Realtime Execution of Automated Plans using Evolutionary Robotics

Tommy Thompson & John Levine

Abstract— **Applying neural networks to generate robust agent controllers is now a seasoned practice, with time needed only to isolate particulars of domain and execution. However we are often constrained to local problems due to an agents inability to reason in an abstract manner. While there are suitable approaches for abstract reasoning and search, there is often the issues that arise in using offline processes in real-time situations. In this paper we explore the feasibility of creating a decentralised architecture that combines these approaches. The approach in this paper explores utilising a classical automated planner that interfaces with a library of neural network actuators through the use of a Prolog rule base. We explore the validity of solving a variety of goals with and without additional hostile entities as well as added uncertainty in the the world. The end results providing a goal-driven agent that adapts to situations and reacts accordingly.**

I. Introduction

Machine learning applications for intelligent agent formulation is a broad area of research. We have learned how to generalise vast quantities of raw data and process them in an intelligent and effective manner. Resulting in software and hardware that can adapt to and rationally act against sensor data provided appropriate training mechanisms were applied. One technique used frequently for this generalisation process is neuro-evolution; the application of evolutionary computation to train artificial neural networks (ANNs). ANNs are ideal for processing low-level real-time data and can generate effective, robust and computationally cheap solutions to a myriad of problems. However given the nature of the data presented and reactive nature of ANNs to stimuli, this often constrains these agents to dealing with tasks of limited scope.

This is seldom an issue with many applications, however it does effectively limit the potential capability of these agents. Research in non-player characters (NPCs) for video games provide a prime example of this. Neural network control has became a prominent development tool in a variety of game-related research ranging from combat NPCs [1], [2] to car racing and navigation [3], [4]. While we do not wish to detract from these contributions we seldom see controllers designed to solve larger problems that require sequences of individual and varied actions to solve. What becomes clear is that there is no means of abstract reasoning to occur. The lack of internal state prevents reasoning and deliberation regards the agents current state. Thus preventing the chaining of actions in sequence unless control is exerted outwith the controller itself.

Tommy Thompson and John Levine are with the Strathclyde Planning Group, University of Strathclyde, Glasgow, G1 1XH, UK, email: [tommy, john.levine]@cis.strath.ac.uk).

Perhaps one of the most renowned methods of abstract reasoning is the field of automated planning. Using a restricted model of the world and an explicit description of the state-action space, search methods are applied to produce a chain of actions that will traverse from the initial state to a state representing the desired end-goal conditions. While this is a highly effective tool, traditional planning methodologies (typically referred to as classical planning) must search offline utilising abstract, fully deterministic and observable models of the world without interaction. Many find that this feature detracts from the potential of this application should it be used in real-time environments.

ANNs and planning are perhaps prime candidates to compare and contrast the drawbacks and benefits of reactive to state-based behaviour, however there is potential in utilising these together. Planning can assist in overcoming the lack of internal state within an ANN-driven agent, while said perceptron can help a plan-driven agent interact with a world it can only see through abstract models. However this is not a simple intuition-based decomposition; there are significant issues that must be addressed ranging from the use of an offline search process in a real-time environment to gaps in knowledge that can not be expressed in neither the neural network nor the planners domain model.

In this paper we explore the feasibility of combining classical planning, modelling and deliberation with the reactive control of ANNs in a decentralised agent architecture for game environments. In order to do so we create a new testbed environment called *BruceWorld* that is sufficient for our needs. What follows is an account of our intent to create an agent capable of reasoning over long-term goals, devising plans to solve these goals and executing them through the use of reactive controllers. Furthermore, we explore means of coping with holes in the knowledge of both systems and how this leads to a goal-driven agent that reacts to local stimuli effectively.

This paper is laid out as follows; in Section II we provide detail of our testbed simulation *BruceWorld* and how it provides sufficient challenge for our agent, followed by an offering of related research in Section III. We then explore how we create our decentralised architecture in Section IV. A series of test-problems and the results of our agent against them are given in Section V and a discussion of the overall effectiveness of our controller as well as suggestions for further research in Section VI. We then provide our concluding remarks in Section VII.

978-1-4244-4815-9/09/$25.00 ©2009 IEEE

II. *BruceWorld*

BruceWorld is a testbed environment designed specifically to assess the validity of our research in this paper. The original concept being to present a domain suitable for neural network controllers but can also be reasoned about in an abstract manner. BruceWorld is a java developed game that operates in discrete time-steps occurring every 15 milliseconds. Confined to the interior of an office building and inspired in part by the movie *Die Hard*, each BruceWorld instance (such as the example shown in Figure 1) consists of a series of rooms, all connected either by corridors, doorways or by ventilation ducts. If a door in a doorway is closed it will typically require a switch nearby to be hit in order for it to be opened.

