GRUE: An Architecture for Agents in Games and Other Real-Time Environments

Elizabeth Gordon, BS, MSc.

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

March 2005

Abstract

This thesis presents an architecture, which we call GRUE, for intelligent agents in real-time dynamic worlds. Such environments require agents to be able to flexibly adjust their behaviour to take into account changes in the environment or other agents' actions. Our architecture is based on work done in robotics (Nilsson, 1994; Benson and Nilsson, 1995; Benson, 1996), which also deals with complex, dynamic environments. Our work focuses on goal arbitration, the method used by the agent to choose an appropriate goal for the current situation, and to re-evaluate when the situation changes. In the process, we have also developed a method for representing items in the environment, which we call resources, in terms of their properties. This allows the agent to specify a needed object in terms of required properties and use available objects with appropriate properties interchangeably. We show that the GRUE architecture can be used successfully in both a typical AI test bed and a commercial game environment. In addition, we have undertaken to experimentally test the effects of the features included in our architecture by comparing agents using the standard GRUE architecture to agents with one or more features removed and find that these features do improve the performance of the agent where expected.

Acknowledgments

First of all I would like to thank Sony Computer Entertainment Europe, who provided the majority of my funding, and express my gratitude to Ian Wright and Dave Ranyard who made it possible. I would also like to thank the School of Computer Science for providing the remainder of my funding.

Next I would like to express my gratitude to my supervisor, Brian Logan, for all his help in getting to this point.

I would also like to thank Nick Hawes, for allowing me to use his Sim_Agent code for connecting to Unreal Tournament, and Mike Lees, who allowed me to use his Tileworld implementation. Thanks also to Neil Madden and Mike Lees for help proofreading this document.

Finally, thanks to Jamie Strachan for proofreading above and beyond the call of duty, as well as help, support, advice, and generally putting up with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents a novel agent architecture for use in computer games and related environments. We begin this chapter by motivating our use of computer games for AI development before discussing our architecture specifically.

Computer graphics have become increasingly sophisticated, to the point where three dimensional simulated worlds are common on the average desktop computer. These types of worlds occur in the form of virtual reality entertainment environments or art works, training simulations, and computer games. Modern computer games provide dynamic, compelling simulated worlds. While many early computer games were text-based adventure games that give a description of a location and allow the user to type in commands e.g. *Zork II* (Infocom, PC 1981) [1], modern games have evolved into complex worlds rendered with high-quality three dimensional graphics.

Computer games are becoming an increasingly popular platform for artificial intelligence research. By their very nature, games are compelling enough to hold the interest

[1]Games given as examples will be followed by the name of the company that published the game, the year of publication, and if relevant, whether the game is for a standard PC or a games console. Games consoles that may be mentioned are the Playstation2®, the Gamecube™, and the Gameboy Advance™

of human beings. This is largely because they provide complex environments with challenging problems. These same properties make games good environments for evaluating AI techniques and cognitive models. Laird argues that games are “AI’s killer application” because in addition to having the right properties for testing AI systems, the computer games industry is a place where good AI can make a significant difference (Laird and van Lent, 2001). Games have been used in artificial intelligence research at least as far back as Pengi (Agre and Chapman, 1987), which was a reactive agent that played the game Pengo (Sega, 1982 arcade machine). Agre and Chapman explain that they chose to use Pengo as a domain because “events move so quickly that little or no planning is possible, and yet ... human experts can do very well” (Agre and Chapman, 1987). They go on to point out that this is also true of many everyday activities. This description, of course, applies to many games besides Pengo and emphasizes the relevance of games research to mainstream AI. The Soar games group has implemented computer game characters using the Soar cognitive model. They have created agents to play Quake II (Activision, PC 1996), Descent 3, and to operate a military flight simulator (van Lent et al., 1999; Laird and Jones, 1998). Soar will be discussed in more detail in Section 2.3. The EXCALIBUR project (Nareyek, 1998) is a planning system specifically designed for dynamic and/or multi-agent game environments. This system has the ability to reason about and even exploit events that happen independently of the agent. Hawes (Hawes, 2000; Hawes, 2003) proposes an anytime planning system to be used in computer game agents. An anytime planner constructs plans with the caveat that the planning process may be stopped at anytime. The planner is expected to return an executable plan no matter how little time there was to create the plan. If the planner is allowed to spend more time on the plan, the quality of the plan increases correspondingly.

Other work on computer game agents includes that by Khoo (Khoo et al., 2002), Zubek (DePristo and Zubek, 2001; Horswill and Zubek, 1999), etc.. Researchers at CMU have developed an interface to the popular game Unreal Tournament (Infogrames, PC 1999), which they are using for research on teamwork. See for example (Vu et al., 2003), which describes an architecture for multi-agent control. Interactive entertainment applications, while slightly different from games, require agents with similar capabilities to computer game characters. CMU's Oz project has focused on building agents for virtual worlds. Overviews are given in (Bates, 1992b; Bates, 1992a). Pattie Maes (Maes, 1994) describes several entertainment applications that include agents. MIT's Synthetic Characters group has done much work in the area of entertainment applications; some examples are (Blumberg et al., 1996; Blumberg et al., 1999; Kline and Blumberg, 1999; Yoon et al., 2000; Blumberg et al., 2001). Other work on agents for entertainment applications includes (Cavazza et al., 2002; Mateas and Stern, 2002; Logan et al., 2002).

Certain other types of environments are similar to games. There has been much work done on military training simulations. Many of these focus directly on social issues, such as directing troops effectively (van Lent et al., 2004) or dealing with distressed civilians (Traum et al., 2003). In fact, many games are based around military scenarios e.g. *Command and Conquer* (Electronic Arts, PC 1996) and the US army has gone so far as to release a game to attract potential recruits (America's Army, 2004). Training simulations for non-military applications have also been used for similar research, see (Rickel and

Johnson, 1998) for one example.

A game for our purposes is a virtual environment in which one or more human players interact with each other and with computer-controlled characters in order to achieve some number of objectives. While some computer games are turn-based, we are concerned in this thesis with those games that are real-time. A computer game (or any real-time simulated environment) contains a main loop which repeatedly displays graphics on the screen, checks for input from players, and makes any necessary changes in the game world, such as movement of characters and objects. A frame rate that is too slow will look jerky and feel unresponsive to the player. Typically most of the processing time is taken up by the graphics and physics computations, leaving about ten percent of the processing time for the artificial intelligence subsystem (Funge, 1999). On the bright side, characters in a computer game do not actually need to respond as fast as the graphics are updated. In fact, slower response times may make a character seem more believable (Freed et al., 2000).

