comes to determining the consequences of unexpected events. Although this linkage of events to preconditions allows events to activate under known situations, it defeats the purpose of the unexpectedness of events because now the agent knows when particular events will occur and under what conditions. In order for realistic event modeling, events should be able to occur without any apparent reason to mimic the sudden appearance of problems in the real world.

2.2.2.2 SimPlanner

As an extension of GraphPlan, Onaindia et al. [30] presented the SimPlanner simulator that permits an agent to perform planning, execution monitoring, and replanning throughout the duration of the agent operation. This interleaving of planning and execution provides several advantages. First, any change in the environment, such as the occurrence of an unexpected event during execution, is detected by the monitoring module and subsequently incorporated into the plan, invoking the replanning module if necessary. When an unexpected event occurs, the system assumes the current action can finish unaffected, but the remaining action sequence is dynamically analyzed to see how the event possibly impacts the rest of the plan. If the plan is affected as determined by unsatisfied preconditions (i.e. the event prevents future planned actions from executing), the planner attempts to incrementally change the plan so as to follow the original plan as close as possible without sacrificing a more effective solution. In effect, SimPlanner recognizes the consequences of the event and attempts to shift the plan a little bit at a time until the consequences disappear. Once a valid solution is reached, the agent can continue to execute the repaired plan and achieve the goal.

Through this method, SimPlanner can monitor the execution of the plan, anticipating the consequences of events and readjusting the plan as necessary. While this represents a good start, several problems become evident. First, the assumption that the event will not impact the currently executing action may not always be valid. In some situations, such as a missile launch by an enemy agent, the agent may want to immediately terminate the current action and pursue a more reasonable plan of action, such as evasion. Another limitation of the model is its inability to reason about the availability and consumption of resources required by actions. Finally, all actions are assumed to be instantaneous, which limits the feasibility of using such an approach in the real world.

2.3 SUMMARY

Throughout this chapter, planning, agents, multi-agent systems, and classical planning paradigms are described to provide the background material necessary to contribute to the overall understanding of the goals of this work. Additionally, state-of-the-art literature addressing how consequences are detected and resolved has been reviewed. Now that this background has been discussed, the subsequent chapters will draw from relevant ideas and concepts in this chapter to develop a model that automatically determines consequences of unexpected events.

CHAPTER 3: PROBLEM STATEMENT

This chapter presents a concise description of the problem addressed in this thesis.

3.1 PROBLEM DEFINITION

Historically, simulations have provided an abstracted environment in which to test an algorithm or approach without the time or cost overhead of a physical or real-world system [31]. Often, simulations were preferred because they did not suffer from physical sensor deterioration or the noise often found in the real world; in essence, assumptions were made to simplify the research [32] and to cope with limitations in processing power. As a result, agents in a simulation always knew what to expect and how to deal with any situation encountered. However, with demands for more accurate results in realistic environments and the ability to process more data through increases in computing resources, simulations grew more complex to better emulate real-world conditions. This increased availability of high power computing machines resulted in an increasing number of agents moving out of simple simulations into real-world systems, which caused many existing approaches to fail [32]. Analyzing these failures led to the conclusion that the inherent uncertainty of a real-world environment introduced events and situations that the agent was not able to handle. Agents could not effectively operate in the real world because existing models insufficiently addressed issues such as the dynamic nature of the environment and the incompleteness of an agent's knowledge.

The problem this thesis addresses is that when faced with an unexpected event that dynamically affects the environment, how can an agent respond intelligently? More specifically, if an agent becomes aware of an unexpected event occurring either currently or at some point in the future, how can the agent automatically determine the consequences of the unexpected event on its plan in order to replan more effectively? Two important considerations play a role in this problem. First, this process must be automatic, eliminating static contingency plans designed for pre-programmed plan interruptions. Secondly, this process must aid replanning. This means that the results of determining the consequences must be able to be interpreted by the agent. By gauging the future consequences, an agent should be in a better position to make more effective decisions.

3.2 Hypothesis

Agents can automatically determine consequences of unexpected events by utilizing a model that incorporates the plan representation of GraphPlan, dynamic transitions of CxBR, and the relationship debugging features of slicing.

3.3 Contributions

This research into automatically determining consequences of unplanned events can result in several important contributions:

- A new model that provides an effective method for agents to deal with unanticipated situations through a framework that reasons about the consequences of unexpected events

- A test bench prototype from which “what-if” scenarios can be developed, tested, and deployed
- A reusable programming library that other intelligent agents can utilize

CHAPTER 4: APPROACH

Now that the problem has been concisely described and the relevant literature reviewed, this chapter proposes an approach to enable agents to reason automatically about unexpected events occurring during the execution of a plan. Through the development of the approach, the design phase of the proposed model as a reusable library is discussed. First, the purpose and benefits of developing a reusable library for the model is examined. Second, approach requirements are examined to determine what existing paradigms can be incorporated into the solution. Third, the proposed approach framework, named ADCUE, is introduced and overviewed. Fourth, each component in the ADCUE model is presented in detail to completely describe the foundation of the model. Fifth, planning and replanning as related to ADCUE agents is discussed. Sixth, the underlying technique of *slicing* is introduced as the major component of the proposed approach that aids in determining consequences. Seventh, the application of slicing to planning is described so as to facilitate the transition of this technique to a new domain. Eight, pseudo code for the algorithm that determines the consequences of unexpected events is provided to gain a deeper understanding of how the proposed approach works. Finally, examples are dispersed throughout the chapter for clarity's sake.

4.1 PURPOSE

At the outset of this research project, the goal was to develop an initial model that could endow agents with the cognitive abilities to reason autonomously about how unexpected events affect the plan. It was envisioned that this work would be expanded with subsequent research. Thus, in order for further work to be easily added and allow room for model expansion, a clear separation between the testing prototype and the actual core model implementation is necessary. By implementing the model as a library, the initial limited prototypes can later be abandoned while preserving and upgrading the library. Thus, the implementation performed for the purposes of this thesis is a multi-stage process: first, create a generic library encapsulating the model and reasoning methods and secondly, develop test cases. Each test case developed would be a separate application that utilized the functionality provided by the model. By using this approach, anybody desiring to develop a simulation where agents need to reason about the consequences of unexpected events can easily integrate it with the library.

4.2 Combination of Paradigms

When designing an approach to solve the problem of interpreting the implications of unexpected events, preexisting paradigms described in the literature were examined for suitability to the problem at hand. For various reasons mentioned previously, the approaches overviewed in Chapter 2 were found unsuitable. However, while no single paradigm could be applied directly to solve the problem, many of the approaches provided valuable contributions that could be incorporated into a future, more comprehensive model. In order to find the relevant paradigms that apply to the problem, a typical use case is examined and several core requirements of the final desired model are now specified.

4.2.1 *TYPICAL USE CASES*

Two typical use-cases are envisioned. First, automatic consequence determination may be quite useful when creating and testing "what-if" scenarios. In this use-case, a user would create a plan (or help the system to generate a plan) and subsequently inject events into the system to observe the results of "what if so and so happens" scenarios. Employed in this manner, a user is more interested in the plan itself than the actual execution or simulation of the plan; the benefit to the user in this case is testing a plan for robustness under different operating scenarios and correcting weak links in the plan sequence.

A second typical use-case scenario consists of an intelligent agent operating within a real or simulated environment. In this use-case, an intelligent agent is usually generating plans autonomously and must subsequently reason and respond to unexpected events in real-time. When this scenario is utilized by an agent, the execution or simulation of the plan to achieve some objective is considered more important. Here the benefit to the agent is the ability to recognize when unexpected events affect the plan, and then be able to aid the replanning process by determining the effects of events.

4.2.2 *APPROACH REQUIREMENTS*

From the use-cases mentioned in the previous section, it is clear that any developed model that automatically determines the consequences of unexpected events must satisfy several requirements. These requirements were used to evaluate existing paradigms for suitability to the specific problem at hand. Five core requirements of the final system were drawn up. For each of the first four requirements, a paradigm has been selected to address the requirement and to be incorporated into the approach. The requirements are as follows:

1. **Plan Generation/Representation:** Before any reasoning about events can take place, a plan must be provided by the user or generated by the agent. A plan must consist of a sequence of steps using environmental components such as goals, actions, and resources available.

   Selected Paradigm: Although it has some fundamental limitations as described in the literature review, GraphPlan is a well-respected planning approach in the STRIPS domain that can be used within static environments to build a plan. The representation of a plan in a graph format with each node representing a step to execute and each branch representing an alternate series of steps is particularly well suited to analysis. In addition to utilizing a graph representation, the key concepts drawn from GraphPlan include the concept of using sequences of actions to change an initial state to the agent's goal state.

2. **Event Analysis**: When an unexpected event does occur, the agent must have some method of analyzing the event and determining whether this event affects the plan and if so, the nature of the effects.

   Selected Paradigm: Through the relationship established between a program consisting of sequential instructions and a plan consisting of sequential steps, the technique of slicing used in debugging domains can be applied to analyzing a plan. The primary use for this technique in the planning domain is to analyze the dependencies between actions and events and preconditions. Employing the technique of slicing to a plan can yield answers to the questions: Does this event affect the plan? If so, how?

3. **Dynamic Transitions**: To adequately handle real-world environments with unexpected events, an agent must be able to dynamically transition from the current action to a new solution when an unexpected event renders the current plan untenable. Another important aspect of dynamic transitions is that the transitions are not hardcoded, i.e. the agent or the system builder should not directly link the occurrence of specific events to specific contingency plans.

   Selected Paradigm: Because of its definition of states and flexible transitions between states, Context-Based Reasoning (CxBR) has proven suitable for controlling the behavior of real-time agents in dynamically changing environments. While previous attempts to deal with event interruptions have relied on hardcoded contingency plans, incorporating the concept of hierarchical states (or contexts in CxBR terminology) and flexible transitions will enrich the final model. CxBR concepts will enable agents to adjust their behavior dynamically to situations they encounter.

4. **Incremental Updatability**: One of the key aspects of any planning system is the flexibility to change quickly and adapt to new scenarios. In this case, plans should able to be incrementally updated through the introduction of events or new actions. For instance, during "what-if" scenarios, it is useful to add/remove/modify resources or inject events into the environment dynamically to test the robustness of the plan.

   Selected Paradigm: Because it attempts to model processes that are often subject to change, Contextual Graphs (CxG) was chosen as a paradigm keeping data up-to-date and consistent. By applying CxG's feature of

Incremental Knowledge Acquisition, it is possible to easily modify the environment or plan. To support dynamically updating the environment components at runtime, this concept is extended to include injecting events into the model at any point in time.

5. **Agent Simulation**: The final requirement of the proposed approach is its support for accurate simulation of an agent in a dynamic, changing world. This would require the inclusion of concepts such as time durations for actions, resource usage needed to accomplish a goal, and close to real-time execution.

   Approach: The final requirement of real-time agent simulation is a feature exhibited in many systems, but is unique to the final system because of the combination of various architectures that have differing levels of support for the simulation of agents.

4.3 OVERVIEW OF ADCUE

The approach proposed in this thesis is the Automatically Determining Consequences of Unexpected Events (ADCUE). ADCIE incorporates aspects from the aforementioned paradigms in addition to new features described in this chapter. ADCUE provides a simulation environment in which an agent can exist and operate. Furthermore, ADCUE provides an agent with built-in abilities to reason about the consequences of unexpected events. This is accomplished through the interaction between the various ADCUE components:

- **Attribute:** A feature of a component represented by a name, type, and value

- **Resource:** A non-intelligent, material entity that can be used, produced, or synthesized by an agent through the execution of actions
- **Condition:** A Boolean relationship between an attribute and a value that evaluates to true or false depending on the current state
- **State:** The presence of one or more conditions, allowing a situation to be quantified and uniquely identified
- **Objective:** The goal of an agent represented by a state that specifies the conditions that should be true for the agent to have accomplished the goal
- **Action:** A black box, executable by an agent under necessary preconditions, that produces a change in the environment through resulting postconditions
- **Event:** An action, without preconditions, executed by the environment or another agent, as perceived by the current agent
- **Agent:** An intelligent entity with objectives to achieve by executing actions that produce/use/synthesize resources
- **Environment:** Describes the system as a whole in terms of the previous components and any custom data attached to the simulation

Using these components, a complex simulation can be constructed to represent real-world environments and situations. The ADCUE system is capable of representing objective-driven agents that plan, handle events, and replan when necessary. To further understand how each of these components contributes and enhances ADCUE, they are subsequently examined in detail.

4.4 Model Components

Each component in ADCUE has specific features that enable the construction of a realistic simulation that functions smoothly and correctly. In this section, the features of each component are described and examples are given to clarify how the component integrates within the model as a whole. The convention for a component name is to use *italicized* font, such as the resource *Milk*. To refer to attributes, the dot operator will be used; for instance, *Milk.Spoiled* might represent an attribute that becomes true when the resource *Milk* expiration date passes.

4.4.1 Attribute

Attributes form the core of all other components and are used extensively throughout ADCUE. An attribute describes one aspect or feature of a component. It has three parts:

1. **Name:** The name uniquely identifies the feature of the object. One example for the resource *Milk* might include *Spoiled.*
2. **Type:** The value of the feature may be numerical, categorical, or Boolean. An attempt to set or modify the value of an attribute must abide by the type of the attribute (i.e. it should be impossible to set a categorical attribute to a numerical value).
3. **Value:** The last part of an attribute is the actual value of the attribute. This describes the component in terms that the rest of the simulation can understand and typically changes to reflect the passage of time. The format of the value depends on the type. Numerical values may take on a value within the range of real or floating point numbers. Categorical data

is represented textually, and the type Boolean may be considered categorical with only two valid categories: true or false.

4.4.2 Resource

A resource in a general sense can be defined as “an available means” [33]. In ADCUE, a resource represents a non-intelligent, material entity that is potentially useful to an agent in the system. Agents can consume resources, produce resources, or synthesize one resource into another through the execution of actions. Two pre-defined attributes are particularly important when considering a resource: amount and ownership. Any resource must have a finite amount or quantity; this value is likely to change throughout the execution of the simulation as the resource is produced and/or consumed. For future multi-agent considerations, ownership will play an important role. For example, if used in a battle simulator, ADCUE would need to clearly separate resources between opposing agents in addition to limiting or preventing the simultaneous usage of resources.

4.4.2.1 Resource Example

As mentioned earlier, the resource *Milk* can be viewed as a resource. It can be said to be produced by a *Supermarket* agent and consumed by a *Person* agent. Extending the example, a *Person* agent might synthesize *Milk* into another resource such as *Chocolate Milk* or *Buttermilk Pancakes*. An example of attributes for this resource might include those seen in Table 1.

Table 1: Example attributes for the resource Milk

Attribute	Type	Value
Amount	Numerical	Initial value: 1 gallon
Location	Categorical	Refrigerator
Ownership	Categorical	Agent.Smith
Spoiled	Boolean	Initial value: false
Temperature	Numerical	Initial value: 40° F

4.4.3 CONDITION

A condition can be defined as a “state of being” [33], and must always evaluate to true or false: either the condition exists or it does not. In real life, a condition might be represented with the question “Is the milk spoiled?” This condition is true or false. In ADCUE, a condition is represented with a relationship between an attribute and the value for the attribute. Thus, a condition has three parts: the attribute, relationship, and value. Relationships available in ADCUE are: equals, not equals, greater than, less than, greater than or equal, and less than or equal. Conditions are used extensively when determining whether an action can or should be executed. In our real-world “Is the milk spoiled?” example, the condition might indicate whether you can have cereal in the morning. ADCUE represents this condition as “*Milk.Spoiled* not equal *true*”. Although it reads slightly differently for ease of computation, the semantic meanings are the same.

4.4.4 STATE

Because a single condition will typically not convey all the information necessary to define a situation, multiple conditions may be combined together to form a state. Thus, a state can be used by an agent to identify situations. It can be said that an agent is in a state (or situation) only when all the conditions that form the state are true. It is important to note that states do not usually specify all possible conditions that can occur at any one given time. Often only a few key conditions are necessary to determine the state, such as "*Tire.Flat* equals *true*" and "*Jack.Broken* equals *true*". The fact that the condition "*Car.Color* equals *white*" has little relevance to the fact that the agent is in a state with a broken car that cannot be easily repaired. This technique replicates CxBR's method of limiting the scope of what the agent considers to only what is relevant in the current state or context. An example might include state named *Enough Milk* that combines the condition as "*Milk.Spoiled* not equal *true*" with "*Milk.Amount* greater than *1 cup*". By checking whether these two conditions are true, the agent can determine whether or not it is in the state *Enough Milk*. An agent is only in the *Enough Milk* state when both conditions evaluate to true.

4.4.5 Objective

When planning, usually a desired outcome exists as a set of goals or objectives. ADCUE uses the component objective to describe the final state which an agent is trying to bring about by executing actions. From the ADCUE point of view, there is no difference between a state and an objective except the special connotation that an objective is the desired outcome or state that the agent wishes to exist. In fact, achieving the objective might signal the end of the simulation. Because the conditions contained within the objective may or may not be true depending on whether the objective has been achieved, the agent's goal is to find a way to make all the conditions true and bring the state into existence. Another way to view objectives is the end result of executing a series of actions that incrementally change the state of the environment until the state matches the state specified as the objective.

4.4.5.1 Example

A sample objective, in natural language terms might be “agent Smith wants to eat after he wakes up in the morning.” Translating this to the ADCUE model, the objective might be represented with a state named *Not Hungry* that consists of two conditions: “*Smith.Location* not equals *bed*” and “*Smith.Hungry* equals *false*.” This objective would guide the agent Smith to execute the action *Get out of Bed* followed by the action *Eat Breakfast* to change the initial state of the environment satisfy the conditions specified by the objective.

4.4.6 Actions

In real life, an action may be defined as a set of steps performed. For simplification purposes, ADCUE views actions in a similar manner. One exception is that ADCUE treats all actions as “black boxes” in that an agent can execute an action to achieve some effect on the environment. This black box treatment means ADCUE would happily accept an action named *Transport Item* that instantaneously transferred an item to the moon without questioning how the action worked. Neither the agent nor ADCUE care what physical processes or steps need to be taken in order to bring about the conditions.

To execute an action, an agent must ensure the set of conditions associated with the action are all true. Every condition within this set of conditions, called the a precondition, must be satisfied before the action can be run. For example, the action *Cook Pancakes* may require the resources *Pancake Mix* and *Stove*. These would correspond to preconditions “*Pancake Mix.Amount* greater than *1 package*” and “*Stove.Broken* not equals *true*”. Without these preconditions being true, it is impossible to execute the action *Cook Pancakes.* Once the preconditions for an action become true, the agent is free to execute the action to bring about the effects, or postconditions. Postconditions represent the conditions that are true after the action executes and are defined per action. It is possible for multiple actions to accomplish similar goals, but differ in resources used, leading to the realistic and complex interaction between resources and actions available to the agent.

4.4.6.1 Example

Returning to the cooking example, agent *Smith* may have a number of actions available to achieve the objective *Not Hungry*, such as *Pour Cereal and Milk*, *Cook Pancakes*, and *Order McMuffin*®. Execution of any of these actions will result in the same basic effect or postcondition, namely *Smith.Hungry*, will become *false*. However, they differ in preconditions and other postconditions. *Pour Cereal and Milk* will reduce cereal and milk resources. *Cook Pancakes* will consume extra time and require additional cooking utensils, such as a pan. The action *Order McMuffin*® has preconditions of *Car* and *Money* and may have a postcondition of higher cholesterol. As can be seen, complex interactions can be built with relatively little effort.

4.4.6.2 Simultaneous Actions

Modeled after CxBR's "one active context" limitation, ADCUE permits agents to execute only one action at a time; thus, an agent can never perform two actions simultaneously. Since the functionality of executing simultaneous actions may be desired in some situations, such as executing the actions *Talking* and *Driving* at the same time, the concept of multiple inheritance is introduced. Inheritance allows a parent action to pass the full properties of itself to a child action. The child action can be viewed as a copy of the parent action with the addition of any customizations or additions specific to the child. Multiple inheritance allows an action to automatically incorporate the characteristics of more than one distinct parent action. Used in this manner, multiple inheritance allows the system designer to combine the effects of multiple actions into a single new action. For example, the action *Talking and Driving*, a child of both the *Talking* action and the *Driving* action, embodies all the characteristics of its parent actions, including postconditions. The new action's postconditions are the combination of each individual action. Thus, from the perspective of an ouside observer, the execution of the child action *Talking and Driving* is no different than executing the parent *Talking* action and *Driving* action simultaneously. This effectively achieves the same end as simultaneous actions. While more work is required to manually create child actions and specify inheritance, this approach avoids the compatibility problem that may occur under automatic action composition whereby an agent may try to execute *Get Dressed* and *Drive to Work* simultaneously to save time.

4.4.6.3 Action Levels

CxBR employs a hierarchical structure for contexts that lends itself well to organizing states into logical clusters of differing detail. ADCUE applies this functionality to actions by allowing levels to be associated with actions. Currently, "high" level and "low" level actions are utilized. For the allowance of more detailed actions in future expansions of ADCUE, "lower" level and "lowest" level actions are reserved, but not implemented. High level actions represent the lowest granularity of an action, and decreasing action levels (low, lower, lowest) indicate increasing amounts of details. The hierarchy allows high level actions to be composed of many lower level actions. For instance, a high level action might be *Eat Breakfast*, while a low level action for *Eat Breakfast* might be *Cook Pancakes* or *Chew*. Consequently, the lower level action specifies how to accomplish the high level action. This hierarchical grouping of actions has a number of far reaching impacts. First, this organizes actions into more natural and manageable structure. Secondly, action levels provide a means of defining the amount of detail within a simulation. While one simulation might only require high level actions, another might require lower level actions to physically control the movement of the agent. Finally, this allows the planner to plan on several levels. The planner may initially generate a plan using only high level actions and then expand the plan using the lower level actions, thus saving computational time. This also allows the planner to generate a high level plan and then fill in the details later by expanding high level actions with lower level actions as necessary.

4.4.7 EVENTS

Events have many similar properties to actions. Both affect the world through predefined postconditions. However, there are some key differences. First, events have no preconditions, i.e. they can execute at any time. Events are triggered randomly (to simulate the randomness found in the real world) or by the user, who may be interested in seeing how the agent will react to the event. Another important difference is the frame of reference. An event can be considered an action that is perceived, but not initiated, by the current agent. Under this definition, the action executed by agent *Jones* is perceived as an event by agent *Smith*. Events are said to belong to and be executed by the environment. For example, the event *Meteorite Strike* may be triggered with a certain probability by the environment or by the user of the simulation.