In a given problem instance there are two other types of agent that may interact with our Bruce agent, known as terrorist and hostage. Terrorist agents are hostile to Bruce and will seek to eliminate him once he enters their immediate vicinity. Meanwhile hostages may be somewhere within the building and need to be rescued. Each problem instance is potentially filled with enemy agents whose goal is to eliminate any moving targets in the vicinity. Hostages will be found in varying areas of the world and will often need to be rescued by Bruce. Hostages are typically very cooperative when in Bruce's vicinity, however should the agent be injured or be in a state of panic their range of capabilities is drastically reduced. Unless these issues are addressed they are often unable to act, forcing Bruce to interface with them in different ways, ranging from patching up wounds with aidkits found throughout the world to carrying them throughout the level.

To further impede Bruce and his attempts to achieve his goals, small explosives will occassionally be littered around the world that may require defusing if they pose a significant threat.

Bruce can be charged with goals such as navigation through a series of corridors, air vents and doorways, retrieval of resources such as aidkits, rescuing of hostages or defusing of bombs. Note that no terrorist goals are ever imposed upon Bruce, this is a deliberate design decision as unlike hostages, terrorists cannot be seen in the plan model at initialisation (since they cannot be effectively modelled in a planning domain). Bruce can only 'see' hostile agents once he had entered the same location as them.

Each agent is capable of basic movement throughout the environment; forwards, backwards, turn left or right and is capable of activating one or more of these actions at each discrete time-step of the simulation. Movement is fixed to a predefined distance (2 pixels) or angle (4 $\deg$) per update. Once any navigation action is committed, there is a minimum delay of 1 cycle before that same action can be committed again. Each agent will start with a maximum health of 100 points, and will suffer a loss of 25 points for each collision with the environment or if hit by incoming fire. Terrorists and Bruce can potentially wield knives or guns and use them against opponents. While knives only operate close range, guns may be fired at a distance in the direction they are facing. Should any rounds be fired there is a 50 cycle delay before another round can be fired

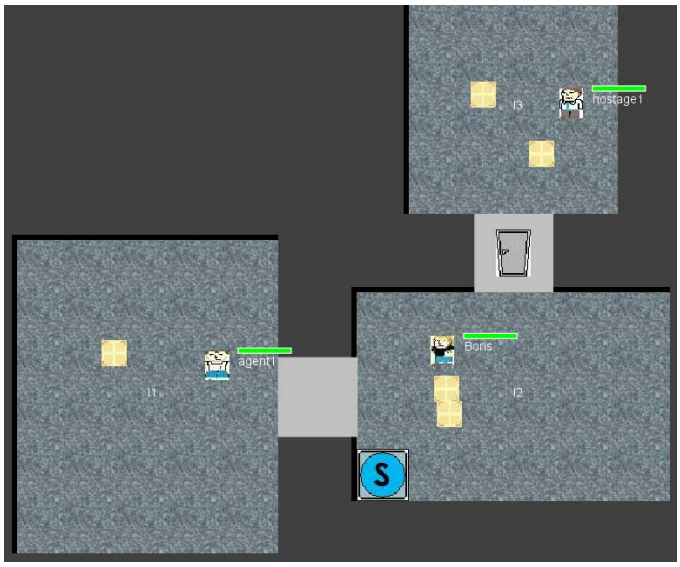

Fig. 1. A screenshot from the BruceWorld game. The game in progress challenges Bruce to rescue the hostage from behind the locked door. However there is an enemy terrorist in the next room, and clutter in the rooms that the agent must navigate around.

III. Research Background & Related Work

The concept of a layered architecture that incorporates deliberation with reactive control is far from a novel concept. With a range of systems that have emerged typically within engineering disciplines. The approach we report on in Section IV, specifically the implementation of classical planning with reactive control is relatively unique in the field of computational intelligence and games. CI in games is a useful domain for experimentation as we bypass many low-level issues of execution through the abstractions and assumptions created by game engines. Thus opening up oppurtunities for automated planning systems that are often avoided in engineering problems. A disappointing fact given that many of these problems result in restricted, problem specific implementations. So far as related research is concerned, we find our interests coincide with work conducted in execution monitoring of robot missions. Rover agents utilised for planet navigation and observation typically utilise a neo-classical or HTN plan-based control system. Given the large investments in creating and launching rover agents, a substantial amount of research is focussed on compensating for issues that occur in realtime execution. Research ranges from stronger understanding of the plan actuators to expressing greater control over the process from the plan level.