Because games are real-time, agents within the environment must react on approximately the same time-scale as a human would. This is a separate requirement from the general display of graphics and processing of player input, which must be done fast enough to feel responsive and appear smoothly animated. Games are also dynamic, by which we mean that changes may occur within the world that do not depend on the actions of the agent. Games often include objects that appear or disappear at random, platforms that move back and forth, and special effects that are activated by agent actions but terminate after a specified length of time. Agents that operate in such environments must be able to adjust their behaviour according to changes that occur within the world.

Many games also have an important inventory management component. That is, players within the game can acquire various objects, but may only carry a certain number at any one time. Objects may be hidden in the game world, given to players by game characters or other players, or alternatively the player may purchase them in shops within the game world. Almost all games include some idea of health, which goes down when the player takes damage and can be restored using health items or magic spells among other methods. Games may also require players to eat, sleep, refuel vehicles and so forth.

For the purposes of this thesis, the terms world and environment are regarded as interchangeable. The term agent will be defined as an entity within the environment which obtains sensory information about the world and is capable of performing some set of actions. Crucially, agents (for our purposes) are autonomous, meaning they are not controlled by a human (although they may take suggestions from or react to human players). Most agents are present within a world; that is, they have location. Most agents will also store some kind of state (which may include their location, a representation of their health, etc.). All of our examples will assume that the agent is a character within the game or simulation; either a simulated human or some other creature with approximately human capabilities.

1.1 The Problem Addressed

Currently, game characters often exhibit formulaic, unrealistic behaviour. As game developers try to draw in new audiences and tell more sophisticated stories, they will require characters capable of handling complex situations in believable ways.

Factors influencing the believability of animated characters have been analyzed by a number of authors. Loyall takes inspiration from film and theatre to state that believability requires the ability to express a recognizable personality (Loyall, 1997). Reilly suggests that building believable characters is an artistic problem rather than an AI problem (Reilly, 1996). Sengers argues that agents must be placed within a sociocultural context, and that they must provide narrative cues in order to be understandable (Sengers, 1998). Blumberg suggests that animated characters must be capable of conveying their current motivations to human observers (Blumberg, 1997).

At a lower level, game characters need a minimum level of competence. A current problem for game developers is creating agents that do not exhibit obviously stupid behaviour. Some common AI problems in games are described in (Wetzel, 2004), and include attacking repeatedly with ineffective weapons, using a path that is known to be blocked by strong enemies, and not reacting until the player is actually visible (which allows the player to attack from a point out of sight). Our personal experience includes video game fights in which an enemy uses the same three weapons, in sequence, repeatedly. We have also seen games where magic using characters use inappropriate spells (i.e. casting a magical shield around a character who is too far away to be attacked). Another common complaint about the AI used in games is that the enemies “cheat” using information not available to the player. Bererton argues that the best way to give characters the appearance of intelligence is to actually make them more intelligent (Bererton, 2004).

One way common approach to making intelligent agents is to use a goal-based agent architecture; that is, an architecture that explicitly represents and reasons about its goals. Norman and Long argue that agents that interact with unpredictable environments need to be *goal autonomous* (Norman and Long, 1996) where goal autonomy is defined as the ability to generate new goals on the fly combined with the ability to adjust the focus of attention from one goal to another. Game worlds are good examples of unpredictable environments, although the necessity for goal autonomy reaches into other applications. In particular, robots operate in the real world which is inherently unpredictable. Multi-agent applications may also require goal autonomy if the actions of other agents are not predictable. Certainly any simulation which includes human participants will have the property of unpredictability. The use of a goal-based architecture for game agents has been advocated (O'Brien, 2002) for the reason that it makes agents that "feel more human". A more detailed description of a goal-based system is given by Orkin (Orkin, 2004). This system is being used to develop two commercial games, and has been successfully used to control a number of characters despite the computational limitations of commercial games. Orkin advocates the use of a goal-based system because goals can be sequenced to create complex behaviours and because a modular architecture facilitates sharing of behaviours between characters or projects.

In this thesis, we present a goal-based architecture for computer game agents. For our purposes, a goal is a data structure containing the following elements: a condition to be achieved or maintained in the world, a priority, and a type. The most fundamental property of the goal is the condition to be achieved. This is the definition of goal as used conversationally, and allows the agent to choose or construct a plan for achieving the goal. The priority is used to make decisions about which plans to execute, and the type specifies whether the goal achieves or maintains its associated condition. Goals are achieved by plans. We use the word plan to mean a set of actions that when performed by the agent achieve the condition of the associated goal. The priority allows the agent to decide how important the goal is compared to its other goals. We allow this priority to change in reaction to changing circumstances in the world. This allows the agent to adjust its behaviour appropriately without forcing the programmer to hard-code rules to handle all possible situations. These elements are described further in Chapter 3.

Our architecture is *goal autonomous* in the sense described by Norman and Long (Norman and Long, 1996) because it has the following properties: agents created using our architecture may generate new goals in appropriate circumstances, the priorities of these goals can change as circumstances in the world change, and goals stop being considered when they are no longer appropriate.

1.2 Summary of Work Done

The aim of this thesis is to develop mechanisms to support the creation of characters for games and similar applications. However, we have chosen to focus on underlying mechanisms that can underpin the development of many different kinds of agents. The central idea of this thesis is a a resource-based goal arbitration method. Resources have been used to represent real-world objects and quantities (such as money, health or fuel) in a number of planning systems e.g. (Drabble and Tate, 1994; Kvarnstrom et al., 2000; Brenner, 2001; Koehler, 1998). Such systems do not, however, consider the problem of goal arbitration. Our method uses the current state of the world, including presence or absence of particular resource types, to constrain the agent's choice of goals.

Specific abilities supported by our goal arbitration method are the ability to generate new goals with the requirement that goals are only generated when they are appropriate, the ability to assign priorities to new goals that vary according to the current situation, and the ability to adjust the relative priority of goals to which the agent has committed as that situation changes. Our priority values are similar to Norman and Long's motivations (Norman and Long, 1995; Norman and Long, 1996). We also allow the agent to work on multiple goals in parallel, subject to resource availability.

We adopted an empirical approach to the evaluation of this goal arbitration method. During the course of this project, we designed and implemented an agent architecture, which we call GRUE (Goal and Resource Using Entity), which embodies our resource-based approach to goal arbitration. We have based GRUE on previous work in robotics. Robots operate in the real world which shares the properties of complexity and unpredictability with game worlds. GRUE uses pre-written plans in the form of teleo-reactive programs (TRPs)(Nilsson, 1994) and is an adaptation of the architecture described in (Benson and Nilsson, 1995; Benson, 1996) which we discuss further later in Chapters 2 and 3. A secondary aim of this thesis is simply to show that teleo-reactive programs can be used in a commercial game environment. We also experimentally tested a range of features included in GRUE. Some features are intrinsic to the execution of the architecture; for these features we have included a mechanism for switching the feature off. Other features require the agent designer to take advantage of them, for example the architecture runs the goal generators but the exact functionality of the goal generator, including whether or not it updates the priority of existing goals, is left for the programmer to decide. By making use of the switches for intrinsic features and creating alternate versions of the environment-specific portions of the agent, we can test each feature of the architecture individually.