4.4.8 Agent

An agent is an intelligent entity of a particular class (person, tank, etc.) that is capable of carrying out actions and reasoning about events. An agent uses actions and resources to change the current state to reach some end state, or objective. The actions the agent will tentatively execute in the future are referred to as the plan and are generated with the help of the reasoning model, which is described later.

4.4.9 Environment

The environment encapsulates all the previously described model components and provides the interface to the simulation. It also contains global attributes such as the current time, location, or weather. The environment is also responsible for handling the execution of events, either probabilistically or upon the command of the user. Finally, the environment may specify application-dependent information such as geography, opposing military movements, etc.

4.4.10 Self-Describing Components

One important requirements of ADCUE is that all components must be completely *self-describing*. Self-describing components keep track of their status and perform internal housekeeping, thus encapsulating both data and functionality within the component. To illustrate, an agent operating within a traditional rule-based system might check the expiration date on a milk container to see whether the resource *Milk* is still usable. However, this encourages sprawling dependencies between components; furthermore, the check must be duplicated for each agent that uses the resource *Milk*, a practice discouraged when building software systems. In

ADCUE, the burden of knowledge is shifted to the component itself. Under this approach, the milk container should describe itself completely, including the knowledge of when it is expired. For example, when the milk container detects it has expired, it should update its attributes to reflect the new condition of itself (i.e. the attribute *Milk.Spoiled* should be set to *true*).

The benefit to this approach is that it modularizes the design by placing the burden of knowledge, not on the agent, but on the individual component. By enforcing this requirement, it is possible to add, remove, modify, or fine-tune components without necessarily requiring any modification of any other component, including the agent. Furthermore, this allows the modification of the system to occur dynamically at runtime, even when the agent is in the middle of completing a mission. This is particularly important for "what-if" scenarios where a user of the system may want to add or remove resources or other components to the system to see the effects of such modification.

4.4.10.1 EXAMPLE

Because of the self-describing requirement, two attributes of *Milk*, *Spoiled* and *Temperature*, must be updated to keep the state of the component up-to-date. For example, if attribute *Location* changes from "Refrigerator" to "Counter," the resource must periodically update the *Temperature* attribute to reflect the fact that the milk is getting warmer.

4.4.11 DYNAMIC COMPONENTS

Because of the nature of the model and the self-describing requirement, ADCUE is designed so that a component can be modified dynamically during runtime while still preserving the integrity of the system. This is achieved by the injection of events into the system. For example, in a simulation of an Unmanned Aerial Vehicle (UAV) surveying geographic locations, the user of the simulation may want to inject an event predefined by the system developer, such as *Hostile Missile Launch*. However, consider the event *Rats Chewed on Wires* – an event certainly not pre-defined. If the user wishes to inject such an event to see how it affects the agent, the user could define the effects as "disable a random piece of equipment on the UAV." After inputting the event into ADCUE, the user could subsequently inject the event into the environment and observe the agent's response. This allows scenarios to be built by adding/removing components and observing the effects on the system.

4.5 PLANNING

Since the goal of every agent is to achieve its assigned objectives, planning is the first step in calculating how the agent can use the actions and resources available to it to bring the end state into existence. Given a state, or set of conditions, that must exist for the agent to consider its mission a success, what sequence of actions should the agent schedule to run? Because ADCUE draws a significant amount of conceptual material from the GraphPlan paradigm in terms of specifying how agents use actions to achieve goals, the approach presented here is similar to that found in GraphPlan papers. Both GraphPlan and ADCUE employ a backward chaining, demand-based planning system.

4.5.1 Planning Process

The planning process of finding a sequence of actions can be described as an incremental process that finds actions in reverse chronological order, leading from the objective to the current agent's state. The goal of the process is to produce a directed planning graph that links the single entry node representing the current state to the single exit node representing the end state (objective) via a series of actions. Each unique path from the start node to the end node represents a potential plan that the agent can execute to achieve the objective. The process involves four steps:

1. Check the difference between the current state and the goal state. The difference between the two states will be the attributes that need to be changed in order for the objective to be achieved. These attributes are referred to as *objective attributes*. In some instances, an objective might be partially completed when it is assigned to the agent, in which case the number of objective attributes will be less. If no objective attributes are found, no difference exists between the initial and goal states. Thus, the agent has already achieved the objective and no plan is necessary.
2. Perform demand-driven action searching. From step 1, the process has compiled a list of objective attributes. A search is initiated for actions whose postconditions modify these objective attributes. It can be said that the objective demands actions that affect the objective attributes. The actions found are referred to as candidate actions because they represent actions that may result in the accomplishment of the objective. Generally speaking, it is not possible to determine whether any of these candidate actions will definitely lead to the objective. This will be

determined later. However, depending on the complexity of the postconditions, it may be possible to eliminate candidate actions that have no possibility of leading to the objective by optimizing action postconditions with objective preconditions. These candidate actions are linked to the objective, forming the end of a directed graph with the objective representing the terminal node.

3. Calculate the backwards state. For every candidate action linked to the objective, it is possible to calculate the state existing before the action was executed (this is known through the preconditions). The result is a list of intermediate states; an agent in one of these states could execute the appropriate candidate action and achieve the objective. If the agent's current state matches one of these intermediate states, the agent has found a potential plan of some sequence of actions that may lead to the goal state. Since a path between the current state to the end state has been achieved, this path is considered finished.
4. Develop the backwards chain. From each set of intermediate states that do not match the current state (i.e., the action cannot be directly executed by the agent), go to step 2, treating the intermediate state as the objective. This recursively traces plans from the objective to the agent's current state. A limit on the number of steps in a plan will prevent infinite backwards chaining.

This process may incorporate all action levels, or hone in and plan at a specific action level to build plans of different granularity. At the end of this process, all "dangling" paths that did not find a way to reach the agent's current state are eliminated. This may occur because a particular resource was not available. For instance, the action *Drive* can be used to transport items, but only if a car is available. If no graph path backtracked completely from the objective to the agent's current state, two possibilities exist. First, no combination of actions that the agent can execute will accomplish the objective. Second, a combination of actions to reach the objective exists, but was not found because the maximum number of steps allowed in the plan was set too low in step 4. If a path exists between the agent's current state and the objective state, at least one potential plan exists. Each unique path represents a potential plan. Validating the plan can be done by analyzing each action, starting with the current state to the finish state to ensure all the preconditions and postconditions match. Alternatively, the agent can perform the alternative plans in hyper-real-time. The actual path selected among the potential plans is agent dependent. For instance, the agent may want to maximize time or minimize monetary cost.

There are several important advantages to using this approach. First, the backwards chaining constrains the plans generated so as to reduce the combinatorial explosion that would occur if planning in a forward chaining manner. Secondly, the planning graph is a convenient way to organize plan alternatives as each path is unique path from the current state to the object represents a different plan that will achieve the goals of the agent. All precondition dependencies are also explicitly laid out and specified in the planning graph. Finally, the graph provides an easily traversable structure for further analysis, such as determining the effects of events.

4.5.2 EXAMPLE

Consider the agent *Smith* waking up in the morning wanting to eat breakfast. Since *Smith* is hungry when he wakes up, the objective state that *Smith* wants to accomplish is defined as the condition “*Smith.Hungry* equals *false*”. The environment and other ADCUE components, including available resources and actions, are included in Figure 1. Using only high level actions, the planning process may initially develop a plan with coarse granularity.

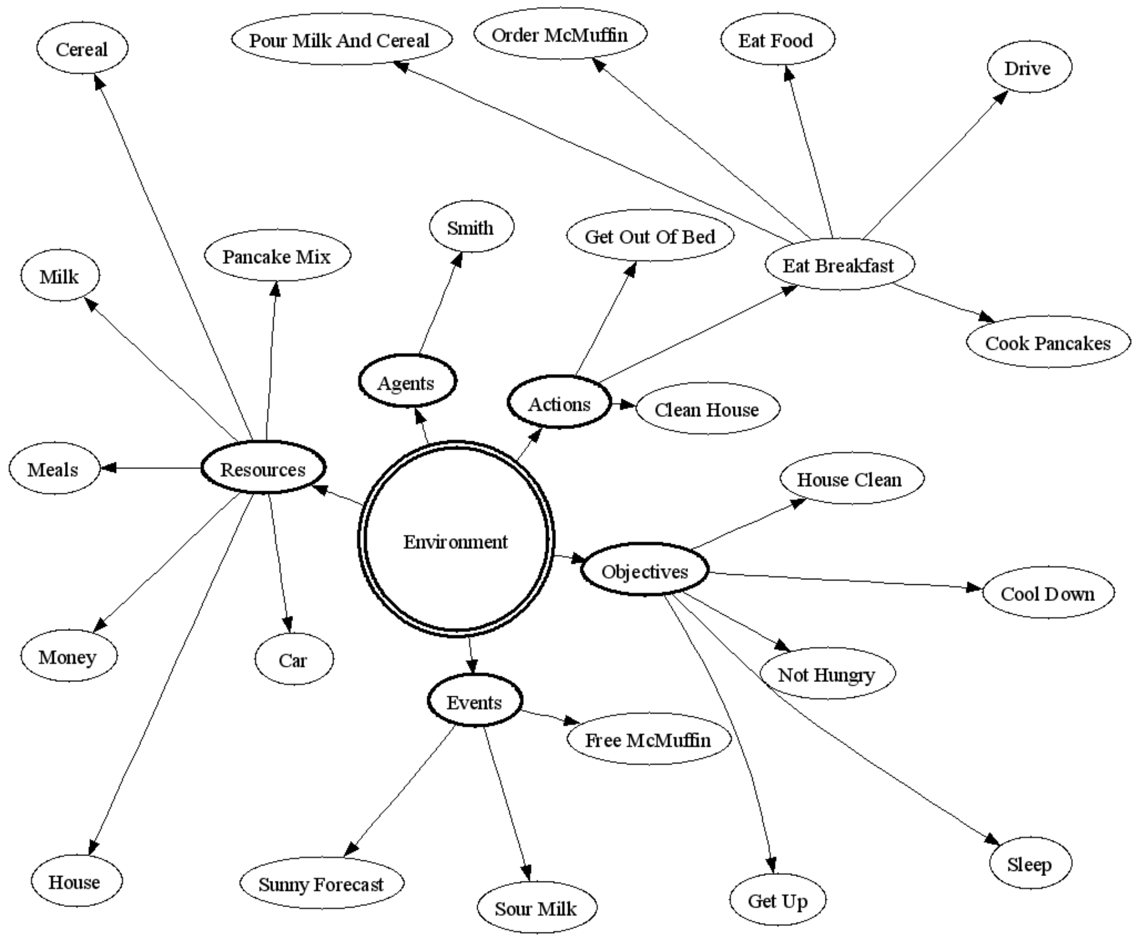


Figure 1: Breakfast scenario components

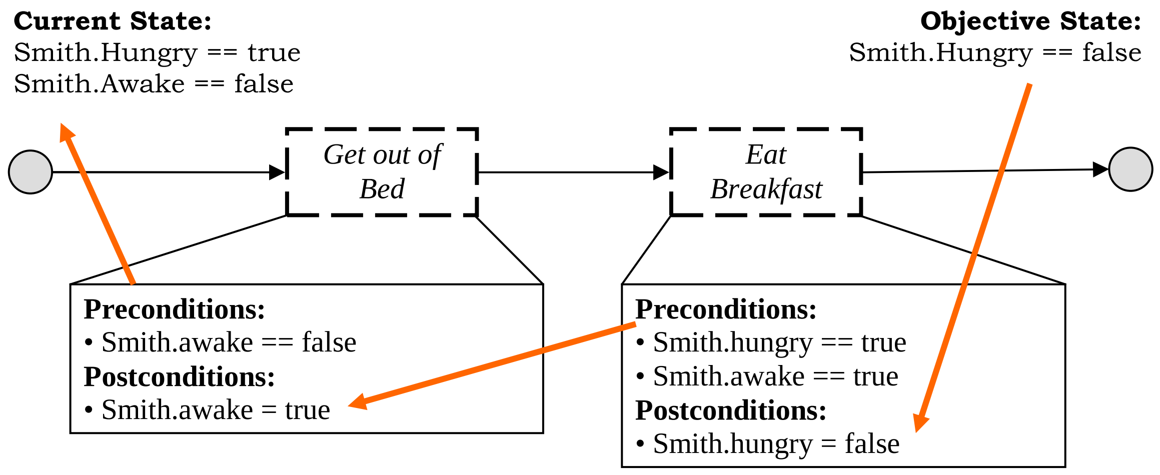


Figure 2: Planning graph representing the high level plan

As seen in the Figure 2, the high level planning graph consists of matching high level actions that lead from the agent's current state (*Smith* is in bed) to the objective state (*Smith* is not hungry). Notice that the attribute in the objective state, *Smith.Hungry* was traced back to the action *Eat Breakfast*, which in turn, was traced back to the action *Get out of Bed*. Since this action can be directly executed by the agent in its current state, the planning process finishes, yielding a high level plan. If desired, this plan can be expanded in more detail by running the process on the low level actions associated with *Eat Breakfast*. To do this, the action *Eat Breakfast*'s preconditions are treated as the current state and the postconditions are treated as the objective state. Figure 3 is the corresponding low level planning graph. Notice that agent *Smith* has several different options for eating breakfast now.

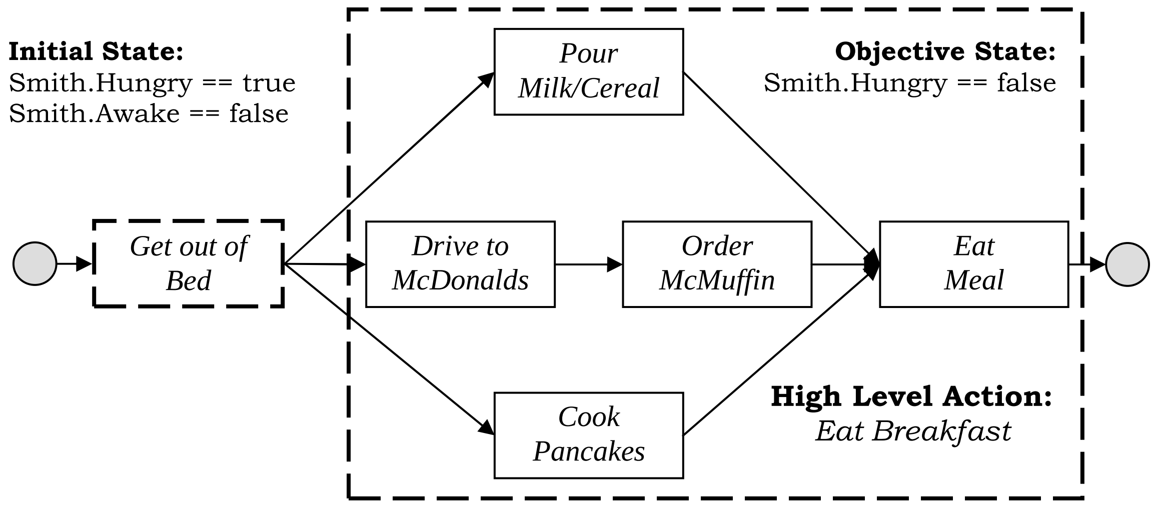


Figure 3: Planning graph representing low level plan

4.5.3 Replanning

In ADCUE, replanning because of the occurrence of an expected event can be performed via two methods. If multiple plans exist on the planning graphs (more than one unique path from the current state to the objective exist), it may be possible to select a different, already-generated plan path that is unaffected by the event. For instance, if it is known that *Milk* is spoiled, the agent may select the action *Order McMuffin*® instead. If the event has far-reaching effects, it may be better to start the process of planning from scratch, considering the postconditions of the event from the start of the process. In this situation, it is possible that the alterations from the event postconditions will give the planner the information necessary to generate new action links between states that plan around the effects of the event.

4.6 Introduction to Slicing

Now that the ADCUE framework has been presented, each component described in detail, and the methods of planning have been described, it is important to investigate the mechanism that operates within the framework and enables the agent to automatically determine the consequences of events. This mechanism is the technique of *slicing*.

4.6.1.1 Historical Perspective

Originally designed as a technique to aid debugging programs, *slicing* involves analyzing a particular statement in terms of data and control dependencies [34]. Slicing constricts the view of a program's source code to a small piece, typically a single statement called a slice, in conjunction with all the other statements in the program that can potentially affect the slice. In essence, by removing or "slicing away" everything irrelevant to the statement under analysis, a debugger can ignore all extraneous details. The remaining statements can be categorized as data dependencies, statements that modify the data used by the slice, or control dependencies, statements that define the flow of execution to the slice. Slicing can be performed in a backwards or forwards manner. Backwards slicing shows how previous statements affected the current slice; forward slicing shows how the current slice influences future program statements.

4.7 Slicing Applied to Planning

In addition to applications in debugging, slicing is a technique that can be applied to planning. Instead of working on statements in a program, slicing can be performed on the steps in a plan. Data dependencies become dependencies in environment conditions and control dependencies become dependencies in previously executed steps. By isolating single actions, a slice will show the relationship between conditions in the environment (data dependencies) and the flow of previous actions (control dependencies). In a similar way that slicing can be used to analyze a running program, this technique of slicing can be applied when introducing an event into an already executing plan. By tracing the causes of the event upstream (backwards slicing) and the effects of the event downstream (forwards slicing), the dependencies can show which parts of the plan will be affected by the event. Adapting the process of slicing to planning in this manner, an agent can begin to dynamically analyze the relationships between the effects of an event and planned actions, inferring consequences along the way.

4.7.1 PLAN REPRESENTATION

To apply the technique of slicing to the domain of planning, plans must be in a format similar to that of a program's source code. A program always has a defined start and end of execution with a sequence of instructions to execute in between. Likewise, plans can be represented as steps, or actions, stored in nodes of a graph. The entry node represents the beginning of the plan (the current state) and the exit node represents the end of the plan (the desired or goal state). Relationships between nodes are specified by action preconditions and postconditions. The execution of an action depends upon preconditions and is characterized by the resulting postconditions. Preconditions represent control dependencies because they define when it is possible for an action to run; postconditions represent data dependencies because it reflects the effects of the action's execution on the environment. Through this representation, the technique of slicing can be applied to the graph to analyze the plan when unexpected events occur.

4.8 Proposed Approach Algorithm

When an event occurs, the proposed approach needs to answer two questions: 1) Does the event affect the plan? 2) If so, what are the effects of the event on the plan? To answer these questions, a customized intersection operator is proposed for use in conjunction with the slicing technique described in previous sections.

4.8.1 Intersection Operator

The intersection operator ∩ is used to detect whether an event affects the plan. It takes two parameters, an action and an event. Conceptually, the intersection operation detects conflicts between event postconditions and action preconditions/postconditions. If the intersection operator results in an empty set, the event does not directly affect the action; if the intersection results in a non-empty set, a potential conflict exists between the action and the event. Mathematically, the intersection operator between $Event\ E$ and $Action\ A$ is defined as:

$$\begin{aligned} E \cap A &= Pre(E \cap A) \cup Post(E \cap A) \\ &= (E.Postconditions \cap A.Preconditions) \\ &\cup (E.Postconditions \cap A.Postconditions) \end{aligned}$$

$$where\ E.Postconditions, A.Postconditions, A.Preconditions\ are\ attribute\ sets$$

As can be seen, the intersection operator is composed of two separate intersections that are subsequently joined with a union. This represents the two different conditions that must be true for an action to remain unaffected by an event. First, the event must not affect any of the action's preconditions and second, the event must not affect any of the action's postcondition attributes. In laymen's terms, the first part of the intersection operator can be read as: if the event modifies attributes used to determine if the action can be executed, the event may prevent the agent from using the action in the plan because the preconditions may no longer be satisfied. The second part detects a potential conflict between the event and action postconditions: both are attempting to change the same attribute to potentially different values.

4.8.1.1 Testing Intersection Hypothesis

In effect, an intersection operation between an event and action that results in a non-empty attribute set is proposing a null hypothesis. This null hypothesis states that based on the overlap between event and action attributes, the action affects the event. For instance, consider Case 1 in Figure 4 with action *Sunbathe* and the event *Rainburst*. Both set the attribute *Smith.Wet*; the difference is that *Sunbathe* sets this attribute to false and the event *Rainburst* sets the value to true. Because the attributes are being set simultaneously by different components, a situation which is referred to as attribute overlap, the intersection operator will hypothesize that the event affects the action. For Case 1, this is a valid hypothesis. However, in other situations the null hypothesis may not be true. If the action *Sunbathe* is swapped with *Swim* as seen in Case 2, the both the action and event modify the attribute *Smith.Wet* – but this time they both set the value to true. In this case, the null hypothesis should be rejected because although the attributes overlap, there is no conflict between the action and event.

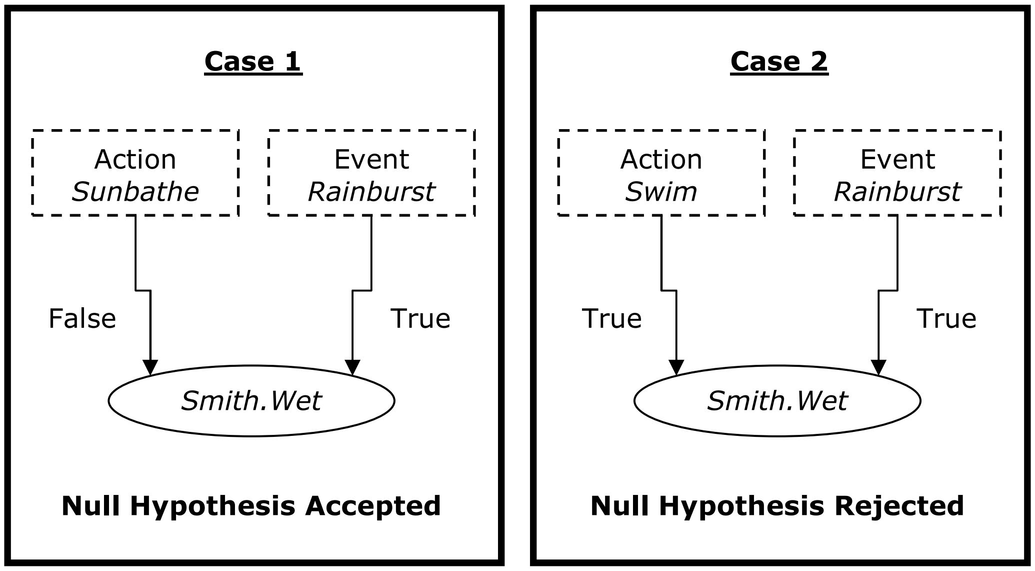


Figure 4: Null Hypothesis Validation

Since the intersection operator only reveals a potential conflict between an action and event, the null hypothesis generated must be tested. In simple cases, such as the beach example, this can be done without much effort. However, in more dynamic situations where actions may result in different effects depending on various environmental factors, it may not be as straightforward. In these instances, it may be necessary to simulate the rest of the plan and examine the expected future values of the attributes to see if a conflict exists. Based on calculated future attribute values, it is possible to accept or reject the null hypothesis that the event affects the action.