Understanding of actuators is sought to provide tighter and more cohesive performance. A fine example of this is found in [5], where the authors seek to learn performance models of plan actuators for a football playing agent. The goal being to improve the efficiency of the agents navigation as it moves towards the ball, turns to face the goal and dribble the ball

into the net. While it is possible to achieve this, often the behaviour looks disjoined since it is an *n*-step plan performed in sequence. Utilising performance models and subgoal refinement results in improved performance, with the execution appearing more cohesive. A second example is found in [6], where the authors attempt to learn behaviours as structured stochastic processes through the use of dynamic bayesian networks. Knowledge of behaviour capabilities allows for better understanding of the agents functionality.

Moving into different territory, research found in [7] and [8] seek improved monitoring and control from a planning perspective. In [7] we are introduced to *lxTeT*; a partial-order-causal-link (POCL) temporal planner that is based on CSPs for inclusion within a distributed robot control architecture. When executed the lxTeT operates as a temporal executive, maintaining control over continued operation of actions and as a procedural executive expands and refines the actions into commands at a functional level. While in [8] introduces a planning system dubbed *Kirk*, decomposing task-level commands expressed in Reactive Model-Based Programming Language (RMPL) into Qualitative-State plans (QSPs) through the use of Temporal Planning Networks.

Finally, research found in [9] and [10] reports on their advances in rover test bed systems. The authors comment that in complex, critical systems almost every component provides a potential point of failure; a by-product of the size and complexity of the systems required in order to approach this task. Providing error proof code that compensates for flaws is difficult due to the sheer size of the systems and the number of known (and unknown) cases that can arise. The importance of this research is three key observations that tie strongly to our interests in this paper:

- Often assumptions made by control software prove to be false during execution.
- Software may be attacked by a hostile agent, seeking to interrupt its execution or may simply be reacting to its presence.
- Continued changes to software often introduces compatibility issues between components.

In order to compensate for these failures, software must be able to recognise and diagnose failures, isolating the component where failure has occurred and find an alternative means of execution. However to achieve this we require numerous models to successfully monitor the execution. Ranging from models of component relationships, to models of intended behaviour and known errors. Ultimately, an agent or system must be able to sense it's own state and reason over that information. We strongly agree with these concepts and as shown in Section IV attempt to adhere to these values when designing our agent architecture.

IV. Agent Design

In this section we provide a description of the agent architecture utilise for the *BruceWorld* domain. We also provide an insight into the execution behaviour of our agent. Our architecture is programmed in Java and built from three specific components; the *Plan Manager* (PM), *Rule Controller* (RC) and the *Controller Library* (CL). The system navigates through these components in the order which we have provided them above and is shown in Figure 2. Throughout the following subsections we will explain each individual components design and form of execution.

A. Plan Manager

The PM is responsible for all interactions with the planning model as well as dictating the overall flow of execution for the agent. At the beginning of any instance, given a specified goal, the system must generate a plan of action for the agent to execute. This is carried out utilising the JavaFF planner; a object-oriented implementation of Fast-Forward (FF), a forward chaining heuristic state space planner [11].

Using the provided domain file representing the BruceWorld game, the plan manager requires a representation of the specific problem instance. To achieve this, each object within the game is queried to generate PDDL (Planning Domain Definition Language) predicates that will provide information as to the current state of that object. It is important to note at this juncture that it is not required for *every* object in the environment to provide information, on the condition it is not goal critical. For example, enemy agents (such as the one in location 2 of Figure 1) cannot be expressed in the plan-model, hence we do not query them for information. This in fact results in a real problem for execution which is resolved via the Rule Controller, described in the following subsection.

Once planning is completed we take two pieces of information; the grounded initial state of the problem and the sequence of actions that compose the plan. Given that these are stored using the JavaFF classes, this allows for quick and clean querying and manipulation when necessary.

When execution begins, the PM is then responsible for ensuring consistency between the plan model and the BruceWorld instance. We take the first action in the plan queue for execution and quickly assess whether the current state of the plan model accurately represents the BruceWorld instance. By querying the BruceWorld objects in the instance, we formulate a similar grounded state and compare them. If the plan-model is accurate then we commit the current action in the plan for execution and move control to the Rule Controller. Should there be inaccuracies between the plan-model and what exists within the instance (bearing in mind we only consider objects that can be observed by the planner), then the plan-model is re-generated and a re-plan is committed. Resulting in a new current state and queue of actions to execute, forcing the deliberation process to start again from the beginning.