In addition to our resource-based arbitration method and the goal autonomy features supported by GRUE, we have developed a flexible representation of resources. Each resource is stored as a list of properties. Goal arbitration within GRUE is not dependent on any particular set of properties or property values. This means that resources can be represented in the most convenient manner for a particular game world or application. We also make a distinction between properties that a resource must have in order to achieve a goal, and properties which are preferred by the agent but not necessary. This allows an agent to request resources that will allow a goal to be achieved most efficiently or effectively without preventing it from using less preferred resources when necessary. Preferences could also be used to give agents the appearance of personality. Some representations of resources in the planning literature contain some idea of preferences (Kvarnstrom et al., 2000), but we do not know of any that are as expressive as the notation used in GRUE.

The full list of features supported by GRUE that we evaluate is as follows:

1. Goals are generated only in appropriate situations.

2. Goal priorities are set according to the current situation.

3. Goal priorities are updated continuously.

4. Plans request resources in terms of required and preferred properties.

5. A plan can specify that it needs exclusive access to a resource.

6. A plan that makes the same request during consecutive cycles will receive the same resource each time.

7. Numerical quantities can be divided and used for more than one goal.

8. Plans can use special notation to request numerical values within a range.

9. Multiple actions can be executed during each execution cycle.

Goal autonomy is provided by the first three of these features. The designer may use these features or not as appropriate for each goal. The use of resources is integral to our goal arbitration method, in that resource availability is used together with goal priority to determine which plans will execute during each cycle, and is necessary to support believable behaviour in game worlds. Allowing exclusive access to resources is necessary in situations where the agent is performing actions that may change a resource. The sixth feature, allowing plans to receive the same resource during consecutive cycles, is provided because an agent that constantly stops in the middle of plans and selects new resources may not be very believable or effective. Agents that use resources must also be able to intelligently use divisible quantities. In a game that includes the buying and selling of items, believable behaviour requires the ability to use money for the purchase of multiple items that are needed to satisfy different goals. We also provide new notation for representing ranges of numerical values which allows agents to encode sophisticated resource requirements such as “I want as much as possible, but not more than 10”, or “I need at least 5, but more would be better”. Finally, GRUE allows an agent to work on several plans simultaneously and (if possible) every plan may execute actions in the same execution cycle. This last feature allows an agent to recognize when two plans are compatible and execute them in parallel. Three of these features, parallel execution of actions, the use of the same resource during consecutive cycles, and exclusive access to resources, can be disabled for the entire architecture (which is useful for testing purposes).

We show that resource-based arbitration works in two different environments. The first of these environments, Tileworld (Pollack and Ringuette, 1990) is an AI test bed in which we test all of the features of the architecture. The second environment, Unreal Tournament (Infogrames, PC 1999), is a commercial game and supports a wider range of goals than does Tileworld. We use Unreal Tournament to evaluate the goal autonomy features of the GRUE architecture. The experimental results in Chapter 5 allow us to draw some conclusions about the usefulness of our architecture's features. The goal autonomy features boost performance in all of the environmental conditions we used. The use of preferred properties and the special notation for numerical values prove to be useful in those environments where there are opportunities to use them. The results of testing exclusive access to resources, the use of the same resource in consecutive cycles, and multiple execution of actions show that these features are less helpful for our implemented agents, and we suggest that more investigation is necessary to discover under what conditions these features should be used.

1.3 Applications to Other Problem Domains

Resource-based goal arbitration is potentially applicable to AI research outside of games and simulations. Resources are designed to model reasoning about properties of objects rather than the specific objects themselves. This is of course an ability that humans have, allowing us to substitute equivalent objects for each other in various situations. Using resources for goal arbitration allows an agent to defer a task until the appropriate resources are available for it. This is another aspect of general intelligence. Resource-based goal arbitration can be used in any problem domain in which the information available to the agent can be represented in resource form.

Robotics is one field with obvious correlations to autonomous game agents. Robots may be autonomous and operate in real-time (as defined above) depending on their intended function. Many robots (i.e. mobile robots rather than assembly robots) operate in a complex, unpredictable environment. In particular, robots entered in the Robocup soccer competition (except the small robot league (Robocup, 2004)) are expected to operate quickly in an environment containing other mobile agents and a moving ball. In fact, the Robocup simulation league could be considered a computer game, except that there is no human player. The GRUE architecture is based on work originally done in robotics (Nilsson, 1994; Benson and Nilsson, 1995; Benson, 1996), although this was not resource based. Our goal arbitration method could be used for robotics applications where it is possible to transform the robot's sensor data into resources.

Having said that, the usefulness of a particular goal arbitration method and of other specific features of an agent architecture are partially dependent on the problem domain. In the course of evaluating our architecture we will consider several properties of environments that affect the utility of our architecture design. The first is the speed of the environment. This has to do with how fast things change. Changes can be movements or actions of other agents, the set of objects and/or agents available, and any other changes that affect an agent's actions. If the rate of change is so high that by the time the agent makes a decision the situation that prompted that decision no longer exists, then there is no point even attempting to make decisions. Conversely, if changes occur only extremely rarely, the agent can afford to spend a lot of time planning what to do next. This is related to the duration of the environment. A game will probably last a relatively short amount of time (up to about an hour or so), a simulated world could last for an entire day (depending on the specific application), while a robot may be expected to go about its business for days or weeks. This again affects the type of abilities that are useful in the environment. A robot may need to be able to schedule a task for later in the day or even for several days later, while for an agent playing a 20 minute game this ability is completely irrelevant.

A second property of environments is density. We use density to refer to the number of resources available to be used by the agent, and in particular to the number of consumables and movable objects (as opposed to stationery features or information that may be available as resources). If an agent is operating in an environment with very few objects, providing the ability to select the optimal object is not especially helpful. If the agent is in an environment that is always full of objects it will always be easy to find an object with the most desirable properties, so complex reasoning about which object is best will not be very helpful in this type of environment either.