In general, the intersection operator can affect actions in one of three possible levels: impossible, delta, or null. An event may change attributes in the environment such that the preconditions to an action are no longer satisfied, thus making it impossible to execute the action. Another possibility occurs when the event changes the postconditions of the action, thus producing a delta, or change in the effect of the action. Finally, a null result indicates the event has no effect on the action.

4.8.2 PLAN SLICING

The combination of the slicing technique and the intersection operation provides a powerful pair of tools for analyzing the plan stored in the planning graph to find any consequences of unexpected events. The algorithm for determining whether an event affects the plan can be described with the pseudo code in Table 2:

Table 2: Pseudo code for determining plan affectedness

```
Determine Plan Affectedness by Event E

Set Plan P.Affected to false
Foreach Action A in Plan P
    If (A ∩ E) not empty
        Set Plan P.Affected to true
        Stop
```

This pseudo code checks every step (action) in the plan to see if the action will be affected by the event. If none of the steps in the plan are affected, the plan as a whole is unaffected. Once it is determined that an event will affect the plan, the consequences of the event can be inferred by determining the difference between executing the plan with the event versus without the event. This difference can be extracted from the list of conflicted attributes provided by the intersection operator. The pseudo code for this procedure can be found in Table 3.

Table 3: Pseudo code for determining event consequences

```
Determine Consequences of Event E

Foreach Action A in Plan P
    If (A ∩ E) not empty
        Foreach Attribute Attr in (A ∩ E)
            Add Attribute Attr to List diff

Return List diff
```

4.8.3 Example

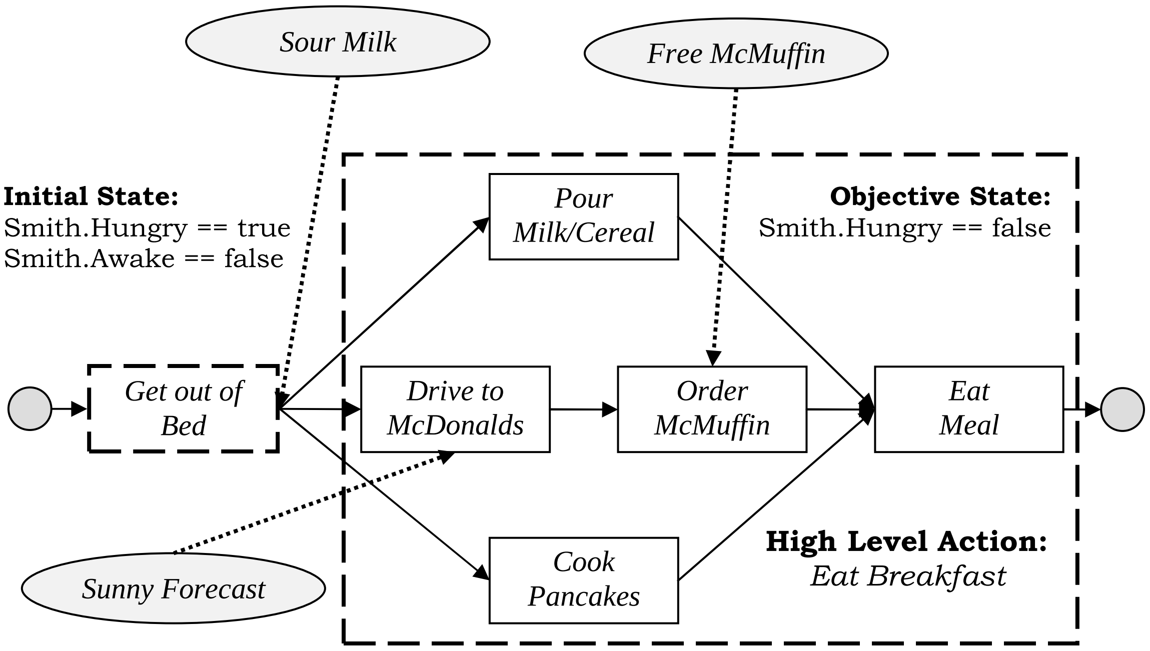


Figure 5: Planning graph with events

Consider a plan for Agent *Smith* to wake up in the morning and eat breakfast. A number of different actions are available, such as *Pour Milk And Cereal*, *Cook Pancakes*, and *Order McMuffin*®. Several events can affect the plan, including *Sour*

Milk, *Free McMuffin®*, and *Sunny Forecast.* See Figure 5 for a graphical representation of the plan with injected events listed in bubbles.

Given the above scenario, it is desirable to know how each of the three events affects the system:

- *Pour Milk and Cereal* ∩ *Sour Milk*: This event represents *Agent.Smith* walking to the refrigerator in the morning and discovering that the milk gone bad. The event *Sour Milk* will make it impossible for the action *Pour Milk and Cereal* to be executed since one of the preconditions of the action *Pour Milk* is "*Milk.Spoiled* not equals *true*".
- *Order McMuffin®* ∩ *Free McMuffin®*: This event simulates *Agent.Smith* receiving a "buy one, get one free" McMuffin® deal at McDonalds. The event *Free McMuffin®* produces a delta or change in the postconditions of the action *Order McMuffin®* by doubling the number of meals. Notice, that not all unexpected events are detrimental; a free McMuffin® is a pleasant surprise (unless agent *Smith* is on a diet).
- *Drive* ∩ *Sunny Forecast*: The final possibility is null and is seen when the event *Sunny Forecast* is heard during the agent's *Drive* action. In this case, the event has no effect whatsoever on the action. Thus, the event effects on the plan is null and the affected attribute set is empty.

4.9 SUMMARY

This chapter presented the methods used to determine the consequences of an event on the plan. The two main components introduced were slicing and the intersection operator. While slicing provides an intuitive method of analyzing plan dependencies, the intersection operator calculates if and how much an event affects steps within the plan. The proposed solution of utilizing slicing and the custom defined intersection operator to within a plan consisting of nodes in graph creates a powerful reasoning mechanism whereby consequences of unexpected events can be automatically determined. Furthermore, the ADCUE framework was presented as the combination and logical extension of several preexisting paradigms. These paradigms include CxBR, slicing, CxG, and GraphPlan. ADCUE emphasizes robust planning, the ability to analyze events, dynamic transitions that do not require hardcoded contingency plans, easy upgrades, and real-time simulation. To develop such a model, ADCUE uses the components Attribute, Resource, Condition, State, Objective, Action, Event, Agent, and Environment. By augmenting proven techniques and applying them to new problems, the reasoning model provides a comprehensive solution to the problem of automatically determining the consequences of unexpected events.

CHAPTER 5: IMPLEMENTATION

This chapter describes the implementation of the ADCUE approach presented in the previous chapter. First, the environment and the encapsulation of ADCUE into a library are discussed. Secondly, desired usage and system input/outputs are overviewed. Thirdly, the architecture of the code are investigated, including class diagrams of key components, followed by a fourth section providing how the ADCUE algorithm operates.

5.1 LIBRARY ENVIRONMENT

The environment to implement ADCUE in was Microsoft Visual Studio C++ 2005 on a Windows platform. C++ was chosen for purposes of efficiency and speed, and the Microsoft IDE was chosen for productivity. While the test cases should not require heavy computational resources, it is expected that in a non-trivial real-world situation, ADCUE will be expected to perform in real-time in complex simulations. While C++ is not inherently as portable as other popular languages such as Java, effort was extended to ensure that all code written was cross-platform so that future expansions could have a wide range of operating environments from which to select. To encourage use and reuse in additional projects, a strict coding and commenting standard was followed. The freely available tool Doxygen was used to generate extensive documentation from specially formatted comments within the code. With these aids, it should prove relatively easy for a competent C++ programmer to review the documentation and write a program using ADCUE with minimal work.

5.2 Library Usage

The ADCUE library allows the simulation of goal-oriented agents operating within an environment as described earlier. It is composed of the library interface and the simulation. To keep consistent terminology, the term simulation will be used to refer to the set of ADCUE components grouped together as a cohesive package. It also refers to the execution of the ADCUE components through time and the injection of events into the system. The library provides methods for loading created simulations, controlling the simulation, and injecting events into the system for the agent in the simulator. The simulation is created and stored through a series of files located in a directory, each one representing a component in the final simulation. In this manner, it is easy to create and modify the simulation data. At a later point in time, a Graphical User Interface may be built to further simplify the creation of ADCUE simulation components. To use the ADCUE library, one must load the directory containing the simulation files and then run the simulation, optionally injecting events and retrieving the current state of the simulation. The ADCUE library provides an agent access to a rich representation of the environment and the ability to use ADCUE's built-in functionality to reason about the consequences of unexpected events.

5.2.1 System Inputs

As mentioned, the inputs to ADCUE are the simulation files stored in a single folder. The purpose is to provide an easy way to create/modify/delete ADCUE components. The prefix for each file is the type of component. For instance, the file containing the representation for the resource *Milk* would be named

"Resource.Milk.txt." Each component file should completely describe the component; as such, each component has an individualized file format

It is important to mention the notation used to describe the file formats. Generally, each non-empty line in the component file specifies an attribute or some other piece of information about the component. Most lines begin with a label followed by a colon; this identifies the rest of the line as one of the pre-defined pieces of information. The remaining line usually has a number of pre-defined tags that must be supplied. For example, text enclosed between <less than and greater than signs> is a tag representing a description of the user-defined text that must be inserted. For example, when the tag <Attribute Name> is encountered, the user building the component insert a name for the attribute to replace the tag, less than and greater than signs excluded. [Square brackets] operate in a similar fashion, except they represent optional tags. Thus the tag [Units] can either be supplied or omitted depending on if the user wants to specify units for the attribute. Quotes are used to allow spaces and must be used where specified. For instance to use the name *Weather Conditions* for the tag "<Name>", the text in the file should read "Weather Conditions". { Curly brackets } surrounding a line represent a block: the line is a template that can be used several times in a row (with different values for the tag). For example, curly brackets are used to associate many attributes with a component.

5.2.1.1 Attribute

Because most ADCUE components can be described with attributes, it makes sense to first describe how an attribute for a component is specified. A component spans one line; the template is as follows:

```
{ Attribute: <Type> “<Name>” <Initial Value> [“<Units>”] }
```

The four tags are:

- **<Type>**: Specifies the type of attribute. Valid values include Number, Text, and Boolean. Number includes both integer and floating point values. Text may be any sequence of characters enclosed between “quotation marks”. Boolean types may only be true or false.

- **“<Name>”**: A string representing the name of an attribute. For clarification, the space is considered a valid character so valid attribute names may include “Net Worth”, “Last Name”, etc.

- **<Initial Value>**: Attributes have an initial, default value when the ADCUE system is first started. The value must match the type, so if the type is Number, values such as 3.14 or 10 are valid, but “pi”, “1”, or false are not.

- **[“<Units>”]**: It is beneficial to allow the user to link units of measure to a particular attribute. This is optional as indicated by square brackets. Note that units are specified within a string; this is done so complicated units may be used, such as “kg m/s.” Omitting the units results in a unitless attribute. It is encouraged to specify the units if it is known to clarify the system.

5.2.1.2 ENVIRONMENT

The environment file contains attributes globally affixed to the whole simulation and a list of agents to use in the simulation. Because only one environment may exist at a time, there is no need to differentiate between multiple environments. As a result, the environment is not named and is stored in the file “Environment.txt.” The format for this file is as follows:

```
{ Attribute: <Type> “<Name>” <Initial Value> [“<Units>”] }
{ Use: <Agent Name> }
Update:
```

From this file format specification, three blocks are used: a list of attributes, a list of agents to use, and an update function. The attribute template has already been described. The *Use* statement imports agents into the simulation. For example, the statement:

```
Use: Smith
```

Loads the agent specified in the file “Agent.Smith.txt” into the ADCUE system and activates it so it can begin to execute actions. Other agent files may be stored in the directory and loaded by the system, but they will not be used until the environment explicitly places them in the simulation via the Use command. The Update is a standalone line that starts the final block of the file. Every line following the block is part of an update routine that is designed to run every cycle in the simulation. This allows the environment to regulate itself, such as raising the temperature as time elapse during the day. However, this is not currently implemented; it is provided only as a provision for future implementations. An example environment component file is:

```
Attribute: Number "Temperature" 75 "deg F"
Attribute: Number "Refrigerator.Temperature" 40 "deg F"
Attribute: Text "Forecast" "Night"

Use: Agent.Smith

Update:
```

Through this example, the environment is described with attributes for the temperature of the room, the temperature of the refrigerator, and the weather forecast. Initial values are specified as 75° F, 40° F, and “night”; furthermore, these attributes can take on new values during the execution of the simulation. The environment is further described by introducing the agent *Smith* into the system.

5.2.1.3 AGENT

The agent contains two blocks: objectives and attributes:

```
{ Objective: <Name> }

{ Attribute: <Type> “<Name>” <Initial Value> [“<Units>”] }
```

An objective block lists objectives that must be achieved in the order listed. The conditions that comprise the objective state are listed in the respective files named “Objective.<Name>.txt”. An example agent component file is “Agent.Smith.txt”:

```
Objective: Not Hungry

Attribute: Text "Location" "Bed"
Attribute: Number "Weight" 150 "lbs"
Attribute: Boolean "Hungry" true
```

5.2.1.4 OBJECTIVE

The objective file format has one block that lists conditions that must be true before the objective can be considered achieved. The conditions in the objective need not be satisfied in any particular order; once all conditions are true, the agent considers the objective accomplished successfully.

```
{ Condition: <Attribute Name> <Relationship> <Attribute Value> }
```

As mentioned in the previous chapter, a condition is the relationship between an attribute and its value. The attribute name must be fully qualified. This means that the component type must be specified as well as the name. For instance, if the condition attribute we want to satisfy is *Smith.Hungry*, the component type must be prefixed: *Agent.Smith.Hungry*. Because objectives are assigned to particular agents, it is desirable to have reusable objectives that can be assigned to more than one agent. Thus, the *me* component type shortcut was introduced. Instead of statically setting the agent type at runtime, the agent can be determined dynamically at runtime. For instead of limiting the objective to *Agent.Smith.Hungry*, the shortcut *me.Hungry* can be used. When the objective is assigned to any agent, the prefix is automatically added by ADCUE. In this manner, the objective can be assigned to *Agent.Smith* and *Agent.Jane* without modification, as seen in Figure 6.

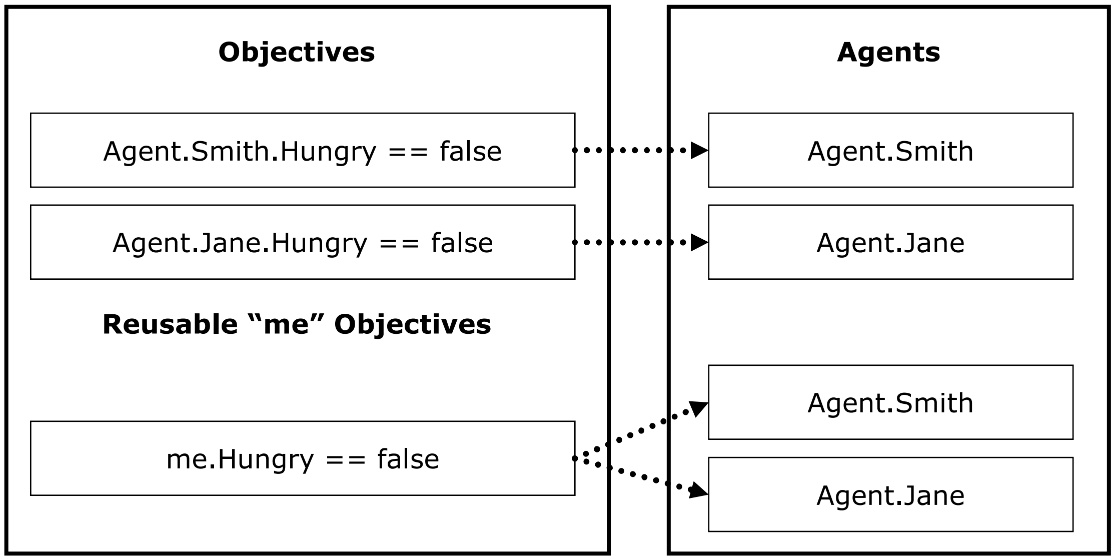


Figure 6: Assigning reusable “me” objectives

The relationship may be explained in the following way: Check if <Attribute Value> is <Relationship> to the value contained in <Attribute Name>. The relationship between the attribute’s value and the value specified by the user condition may be one of the following values (the symbols are listed in parenthesis).

- Equals (==)
- Not Equals (!=)
- Less or Equal (<=)
- Greater or Equal (>=)
- Less than (<)
- Greater than (>)

An simple example of an objective is “Objective.Not Hungry.txt”:

```
Condition: me.Hungry == false
```

Notice this example makes use of the *me* shortcut so the *Not Hungry* objective can be assigned to multiple agents. It is important to note that in these cases, any agent to which this objective is assigned must have a Boolean attribute named *Hungry*.

5.2.1.5 Resource

The resource file format is similar to the environment format seen earlier. It contains the initial amount for the resource, the attributes associated with the amount, and an update routine (currently unused). The file format is:

```
Amount: <Number Value> [“<Units>”]
{ Attribute: <Type> “<Name>” <Initial Value> [“<Units>”] }
```

```
Update:
```

All resources must start out with an initial numeric amount with the units optionally specified. A list of attributes may also be associated with the resource. The update routine, as is the case with the environment update routine, is executed once per simulation cycle and is used to allow the resource to manage itself. In the example of the resource *Milk*, the milk carton can detect its location and subsequently update its temperature depending on whether it is located in or out of the refrigerator. This update functionality is currently provided for future expansion of the model. An example of the resource file is:

```
Amount: 1 "gal"

Attribute: Boolean "Sour" false "gal"
Attribute: Number "Temp" 40 "deg F"
Attribute: Text "Location" "Refrigerator"

Update:

if (me.Location == "Refrigerator ")
{
    me.Temp -= (me.Temp - Environment.Refrigerator.Temp) / 100;
}
if (me.Location != "Refrigerator ")
{
    me.Temp += (Environment.Temp - me.Temp) / 100;
}

if (me.Temperature > 80)
   inject Event.Sour Milk;
```

As can be seen, self-describing components are able to update their own attributes and subsequently inject events such as *Sour Milk* if the carton was left outside the refrigerator too long. Notice the use of the shortcut *me* is used to refer to the current resource, thus allowing the resource to be easily renamed at a later point in time..

5.2.1.6 Action

Actions consist of preconditions and postconditions. The format is:

```
{ Condition: <Attribute Name> <Relationship> <Attribute Value> }

Postconditions:
{ <Attribute Name> <Operator> <Value> }
```

Because preconditions are similar to objective conditions in that they must be satisfied before the action can be executed, the preconditions are listed in the same format as those in the objective component files. The postconditions consist of operations on attributes in the environment, referenced by the attribute name. The <Operator> can take on one of the following operators: "=", "+=", "-=", "*=", "/=". Currently, only constant values are allowed. However, for increased flexibility in the future, values stored in attributes will be allowed. An example of an action is "Action.Eat Food.txt":

```
Precondition: me.Hungry == true
Precondition: Resource.Meals.Amount > 1

Postconditions:

Resource.Meals.Amount -= 1
me.Hungry = false
```

Again, the shortcut *me* is used to refer to the agent executing the action.

5.2.1.7 Event

The file format for an event is the same as an action, except the event file format omits the preconditions. Thus, an event such as "Event.Sour Milk.txt" migh consist of:

```
Postconditions:
        Resource.Milk.Sour = true
```

5.2.2 SYSTEM OUTPUT

The output of the system is a planning graph and, optionally, the parts of the planning graph that are affected by events. To facilitate the clarity of the generated graphs and the consequences of events on a plan, ADCUE uses an external tool to generate graphical representations of the graphs and the effects of events on actions in the plan. This is described in more detail in the testing section, as it is used to validate ADCUE's operation.

5.3 ADCUE ARCHITECTURE

The ADCUE system consists of several different packages, each with a separate functionality. The modules are divided into three separate levels as seen in Figure 7. The key package is the ADCUE Core, which contains the functionality to load/manipulate/reason with the ADCUE components. This level is referred to as the model level as it contains the implementation of the ADCUE approach. At the scenario level, any number of scenario packages can be developed which use ADCUE to implement situations in which agents automatically reason about the consequences of events. Two supporting packages, the Zebulon base library and the Google hashmap are used extensively. The Zebulon base library is a library developed at the Robotics Laboratory at UCF to provide high-level cross-platform commonly used functionality for C++ programs. It includes classes to work with arrays, files, and parsing. The Google hashmap, developed by Google, is widely regarded as an extremely fast and efficient hash table, which is used to store/retrieve ADCUE components by names.