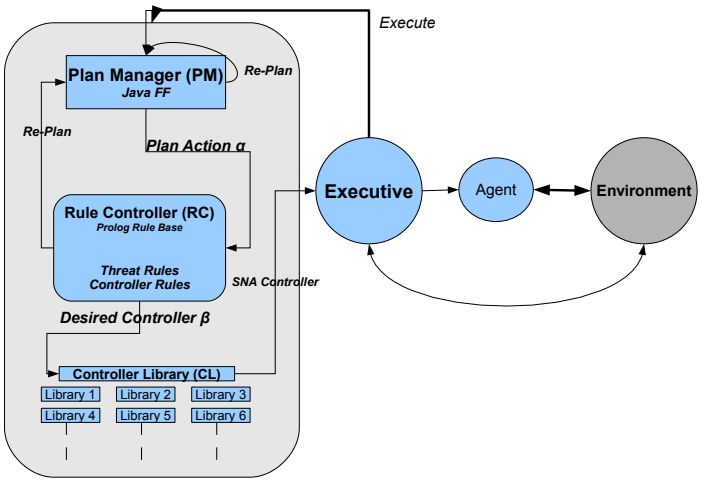

Fig. 2. This diagram shows the components of our architecture as well as the flow of execution. The executive will query the Plan Manager for the next action provided there is no need to force a re-plan. Once an action is found, the Rule Controller ascertains whether there is any external threat to the agent. Once the Rule Controller has made a decision on what action to take, a request is sent to the Controller Library. Resulting in a Subsumption Neural Architecture (Figure 3) or hardcoded script being sent to the agent for execution.

B. Rule Controller

The RC is responsible for reactive deliberation that the PM is incapable of. The RC incorporates two rule bases; a threat rule base and a controller rule base. Threat rules highlight whether particular features of the environment or other agents represent a potential threat to our executing agent. While the controller rules provide an association with particular action commands (either dictated by the PM or the RC) to desired behaviours that can be retrieved from the Controller Library.

To allow for ease of access and fast query times with the rule base, the rules are encoded in Prolog and interfaced to our Java source code using the JPL interface provided in releases of SWI-Prolog.

This component is activated when we have received a executable action from the PM. At this point classification takes place across all relevant entities. The agent queries the environment for information about hostile or potentially dangerous objects within a limited range of the agent. In the case of *BruceWorld*, threats such as bombs or terrorists are only visible if they are in the same room as the agent. Once information about a specific object is gathered, it is put through a phase of classification by fuzzifying the gathered data. Once we have classified items of interest, we formulate a Prolog query and generate the appropriate threat-level for that entity courtesy of the threat rule knowledge base.

Once all potentially hostile entities (if any) are classified, we then run our controller rules to devise what specific behaviour we wish for our agent to execute. The first priority is assessing whether any of the threat-classified entities pose a sufficient threat for our agent to deal with it. If this be the case, our rule base will provide the desired behaviour that will deal with this threat. The next phase is to assess whether any supplementary actions are required; this is an important component of execution that helps resolve issues that arise from using the state-based representation of the plan-model against a two-dimensional game world. A good example of this is a 'defuse-bomb' action in our planning model. Given that the plan-model has no understanding of the physical dimensions of a room, we simply require the agent be in the same room as the bomb to defuse. However in the real world this is not the case; the agent must be physically next to a bomb in order to be close enough to defuse it. Hence there is a series of rules that dictate supplementary or *bridge* actions that are required before the desired action is executed. If any bridge actions are required, then this request is sent to the CL for retrieval. However if neither threat actions nor bridge actions be requested, the controller rules will provide the desired behaviour based on the current plan-action that we wish to execute and will be sent to the CL for retrieval.

C. Controller Library

The CL provides an interface to a variety of robust controllers, providing behaviours that will achieve tasks decided by the RC. There are two forms of actuator that can be retrieved from the CL; hardcoded control and pre-trained behaviors.

Control scripts are provided for instances where the desired action typically results in simple modification of an entity's state and does not require any intelligent behaviour. For example, agent interactions with hostages (such as healing and slapping) are hardcoded given that these actions only result in changes to the state of the hostage and have neither extra effects nor interactions with the physical environment.

Meanwhile the pre-trained behaviours provide a range of control to complete desired actions of the executive. Stored on disk is a collection of chromosome libraries, where each chromosome encodes weights of an Artificial Neural Network (ANN) phenotype. Each library provides a different form of control and is explained below.

- *Visit Waypoint*: Pre-trained to permit an agent to move across an environment to a specified (x,y) coordinate.
- *Destroy Target*: Given an assigned target, the controller will maneuver and fire a loaded gun to eliminate the target as efficiently as possible.
- *Grab Item*: Pre-trained to navigate an environment to pick-up a specified item. This controller operates similar to the Visit Waypoint control, however it also has a separate 'grab' command to retrieve items.
- *Detect Obstacles*: A supplementary controller that permits an agent to react to nearby obstacles and maneuver around them.
- *Dodge Shells*: A supplementary controller that permits an agent to react to oncoming enemy fire.