The third property of environments that we will mention here is the severity of consequences for neglecting tasks. This actually has a large effect on the best type of goal arbitration to use. In slowly changing environments where the consequences for neglecting a task are never severe, it is possible to allow agents to always finish working on their current task before starting a new one. If consequences can be severe, then it is better to switch to a high priority task immediately, even if it results in some work needing to be done over later.

The real world contains task that vary on all of these parameters. For instance, office environments are typically slow, with tasks that can be safely neglected for some amount of time, while firefighters work in a quickly changing environment (the smoke and fire move around, the building may collapse, etc.) with potentially severe consequences (if a firefighter fails to rescue a trapped person right away, that person may die). Fires are also typically limited in duration, while office tasks may last for days.

We aimed GRUE at environments which change fast enough to make spending a lot of time deliberating a bad idea, but slowly enough that some deliberation is possible. Because our target domain is games, we expect GRUE agents to be operating on a limited time scale (about 5 minutes up to about an hour). We also expect GRUE agents to find enough resources to make their tasks possible, but few enough that optimizing resource use is a good idea. This does not make it impossible to use GRUE in other situations, just suggests that in other types of environments a different architecture may be a better choice. Our evaluation of GRUE in Chapter 5 examines the utility of various features included in the architecture in different types of environments, particularly environments varying in density and rate of change. We also present a comparison between GRUE and several similar architectures in Section 3.4, in which we discuss the relationship between severity of consequences and goal arbitration methods.

1.4 Overview of Thesis

In this chapter, we have summarized the features of computer games and other real-time simulations, and motivated their use as a research platform. We have argued that creating goal-oriented agents is a good way of avoiding the simple, formulaic behaviour typical of characters in many games. Also, we have listed the features of our architecture, GRUE, and suggested reasons why these features ought to be advantageous for agents in dynamic simulations.

The following chapter gives a brief overview of some of the main approaches to the construction of intelligent agents. Some of these approaches have been used in games, and we summarize these where applicable. Chapter 3 describes the GRUE architecture in more detail, including complete descriptions of the algorithms and data structures used, as well as discussing the motivation for the design choices in detail. In Chapter 4 we describe the implementation of the GRUE architecture, and the two agents that have been developed using GRUE. Chapter 5 details the experiments we have performed with our two agents to evaluate the effects of the different features of the architecture. Chapter 6 suggests some useful extensions of the architecture, some of which could form the basis for future research. Finally, Chapter 7 summarizes our findings and discusses the implications of our work.

CHAPTER 2

Approaches to Agent Development

Tyrrell defines the problem of action selection as "choosing the most appropriate action out of a set of possible candidates" (Tyrrell, 1992). This is the central problem for intelligent agents. There are many different ways of solving it, and Humphrys even suggests learning algorithms to learn the solution (Humphrys, 1996). Blumberg identifies three important properties of action selection mechanisms: the ability to limit the amount of time for which the agent pursues a goal, the ability to time-share between goals of different priorities, and which uses a winner-take-all strategy for arbitration (Blumberg, 1994). While some arbitrate between competing actions (Agre and Chapman, 1987), we work at the level of competing goals. Our solution allocates resources to goals, then allows each goal to propose an action. The highest priority goal always gets first choice of both resource and actions, but other goals are allowed to execute actions also if they do not conflict. Our architecture has two of the properties identified by Blumberg, and time limits on goals can be implemented where necessary (Section 4.4 contains an example of this).

In Chapter 1 we gave the following list of features supported by GRUE:

1. Goals are generated only in appropriate situations.
2. Goal priorities are set according to the current situation.
3. Goal priorities are updated continuously.
4. Plans request resources in terms of required and preferred properties.
5. A plan can specify that it needs exclusive access to a resource.
6. A plan that makes the same request during consecutive cycles will receive the same resource each time.
7. Numerical quantities can be divided and used for more than one goal.
8. Plans can use special notation to request numerical values within a range.
9. Multiple actions can be executed during each execution cycle.

In this chapter, we will discuss GRUE's place amongst the various approaches to agent design, and the features that are supported by closely related architectures.

There are too many agent architectures represented in the literature to discuss them all. Instead, we have chosen systems that are representative of general trends or schools of thought in artificial intelligence. We have also chosen certain systems that are closely related to our own work, specifically BOD which falls into the category of behaviour based systems, MINDER1 which is part of the Cognition and Affect project, and the Teleo-Reactive Architecture, which was developed for use in robotics. We give a detailed comparison between these latter systems and our own architecture in Section 3.4.

2.1 Behaviour-Based Architectures

Several researchers have argued that complex behaviour may emerge from a large number of simpler behaviours which interact (Minsky, 1988; Agre and Chapman, 1987). Brook's subsumption architecture (Brooks, 1985; Brooks, 1991) is one such approach. The subsumption architecture consists of layers, where each layer contains one or more finite state machines augmented with timers. The architecture was designed to control robots and is assumed to be implemented directly at the hardware level. The robot reacts directly to the world, without any kind of deliberation. However, higher layers of the architecture can suppress both the inputs and the outputs of lower level behaviours. In this way, new layers can be added to the agent without modifying the existing behaviours.

Important properties of the subsumption architecture are that it allows agents to be built incrementally, agents should be robust, and the world is used as its own model instead of storing a world model inside the agent (Brooks, 1990). The subsumption architecture represented a new direction for artificial intelligence research (Arkin, 1998) at the time it was developed. Since then, many others have used behaviour based approaches to robot or agent design.

One similar approach is Maes' spreading activation networks (Maes, 1994). In this approach, all behaviours exist at the same level. Activation comes in from the environment, and spreads along links from one behaviour to another. The overall behaviour of the agent emerges from the interactions of the individual behaviours.

Bryson describes a development process called Behaviour Oriented Design (*BOD*) (Bryson, 2001). This is designed to help agent designers create agents using behaviour oriented techniques. She identifies a number of useful components for use in such agents. The first is an *action pattern* which is a list of actions to be done in sequence (or optionally in parallel). A basic reactive plan (*BRP*) is a list of production rules where each rule contains a condition, an action, action pattern, or BRP to be executed, and a priority for the rule. Rules are considered in order of decreasing priority. BRPs are almost identical to teleo-reactive programs (see Section 2.5), although BRPs can have more than one rule with the same priority which is not possible in a teleo-reactive program. Additionally, Bryson describes a *competence*, which is a BRP which additionally specifies the number of times a rule can be tried in a row before the agent gives up. Competences also return a boolean value indicating whether the competence executed successfully or whether none of the rules could fire. Finally, Bryson defines a *drive collection* which controls the other elements of the architecture. A drive collection consists of a number of *drive elements*, each of which contains a priority and a precondition like the production rules in a BRP. However, a drive element also contains the root of a BRP hierarchy (which when executed satisfies the drive), the current active element in the hierarchy, and optionally a frequency determining how often the drive element triggers. The current active element can be a competence or an action pattern. If a competence triggers another competence or action pattern, the new element becomes the current active element. If, on the other hand, the active element is an action which terminates, the current active element is reset to the top of the hierarchy. Bryson remarks that a drive collection effectively allows several behaviours to operate in parallel (Bryson, 2001). Because drive elements store the competence or action pattern they are currently running, drive elements can be interrupted and then resume exactly where they left off.