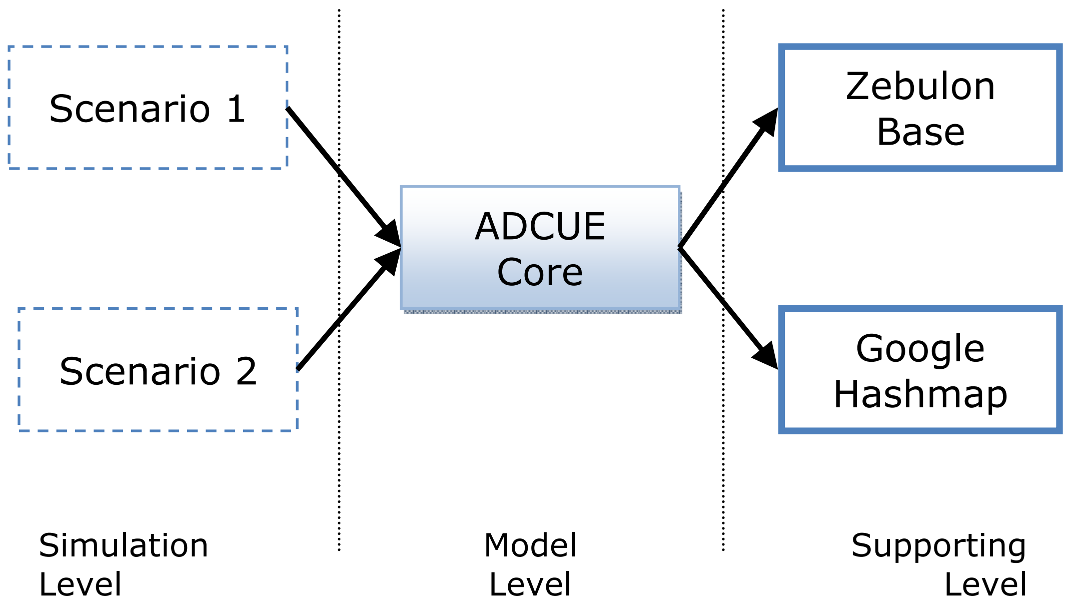


Figure 7: High level ADCUE package structure

The remainder of this implementation chapter focuses on the model level. The ADCUE Core is composed of a diverse set of classes. The first set of classes handles the representation of the ADCUE components. These classes include: Attribute, ADCUEComponent, Environment, Agent, Action, State, Condition, Routine, Block, and State. Each class is examined in detail in upcoming sections. The relationship between these classes can be seen in Figure 8. Notice all components derive from ADCUEComponent, which has an array of Attributes. The other set of classes are for generating/analyzing plans and is discussed later.

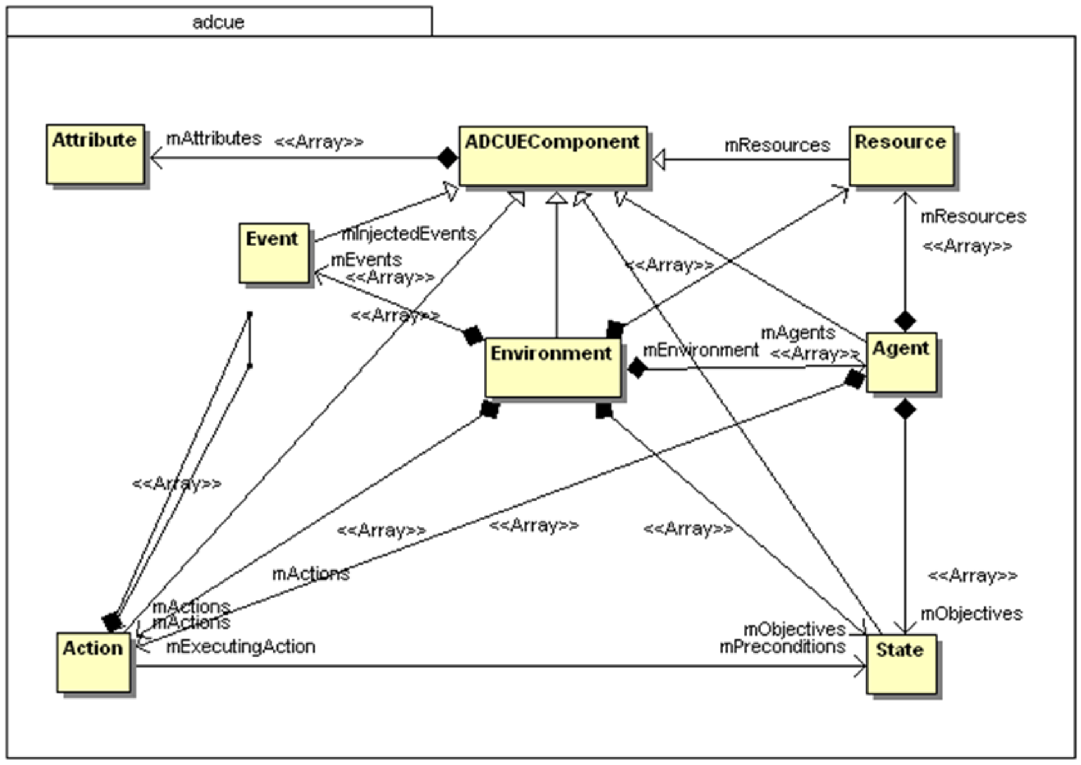


Figure 8: ADCUE representational class diagram

5.3.1 Attribute

The **Attribute** class (see class diagram in Figure 9) is the core of the ADCUE system. Nearly all ADCUE components can be described at least partially through a set of attributes. An attribute can hold textual, numeric, or a Boolean value as specified by the **AttributeType** enumeration. The Attribute class stores two sets of values: the initial value and the current value. The initial value is the value the attribute was first assigned and is stored because of its potential usefulness later in the simulation. The current value tracks how the attribute changes over time. The actual values are stored in an **AttributeValue** union that stores a text string, number, or Boolean value.

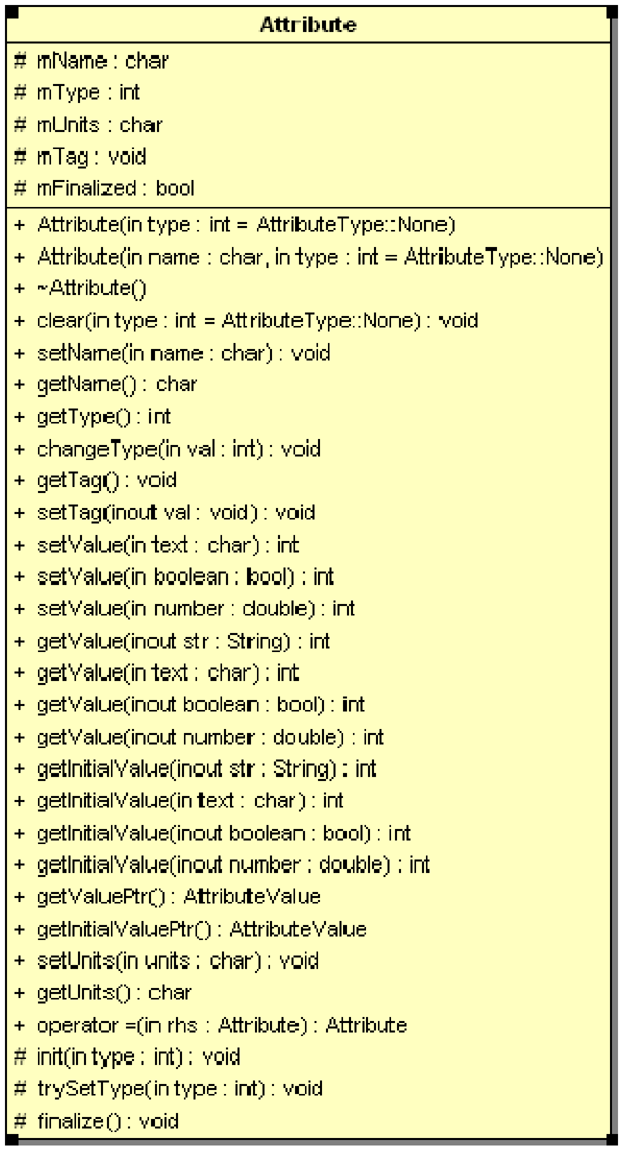
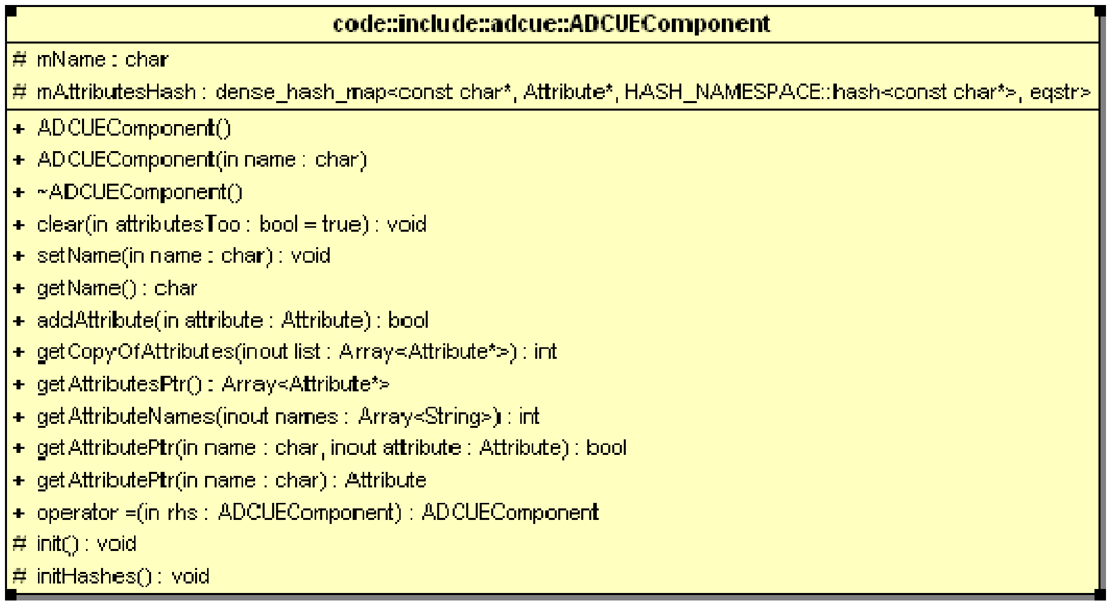

Attribute
# mName : char
# mType : int
# mUnits : char
# mTag : void
# mFinalized : bool
+ Attribute(in type : int = AttributeType::None)
+ Attribute(in name : char, in type : int = AttributeType::None)
+ ~Attribute()
+ clear(in type : int = AttributeType::None) : void
+ setName(in name : char) : void
+ getName() : char
+ getType() : int
+ changeType(in val : int) : void
+ getTag() : void
+ setTag(inout val : void) : void
+ setValue(in text : char) : int
+ setValue(in boolean : bool) : int
+ setValue(in number : double) : int
+ getValue(inout str : String) : int
+ getValue(in text : char) : int
+ getValue(inout boolean : bool) : int
+ getValue(inout number : double) : int
+ getInitialValue(inout str : String) : int
+ getInitialValue(in text : char) : int
+ getInitialValue(inout boolean : bool) : int
+ getInitialValue(inout number : double) : int
+ getValuePtr() : AttributeValue
+ getInitialValuePtr() : AttributeValue
+ setUnits(in units : char) : void
+ getUnits() : char
+ operator =(in rhs : Attribute) : Attribute
# init(in type : int) : void
# trySetType(in type : int) : void
# finalize() : void

Figure 9: Attribute class diagram

It is also possible to store units with an Attribute. Units are represented using strings, thus it can store any unit of measure. Additionally, any custom information can be associated with the attribute.

5.3.1.1 ADCUECOMPONENT

Since all ADCUE components have a name (except the environment) and a list of attributes, it is beneficial to encapsulate this functionality once within an abstract base class. The abstract base class created for this purpose is the **ADCUEComponent**. This class takes care of the tedious management of storing, adding, and modifying attributes in addition to storing and retrieving the component name. A class diagram of this base class can be seen in Figure 10.

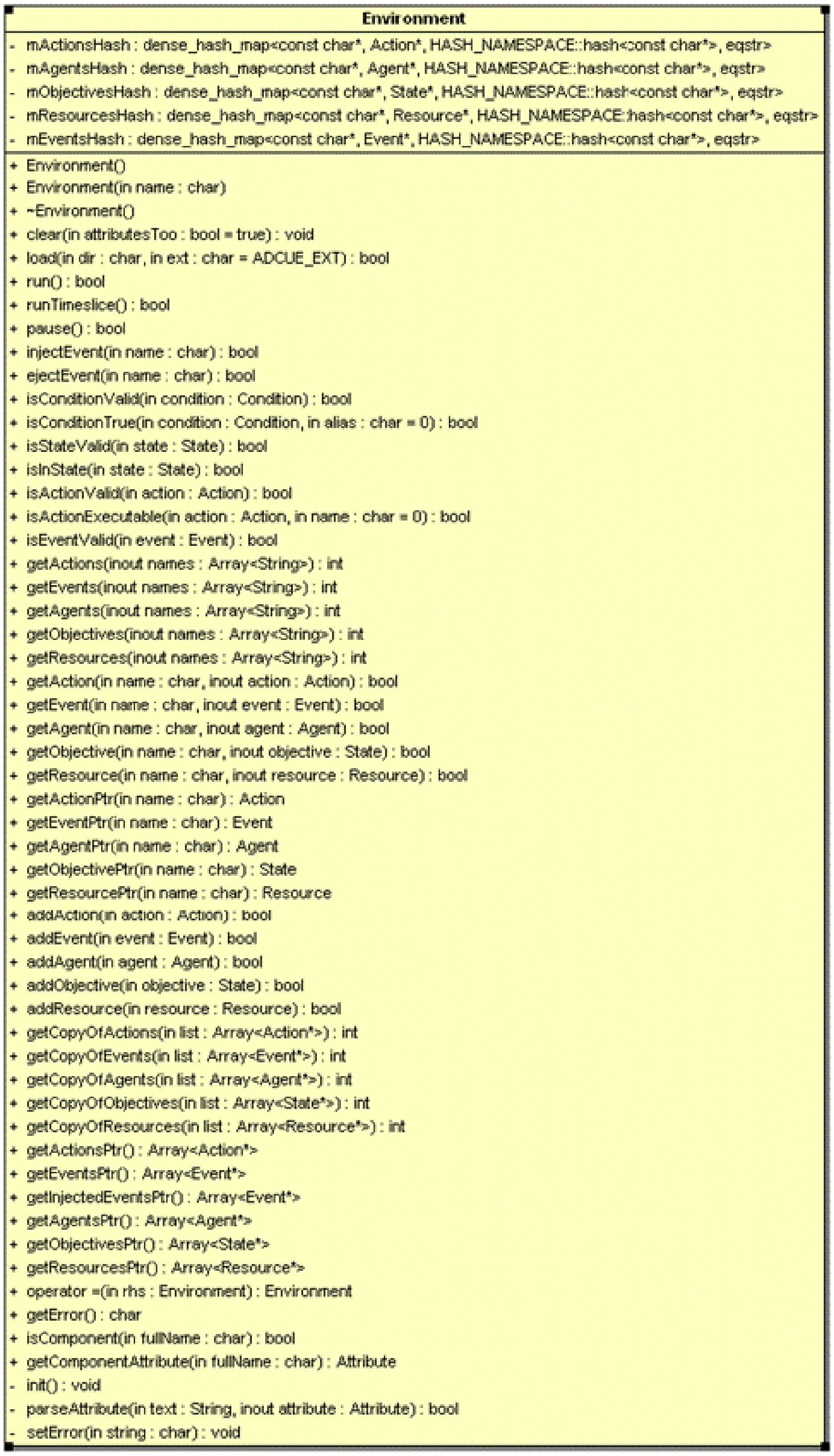


```
code::include::adcue::ADCUEComponent
# mName : char
# mAttributesHash : dense_hash_map<const char*, Attribute*, HASH_NAMESPACE::hash<const char*>, eqstr>
+ ADCUEComponent()
+ ADCUEComponent(in name : char)
+ ~ADCUEComponent()
+ clear(in attributesToo : bool = true) : void
+ setName(in name : char) : void
+ getName() : char
+ addAttribute(in attribute : Attribute) : bool
+ getCopyOfAttributes(inout list : Array<Attribute*>) : int
+ getAttributesPtr() : Array<Attribute*>
+ getAttributeNames(inout names : Array<String>) : int
+ getAttributePtr(in name : char, inout attribute : Attribute) : bool
+ getAttributePtr(in name : char) : Attribute
+ operator =(in rhs : ADCUEComponent) : ADCUEComponent
# init() : void
# initHashes() : void
```

Figure 10: ADCUEComponent class diagram

5.3.2 ENVIRONMENT

The **Environment** class (see Figure 11 for class diagram) is the class that has the responsibility of managing all other components in the ADCUE system. It handles loading the ADCUE components from file, linking and storing all components, retrieving them as necessary, and many other miscellaneous functions to control the environment. In fact, the Environment class serves as the point of contact for all other components in the system as it is the container in which all components are stored. The environment also calculates whether objective states have been reached yet so the agent knows when to finish the simulation.

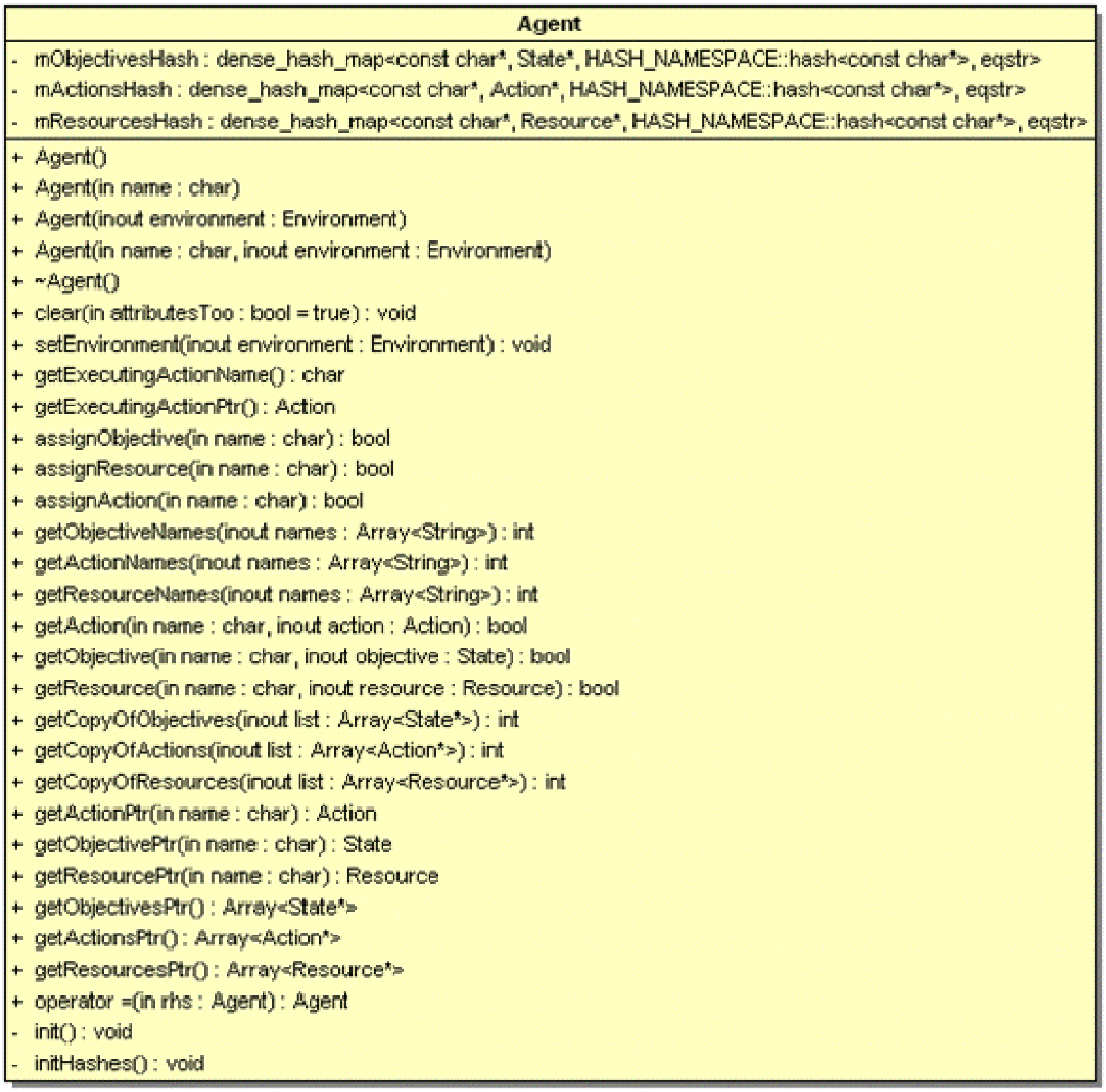


```
Environment
- mActionsHash : dense_hash_map<const char*, Action*, HASH_NAMESPACE::hash<const char*>, eqstr>
- mAgentsHash : dense_hash_map<const char*, Agent*, HASH_NAMESPACE::hash<const char*>, eqstr>
- mObjectivesHash : dense_hash_map<const char*, State*, HASH_NAMESPACE::hash<const char*>, eqstr>
- mResourcesHash : dense_hash_map<const char*, Resource*, HASH_NAMESPACE::hash<const char*>, eqstr>
- mEventsHash : dense_hash_map<const char*, Event*, HASH_NAMESPACE::hash<const char*>, eqstr>
+ Environment()
+ Environment(in name : char)
+ ~Environment()
+ clear(in attributesToo : bool = true) : void
+ load(in dir : char, in ext : char = ADCUE_EXT) : bool
+ run() : bool
+ runTimeslice() : bool
+ pause() : bool
+ injectEvent(in name : char) : bool
+ ejectEvent(in name : char) : bool
+ isConditionValid(in condition : Condition) : bool
+ isConditionTrue(in condition : Condition, in alias : char = 0) : bool
+ isStateValid(in state : State) : bool
+ isInState(in state : State) : bool
+ isActionValid(in action : Action) : bool
+ isActionExecutable(in action : Action, in name : char = 0) : bool
+ isEventValid(in event : Event) : bool
+ getActions(inout names : Array<String>) : int
+ getEvents(inout names : Array<String>) : int
+ getAgents(inout names : Array<String>) : int
+ getObjectives(inout names : Array<String>) : int
+ getResources(inout names : Array<String>) : int
+ getAction(in name : char, inout action : Action) : bool
+ getEvent(in name : char, inout event : Event) : bool
+ getAgent(in name : char, inout agent : Agent) : bool
+ getObjective(in name : char, inout objective : State) : bool
+ getResource(in name : char, inout resource : Resource) : bool
+ getActionPtr(in name : char) : Action
+ getEventPtr(in name : char) : Event
+ getAgentPtr(in name : char) : Agent
+ getObjectivePtr(in name : char) : State
+ getResourcePtr(in name : char) : Resource
+ addAction(in action : Action) : bool
+ addEvent(in event : Event) : bool
+ addAgent(in agent : Agent) : bool
+ addObjective(in objective : State) : bool
+ addResource(in resource : Resource) : bool
+ getCopyOfActions(in list : Array<Action*>) : int
+ getCopyOfEvents(in list : Array<Event*>) : int
+ getCopyOfAgents(in list : Array<Agent*>) : int
+ getCopyOfObjectives(in list : Array<State*>) : int
+ getCopyOfResources(in list : Array<Resource*>) : int
+ getActionsPtr() : Array<Action*>
+ getEventsPtr() : Array<Event*>
+ getInjectedEventsPtr() : Array<Event*>
+ getAgentsPtr() : Array<Agent*>
+ getObjectivesPtr() : Array<State*>
+ getResourcesPtr() : Array<Resource*>
+ operator =(in rhs : Environment) : Environment
+ getError() : char
+ isComponent(in fullName : char) : bool
+ getComponentAttribute(in fullName : char) : Attribute
- init() : void
- parseAttribute(in text : String, inout attribute : Attribute) : bool
- setError(in string : char) : void
```

Figure 11: Environment class diagram

5.3.3 Agent

An **Agent** class (see Figure 12 for class diagram) is described through a set of attributes and available actions to execute. It operates within the environment and can produce/consume/synthesize resources. The Agent class does not select which action to execute, this processes is done through the Environment class. However, the Agent class does allow access to the action the agent is currently executing.