When a controller request is sent to the library interface, the resulting controller is a Subsumption Neural Architecture

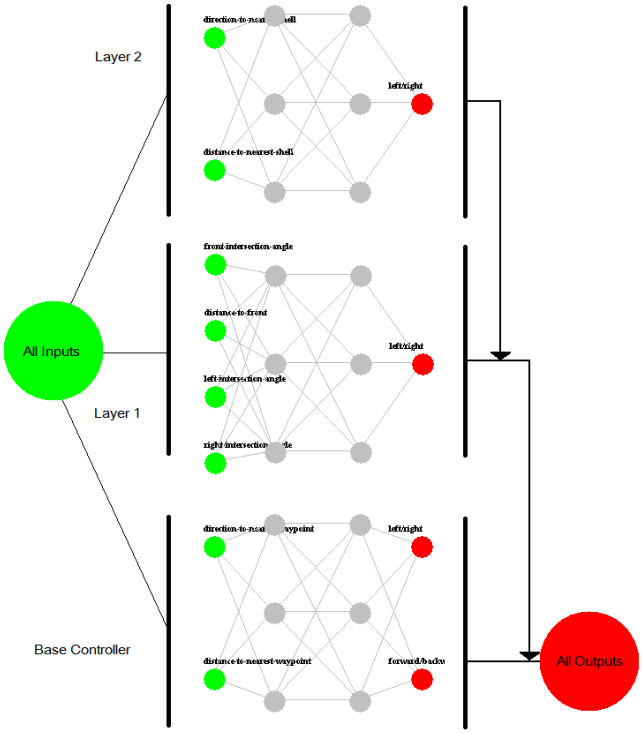

Fig. 3. An example Subsumption Neural Architecture (SNA), where three individual controllers are placed atop one another to dictate a hierarchy of execution. Note that agents utilise a subset of all available inputs and activate a subset of the outputs.

(SNA) that will feature one or more of the controllers stated above. The SNA controllers (shown in Figure 3) were devised in our previous research found in [12] and are inspired by the seminal works by Brooks in [13], [14]. Each controller is composed of a series of ANNs existing in a hierarchy of precedence. Should any layer of the architecture express desire to control a particular output, it will block or 'subsume' all layers beneath it from committing output. This allows for more intelligent, reactive control and provides a 'plug 'n' play' approach to behaviour formulation. Allowing us to create unique behaviours through simple combinations. Please see [12] for further information on the controller construction and training methodology.

We provide multiple instances of each controller, hence when a request is made to the CL the desired controllers will be selected at random to be placed in the SNA. This provides variety in execution since the behaviour will not necessarily be the same in each instance. Each controller is pre-trained and ready to use straight away, given the time available for development and testing we borrowed resulting chromosomes from our experiments in the *EvoTanks* domain [1], [12] to test our architecture. While there are differences in the environments, our results in Section V show that these were still sufficient for our testing purposes.

When a controller is requested for action, the interface also requires a focus entity that is the reason for the desired control. For example, for navigation we require the destination entity, or the target entity for a destroy target control. The interface pre-loads our SNA with all required information to allow for immediate execution. Hence in the instance of navigation, a pre-computed series of navigation points are provided for the agent to move towards.

V. Testing & Results

In this section, we provide an account of running our architecture against a variety of problems. We provide a brief introduction to the test problems and the recorded data from each run. The resulting data is then discussed at length in Section VI.

For this paper we provide problems that cover a range of agent functionality. In terms of challenge, all are solvable when no extra information is added to the problem. When enemy agents are added then there is a possibility that the agent can fail to solve a problem. We provide a brief description of each problem instance below:

- *Problem 1*: Navigate through a sequence of locations and corridors to reach a destination location.
- *Problem 2*: A hostage is trapped in a nearby location with an armed bomb. Agent must defuse the bomb and escort the hostage back to the starting position.
- *Problem 3*: A hostage is trapped inside a nearby room blocked off by a locked door. The agent must unlock the door and allow the hostage to escape.
- *Problem 4*: This is similar to problem 3 except there are two hostages trapped in separate rooms. Agent must rescue and escort both hostages to the goal location.
- *Problem 5*: Our agent must escort two hostages to a nearby location. However one hostage is unconscious and the other is an uneasy mental state.

We focus many of our problem instances on hostage retrieval, since potential uncertainty will have an impact on overall performance. We assess each problem instance 10 times in the standard mode (i.e. no modifications to the problem) and a further 10 times with the hostile entities and uncertainty added to the model. The purpose of this is to assess how well our agent performs when faced with elements that may impede progress, or contradict the plan-model and force the agent to rethink the plan of action.

Results of running our problem instances standalone and with threat and uncertainty can be found in Table I and II respectively. The data provided shows that an agent will often solve a problem instance and will continue to complete most instances when faced with added adversity and uncertainty. Note that there is a significantly larger number of actions committed on average in each instance when we have threat and uncertainty options active. This in turn results in the

TABLE I

STATISTICS FROM 10 RUNS OF EACH OF OUR PROBLEM INSTANCES WITH NO MODIFICATIONS. WE PROVIDE RELEVANT STATISTICS REGARDS CONTROLLER PERFORMANCE, AS WELL AS THE NUMBER OF TIMES THE SYSTEM INTERACTS WITH JAVAFF AND THE PROLOG RULE BASE. THE AVERAGE TIME TAKEN TO INTERFACE WITH THESE COMPONENTS BEING AN IMPORTANT POINT TO CONSIDER.