We based GRUE on Benson's work using teleo-reactive programs, which is extremely similar to BOD (see Section 2.5 below). It follows that BOD also has similarities to GRUE. While BOD does not explicitly represent goals, the drive elements serve the same purpose, and have conditions controlling when they can trigger. This effectively gives BOD the property that goals are only generated in appropriate situations. However, no mechanism is provided for changing the priority of goals. Bryson instead controls action selection using a combination of preconditions and frequency parameters. A drive element can trigger if its precondition is true and if it has not triggered within the amount of time specified by the frequency setting. This can have the same effect as changing priorities if set up correctly. For instance, consider the case of a robot with a battery that needs recharging. If the charge is very low, the robot needs to recharge before doing anything else. If the charge is fairly high, we would like the robot to recharge only if it is not busy. Using variable priorities, we would set the priority to increase as the charge decreases. In BOD we would need to give the drive element a high priority all the time, and a fairly low frequency so the robot can get other tasks done. To make this work properly, we need to state the task as "check if batteries need to be charged" rather than "charge the batteries". Then if the batteries do not need to be charged, the associated competence will simply exit without doing anything. So BOD does not really support the variable goal priority features of GRUE, but has equivalent functionality.

BOD does not support the resource features of GRUE. While a representation of resources could be used in BOD, the resource-based goal arbitration mechanism we designed for GRUE is incompatible with the functionality of Bryson's drive collections. We will elaborate on this issue in Chapter 3. The remaining feature of GRUE is that multiple actions can execute in parallel. In GRUE, we allow true parallelism: during each execution cycle, all plans may propose actions and all of them will run unless they conflict. BOD does not support true parallelism. It simply runs the highest priority drive element, subject to frequency constraints, while all other active drive elements are suspended. BOD does allow a single drive element to execute multiple actions in parallel.

2.2 BDI Architectures

The BDI framework is another popular method for building agents, and includes a philosophical model of human reasoning, a number of implementations, and an elegant abstract logical semantics (Georgeff et al., 1998). The basic components of the framework are beliefs, desires, and intentions (thus BDI). BDI has been applied to a number of applications, including a player for the game Quake II (Activision, PC 1996) (Norling, 2003).

Beliefs correspond to the agents memory. The argument is that an agent with no internal representation of the world is limited in that it can only sense a small portion of the world at once (Rao and Georgeff, 1995). By storing an internal world model, the agent can integrate different pieces of sensory information and build up a complete picture. The BDI framework does not have any requirements about how the beliefs are represented, only that they are stored within the agent. Desires represent the agent's current objectives and the associated priorities of those objectives (Rao and Georgeff, 1995) and are equivalent to goals (Georgeff et al., 1998). Rao and Georgeff state that they use the term desire rather than goal because the agent may have many desires and those desires may be mutually exclusive, while the word goal has sometimes been defined as not having these properties (Rao and Georgeff, 1995). Intentions, in contrast represent the current course of action. In other words, once the agent commits to working towards a desire, the desire becomes an intention.

The BDI framework requires that an agent have some representation of each of these components, but places no restrictions on how they are represented and processed. The GRUE architecture contains a memory component which corresponds to the set of beliefs used by a BDI agent, goals which correspond to BDI desires, and tasks which serve the same purpose as BDI intentions. Thus, our architecture could be considered a BDI architecture. We did not, however, specifically base our architecture in the BDI philosophy and we believe that the terms desire and intention could be confusing to those unfamiliar with the BDI framework. We should also note that other architectures, such as Soar (see Section 2.3), can also be mapped onto the BDI framework (Georgeff et al., 1998), but that the main thrust of research done on BDI has focused on the "logical and philosophical underpinnings" of agent architectures (Georgeff et al., 1998) while other research has taken different approaches.

2.3 The Cognitive Modeling Approach: Soar

Cognitive models are models of human cognition. That is, they provide common mechanisms to support cognition, and those mechanisms are based on theories of human cognition. It is possible to create agents for many types of applications as well as to model human problem solving behaviour with an architecture designed using a cognitive modeling approach. The two best-known cognitive models are Soar and ACT-R. We choose to describe Soar rather than another cognitive architecture for two reasons. One is that Soar is extremely popular. Many researchers are developing Soar, and a large number of related research papers have been written (Soar, 2004). This alone makes it worthy of mention as one of the major approaches to agent development. Secondly, much work in games and real-time simulations has been done using the Soar architecture (e.g. (Laird and Duchi, 2000; Laird, 2000; van Lent et al., 1999; Magerko and Laird, 2004)). This makes it directly relevant to the work described in this thesis.

We give a brief description of Soar here; an overview is (Lewis, 2001). The Soar architecture uses a long-term memory which stores production rules representing general knowledge. During each cycle, the architecture fires all production rules in long-term memory that match the contents of the short-term memory. Short-term memory contains information about the current state, including the current goal, sensory input and the results of firing production rules. The rules all fire in parallel, avoiding any decisions about which rules should fire first. This firing of rules is repeated until no more rules can match. During this phase, the rules may propose actions to work toward goals, which are stored in the short-term memory. These actions have preferences which are set when they are proposed, and also stored in short-term memory.

Once the system is no longer able to fire any production rules, Soar ranks all of the actions that have been proposed. The most preferred action is selected and executed. Actions can have a variety of preference values (Laird and Congdon, 2004), which may specify either the agent's attitude toward that particular action, or the relative priorities of two actions. An action may be designated the best or worst action, specified as better than a competing action, or specified as worse than a competing action. The preference value may indicate that the action must be selected for the current goal to be achieved, or that the action cannot be selected if the agent is to successfully achieve the current goal.

If Soar ends up in a state where no rules can fire, or has two proposed actions with equal preference values, Soar creates a sub-goal. This sub-goal is simply to resolve the problem situation. When the sub-goal is resolved, Soar may create a new production rule so that the problem situation will not occur again.