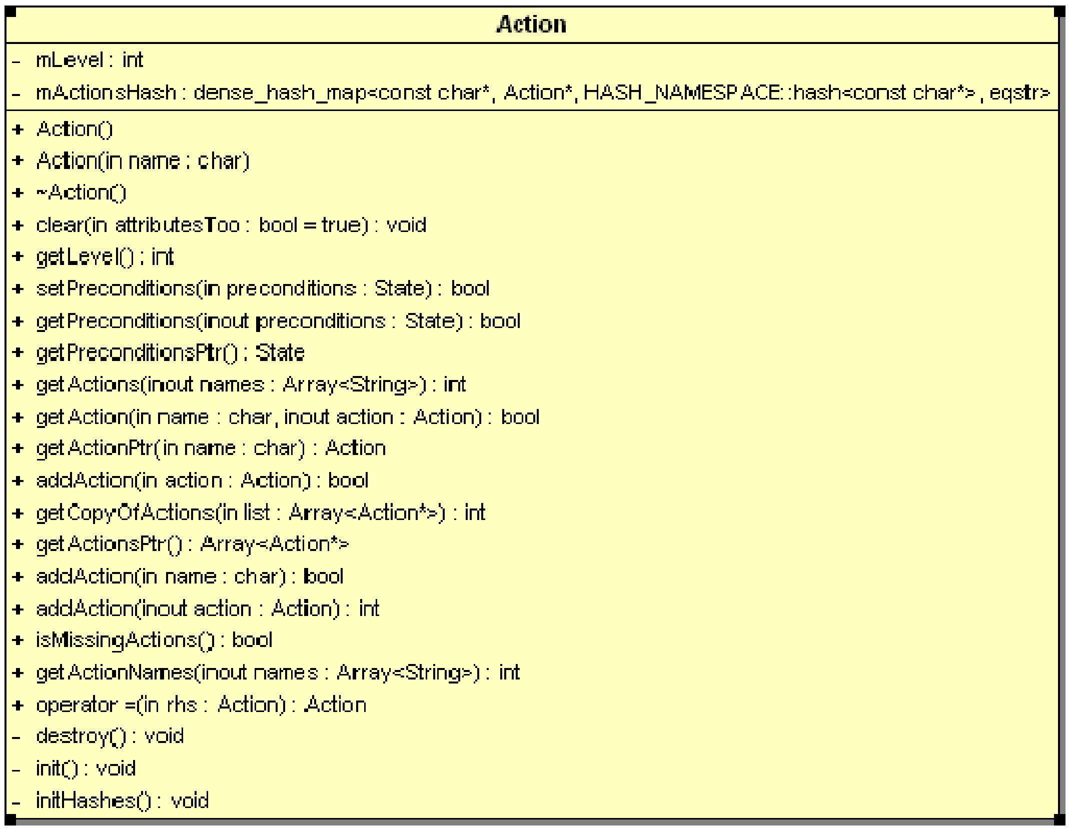


```
Agent
- mObjectivesHash : dense_hash_map<const char*, State*, HASH_NAMESPACE::hash<const char*>, eqstr>
- mActionsHash : dense_hash_map<const char*, Action*, HASH_NAMESPACE::hash<const char*>, eqstr>
- mResourcesHash : dense_hash_map<const char*, Resource*, HASH_NAMESPACE::hash<const char*>, eqstr>
+ Agent()
+ Agent(in name : char)
+ Agent(inout environment : Environment)
+ Agent(in name : char, inout environment : Environment)
+ ~Agent()
+ clear(in attributesToo : bool = true) : void
+ setEnvironment(inout environment : Environment) : void
+ getExecutingActionName() : char
+ getExecutingActionPtr() : Action
+ assignObjective(in name : char) : bool
+ assignResource(in name : char) : bool
+ assignAction(in name : char) : bool
+ getObjectiveNames(inout names : Array<String>) : int
+ getActionNames(inout names : Array<String>) : int
+ getResourceNames(inout names : Array<String>) : int
+ getAction(in name : char, inout action : Action) : bool
+ getObjective(in name : char, inout objective : State) : bool
+ getResource(in name : char, inout resource : Resource) : bool
+ getCopyOfObjectives(inout list : Array<State*>) : int
+ getCopyOfActions(inout list : Array<Action*>) : int
+ getCopyOfResources(inout list : Array<Resource*>) : int
+ getActionPtr(in name : char) : Action
+ getObjectivePtr(in name : char) : State
+ getResourcePtr(in name : char) : Resource
+ getObjectivesPtr() : Array<State*>
+ getActionsPtr() : Array<Action*>
+ getResourcesPtr() : Array<Resource*>
+ operator =(in rhs : Agent) : Agent
- init() : void
- initHashes() : void
```

Figure 12: Agent class diagram

5.3.4 ACTION

The **Action** class (see class diagram in Figure 13) is an executable black box that runs only under specific preconditions and changes the world through a set of postconditions. For more efficient processing and to organize the actions in a logical fashion, actions are grouped hierarchically by the **ActionLevel** enumeration. At the top level, a High level can have a number of Low level actions. The High level action can be thought of the abstract "what needs to happen" while the Low level actions can be thought of as "how to accomplish what is needed."



```
Action
- mLevel : int
- mActionsHash : dense_hash_map<const char*, Action*, HASH_NAMESPACE::hash<const char*> , eqstr>
+ Action()
+ Action(in name : char)
+ ~Action()
+ clear(in attributesToo : bool = true) : void
+ getLevel() : int
+ setPreconditions(in preconditions : State) : bool
+ getPreconditions(inout preconditions : State) : bool
+ getPreconditionsPtr() : State
+ getActions(inout names : Array<String>) : int
+ getAction(in name : char, inout action : Action) : bool
+ getActionPtr(in name : char) : Action
+ addAction(in action : Action) : bool
+ getCopyOfActions(in list : Array<Action*>) : int
+ getActionsPtr() : Array<Action*>
+ addAction(in name : char) : bool
+ addAction(inout action : Action) : int
+ isMissingActions() : bool
+ getActionNames(inout names : Array<String>) : int
+ operator =(in rhs : Action) : Action
- destroy() : void
- init() : void
- initHashes() : void
```

Figure 13: Action class diagram

5.3.5 Routine

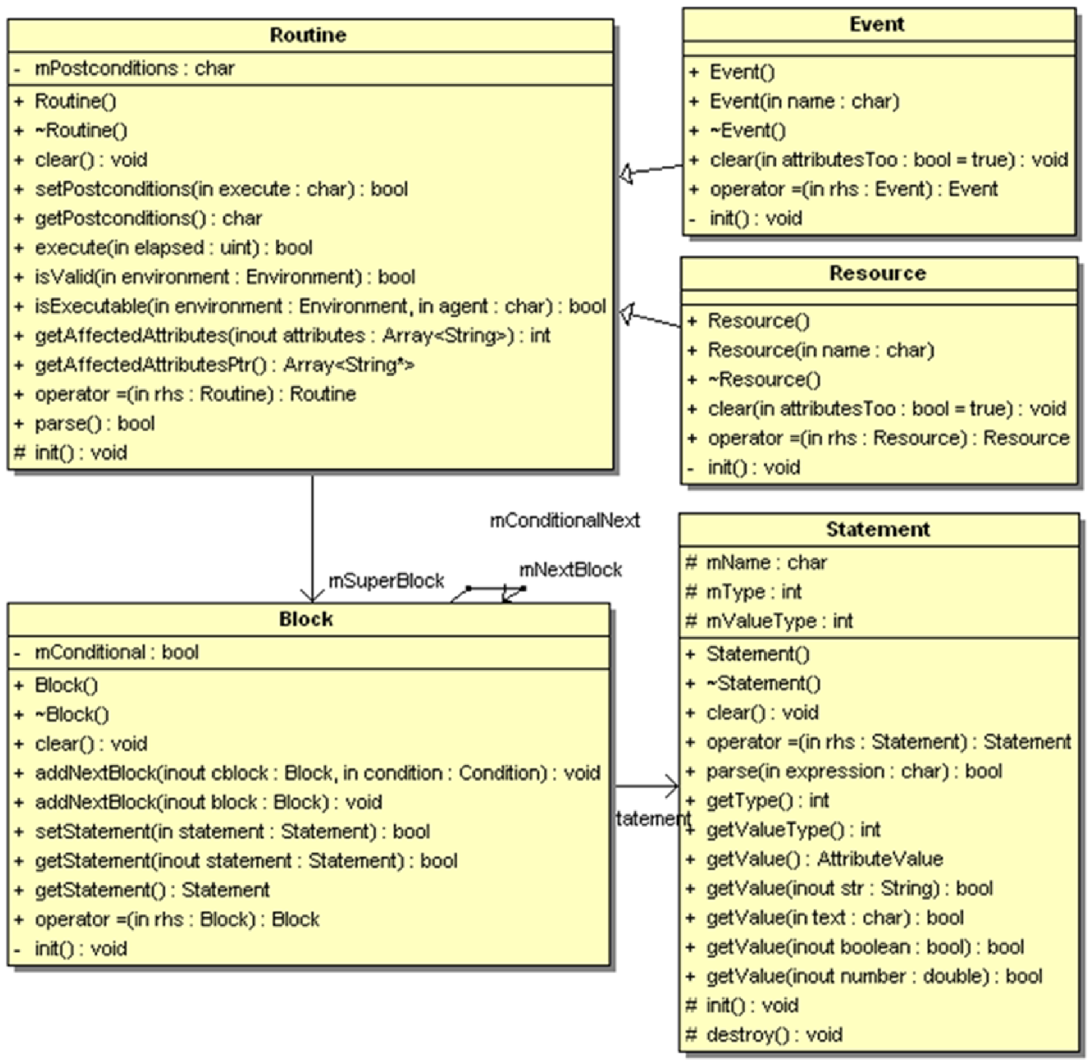

Routine
- mPostconditions : char
+ Routine()
+ ~Routine()
+ clear() : void
+ setPostconditions(in execute : char) : bool
+ getPostconditions() : char
+ execute(in elapsed : uint) : bool
+ isValid(in environment : Environment) : bool
+ isExecutable(in environment : Environment, in agent : char) : bool
+ getAffectedAttributes(inout attributes : Array<String>) : int
+ getAffectedAttributesPtr() : Array<String*>
+ operator =(in rhs : Routine) : Routine
+ parse() : bool
# init() : void
Event
+ Event()
+ Event(in name : char)
+ ~Event()
+ clear(in attributesToo : bool = true) : void
+ operator =(in rhs : Event) : Event
- init() : void
Resource
+ Resource()
+ Resource(in name : char)
+ ~Resource()
+ clear(in attributesToo : bool = true) : void
+ operator =(in rhs : Resource) : Resource
- init() : void
mConditionalNext
mSuperBlock
mNextBlock
Block
- mConditional : bool
+ Block()
+ ~Block()
+ clear() : void
+ addNextBlock(inout cblock : Block, in condition : Condition) : void
+ addNextBlock(inout block : Block) : void
+ setStatement(in statement : Statement) : bool
+ getStatement(inout statement : Statement) : bool
+ getStatement() : Statement
+ operator =(in rhs : Block) : Block
- init() : void
tatement
Statement
# mName : char
# mType : int
# mValueType : int
+ Statement()
+ ~Statement()
+ clear() : void
+ operator =(in rhs : Statement) : Statement
+ parse(in expression : char) : bool
+ getType() : int
+ getValueType() : int
+ getValue() : AttributeValue
+ getValue(inout str : String) : bool
+ getValue(in text : char) : bool
+ getValue(inout boolean : bool) : bool
+ getValue(inout number : double) : bool
# init() : void
# destroy() : void

Figure 14: Routine, Block, Statement, Event, & Resource class diagram

Many of the ADCUE components, such as Action, Event, Environment, Resource, etc, have the ability to change the environment. This is accomplished through postconditions stored in a **Routine** class (see Figure 14 for the class diagram). A routine is divided into a number of **Block** classes, each of which contains one or **Statement** classes. A statement is the lowest level and is executed as a single logical function. The purpose of a block is to allow conditional execution of groups of statements. For example, a group of statements may need to be executed only if the temperature dips below freezing.

5.3.6 EVENT & RESOURCE

Both the **Event** and **Resource** classes inherit from Routine and ADCUEComponent, as can be seen in Figure 14. From the ADCUEComponent, the event or resource inherits a name and attributes from the Routine class, an event inherits functionality to define postconditions and execute them while the resource inherits postconditions to update itself on a timed interval.

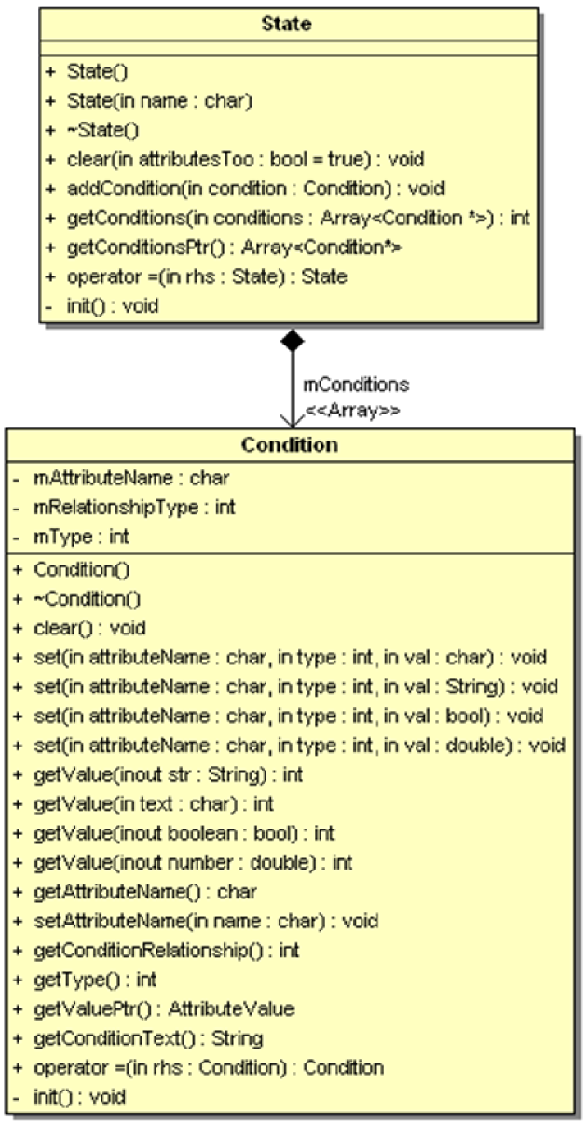

State
+ State()
+ State(in name : char)
+ ~State()
+ clear(in attributesToo : bool = true) : void
+ addCondition(in condition : Condition) : void
+ getConditions(in conditions : Array<Condition *>) : int
+ getConditionsPtr() : Array<Condition*>
+ operator =(in rhs : State) : State
- init() : void
mConditions
<<Array>>
Condition
- mAttributeName : char
- mRelationshipType : int
- mType : int
+ Condition()
+ ~Condition()
+ clear() : void
+ set(in attributeName : char, in type : int, in val : char) : void
+ set(in attributeName : char, in type : int, in val : String) : void
+ set(in attributeName : char, in type : int, in val : bool) : void
+ set(in attributeName : char, in type : int, in val : double) : void
+ getValue(inout str : String) : int
+ getValue(in text : char) : int
+ getValue(inout boolean : bool) : int
+ getValue(inout number : double) : int
+ getAttributeName() : char
+ setAttributeName(in name : char) : void
+ getConditionRelationship() : int
+ getType() : int
+ getValuePtr() : AttributeValue
+ getConditionText() : String
+ operator =(in rhs : Condition) : Condition
- init() : void

Figure 15: State and Condition class diagram

5.3.7 STATE & CONDITION

The **State** class defines a set of conditions that must be true to be in the state. Each condition is stored in a **Condition** class that links an attribute in the environment to a value. Once all conditions associated with a state are evaluated to be true, the state is said to be active. States represent objectives and allow the agent to determine when it has reached the objective state. The class diagram can be seen in Figure 15.

5.4 ADCUE Algorithm Operation

The ADCUE algorithm that determines the consequences of events is the combination of the intersection operator and the technique of slicing. Remember that the intersection operator can determine whether or not an action is affected by an event. However, this operation only checks one action (or step) in the plan. Slicing allows us to traverse the planning graph (forwards and backwards) and use the intersection operator on relevant nodes, thus examining upstream and downstream effects. Several classes aid with this process: Planner, PlanGraph, and PlanNode. The relationship and class diagrams for these classes can be seen in Figure 16.

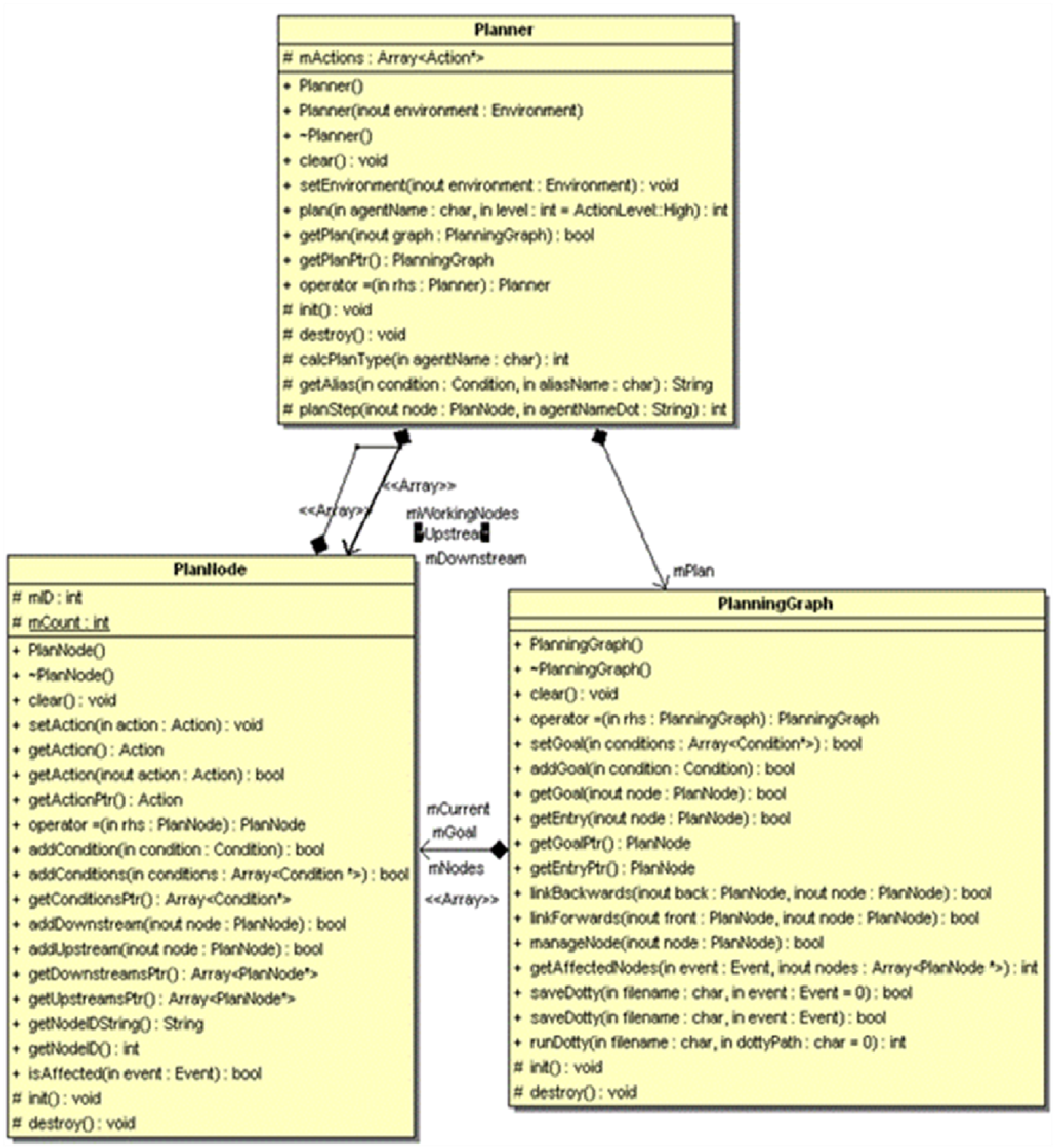

Planner
# mActions : Array<Action*>
+ Planner()
+ Planner(inout environment : Environment)
+ ~Planner()
+ clear() : void
+ setEnvironment(inout environment : Environment) : void
+ plan(in agentName : char, in level : int = ActionLevel::High) : int
+ getPlan(inout graph : PlanningGraph) : bool
+ getPlanPtr() : PlanningGraph
+ operator =(in rhs : Planner) : Planner
# init() : void
# destroy() : void
# calcPlanType(in agentName : char) : int
# getAlias(in condition : Condition, in aliasName : char) : String
# planStep(inout node : PlanNode, in agentNameDot : String) : int
<<Array>>
<<Array>>
mWorkingNodes
Upstream
mDownstream
mPlan
PlanNode
# mID : int
# mCount : int
+ PlanNode()
+ ~PlanNode()
+ clear() : void
+ setAction(in action : Action) : void
+ getAction() : Action
+ getAction(inout action : Action) : bool
+ getActionPtr() : Action
+ operator =(in rhs : PlanNode) : PlanNode
+ addCondition(in condition : Condition) : bool
+ addConditions(in conditions : Array<Condition *>) : bool
+ getConditionsPtr() : Array<Condition*>
+ addDownstream(inout node : PlanNode) : bool
+ addUpstream(inout node : PlanNode) : bool
+ getDownstreamsPtr() : Array<PlanNode*>
+ getUpstreamsPtr() : Array<PlanNode*>
+ getNodeIDString() : String
+ getNodeID() : int
+ isAffected(in event : Event) : bool
# init() : void
# destroy() : void
mCurrent
mGoal
mNodes
<<Array>>
PlanningGraph
+ PlanningGraph()
+ ~PlanningGraph()
+ clear() : void
+ operator =(in rhs : PlanningGraph) : PlanningGraph
+ setGoal(in conditions : Array<Condition*>) : bool
+ addGoal(in condition : Condition) : bool
+ getGoal(inout node : PlanNode) : bool
+ getEntry(inout node : PlanNode) : bool
+ getGoalPtr() : PlanNode
+ getEntryPtr() : PlanNode
+ linkBackwards(inout back : PlanNode, inout node : PlanNode) : bool
+ linkForwards(inout front : PlanNode, inout node : PlanNode) : bool
+ manageNode(inout node : PlanNode) : bool
+ getAffectedNodes(in event : Event, inout nodes : Array<PlanNode *>) : int
+ saveDotty(in filename : char, in event : Event = 0) : bool
+ saveDotty(in filename : char, in event : Event) : bool
+ runDotty(in filename : char, in dottyPath : char = 0) : int
# init() : void
# destroy() : void

Figure 16: Planner/PlanGraph/PlanNode class diagram

5.4.1 PLANNINGGRAPH & PLANNODE

A plan in ADCUE is stored in the **PlanningGraph** class as a bi-directional directed graph. As is the case with Contextual Graphs, the planning graph contains one entry node referred to as the initial state and a final state representing the objective state. All other nodes in the planning graph are of class **PlanNode** and represent ordered steps of the plan. Each PlanNode represents an action with preconditions and postconditions. Any path from the initial state to the objective state should include all the steps necessary to bring about the agent's goal. While execution flows forward, the bi-directional links allow the graph to be traversed either forward or backwards for analysis purposes.