Instance	Completed	Initial Plan Length	Avg. No. Actions	Avg. Plan Runs (Avg. Time)	Avg. Prolog Runs (Avg. Time)
Problem 1	10	5	5	1 (85.7ms)	5 (<1ms)
Problem 2	10	3	3.78	1 (182.6ms)	3.78 (<1ms)
Problem 3	9	6	6.78	1 (212.78ms)	6.78 (1.44ms)
Problem 4	10	8	8	1 (209.3ms)	8 (<1ms)
Problem 5	10	5	6	1 (177ms)	7.11 (<1ms)

TABLE II

STATISTICS FROM 10 RUNS OF EACH OF OUR PROBLEM INSTANCES WHEN RUNNING ON WITH THREATS AND UNCERTAINTY. THIS HAS A CONSIDERABLE IMPACT ON THE NUMBER ACTIONS THAT ARE COMMITTED, AS WELL AS THE NUMBER OF INTERACTIONS (AND THE AVERAGE TIME TAKEN) WITH JAVAFF AND THE PROLOG RULE BASE.

Instance	Completed	Initial Plan Length	Avg. No. Actions	Avg. Plan Runs (Avg. Time)	Avg. Prolog Runs (Avg. Time)
Problem 1	8	5	9.86	3.29 (100ms)	21 (1.14ms)
Problem 2	10	3	10.89	3.11 (30ms)	24.9 (<1ms)
Problem 3	8	6	12	3.14 (83ms)	15.4(2ms)
Problem 4	8	8	13.8	3.75(62ms)	14.75(1.13ms)
Problem 5	8	5	14.86	3.86 (47.4ms)	18.71 (<1ms)

PM interacting with the JavaFF planner to force re-plans, as well as the RC utilising the Prolog knowledge base more regularly.

In Table II we see more instances of the agents failing to complete the designated task. In all cases observed, failure arose as a result of the evolved controller being unable to complete the desired task. The one instance in Table I arose due to the agent being trapped in an area surrounded by obstacles that were too close to it. Resulting in a cyclic attempt to reach the destination, this repeated until the permitted time to complete the action timed out. The other instances in Table II were the result of the agent fighting against an enemy agent. The terrorist agent simply overpowered our player.

Interesting observations can be found in the runtimes for the JavaFF planner and the JPL Prolog interface. When we only require the planner in the initialisation of the problem the overhead to achieve this is significant. While a couple of hundred milliseconds is irrelevant on a human level, this is a significant amount of processing time. Especially when we consider how few CPU cycles game developers wish to commit to AI processes per second. However we would hope that any developers who utilise this approach would make attempts to optimise the processing performance of the system, an endeavour we have not attempted. However if we observe the planning times in Table II, we note that the average time to plan is significantly reduced. We observed during testing that keeping the planner in memory drastically reduces the time taken to re-plan. Furthermore, if we observe the JPL interface times, regardless of the number of interactions with the Prolog rule base, the average time taken is incredibly fast. With best performance running at less than 1 millisecond per query on average.

Agent behaviours appear focussed and consistent between instances; with agents moving from one action to the next and maintaining an overall direction. Agents reacted well to nearby hostile elements, ranging from attacking enemy terrorists when in the same room as them to disarming nearby bombs once observed. We were also interested in observing instances where threats such as bombs were noted as non-threatening (a combination of the remaining fuse, blast yield and radius) and were ignored as the agent continued its path of action. Agents would quickly recognise an inaccurate game-state and force a re-plan when required. Once our agents generated a new plan, these would continue without complications and complete the task. As noted previously, all failures occurred from our evolved controllers being unable to complete their task. No failures ever occurred at the planning level.

An interesting observation made during the threat and uncertainty tests for problem-5, was that the planner on rare occassions gave inaccurate plans to the agent, where the resulting state does not reflect the goal state. The agent would execute the plan as described, however during execution, the system would reach a point where the game state did not reflect the state the planner believed would arise. The system then replans and generates a new plan that completes the task. While we are very pleased to see our agent compensate, we have still to devise why the planner generated an erroneous plan.