Both Soar and ACT-R use only a single goal hierarchy(Johnson, 1997). There have been some attempts to use multiple goal hierarchies in Soar, however they require either representing some goals implicitly or forcing unrelated goals into a single hierarchy (Jones et al., 1994). According to (Lehman et al., 1998), Soar creates a top-level goal automatically and all other goals are generated by the architecture to resolve situations in which the choice of action is not clear. In order to force Soar to generate a particular goal then, the agent designer must create a set of production rules that cause a situation in which Soar must generate the desired goal. While it may be possible to create Soar agents that generate goals when appropriate and dynamically update the goal priorities as in our architecture, the design of Soar does not make it easy or natural to do so.

Soar also does not use an explicit representation of resources, although the agent designer could create production rules that do so. The wide variety of preference values supported for actions in Soar means that Soar has equivalent functionality to our preferred and required properties of resources. The other features we specified for our architecture are not explicitly supported by Soar, although they could be implemented in production rules with the exception of parallel execution of actions. Soar is explicitly non-parallel, regarding a choice of actions as a problem to be solved. Of course, this problem could be worked around by causing actions to execute in sequence or collecting actions commonly done together into aggregate actions.

2.4 The Cognition and Affect Project

The Cognition and Affect project at the University of Birmingham is concerned with the study of cognitive architectures. They propose different mechanisms which may be present in different animals, and which might also support cognition in robots or simulated agents. See, for example (Sloman and Logan, 2000; Sloman, 2002; Hawes, 2003; Sloman and Chrisley, 2003; Beaudoin, 1994). Sloman proposes a three-layer architecture (Sloman and Logan, 2000), containing a reactive layer, a deliberative layer, and a meta-management layer.

The reactive layer consists of hard-wired behaviours. These may execute in parallel, and have simple conflict resolution strategies. Only very limited learning is possible in the reactive layer. Some extremely simple creatures may be purely reactive. The deliberative layer includes processes like planning and learning. These are more computationally expensive, but allow much more flexible behaviour and give an agent the ability to adapt to new situations.

Both reactive and deliberative capabilities are common in artificial intelligence research. Some researchers favor approaches that are purely reactive or purely deliberative, and others favor hybrid approaches. Sloman (Sloman, 2002) asserts that we need both kinds of abilities, as well as meta-management capabilities. The meta-management layer gives an agent the ability to reason about its reasoning. Humans can, for instance, notice patterns in their use of plans and readjust their planning accordingly, describe a chain of reasoning to others, deliberately change strategies or standards used for deliberation and so on.

One recurring theme in the Cognition and Affect project is accounting for emotions. Sloman (Sloman, 2002) makes the point that there are different types of emotions that must be supported by different types of underlying mechanisms. Some emotional states can be accounted for with only a reactive layer, while others can not occur without meta-management capabilities. Furthermore, the Cognition and Affect researchers argue that many emotions may occur naturally due to the interaction of other aspects of the architecture. Some of them may even represent error states (Sloman, 2002).

One specific agent architecture developed as part of the Cognition and Affect project is MINDER1 (Wright, 1997). MINDER1 is an autonomous agent architecture that was developed for the purpose of investigating emotional states. MINDER1 agents contain a number of motives, which are structures containing a condition to be made true, an insistence value, and a flag indicating whether or not the motive has moved through an attention filter. While Wright uses the term motive rather than goal, his motive structures are almost the same as our goal structures and his motives serve the same purpose as goals. Wright includes the ability to create a motive structure when appropriate environmental conditions are met. Like GRUE, MINDER1 is based on teleo-reactive programs, which function as plans to be associated with MINDER1's motives. Once a motive is generated, it moves through the attention filter if its insistence value is high enough. The insistence of motives can change over time, and the threshold of the filter is raised and lowered to control the number of motives being processed. Motives that have moved through the filter are processed by the management layer, which determines whether to continue processing the motive, expands it by finding an appropriate plan, and schedules it for execution. While Wright proposes full scheduling capabilities, the implemented system simply chooses one of the set of motives to be processed during the current cycle. Two of MINDER1's capabilities are identified as meta-management level abilities. One of these is the adjustment of the filter threshold, which is raised when there are more than three motives being processed, and lowered when there are fewer than three. The second is the ability to detect perturbant states that emerge while the agent runs. Wright identifies two types of perturbant states: filter oscillation and decision oscillation. Filter oscillation occurs when a rising filter threshold forces all the current motives out of the management layer, which in turn causes the filter threshold to drop, which allows the same set of motives back into the management layer. Decision oscillation is a state in which the agent oscillates between two competing motives.

Of the features supported by our architecture (GRUE), Wright's system shares all of the features relating to goals. It generates motives (goals) only in appropriate situations. The priorities are set initially according to the current situation, but can be updated as the situation changes. However, the system includes no representation of resources. As in GRUE, the plans used by Wright's system are teleo-reactive programs, meaning that our representation of resources could be used by Wright's system without modification.

However, Wright's management layer would need to be modified to take into account the availability of resources. Finally, GRUE runs several plans in parallel and allows multiple actions to be executed simultaneously. Wright's system lacks this parallelism; it only processes one plan at a time, switching between them as their motives become more or less insistent.

2.5 Teleo-reactive Programs and the Teleo-reactive Architecture

Many agent architectures have been developed for use in robotics. Robotics is a tempting area to work in because it involves the development of agents that operate in the real world. Robotics has potentially serious drawbacks, such as the fact that the hardware must work properly if the robot is to be used for anything. It also has difficulties in that current sensors are fairly limited and almost always noisy. However, techniques from robotics can be successfully transfered into other domains, such as computer games and real-time simulations. Examples include (Horswill and Zubek, 1999).

We have already discussed the subsumption architecture, which is one of the most well known architectures in robotics. Another approach from mobile robotics is described in (Firby, 1989). Firby states that it is usually impossible to determine the exact details of a plan ahead of time. He advocates the use of high-level plans and determining the exact details at run time. The high-level plans are constructed from reactive action plans (RAPs), which specify a number of methods of performing a task. Each method is tagged with information about when to use it, and one method will usually be a generic solution that will work in most cases.

In (Nilsson, 1994), Nilsson takes a similar approach to Firby. Here we discuss this work in the context of a full agent architecture based on Nilsson's teleo-reactive programs. The system described in (Benson and Nilsson, 1995; Benson, 1996) was developed to control agents in dynamic environments, and represents an attempt to blend ideas from control theory with standard computer science techniques in order to give the agent the continuous feedback necessary to operate effectively (Nilsson, 1994).