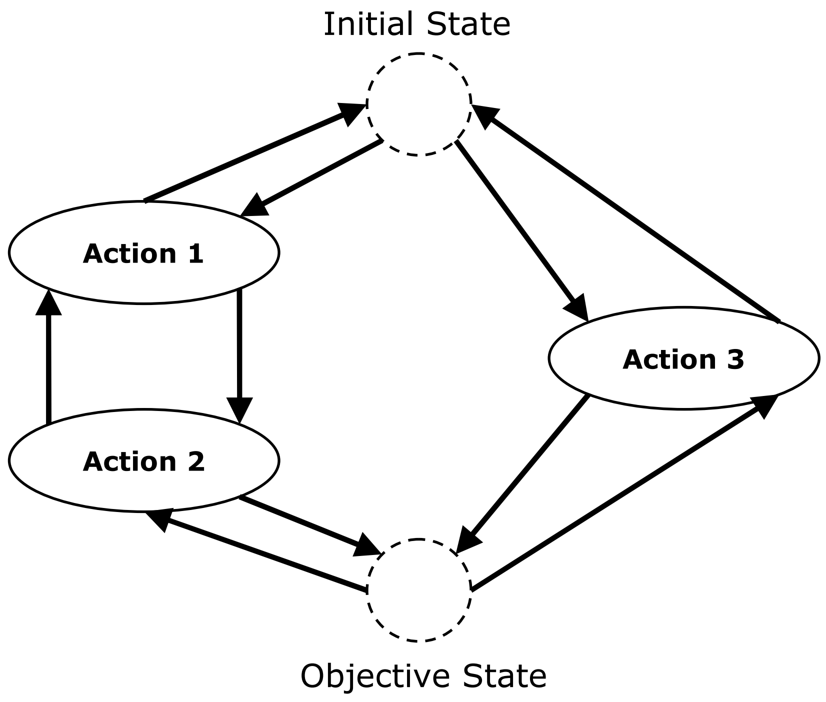


Figure 17: PlanningGraph with three PlanNodes

It is important to note that the PlanningGraph can contain many different paths and action steps to achieve the objective state. It is possible that some paths and actions may never be selected because a resource or condition is lacking. However, these paths and actions must be included in case the resource or condition becomes available at a later time.

5.4.2 PLANNER

The **Planner** class handles generating plans and analyzing plans with respect to an event. Interfacing to the environment, the planner uses the actions from the agent performing the planning to generate a plan that achieves the agent's objectives. Currently, the planner supports only generating simple plans. Larger, more complex plans with multiple objectives are beyond the capabilities of the planner and must be programmatically generated using the PlanningGraph class API.

5.4.3 ADCUE ANALYSIS ALGORITHM

The algorithm for plan analysis that determines the consequences of events on a plan is handled through the previous three classes. When an event is introduced into the plan, the technique of slicing used to apply the intersection operator to nodes in the planning graph. This corresponds to testing each step in the plan to see how affected it is by the event. Based on the results of applying these two techniques to all the paths in the plan, potentially affected actions are flagged with the attribute set returned by the intersection operator. The attribute sets contain which attributes the event affects and are used to determine the exact consequences of the event.

5.5 SUMMARY

The chapter concluded with an overview of topics such as the environment, high level design, and usage of the ADCUE library. The library will serve as the backend to all systems utilizing the ADCUE model, including the test cases presented in the next chapter.

CHAPTER 6: TESTING

By testing ADCUE with several test cases, a measure of how well ADCUE detects and determines the consequences of unexpected events can be obtained. This chapter describes the method used to design, set up, and run the tests. This also provides a metric with which the test results can be evaluated. Two separate testing scenarios, each consisting of several tests, are subsequently investigated and the results are concisely overviewed.

6.1 METHODOLOGY

The test procedure for ADCUE is specified as the evaluation of the ADCUE model operating under different scenarios. Typical use-cases where a user or agent is interested in the consequences of unexpected events are developed by creating a scenario through the various ADCUE components that comprise the simulation. A set of events are also specified for each scenario so they can be injected into the system. The injection of an event into the scenario is treated as a test of how well the system first detects and secondly determines the consequences of the event. For the tests to be unbiased, the expected outcome is documented before the tests are run and the actual results are subsequently compared. The tests performed in this chapter range from easy to difficult so as to accurately determine at what point ADCUE begins to fail.

6.1.1 Graphical Visualization

To aid the testing phase, ADCUE utilizes the AT&T GraphViz programs to generate professional-looking directed graphs from formatted text files. Through this visualization, it is much easier to see the planning graph and the steps of the plan affected by events. Since ADCUE automatically uses Graphviz to produce graphs from the internal testing procedures, these graphs can be considered part of the result of a test. The legend for these graphs is:

- **Boxes** represent states, typically either the initial or objective state. Key conditions that define the state are listed inside the box.
- **Ovals** represent action nodes, or individual steps in the plan.
- **Arrows** represent the flow of execution from one action to another, indicating the passage of time.
- **Diamonds** represent events injected into the system.
- **Filled ovals** represent actions that are affected by events.
- **Dashed arrows** represent connect diamonds with filled ovals, indicating an event is affecting an action.

6.2 Testing Scenarios

Two testing scenarios were developed: a simple and complex one. The simple scenario is the breakfast scenario used earlier and is used to validate the core functionality of the model. The complex one represents a more dynamic environment and is used to stress ADCUE, possibly to the point of failure. Both scenarios are developed by specifying each ADCUE component in a file and storing the file in a directory with the rest of the simulation components.

6.3 Breakfast Scenario

By now, the breakfast scenario used throughout the thesis should be familiar to the reader, making it a good choice for a test scenario. All the expected outcomes to the events mentioned have previously been predicted as an earlier example. To recap, the graphical relationship between the ADCUE components can be seen in Figure 1 in the previous chapter and the events were described in section 4.8.3.

6.3.1 Simulation Description

This section contains a listing consisting of the simulation files for the breakfast simulation. Each component name is bolded and followed by a horizontal rule and the specification of the individual component.

Action.Clean House

Precondition: me.Awake == true
Precondition: Resource.House.Dirty == true

Postconditions:

Resource.House.Dirty = false

Action.Cook Pancakes

Precondition: Resource.Pancake Mix.Amount >= 1
Precondition: Resource.Milk.Sour != false
Precondition: Resource.Milk.Amount >= 0.0625

Postconditions:

Resource.Pancake Mix.Amount -= 1
Resource.Milk.Amount -= 0.0625
Resource.Meals.Amount += 1

Action.Drive

Precondition: Resource.Car.Available == true
Precondition: Resource.Car.Gas > 1

Postconditions:

me.Location = "McDonalds"
Resource.Car.Gas -= 0.25

Action.Eat Breakfast

Precondition: me.Awake == true
Precondition: me.Hungry == true

Use: Cook Pancakes
Use: Drive
Use: Order McMuffin
Use: Pour Milk And Cereal
Use: Eat Food

Postconditions:

me.Hungry = false

Action.Eat Food

Precondition: me.Hungry == true
Precondition: Resource.Meals > 1

Postconditions:

Resource.Meals.Amount -= 1
me.Hungry = false

Action.Get Out Of Bed

Precondition: me.Location == "Bed"

Postconditions:

me.Location = "Bedroom"
me.Awake = true

Action.Order McMuffin

Precondition: me.Location == "McDonalds"
Precondition: Resource.Money.Amount >= 3.29

Postconditions:

Resource.Money.Amount -= 3.29
Resource.Meals.Amount += 1

Action.Pour Milk And Cereal

Precondition: Resource.Cereal.Amount >= 0.125
Precondition: Resource.Milk.Sour != false
Precondition: Resource.Milk.Amount >= 0.0625

Postconditions:

Resource.Cereal.Amount -= 0.124
Resource.Milk.Amount -= 0.0625
Resource.Meals.Amount += 1

Agent.Smith

Objective: Not Hungry
Objective: House Clean

Attribute: Text "Location" "Bed"
Attribute: Number "Weight" 150 "lbs"
Attribute: Boolean "Hungry" true

Environment

Attribute: Number "Temperature" 75 "deg F"
Attribute: Number "Refrigerator.Temperature" 40 "deg F"
Attribute: Text "Forecast" "Night"

Use: Agent.Smith

Event.Free McMuffin

Postconditions:

Resource.Meals.Amount += 1

Event.Sour Milk

Postconditions:

Resource.Milk.Sour = true

Event.Sunny Forecast

Postconditions:

Environment.Forecast = "Sunny"

Objective.Cool Down

Condition: Environment.Temperature < 75

Objective.Get Up

Condition: me.Location != "Bed"

Objective.House Clean

Condition: Resource.House.Dirty == false

Objective.Not Hungry

Condition: me.Hungry == false

Objective.Sleep

Condition: me.Location == "Bed"

Resource.Car

Amount: 1

Attribute: Boolean "Available" false
Attribute: Number "Gas" 12 "gal"

Resource.Cereal

Amount: 2 "bags"

Resource.House

Amount: 1

Attribute: Boolean "Dirty" true

Resource.Meals

Amount: 0 "meals"

Resource.Milk

Amount: 1 "gal"

Attribute: Boolean "Sour" false "gal"
Attribute: Number "Temperature" 40 "deg F"
Attribute: Text "Location" "Refrigerator"

Resource.Money

Amount: 50 "dollars"

Resource.Pancake Mix

Amount: 5 "packages"

6.3.2 TESTS

Three events were tested with the breakfast scenario: *Sunny Forecast, Sour Milk, Free McMuffin®.* All three were tested with the plan shown in Figure 18. To recap, the initial state is Agent *Smith* has woken up in the morning and is in bed feeling hungry. The objective is for Agent *Smith* to no longer be hungry. Three different paths exist for the agent to choose, each one representing a potential plan.

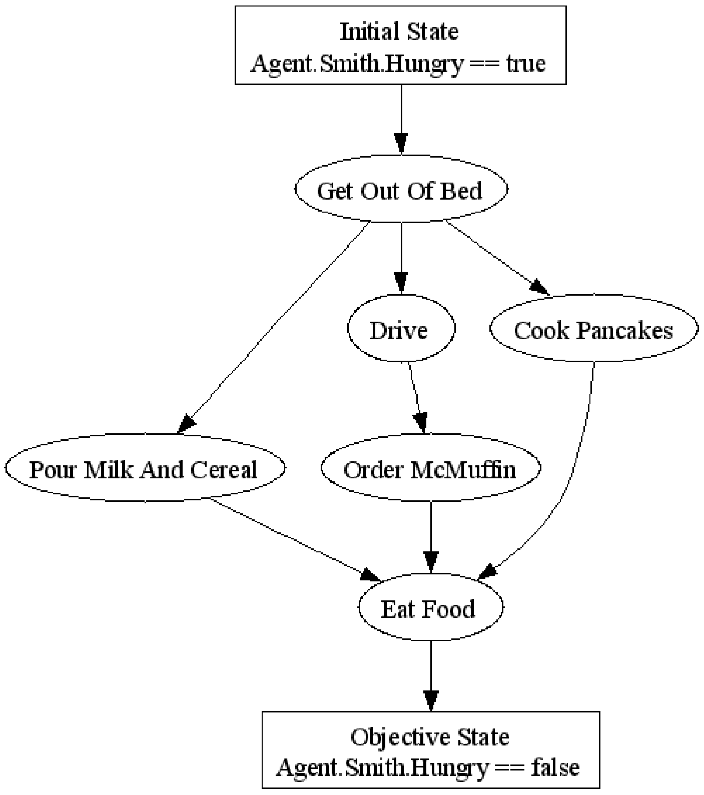


Figure 18: Breakfast scenario plan

6.3.2.1 TEST 1: SUNNY FORECAST

The first test is to inject the event *Sunny Forecast* into the plan and observe the results. The postconditions of this event consist of setting the attribute *Environment.Forecast* to “Sunny”. Because none of the actions in the plan have precondition dependencies on the forecast and none of the actions affect the weather, the expected outcome is for the event to have no consequences whatsoever.

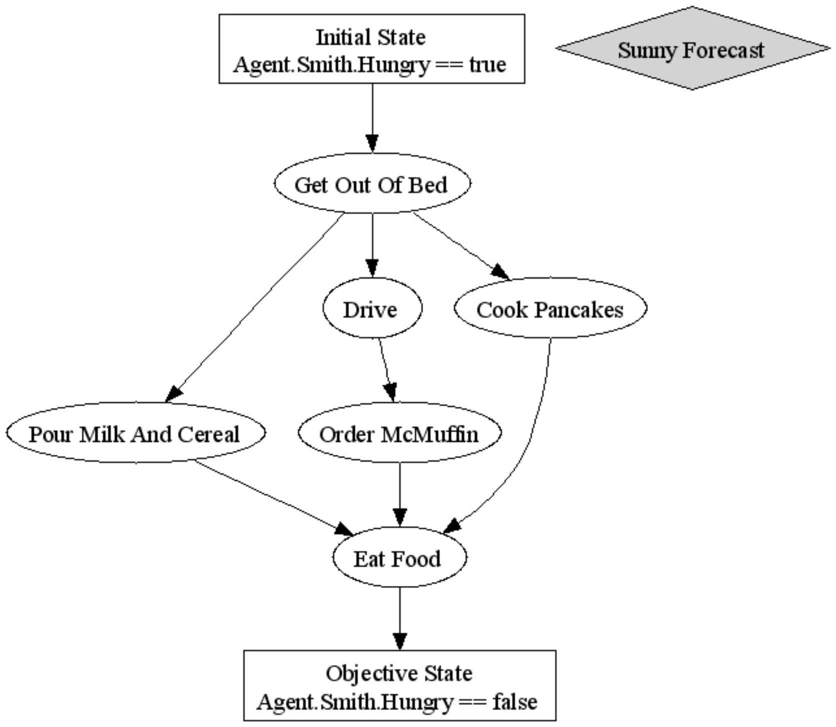


Figure 19: Breakfast scenario with Sunny Forecast event

As seen in Figure 19, the event *Sunny Forecast* is not linked to any actions, nor are any actions shaded, an indication they have been affected by an event. Thus, as expected, the unexpected event *Sunny Forecast* has no consequences on the plan.

6.3.2.2 Test 2: Sour Milk

The second test is to inject the event *Sour Milk* into the plan and observe the results. The postconditions of this event consist of setting the attribute *Milk.Sour* to "true". It is expected that both actions *Pour Milk And Cereal* and *Cook Pancakes* will be affected since they both have preconditions that allow them to execute only if the milk is good.

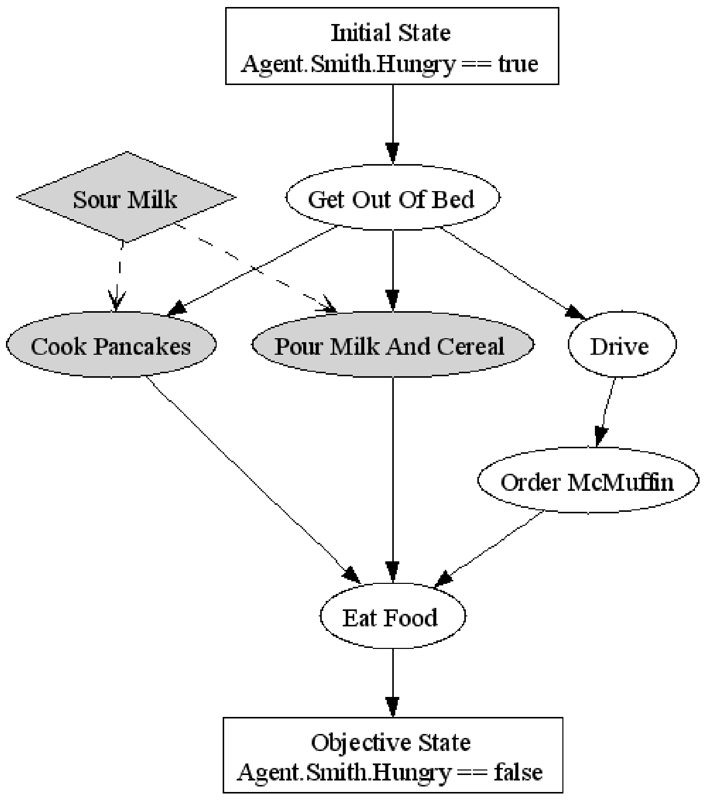


Figure 20: Breakfast scenario with Sour Milk event

As seen in Figure 20, ADCUE correctly detected that Sour Milk affects the predicted actions, leaving only one path open for the plan to succeed. By executing actions in hyper-realtime and propagating the attribute values through time, the agent can determine that the event disables the ability to execute either of the two actions. Thus, agent *Smith* can conclude that the consequences of the unexpected event *Sour Milk* prevents him from making his own breakfast; he is forced to go to McDonalds for a McMuffin®.

6.3.2.3 Test 3: Free McMuffin®

The second test is to inject the event *Free McMuffin®* into the plan and observe the results. This event represents *Agent.Smith* arriving at McDonalds and discovering a “buy one, get one free” deal for McMuffins®. The postconditions of this event consist of incrementing the value of the attribute *Meals.Amount* by 1. It is expected that the action *Order McMuffin®* and *Eat Food* will be affected because agent *Smith* will now be getting two meals instead of one.

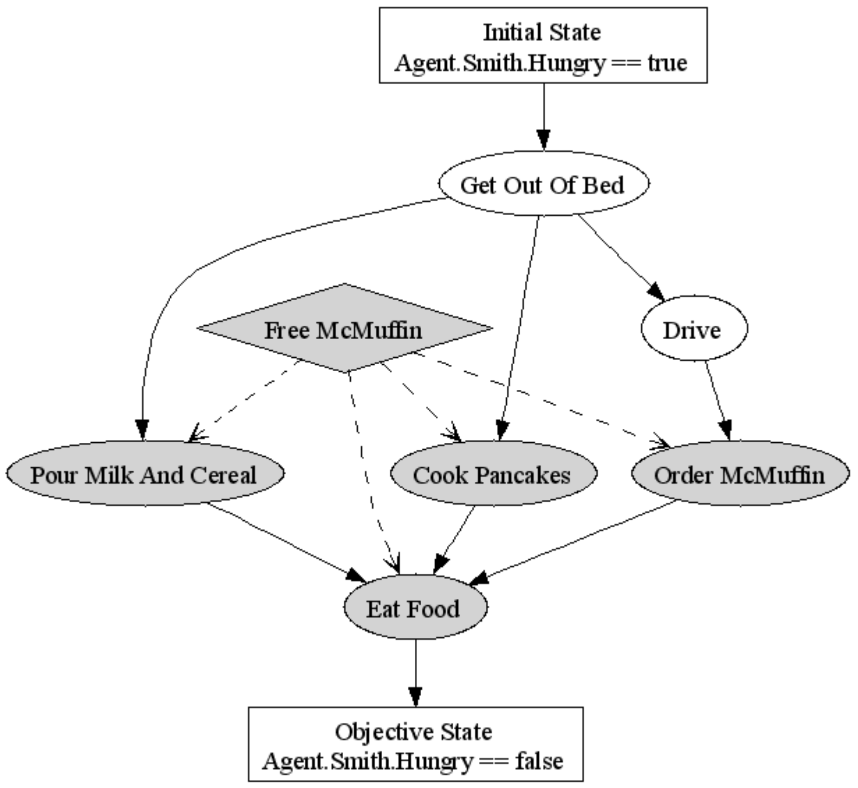


Figure 21: Breakfast scenario with Free McMuffin® event

As seen in Figure 21, the results were not as expected. While both *Order McMuffin®* and *Eat Food* actions are shown as affected, the two additional actions *Pour Milk And Cereal* and *Cook Pancakes* are also shown as being affected by *Free McMuffin®*. Why is this? Upon further inspection, it appears that the specification of the plan components have a logic mistake. While it is was desired that *Agent.Smith* only receive a free McMuffin® through the "buy one, get one free" deal at McDonalds, the postcondition "Meals.Amount += 1" does not do that. Instead, after thinking about it, this postcondition of event *Free McMuffin*® increases the attribute *Meals.Amount* regardless of whether the agent visits McDonalds. In order to rectify this mistake and more accurate represent the "buy one, get one free" event, the event should be injected into the system only for the paths where agent *Smith* visits McDonalds. Viewed through this new perspective, it makes sense that the other two actions were affected; the event *Free McMuffin*® is generating an additional meal no matter what. This reveals an interesting side effect of ADCUE: it also makes errors in specifying the preconditions clearly evident. Thus, while the results were not as expected, the error ultimately lay in human error specifying the preconditions rather than ADCUE. In this test, ADCUE performed as it should have.