VI. DISCUSSION

Our analysis of our agents performance suggest that we can rely on our system to solve the problems we have expressed for it. Our architecture permits the creation of what we consider to be reactive yet goal driven agents. Our agent can commit state-based reasoning to devise plans of action to achieve goals and react to changes in the environment. These

changes can occur at a planning level, where the system can note issues with its understanding of the world at an abstract level and attempt to get back on track through re-planning. Changes can also occur beneath a planning level of observability, however we accomodate for this by utilising a series of rules to suggest a course of action should changes arise. In a sense it provides a reactive step that will interrupt the current plan and resolve any potential conflicts before continuing the course of action. Finally, in order to deal with grounded issues relating to the agents interaction with the environment, we provide a variety of ANN controllers that can be placed within a layered subsumption architecture that provide robust and competent controllers as actuators for the described plan actions.

We have taken onboard points highlighted in [9] and incorporated measures for these within our architecture. We briefly recap this below:

- *Often assumptions made by control software prove to be false during execution.* - Our Plan Manager retains copies of the current plan as well as the current state based on the progression of said plan. We enforce that the planning domain reflects the game world, since when the executive requires a new controller to execute, the system must retrieve a representation of the game state in PDDL to compare against. If any inaccuracies exist, then the a new problem instance is formulated based on the game state, and a plan is devised using the originally assigned goals.

- *Software may be attacked by a hostile agent, seeking to interrupt its execution or may simply be reacting to its presence.* - We have embraced this concept rather literally by having hostile entities exist within our sample domain. The Rule Controller allows for flexibility in dealing with any hostile agents that exist. Rules can be generated that allow for a range of different actions to occur in different situations. In terms of performance we can observe from Tables I and II that interfacing with the Prolog rule base is very fast. At best taking less than one millisecond to find the desired solution.

- *Continued changes to software often introduces compatibility issues between components.* - Each of our components exist isolated from one another. This ensures that any modifications that occur within the Plan Manager and Rule Controller do not have an impact on other parts of the system. Given that the interactions with the current plan and state are carried out via the Plan Manager. Then we could substitute different planners to assess performance. We can also make modifications to our prolog rule base with relative ease as it has no impact on the other components. Lastly the Controller Library provides an ideal means to manage a range of different controllers. The controller factory functionality allows us to store different types of controllers that may vary in topology and design, however is concealed by the factory method.

In summary, this architecture permits state-based reasoning to be merged with reactive control. Utilising individual, decoupled components that are easy to customise and tailor to suit the preference of the designer.

However this is not to say that our approach does not suffer from drawbacks, notably the heavy reliance on expert knowledge at both the planning level and the rule level. We require for the designer to have a strong understanding of all relationships between components within the environment. The designer must also make clear cut decisions as to the controller and threat rules, while these can be changed, in large problem instances it may take significant trial and error for a designer to tailor these to suit. The controllers utilised for the system must also be designed to suit the problem. We incorporated controllers from our EvoTanks domain, and while their performance was admirable in most instances it was clear that they are not trained to deal specifically with these problems. Hence time must be taken to create and train these agent controllers. The benefit being that once they are created they can be used at a later date provided no extra training or modification is required. However a significant amount of training data may need to be amassed prior to developement. There is also the added necessity to have the environment engine be able to represent in PDDL formulations, however we consider this a minor drawback as it is not a significant undertaking to incorporate this into a simulator or engine.

A. Future Work

We have presented our preliminary research in creating our goal-driven reactive agents. However there is a significant amount of further work that we wish to explore. We will now provide a brief insight into potential future work that we wish to carry out.

At present we assign goals to our agents, and while this is suitable for our purposes we would be interested in having these goals be assigned automatically. Research in real-time goal formulation courtesy of motivations instilled within an agent show promise as shown in [15] and [16]. This could help forge agents that can formulate their own goals such as acquiring resources, navigating through locations and solving nearby problems based on their needs and motivations.

Another idea is to incorporate a more intelligent means of planning into the agent architecture. While we have shown that using a relatively simple planner can be effective, it is not without flaws. At present JavaFF will struggle with large problem instances that may require mutliple agents to interact together using limited resources. This is more of a reflection of the current state of the automated planning community than an individual platform. Our immediate goals are to remedy the runtime of the planning module as well as allow for more expansive problems to be solved. An important issue we wish to address is the lack of knowledge of the hostile entities at the planning level, while we can

express bombs in the planning model we do not necessarily assign goals to disarm them, this is carried out at the rule level. While we can continue to rely on this, utilising the motivations concept suggested previously may assist in speeding up this process. Furthermore, we can not model enemy agents and their behaviours into a planning model. An interesting research area would be to explore the potential of creating a planning system that can incorporate enemy goals and their predicted actions based on the goals our agent wishes to achieve. If we have enemy agents who wish to interrupt our flow of execution to a desired goal, then we could possibly predict and counteract them.

At present our controller does not consider the possibility that one approach to solving a problem can fail. If this does occur then it simply declares this particular run of execution as a fail and terminates. We would be interested in having a variety of means to solving particular problems at all levels of the decision process. This could range from having different trained controllers to complete a task (albeit in different ways), to a collection of different actions in the planning domain model that can achieve the same goal condition (though the number of actions required to achieve this may differ).