Benson and Nilsson's architecture comprises a library of plans, an arbitrator, a planner, and a learning system. The plans are teleo-reactive programs. A teleo-reactive program (*TRP*) consists of a series of rules, each of which consists of a condition and an action. The program is run by evaluating all the rules and executing the first rule whose condition evaluates to true when matched against a world model stored in the agent's memory. The conditions are evaluated continuously — ideally by an electric circuit, but otherwise continuous evaluation is simulated by executing the smallest time steps practical for the application (Nilsson, 1994).[1] This allows the agent to respond quickly to changes in the environment. Actions that continue executing as long as a condition remains true are called *durative actions*. When simulating continuous evaluation, a rule whose condition evaluates to true during a time-step either executes an action or starts a durative action. If the durative action has already been started, the rule simply causes the action to continue executing. Durative actions terminate when the rule that started them ceases to fire.

Each TRP achieves a single goal. The first rule in a TRP encodes the goal condition achieved by the program and performs the null action. The next rule contains an action

[1]Nilsson notes that several other systems based on control theory and electrical circuitry have been proposed, primarily Kaebling's GAPPS system (Kaebling and Rosenschein, 1994), but asserts that these systems, because they are pre-compiled, may construct extra circuitry. In contrast, TRPs construct circuitry at run-time, thus creating only what is needed.

that can make the goal condition true, and so on. Each action achieves a condition higher in the list of rules. This is referred to as the regression property (Nilsson, 1994). Teleo-reactive programs may contain free variables.

A TRP may execute another TRP as an action. This can result in a hierarchical tree structure, where each rule is referred to as a *node*. Benson and Nilsson describe heuristic way of executing a TRP tree, which reduces the amount of computation needed. Instead of searching the whole tree every execution cycle, they store the rule currently being executed. On subsequent cycles, they continue switch to the next rule up in the tree if it has become true, and otherwise continue executing the current rule. If both rules become false, then the whole tree is searched. This method reduces computation time, but also reduces the system's ability to take advantage of opportunities. To counter this problem, the system contains a separate process that checks the tree above the current node to see if any of the higher rules can fire. If so, that new rule becomes the current node. This separate process can run more slowly than the rest of the system since opportunities are expected to happen infrequently.

One of Benson and Nilsson's agents attempts to achieve a set of goals which may be provided by a human operator. The *arbitrator* allows the agent to work on several goals at once, by determining which TRP should be allowed to perform an action at each cycle (Benson and Nilsson, 1995; Benson, 1996). TRPs are selected by the arbitrator using the concept of stable nodes. A *stable node* is a rule in a TRP where execution of the program can safely be suspended. That is, the work done up to that point in the TRP is stable with respect to the other TRPs that are running. A rule is stable if running the other TRPs will not cause the condition of the rule to become false. So, for example, a TRP used by a package delivery agent might require the agent to pick up an object and take it somewhere. The condition of having the object is stable with respect to any plan that does not require the agent to drop the object.

The set of stable nodes is compiled before the TRPs start running using STRIPS-style (Fikes and Nilsson, 1971) delete lists for each action. These lists specify all conditions in the world that are made false when the action executes. To find out if a node in a hierarchical TRP structure is stable with respect to another hierarchy, simply check the condition on the node against the delete lists for all of the actions in the other TRP hierarchy. Computing the stable nodes for a set of TRP hierarchies requires checking the condition of every node against the delete lists for every node in all of the other hierarchies. While this is computationally expensive, it only needs to be done when the set of available TRP hierarchies changes.

During each execution cycle, the arbitrator runs the TRP with the best expected reward/time ratio. The reward is the return the agent expects for achieving the goal (which may or may not actually be received) and the time is the estimated time necessary to reach the closest stable node. The arbitrator uses stable nodes to avoid undoing things it has already done. As stable nodes are safe places to stop TRPs, the arbitrator runs each program until it reaches one; it can then switch to another TRP if appropriate. This allows the agent to take small amounts of time to achieve less rewarding goals while it is also working on a more time-consuming but more rewarding goal. When a plan achieves its goal and runs the null action, it is removed from the arbitrator. (Benson and Nilsson, 1995) indicates that rewards are dependent on user priorities, internal, and external conditions and than an expected reward may not correspond to an actual reward. Time estimates are computed using a heuristic function, which will depend on the actions being performed as well as the length of the TRP.

Benson and Nilsson's architecture included capabilities for planning and learning, which are explored in greater detail in (Benson, 1996). The planning system, when given a goal, constructs a TRP to achieve that goal. This is done using teleo-operators, which are data structures that represent actions in a useful format for constructing TRPs. A teleo-operator contains the name of the action, a postcondition that indicates the desired effect of the operator, a precondition that must hold if the operator is expected to work, and a set of side effects. The set of side effects contains conditions that are made true and conditions that are made false by executing the action. The planner uses a backward-chaining method to find a series of teleo-operators that connect the desired goal to the current conditions. The designer may also ask the planner to construct TRPs for a set of goals and store the results. These TRPs can then be used as operators when constructing more complicated TRPs, resulting in a hierarchical tree structure.

The learning system learns the preconditions for teleo-operators. Given an action and the expected effect of the action, the learning system then records any changes in the state of the environment while executing the algorithm. From the environmental states, the learning system derives the precondition of the teleo-operator. The system can also compute side effects of teleo-operators, along with probabilities of those side effects occurring. Benson and Nilsson describe the use of a simple concept learning method (Benson and Nilsson, 1995), while (Benson, 1996) describes the use of inductive logic programming.

Our architecture, GRUE, is based on Benson and Nilsson's agent architecture. Like Benson and Nilsson, we use a library of teleo-reactive programs. However, we have defined as special type of variable which is the only type of variable allowed in our TRPs. Other differences between GRUE and the system described in (Benson and Nilsson, 1995) include the specific provision of architecture components to support the generation of goals, and a novel arbitration method which is based on the availability of resource and allows several TRPs to execute simultaneously rather than switching at stable nodes. We further discuss Benson and Nilsson's work in Chapter 3 along with the specification of our own architecture.

2.6 Summary

There are many architectures available for designing intelligent agents. Architectures can be based on human behaviour, inspired by anything from motor skills to cognitive psychology to philosophical ideas about human psychology. Architectures may also be designed to support complete, although possibly simplistic, agents. Some agents are used primarily for software agents, while others are usually applied to robotics. We chose to base our own architecture on Benson and Nilsson's architecture for several reasons. One is that their architecture was designed to handle the type of dynamic environment typically encountered by robots, which has similar properties to the dynamic environments encountered by agents in computer games. Another is that we find teleo-reactive programs an intuitive way to construct plans for agents. Computer games require agents that are goal autonomous. To meet this requirement, we have redesigned the arbitrator component of the architecture resulting in our novel resource-based goal arbitration method.