6.4 UAV SCENARIO

While the breakfast scenario tested and verified that the core functionality of ADCUE was indeed working as originally envisioned, a more challenging scenario was designed to test the limits of the system and see how ADCUE could handle something more than a toy problem. The scenario chosen was that of the Predator Unmanned Aerial Vehicle (UAV) with a surveillance/reconnaissance mission. The purpose of this scenario is to experimentally verify the validity of the developed model in situations closer to the real world. Additionally, the selection of the UAV scenario is beneficial because of its relative simplicity, clear objectives, and current relevancy.

6.4.1 *PREDATOR UAV*

Unmanned Aerial Vehicles (UAVs) have a wide variety of uses within military operations. UAVs can be considered a long-range airplane controlled remotely by a base station. Among the several UAVs publicly used by the US military, the MQ-1 Predator UAV has achieved notoriety through its involvement in the Balkans [35], Afghanistan [36], and Iraq [37] as an armed reconnaissance spy plane. Created by the General Atomics Aeronautical Systems in 1994 [38], it is often referred to as a MALE (Medium-Altitude, Long-Endurance) UAV. Although originally meant to perform only surveillance tasks, in 2002, the Predator was authorized to carry out armed reconnaissance as well [38]. This makes it an ideal platform to test in a simulation.

6.4.1.1 PHYSICAL SPECIFICATIONS

All Predator UAV specifications have been converted to SI units for ease of calculation and simulation. Because of the nature of the Predator's missions and involvement in various military ventures, it can be difficult to obtain exact figures for the values of the specifications. This problem is further exacerbated by the continued evolvement of the Predator UAV over the past decade. However, while the purpose of the UAV scenario is not to perfectly model the airplane in questions, a best effort has been put forth to ensure the accuracy of the figures listed here. Sources of information include [39], [40], [41], [42], [43].

Table 4: Predator UAV Specifications

Attribute	Value	Attribute	Value
Length	8.22 m	Wingspan	14.8 m
Height	2.1 m	Fuel	378.54 liters
Weight (empty)	512 kg	Normal altitude	4572 m

Attribute	Value	Attribute	Value
Weight (max)	1020 kg	Loiter time at range	24 hours
Velocity (stall)	27.78 m/s	Payload	205 kg
Velocity (cruise)	37.55 m/s	Range	730.64 km
Velocity (max)	60.35 m/s	Ceiling	7620 m

6.4.1.2 Operational & Weather Limitations

The Predator does not tolerate difficult weather conditions very well [43]. During take-off, cross winds should not exceed 7.72 m/s and headwinds should be below 15.m/s. Additionally, the Predator cannot fly in heavy rain or icy conditions. It can handle only light turbulence. In terms of take-off and landing operational requirements, the Predator requires a minimum runway length of 22.86 m by 1524 m.

6.4.1.3 Equipment/Resources

The Predator carries a broad range of equipment in several different categories: surveillance, emergency, and military. In addition to standard equipment, the Predator can carry a payload of 205 kg to include any custom equipment required by the mission.

- **Emergency**: Because the Predator is flown remotely through a data link (either line of sight or satellite), the Predator has a feature that will fly it home to the base station in the event of a data link loss. In the event of an emergency, the Predator can be optionally configured to carry and use a parachute [43].
- **Surveillance**: As surveillance and reconnaissance is the primary objective of the Predator, it is well equipped to handle many different types of surveillance activities. The nose of the plane carries a color video camera

used mainly for flight control. For surveillance purposes, the Predator carries a variable aperture TV video camera and a variable aperture infrared video camera for night-time use. The Predator is also outfitted with synthetic aperture still-frame radar detection equipment.

- **Military**: On the defensive side, the Predator is built with a composite hull, which reduces its radar signature. Furthermore, the Predator flies much slower than typical military aircraft, so enemy radar detection equipment may fail to detect it as a threat [44]. Offensively, the Predator is armed with two AMG-114 Hellfire missiles that can attain speeds of Mach 5 to strike and kill an armored target within a range of 8 km [45].

6.4.1.4 Military Threats

In addition to a poor tolerance to harsh weather, the Predator is susceptible to several military threats. The most pressing threat comes from enemy aircraft and Surface-to-Air missiles (SAMs). Fighter planes, such as the MiG 25, have engaged and successfully shot down Predators. SAMs (Surface-to-Air missiles) have also destroyed Predators. In addition to vehicle-mounted surface-to-air missiles such as the PAC-2 Patriot missile, man portable missiles such as the FIM 92 B/C Stinger can threaten a lower flying Predator. The Patriot missile typically has a range of 70 km and can reach an altitude of 24 km [46], while the smaller, shoulder mounted Stinger has a range of 8 km and can only reach 3.8 km [47]. Given that the ceiling of a Predator is higher than the vertical range of a Stinger missile, the UAV is only susceptible to Stinger missile attacks when flying low.

6.4.2 Missions & Objectives

The Predator's "primary mission is interdiction and conducting armed reconnaissance against critical, perishable targets" [40]. While reconnaissance and surveillance constitute the largest part of its activities, the Predator can perform a variety of roles [39]. These additional roles include target acquisition, target designation, communications, battle-damage assessment, communications & electronics intelligence, jamming, chemical and biological warfare detection, search and rescue, and providing dispensable aircraft. Each of these roles achieves a different objective and will be examined in more detail.

6.4.2.1 Reconnaissance

Reconnaissance usually incorporates exploratory investigations to gather information about a resource or enemy. It can gather a wide range of information about an area, including information about weather, geography, enemy locations, and troop movements. These missions typically cover a large area of interest at cruise speeds, scouting for interesting information.

6.4.2.2 Surveillance

Surveillance constitutes a longer term version of reconnaissance and involves observing a single target for a long time. Typically surveillance takes place after reconnaissance when the points of particular interest have been discovered and require watching for further information and analysis. Examples of surveillance include watching a building, troop, or location. These missions cover a small area and typically fly slow to avoid detection.

6.4.2.3 Target Acquisition & Designation & Attack

During military operations, destroying key enemy equipment and personnel is critical. A Predator is an ideal way to search and find these targets without risking a pilot’s life. The mission can involve high cruise speeds when locating targets, or alternatively loitering in specific areas if intelligence indicates future enemy movement. Once a target is found, the Predator can report target acquisition, use a laser or other designator to guide another vehicle’s munitions against the target, or use one of the Hellfire missiles the Predator carries to attempt to destroy the target.

6.4.2.4 Battle-Damage Assessment

After attempted target destruction by either the UAV or another military unit, the Predator can provide real-time assessment of the damage inflicted upon the target. Such a mission requires the Predator to loiter over the targeted area, using its sensors to analyze the target. This is most important when deciding whether a follow up round is necessary after using long range weapons.

6.4.2.5 Communications Relay & Jamming

In a rapidly advancing military operation, the Predator can use the 205 kg of payload to carry communications relay equipment that enable friendly units to communicate with each other in the absence of usable local or global communication networks. In contrast to providing a communications relay for friendly units, the Predator can be equipped with communication jamming equipment to interfere with enemy communication. Both roles require a high endurance and low speed vehicle, a role the Predator fits into easily.

6.4.2.6 Biological and Chemical Warfare Detection

When suspected biological or chemical weapons have been used and it is dangerous to send manned teams for investigation, the Predator can carry sensitive equipment to determine if traces of biological or chemical weapons are evident. This eliminates the risk that the lingering effects of these weapons might pose to manned investigations. The mission parameters would be very similar to that of reconnaissance, with the exception of using advanced equipment for the detection of weapon remains.

6.4.2.7 Search and Rescue & Person Identification

The infrared camera that the Predator uses is sensitive enough to detect and distinguish a human heat source from 3,048 m [48]. This makes the Predator ideal for search and rescue situations where conditions are nonconductive to ground search and rescue. In these situations, the Predator would most likely perform rapid sweeps between cruise and max speed to most quickly locate the person.

6.4.2.8 Providing Dispensable Aircraft

Although a complete Predator system costs $40 million in 1997 dollars, much of the cost, both monetary and in terms of human life, remains far away in the ground station. Thus, Predators are ideally suited to high risk ventures where the plane may be destroyed. For example, during operations in Iraq, Predators stripped of sensors were flown into deliberately dangerous areas to test enemy anti-aircraft capabilities. Other uses consist of provoking the enemy to scramble fighters in the face of an unknown aerial threat. Such missions require fine control and UAV dashes with maximum velocity. [37]

6.4.3 SCENARIO LOCATION

For an accurate simulation, a location is a necessary component. A location must meet several requirements. First, airports or airfields must be present for the UAV to launch. Secondly, the location must be large and diverse enough to provide a realistic simulation. Because a Predator can fly approximately 805 km for surveillance purposes, the location should have an area of comparable size. The advantage to a diverse location is that it more accurately models situations a UAV operates in the real-world. For example, diversity should include topology, terrain, populations, population, and military bases or movements. Finally, the location must be realistic within the scope of UAV usage, i.e. choosing a location in Antarctica does not utilize the typical functionality of a UAV particularly well.

After careful consideration, the country of Afghanistan was chosen as the scenario location. Afghanistan satisfies the requirements of UAV usage, diversity, airfields, and location size. Furthermore, the US military has used the Predator extensively in Afghanistan. While Predator missions have not been released to the general public, it is beneficial to know that the UAV was simulated in locations that reflect real operations. Afghanistan also offers much diversity in terms of both terrain and population and has both urban and rural areas, including some in mountainous regions that the UAV will be unable to fly over. Finally, because of the guerilla warfare that often took place in the mountains and wilderness, the location is ideally suited for the type of work the Predator does best: locate, observe, and take action on enemy movements.

6.4.3.1 Map & Points of Interest

A Google Earth satellite map of Afghanistan can be seen in Figure 22. During Operation Enduring Freedom in Afghanistan, one of the heavily used U.S. airbases was located in Karshi-Khanabad, Uzbekistan [49]. Named K2, this airbase was only a short flight away from Afghanistan and could reach Kabul easily. The map shown in Figure 22 shows the launch point for the Predator UAV (Karshi) and the approximate range of the Predator. The capital of Afghanistan, Kabul, is located only 575 km away using the simplifying assumptions of a flat world and as-the-crow-flies navigation.

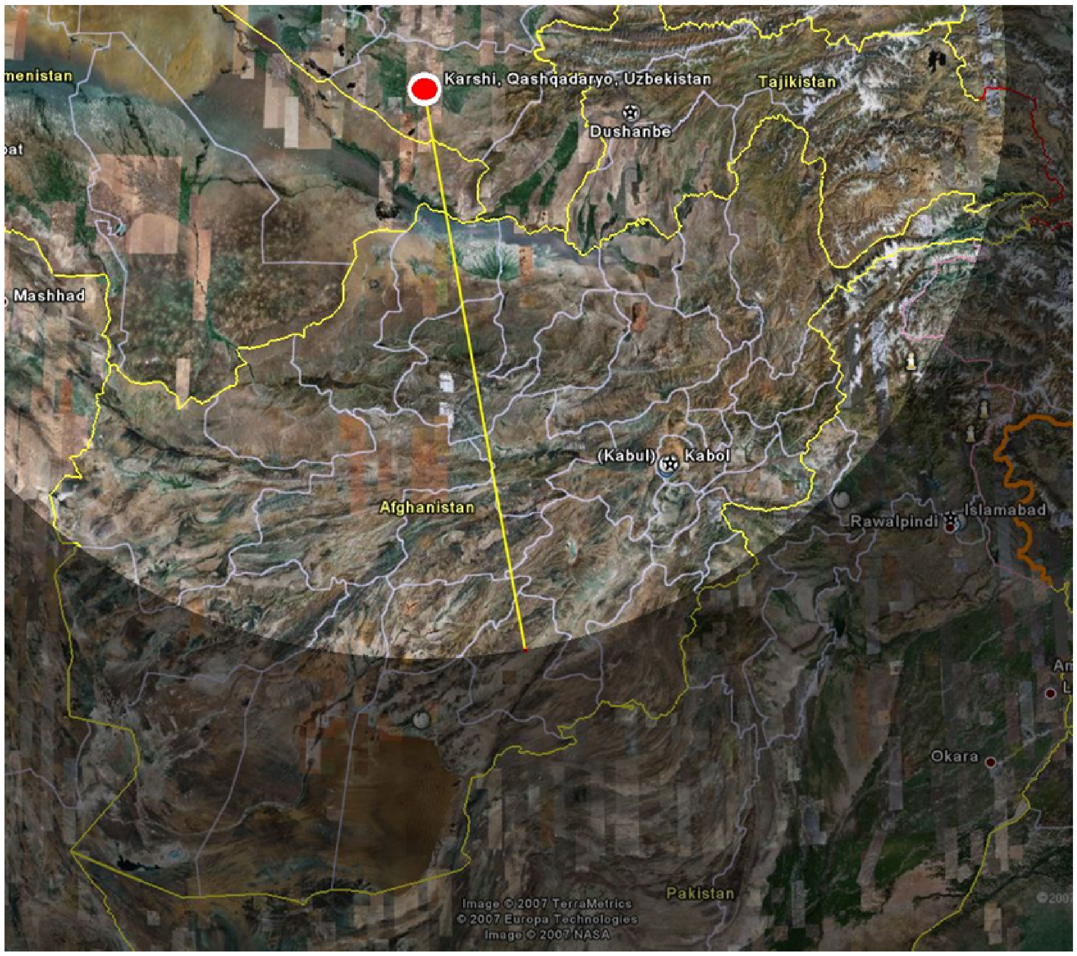


Figure 22: Afghanistan map with Predator launch point and range © Google

6.4.4 Scenario

Now that the Predator UAV has been described in detail and a location has been selected, it is possible to translate general knowledge about UAV missions into the ADCUE model of resources, objectives, actions, events, etc. By first describing the real-world capabilities as was done in previous sections, a solid knowledge base has been laid such that the scenario can be framed realistically in terms of ADCUE.

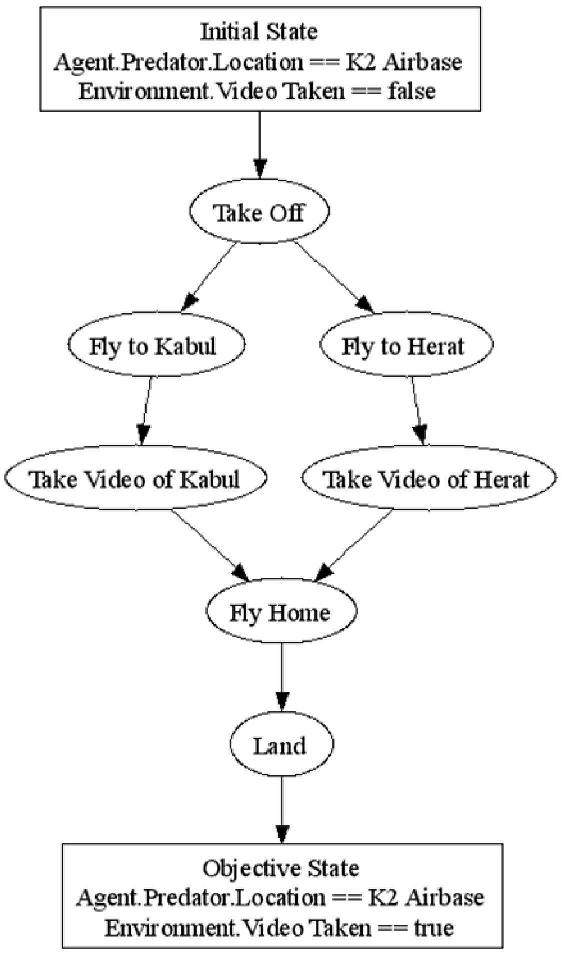

Initial State
Agent.Predator.Location == K2 Airbase
Environment.Video Taken == false
Take Off
Fly to Kabul
Fly to Herat
Take Video of Kabul
Take Video of Herat
Fly Home
Land
Objective State
Agent.Predator.Location == K2 Airbase
Environment.Video Taken == true

Figure 23: UAV base plan

6.4.4.1 Objectives

The objective for the Predator UAV is to launch from the K2 airbase, conduct reconnaissance of the capital city Kabul or Herat by taking some video, and return unharmed to the airbase. The events being injected into the system are gusty conditions with a headwind and poor visibility conditions at Herat.

6.4.5 Scenario Description

This section contains the base plan for the UAV scenario and a listing consisting of the simulation files for the breakfast simulation. Each component name is bolded and followed by a horizontal rule and the specification of the individual component.

6.4.5.1 Base Plan

The base plan seen in Figure 23 calls for the UAV to take off, fly to either Kabul or Herat for reconnaissance, take some video, fly back home to K2, and land safely.

6.4.5.2 Scenario Components

Action.Fly Home

Precondition: me.Alive == true
Precondition: me.Altitude > 0
Precondition: me.Velocity > 0

Postconditions:

me.Fuel -= 10
me.Location = "K2 Airbase"

Action.Fly to Herat

Precondition: me.Alive == true
Precondition: me.Altitude > 0
Precondition: me.Velocity > 0

Postconditions:

me.Fuel -= 10
me.Location = "Herat"

Action.Fly to Kabul

Precondition: me.Alive == true
Precondition: me.Altitude > 0
Precondition: me.Velocity > 0

Postconditions:

me.Fuel -= 10
me.Location = "Kabul"

Action.Land

Precondition: me.Altitude > 0
Precondition: me.Velocity > 0

Postconditions:

me.Fuel -= 10
me.Velocity = 0
me.Location = "K2 Airbase"
me.Altitude = 0

Action.Take Off

Precondition: me.Alive == true
Precondition: me.Altitude == 0
Precondition: Environment.Headwind < 20.43

Postconditions:

me.Fuel -= 10
me.Velocity = 37.75
me.Altitude = 4572

Action.Take Video of Herat

Precondition: me.Location == "Herat"
Precondition: Resource.Camera.Amount >= 1
Precondition: Resource.Camera.Working == true
Precondition: Environment.Herat Visibility > 5

Postconditions:

Resource.Camera.Taken = true

Action.Take Video of Kabul

Precondition: me.Location == "Kabul"
Precondition: Resource.Camera.Amount >= 1
Precondition: Resource.Camera.Working == true
Precondition: Environment.Kabul Visibility > 5

Postconditions:

Resource.Camera.Taken = true

Agent.Predator

Objective: Spy Kabul

Attribute: Boolean "Alive" true
Attribute: Text "Location" "K2 Airbase"

Attribute: Number "Length" 8.22 "m"
Attribute: Number "Height" 2.1 "m"

Attribute: Number "Empty Weight" 512 "kg"
Attribute: Number "Max Weight" 1020 "kg"
Attribute: Number "Wingspan" 14.8 "m"
Attribute: Number "Fuel" 378.54 "l"
Attribute: Number "Normal Altitude" 4572 "m"
Attribute: Number "Loiter Time" 24 "hr"
Attribute: Number "Stall Velocity" 27.78 "m/s"
Attribute: Number "Cruise Velocity" 37.75 "m/s"
Attribute: Number "Max Velocity" 60.35 "m/s"
Attribute: Number "Payload" 205 "kg"
Attribute: Number "Range" 730.64 "km"
Attribute: Number "Ceiling" 7620 "m"
Attribute: Number "Max Headwind" 15.43 "m/s"
Attribute: Number "Max Crosswind" 7.72 "m/s"
Attribute: Number "Min Runway Length" 1524 "m"
Attribute: Number "Min Runway Width" 22.86 "m"

Environment

Attribute: Number "Kabul Visibility" 5 "km"
Attribute: Number "Herat Visibility" 5 "km"
Attribute: Number "Headwind" 0 "m/s"
Attribute: Number "Crosswind" 0 "m/s"
Attribute: Boolean "Turbulence" false
Attribute: Boolean "Icy" false
Attribute: Boolean "Video Taken" false

Use: Agent.Predator

Update:

Event.Headwind

Postconditions:

Environment.Headwind = 18

Event.Poor Visibility

Postconditions:

Environment.Herat Visibility = 0.5

Event.Storm

Objective.Spy Herat

Condition: me.Location == "K2 Airbase"
Condition: Environment.Video Taken == true

Objective.Spy Kabul

Condition: me.Location == "K2 Airbase"
Condition: Environment.Video Taken == true

Resource.Hellfire Missles

Amount: 2 "missiles"

Update:

Resource.Video Camera

Amount: 1 "camera"

Attribute: Boolean "Working" true

Update:

Resource.Video of Herat

Amount: 0 "pics"

Attribute: Boolean "Taken" false
Attribute: Text "Location" "Herat" "city"

Resource.Video of Kabul

Amount: 0 "pics"

Attribute: Boolean "Taken" false
Attribute: Text "Location" "Kabul" "city"

6.4.6 Tests

Two tests are to be performed. The first is to determine how the events *Headwind* and *Poor Visibility* affect the plan.

6.4.6.1 Test 1: Headwind

The first test consists of injecting the *Headwind* event into the simulation. Because the *Headwind* event sets the attribute *Environment.Headwind* to a value greater than the Predator can handle, it is expected that the event will affect the *Take Off* event. The results of simulation using ADCUE seen in Figure 24 shows that it indeed does. The consequences in this case is complete failure of the mission because the UAV is unable to take off the runway.