The plans that our problem instances create are at present not large enough to cause immediate concern. However in future it may be desirable for us to create very large problem instances. If this be the case then we may only wish to plan so far ahead into the future given the uncertainty of a dynamic, multi-agent environment. This form of 'horizon' planning is an area we would wish to explore in the future.

Finally, rules incorporated into the Rule Controller are at present hand-coded and require expert knowledge of the domain. While this is perhaps preferable for some designers, there is the consideration that when dealing with larger problem instances we will be unable to provide rules that cover all possibilities. Further work in this area could explore attempting to learn rules that cover a variety of different scenarios. The use of random or evolved instance generation could help to create test cases that are beyond those that a human would initially conceive of. Thus possibly exploiting holes in the behaviour that the rule controller does not consider.

VII. Conclusion

In this paper we have presented our progress in creating a reactive agent that incorporates classical planning methods to state-based reasoning. Given assigned goals, we devise long-term plans to satisfy those goals through the JavaFF planner and re-plan accordingly if state models do not reflect the environment. Goals are then executed courtesy of pre-evolved neural network controllers combined within a subsumption framework, providing a robustness to their execution. To retrieve desired controllers, we provide a rule system using the JPL interface that provides quick reference to decipher whether current actions are achievable or they require additional actions. The rule system also provides a reactive form of control, allowing for our agent to react to nearby stimuli that warrant its attention and deal with them accordingly. Testing in the *BruceWorld* domain, we find that our system works well in dealing with hostile entities as well as uncertainty. This preliminary research leaves many directions open for future research possibilities.

Acknowledgments

JavaFF is developed by Andrew and Amanda Coles, Maria Fox and Derek Long. Sourcecode and executables are available freely through the GNU GPL via `http://personal.cis.strath.ac.uk/~ac/JavaFF/`.

The authors wish to thank Michelle Galea for her contributions and assistance that proved integral to the progress of this research.

References

[1] T. Thompson, J. Levine, and G. Hayes, "EvoTanks: Co-Evolutionary Development of Game-Playing Agents," *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pp. 328–333, 2007.

[2] G. Parker, M. Parker, and S. Johnson, "Evolving autonomous agent control in the Xpilot environment," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005)*, 2005.

[3] J. Togelius and S. Lucas, "Evolving controllers for simulated car racing," *Arxiv preprint cs.NE/0611006*, 2006.

[4] S. Lucas, "Evolving a neural network location evaluator to play ms. pac-man," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 203–210.

[5] F. Stulp and M. Beetz, "Optimized execution of action chains using learned performance models of abstract actions," in *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 19. LAWRENCE ERLBAUM ASSOCIATES LTD, 2005, p. 1272.

[6] G. Infantes, F. Ingrand, and M. Ghallab, "Learning behaviors models for robot execution control," in *ICAPS*, 2006, pp. 394–397.

[7] M. Gallien, F. Ingrand, and S. Lemai, "Robot actions planning and execution control for autonomous exploration rovers," *WS7*, p. 33, 2008.

[8] R. Effinger, A. Hofmann, and B. Williams, "Progress Towards Task-Level Collaboration between Astronauts and their Robotic Assistants," in *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, ser. ESA Special Publication, vol. 603, Aug. 2005.

[9] P. Robertson, R. T. Effinger, and B. C. Williams, "Autonomous robust execution of complex robotic missions," in *IAS*, 2006, pp. 595–604.

[10] P. Robertson and B. Williams, "Automatic recovery from software failure," *Communications of the ACM*, vol. 49, no. 3, 2006.

[11] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *"Journal of Artificial Intelligence Research"*, vol. 14, pp. 253–302, 2001.

[12] T. Thompson and J. Levine, "Scaling-up Behaviours in EvoTanks: Applying Subsumption Principles to Artificial Neural Networks," *Computational Intelligence and Games, 2008. CIG 2008. IEEE Symposium on*, 2008.

[13] R. Brooks, "A robust layered control system for a mobile robot." *IEEE J. ROBOTICS AUTOM.*, vol. 2, no. 1, pp. 14–23, 1986.

[14] ——, "Planning is Just a Way of Avoiding Figuring Out What To Do Next," *MIT Artificial Intelligence Laboratory Working Paper 303*, 1987.

[15] A. M. Coddington, "Integrating motivations with planning," *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS07)*, pp. 850–852, 2007. [Online]. Available: http://www.aamas2007.org/

[16] A. Coddington, "Motivations as a meta-level component for constraining goal generation," *Proceedings of the First International Workshop on Metareasoning in Agent-Based Systems*, pp. 16–30, 2007. [Online]. Available: http://www.aamas2007.org/workshops.html