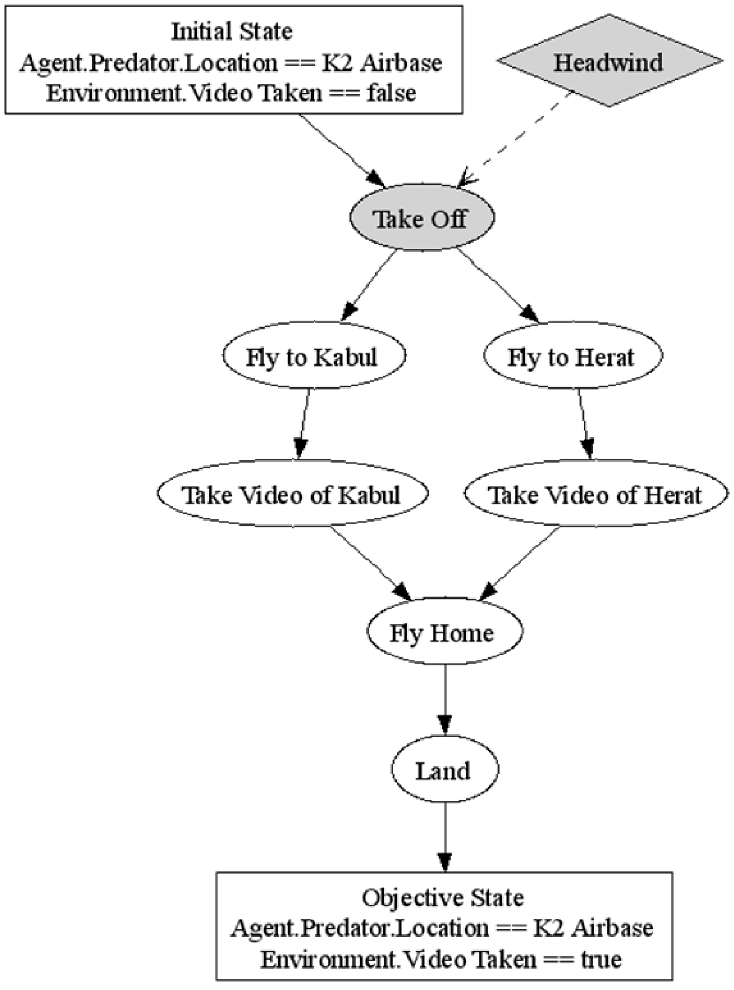

Initial State
Agent.Predator.Location == K2 Airbase
Environment.Video Taken == false
Headwind
Take Off
Fly to Kabul
Fly to Herat
Take Video of Kabul
Take Video of Herat
Fly Home
Land
Objective State
Agent.Predator.Location == K2 Airbase
Environment.Video Taken == true

Figure 24: UAV scenario with *Headwind* event

6.4.6.2 Test 2: Poor Visibility

The second test consists of injecting the event *Poor Visibility* event into the ADCUE scenario. This event lowers the attribute *Envrionment.Herat Visibility* to a value lower than the action *Take Video of Herat* requires for accurate pictures. Because the visibility over Kabul has not changed, it is expected that the event will only prevent the reconnaissance of Herat. As seen in Figure 25, ADCUE simulation shows that this is the case; a mission to Herat fails, but a mission to Kabul is not affected.

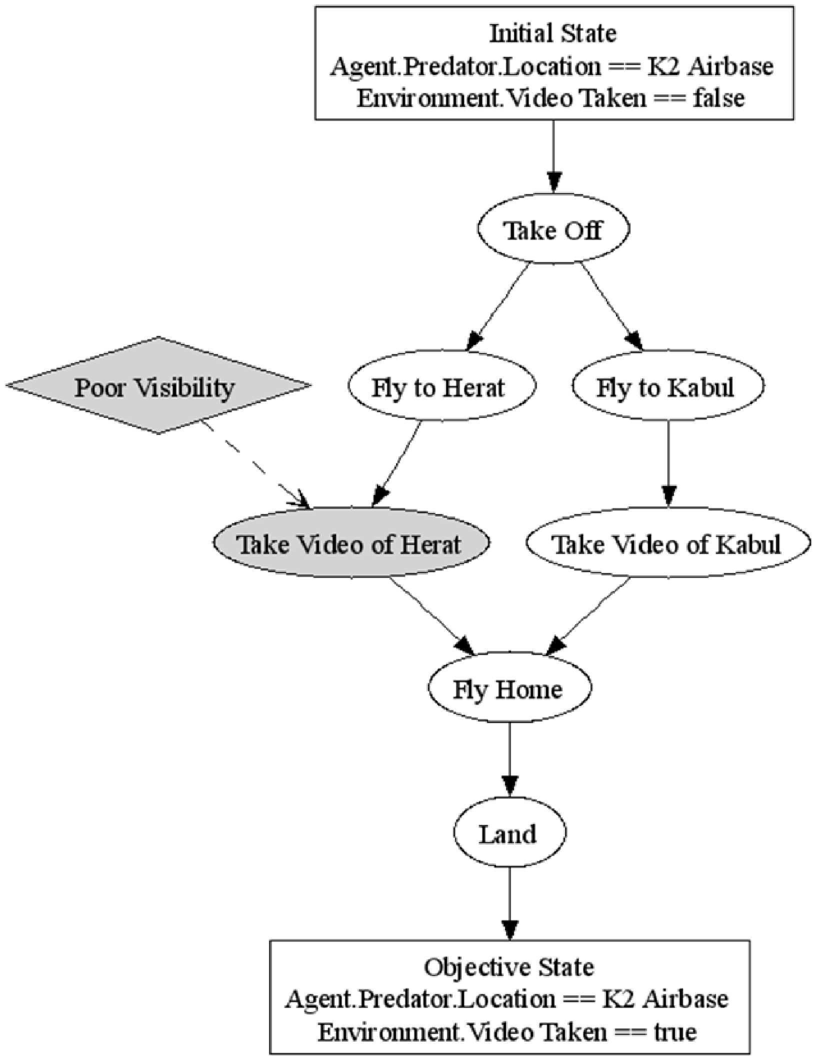

Initial State
Agent.Predator.Location == K2 Airbase
Environment.Video Taken == false
Take Off
Poor Visibility
Fly to Herat
Fly to Kabul
Take Video of Herat
Take Video of Kabul
Fly Home
Land
Objective State
Agent.Predator.Location == K2 Airbase
Environment.Video Taken == true

Figure 25: UAV scenario with *Poor Visibility* event in Herat

6.5 Overall Results

Through the testing of the simple breakfast scenario, expected and unexpected results were observed. First, the tests involving the breakfast scenario demonstrate that ADCUE does automatically detect and determine the consequences of events on a plan. One of the unexpected results was ADCUE's extra validation of the plan and the setup of the plan by catching a logic mistake. The testing of the slightly more complex UAV scenario using static plans showed that ADCUE can easily be applied to a new domain by changing the simulation files. Through the testing summarized in Table 1, it can be said that the most basic functionality of ADCUE works well and in some respects performs additional useful checks.

Table 5: Summary of testing and results

Simulation	**Event**	**Description**	**Actions Affected**	**Results**
Breakfast	*Sunny Forecast*	The daily forecast calls for sun	None	Pass
Breakfast	*Sour Milk*	The milk has gone sour	*Cook Pancakes, Pour Milk and Cereal*	Pass
Breakfast	*Free McMuffin*	A buy one, get one free McMuffin deal	*Free McMuffin, Pour Milk and Cereal, Cook Pancakes, Order McMuffin, Eat Food*	Pass. Unexpected results from human error
UAV	*Headwind*	High headwind	*Take Off*	Pass
UAV	*Poor Visibility*	Visibility at Herat is low	*Take Video of Herat*	Pass

CHAPTER 7: CONCLUSION

This chapter summarizes the thesis as a whole, including the problem, existing approaches, and the proposed system. Weaknesses or opportunities discovered are covered in the future research section. The final section concludes with a forward thinking perspective on results, lessons learned, and applicability to future projects.

7.1 SUMMARY

Throughout this thesis, automatic determination of the implications associated with unexpected events has been investigated and an approach entitled ADCUE was proposed to enable autonomous reasoning about unexpected events. Because this subject deals with a significant amount of planning, background concerning this topic was reviewed to ensure readers were properly familiarized with the material. Expanding upon the information covered in the background, the technical literature was reviewed to see how other researchers have approached this particular problem in the past. Most of the literature focused on dealing with events, or plan interruptions, by associating the events with particular contingency plans that provide workarounds for the consequences of the events. Several reviewed approaches exhibited solutions that used more intelligent behavior by considering the available actions within the world and dynamically changing the plan based on the unexpected event.

Based on paradigms discussed in the background and in the literature review sections, a new model entitled ADCUE was proposed in an attempt to address the problem. The two key components of the model, slicing and the intersection operator, were described and subsequently applied to the domain of planning and detecting unexpected events. Mathematical definitions, pseudo code, and examples were provided to ensure that the proposed approach was sound and could be fully understood. The actual implementation of the ADCUE model, along with all associated components, was detailed, including how each ADCUE component contributed to the overall model. Both automated and manual plans were covered, including the creation of a reusable library that other agent-based simulations can utilize.

To ensure that ADCUE worked as advertised and expected, two scenarios in different domains and of differing difficulty were created for testing purposes. Once imported into ADCUE, events were injected into the scenarios to observe the effectiveness of ADCUE's detection and determination of the consequences of the events on the plan. Each injected event was treated as a test and the predicted results were compared to the actual results to evaluate the model.

7.2 FUTURE RESEARCH

With promising initial results, it is worth following up on this work and researching topics that may lead to enhancements to ADCUE. Several significant subjects that could yield a more robust and capable system include multi-agent systems, probabilistic modeling, and learning.

7.2.1 Multi-Agent System

A simulation of a single agent has limited application. Just as it takes many soldiers to win a battle, oftentimes multiple agents need to work together to achieve the goal. Extending ADCUE with the provision for coordinating the efforts and capabilities of agents when developing a plan could be extremely beneficial. One potential approach assumes a hierarchical authority structure. Not only does this assumption simplify the analysis, but it also mimics reality as can be seen in most military structures. While individual soldiers within a platoon may perform completely independent actions, a platoon as a whole can be viewed from a higher authority as a single entity. As long as the platoon's individual actions contribute to a collective action (such as attack target), individual soldier action can be neglected from the point of a Major. The Lieutenant in charge of the platoon embodies the abilities of the entire platoon from the perspective of the Major. Thus, the Lieutenant acts as the communication layer between individual units of the platoon (soldiers or squads). Extrapolating this concept out into a generic, non-military sense, similar or dissimilar agents can be grouped together under the authority of a boss agent. The boss agent represents the collective actions of the group. Similar to a tiered organization chart or a military structure, multiple boss agents can be grouped together under the authority of a higher boss agent.

7.2.1.1 Potential Reasoning Mechanism

Given a hierarchical authority structure of a boss agent with several individual agents under her authority, the boss agent receives a set of objectives to achieve. To develop a plan to achieve these objectives, the boss queries each agent under her authority to see if it can achieve the objective on its own. If one agent is capable of achieving the objective, a plan involving only that agent is developed and the boss agent reports the objective can be achieved. However, if a single agent is incapable of single-handedly achieving the objective, the agent reports to the boss a partial plan consisting of all the actions the individual agent can contribute towards completing the objectives. The boss can then query other agents, asking them to contribute to the partially built plan. Thus, the plan is built incrementally, with agents committing actions to the plan as they have the ability to do so. This model assumes perfect willingness and cooperation between agents; treachery or laziness of agents is not considered.

This discussion brings up several interesting issues. First, how should the boss agent distribute the load? If a single agent is capable of achieving the objective, should the boss agent consider ordering several additional agents to work in parallel in order to accomplish the objective more quickly? Secondly, if multiple agents are capable of achieving an objective, how does the boss agent determine which agent(s) to assign to the plan? When unexpected events occur, who is responsible for determining the consequences and who is responsible for integrating the replanning across multiple tiers of command? These are all questions that provide interesting and useful future research topics.

7.2.2 PROBABILISTIC MODELING

One of the assumptions thus far is that all actions within ADCUE are completely deterministic, i.e. each action will always produce the same conditions in the same situation. However, this is not a realistic assumption when dealing with the real world. For instance, consider an Unmanned Aerial Vehicle (UAV) on a mission to survey and destroy a target with one of its Hellfire missiles. In this scenario, the action *Launch Hellfire Missile* should not always yield the postcondition

"*Target.Destroyed* equals *true*" because no missile type can hit the target 100% of the time. In the real world, inherent uncertainties lead to non-deterministic actions.

In order to remedy this shortcoming in ADCUE, the concept of probabilistic actions can be introduced. The difference between a normal action and a probabilistic action is that a probabilistic action can initiate an event. By injecting an event into the system, the action can effectively mimic the randomness found in a real world system by modeling uncertain action outcomes. For instance, if a Hellfire missile launched from a UAV has a history of 5% failure, the action *Launch Hellfire Missile* can initiate the event *Missed Target* 5% of the times the action is executed. This has several benefits. First, it provides a richer representation than a simple binary fail or succeed criteria. Second, it allows multiple outcomes to result from the execution of a single action. For instance, the *Launch Hellfire Missile* may also initiate a *Killed Civilians* event to model the accidental and unfortunate side effects of warfare. Through this introduction of probabilistic actions that can initiate events, ADCUE could represent the natural uncertainty found within the real world more accurately.

7.2.3 Learning

Learning is often an important method for augmenting the abilities of an agent over time. While ADCUE does not explicitly define any learning strategies, it can facilitate learning through the concept of an unknown event. An unknown event is an event whose postconditions are not fully known (or perhaps not known at all). Thus, the agent initially has no way of knowing how the event occurring will affect the plan. The obvious goal of the agent is to determine or learn the effects of the agent through experience. The agent first assumes a null hypothesis: the event will have no effect on the plan. As the agent executes the plan, the agent closely monitors the postconditions of actions to determine if any unexpected changes are occurring. If so, they can be attributed to the unknown event. By validating or invalidating the null hypothesis in this manner, a general trend of event effects may be able to be extracted. Because events postconditions may vary depending conditions (location of the event, severity of the event, etc), the agent may need to experience the event multiple times before learning it well. One learning strategy that the agent could employ is Case-Based Reasoning (CBR), where the case contains the attributes of the current conditions, any known information about the event, and the resulting postconditions that the agent notices. After gathering several cases representing the occurrence of unknown events and the postconditions that resulted from the event, the agent may be able to learn and predict postconditions based off of the cases in the case library. Other approaches may yield more advanced algorithms that more accurate or rapidly learn from what is observed in the environment.

7.3 CONCLUSIONS

From the results observed during the testing of ADCUE, it can be concluded that ADCUE performs accurately for simple scenarios and satisfies the proposed hypothesis. Several different types of unexpected events were tested and ADCUE handled them with ease, as was expected. It is interesting to note that the one unexpected result did not arise from ADCUE's inability to handle an unexpected event, but human error.

Currently, several limitations exist in ADCUE. First, time is not well represented in the simulation. Taking time into account would integrate more realism into ADCUE by allowing agents and plans to have constraints not only on resources and environmental conditions, but also temporally dynamic components, such as events. Another limitation of ADCUE is the inability to fully operate dynamically within a simulation. While the framework is laid down, further work is required to integrate the pieces and allow agents to dynamically interact with the environment in temporal manner. One of the most problematic disadvantages to the approach presented in ADCUE is the data intensive nature of describing the components. Because each component must be accurately described in detail for ADCUE to correctly analyze the plan and all facets of the plan, the process of designing and implementing a scenario can be labor intensive. However, with the introduction of inheritance, component libraries, and automation techniques, this disadvantage can be drastically reduced. Furthermore, many future areas of research may be followed to expand the abilities of ADCUE and turn it into an honestly useful paradigm.

This type of research is envisioned to eventually be useful in a diverse range of domains. The ability to reason about unexpected events would be particularly beneficial in the obvious application of planning and scheduling in dynamic environments. For example, highly detailed and dependent projects, such as construction could one day use such a system to determine the consequences of late shipments. Other areas of application include military simulators or training systems that allow events such as unexpected military movements or ambushes to be injected. Modeling “what-if” scenarios of static plans is another application of this research. The successful first tests of the ADCUE model indicate that the model is promising enough to continue the research to improve and expand the approach.

LIST OF REFERENCES

[1] W. V. Wezel, R. J. Jorna, and A. M. Meystel, *Planning in Intelligent Systems: Aspects, Motivations, and Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 2006.

[2] S. C. Perry, "The Relationship between Written Business Plans and the Failure of Small Businesses in the U. S," *Journal of Small Business Management*, vol. 39, pp. 201-208, 2001.

[3] L. D. Interrante and D. M. Rochowiak, "Active Rescheduling for Automated Guided Vehicle Systems," *Intelligent Systems Engineering*, vol. 3, pp. 87-100, 1994.

[4] M. Campbell, A. J. Hoane Jr, and F. H. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, pp. 57-83, 2002.

[5] R. Volpe, T. Estlin, S. Laubach, C. Olson, and J. Balaram, "Enhanced Mars Rover Navigation Techniques," presented at Robotics and Automation, 2000.

[6] M. Baldoni, L. Giordano, A. Martelli, and V. Patti, "Reasoning about Complex Actions with Incomplete Knowledge: A Modal Approach," presented at ICTCS, Torino, Italy, 2001.

[7] A. J. Gonzalez and R. Ahlers, "Context-Based Representation of Intelligent Behavior in Training Simulations," *Transactions of the Society for Computer Simulation International*, vol. 15, pp. 153-166, 1998.

[8] P. Brézillon, "Context Dynamic and Explanation in Contextual Graphs," *Modeling and Using Context (CONTEXT-03)*, pp. 94-106, 2003.

[9] A. Blum and M. L. Furst, "Fast Planning Through Planning Graph Analysis," *Artificial Intelligence*, vol. 90, pp. 281-300, 1997.

[10] D. N. Davis, "Reactive and Motivational Agents: Towards a Collective Minder," presented at Intelligent Agents III—Third International Workshop on Agent Theories, Architectures, and Languages, London, UK, 1996.

[11] P. J. Gmytrasiewicz and C. L. Lisetti, "Emotions and Personality in Agent Design," presented at First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1, Bologna, Italy, 2002.

[12] F. Andriamasinoro and R. Courdier, "The Basic Instinct of Autonomous Cognitive Agents," presented at International Conference on Autonomous

Intelligent System (ICAIS'2002), February 12th-15th, Geelong, Australia, 2002.

[13] D. E. Wilkins, *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann Publishers, 1988.

[14] A. Nareyek, *Constraint Based Agents: An Architecture for Constraint Based Modeling and Local Search Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*. Berlin, Germany: Springer, 2001.

[15] M. Wooldridge and N. R. Jennings, "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, vol. 10, pp. 115-152, 1995.

[16] J. Blythe, "An overview of planning under uncertainty," *AI Magazine*, vol. 20, pp. 37-54, 1999.

[17] A. M. A. Salva, "Situational Awareness Through Context Based Situational Interpretation Metrics," Master's Thesis, University of Central Florida, Fall 2003.

[18] J. McCarthy and P. Hayes, *Some Philosophical Problems from the Standpoint of Artificial Intelligence*: Stanford University, 1968.

[19] J. F. Allen, H. A. Kautz, R. N. Pelavin, and J. D. Tenenberg, *Reasoning about Plans*. San Mateo CA: Morgan Kaufmann Publishers Inc., 1991.

[20] R. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, pp. 189-208, 1971.

[21] R. van der Krogt, A. Bos, and C. Witteveen, "Replanning in a Resource-Based Framework," *Multi-Agent Systems and Applications II*, vol. 2322, pp. 148–158, 2002.

[22] T. B. Dinh and B. Smith, "CPPlanner: Extending Graphplan Framework for Optimal Temporal Planning," presented at European Conference on Artificial Intelligence, Valencia, Spain, 2004.

[23] A. Blum and J. Langford, "Probabilistic Planning in the Graphplan Framework," *Proceedings of the Fifth European Conference on Planning*, pp. 319–332, 1999.

[24] L. D. Interrante, "A Model For Selective Attention In Monitoring And Control Reasoning Tasks," presented at Systems, Man, and Cybernetics Conference, Charlottesville, VA, 1991.

[25] J. Blythe, "Planning with External Events," presented at Conference on Uncertainty in Artificial Intelligence, Seattle, WA, 1994.

[26] C. Grama, E. Pollak, R. Brasch, J. Wartski, and A. J. Gonzalez, "Automated Generation of Plans through the Use of Context-Based Reasoning," presented at Eleventh International Florida Artificial Intelligence Research Society Conference, 1998.

[27] A. J. Gonzalez and S. Saeki, "Using Contexts Competition to Model Tactical Human Behavior in a Simulation," presented at Context 01 Conference, 2001.

[28] C. Micacchi, "An Architecture For Multi-Agent Systems Operating In Soft Real-Time Environments With Unexpected Events," University of Waterloo, 2004.

[29] A. Nareyek and T. Sandholm, "Planning in Dynamic Worlds: More than External Events," *IJCAI-03 Workshop on Agents and Automated Reasoning*, pp. 30–35, 2003.

[30] E. Onaindia, O. Sapena, L. Sebastia, and E. Marzal, "Simplanner: An Execution Monitoring System for Replanning in Dynamic Worlds," in *Progress in Artificial Intelligence Knowledge Extraction, Multi-agent Systems, Logic Programming, and Constraint Solving*: Springer Berlin, 2001, pp. 393–400.

[31] J. Anderson and M. Evans, "A Generic Simulation System for Intelligent Agent Designs," *Applied Artificial Intelligence*, vol. 9, pp. 525-560, 1995.

[32] M. E. Pollack and J. F. Horty, "There's More to Life Than Making Plans," *AAAI Magazine*, vol. 20, pp. 71, 1999.

[33] I. Merriam-Webster, "Merriam-Webster Online Search," 2007.

[34] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, pp. 121-189, 1995.

[35] J. Garamone, " Predator Demonstrates Worth Over Kosovo," in *American Forces Press Service*, 1999.

[36] D. Martin, "The Predator," in *CBS News*, 2003.

[37] "Pilotless Warriors Soar To Success," in *CBS News*, 2003.

[38] C. Wade, "The Warfighter's Encyclopedia: RQ-1 Predator," N. A. C.-W. Division, Ed., 2001.

[39] E. J. Labs, "Options for Enhancing the Department of Defense's Unmanned Aerial Vehicle Programs," C. B. Office, Ed., 1998.

[40] A. C. Command, "MQ-1 Predator Unmanned Aerial Vehicle," U. S. A. Force, Ed., 2007.

[41] G. S. Lamb and T. G. Stone, "Concept of Operations for Endurance Unmanned Aerial Vehicles", 1996, http://www.fas.org/irp/doddir/usaf/conops_uav/index.html

[42] R. McVicker, "Predator System Familiarization Guide", 1996, http://www.fas.org/irp/agency/daro/predator/toc.html

[43] J. Pike, "RQ-1 Predator MAE UAV", 2002, http://www.fas.org/irp/program/collect/predator.htm

[44] K. T. Rhem, "Iraqi Plane Shoots Down American Predator Unmanned Aircraft," in *American Forces Press Service*, 2002.

[45] J. s. I. Group, "AGM-114 Hellfire", 2000, http://www.janes.com/defence/air_forces/news/jalw/jalw001013_1_n.shtml

[46] "Patriot TMD", 2000, Federation of American Scientists, http://www.fas.org/spp/starwars/program/patriot.htm

[47] J. s. I. Group, "Raytheon Electronic Systems FIM-92 Stinger ", 2000, http://www.janes.com/defence/air_forces/news/jlad/jlad001013_2_n.shtml

[48] A. Robinson, "FAA Authorizes Predators to Seek Survivors", 2006, http://www.af.mil/news/story.asp?storyID=123024467

[49] "US Asked to Leave Uzbek Air Base", 2005, BBC News, http://news.bbc.co.uk/2/hi/asia-pacific/4731411.stm