CHAPTER 3

The GRUE Architecture

In this chapter, we discuss the teleo-reactive architecture described in (Benson and Nilsson, 1995; Benson, 1996) which we use as the basis of our work, and highlight some of the problems which motivated the development of our own architecture. We then introduce the notion of a *resource*, which is used to store information in the agent's working memory, and a resource variable, which represents a condition which must be true for the safe concurrent execution of a durative action, and describe our goal processing architecture which extends teleo-reactive programs with resources. We describe the main components of the GRUE architecture and present an algorithm for goal arbitration using resources. Finally, we give a detailed comparison between GRUE and several similar systems.

3.1 Benson and Nilsson's Architecture

We previously discussed this system in Section 2.5. However, we remind the reader that the architecture comprises a library of plans, an arbitrator, a planner, and a learning system, where the plans are teleo-reactive programs. A teleo-reactive program (*TRP*) consists of a series of rules, each of which consists of a condition and an action. The program is run by evaluating all the rules and executing the first rule whose condition evaluates to true when matched against a world model stored in the agent's memory.

Teleo-reactive programs have a number of advantages for controlling agents in dynamic environments. They gracefully handle changes in the environment, whether those changes force the program to go back to previous steps or allow it to jump ahead. The arbitrator allows a teleo-reactive agent to perform actions which work towards multiple goals (Benson and Nilsson, 1995), in that the arbitration algorithm can switch to a different teleo-reactive program for each execution cycle, effectively running all the programs in pseudo-parallel.

The arbitrator switches between teleo-reactive programs at *stable nodes*. Given a set of teleo-reactive programs, a rule is stable with respect to the set if its condition is not made false by the any of the actions in the other programs. Thus, once this condition becomes true, the arbitrator can switch to another TRP and expect that the work done before reaching the stable node won't be undone. The arbitrator always works on the teleo-reactive program with the lowest reward/time ratio, where the time is the estimated time to reach the next stable node and the reward is the expected reward for completing the task. (Benson and Nilsson, 1995) indicates that rewards vary depending on user priorities, the current state of the agent, and external conditions and than an expected reward may not correspond to an actual reward. Time estimates are computed using a heuristic function, which might simply be the number of rules in the TRP or it might be computed based on the actions being executed.

In this section, we will illustrate the use of TRPs in computer games by writing some TRPs for a simple game, *Pac-Man* (Namco, arcade machine). We then discuss some of the limitations of Benson and Nilsson's architecture.

3.1.1 Example: Pac-Man

As an illustration of Benson and Nilsson's architecture, we present a collection of teleo-reactive programs that might be used to play the game Pac-Man. We remind the reader that in the game, the player controls a yellow character (Pac-Man) who moves around a maze eating dots in order to earn points. The maze contains hazards in the form of multi-colored ghosts. The player starts with three lives, and the game is over when all the lives are gone. If a ghost catches Pac-Man, the player loses a life and Pac-Man is placed in a safe position. There are also special dots called power pills which Pac-Man can eat to make the ghosts vulnerable; the ghosts turn blue to indicate this. While the ghosts are blue, Pac-Man can eat them and earn points. The player's score can also be increased by eating fruits, which sometimes appear in the middle of the maze. Note that Pac-Man is always eating—it is not possible for Pac-Man to move over an edible object without eating it. Our Pac-Man agent will play Pac-Man as if it were a human player. That is, it can see the entire maze and all the ghosts at any time.

We have chosen to implement the Pac-Man agent using four teleo-reactive programs which achieve four top-level goals of the game: eating dots; escaping from ghosts, which allows the player to stay alive and continue play; eating blue ghosts; and eating fruit. Pseudo-code for the four programs is given in Figure 3.1.

At any given moment, the Pac-Man agent is running one of the above programs,

Eat Dots

```
if      no dots
then    null

if      there are dots
then    move towards a dot
```

Escape from Ghosts

```
if      no ghost is less than 10 units away OR all ghosts are blue
then    null

if      there is a ghost within 10 units AND the ghost is not blue
then    move away from ghost
```

Eat Blue Ghosts

```
if      no ghosts are blue AND no power pills
then    null

if      there is a blue ghost
then    move towards the ghost

if      no ghosts are blue AND there is a power pill
then    move towards the power pill
```

Eat Fruit

```
if      no fruit
then    null

if      there is a fruit
then    move towards the fruit
```

FIGURE 3.1: Pseudocode teleo-reactive programs for a Pac-Man agent.

say the program for eating blue ghosts. The rules are examined from the top down, so if there are no blue ghosts and no power pills then there is nothing to do. Let's assume there are power pills, so we continue to the next rule. Rule 2 states that if there is a blue ghost present, we should chase it. However, if there are no blue ghosts, we continue down to the third rule, which tells us to eat a power pill to turn the ghosts blue, thus enabling us to switch to using the second rule.

When processing the rules for the *Eat Blue Ghosts* program, Benson and Nilsson's arbitrator checks to see if the highest condition which is currently true is stable with respect to other TRPs, and if so switches to the program with the best reward/time ratio (which may be the *Eat Blue Ghosts* program). However, Pac-Man is a fast-paced game, making most conditions unstable. In the set of programs above, we can identify only a few conditions such as "no fruit" being present which will not be changed by any of the other programs. However, this is not very useful as a stable node as "no fruit" being present is a success condition, so if it is true the *Eat Fruit* program will stop running. Other conditions, such as "there is a fruit" are unstable because any program that causes Pac-Man to move might cause Pac-Man to eat the fruit. This is due to the nature of the game—Pac-Man simply eats anything it runs into.

3.1.2 Limitations of Benson and Nilsson's work

The Benson and Nilsson's architecture has a number of limitations. One problem is that the distribution of stable nodes throughout the active plans impose a minimum latency on responses to changes in the environment or the agent, since the agent will only consider switching task when it reaches a stable node. In some environments, it is difficult to find stable nodes, making it impossible to use them to allow pseudo-parallel execution of TRPs. If there are few stable nodes, arbitration can have the effect of forcing the agent to work towards the unachieved goal with the highest reward to the exclusion of all other goals. More generally, if two programs are the same length (in terms of estimated time to complete) and have the same expected reward, but one of the two has fewer stable nodes, then that program will have a lower reward/time ratio and as a result will tend to be starved of processing time. It would be better to have a general solution that allows the agent to recognize when it is not making progress on a high priority goal and switch to a lower priority goal until the situation changes instead of using stable nodes.

In our Pac-Man programs, there were no stable nodes that were not success conditions. If we were to try to switch between these programs at stable nodes as in (Benson and Nilsson, 1995), the arbitrator would run only one program at a time. However, it is easy to see that running *Escape from Ghosts* can cause Pac-Man to move towards (and hence eat) a power pill, which also makes progress towards the goal of *Eat Blue Ghosts*, suggesting that another approach may be more effective in this environment.

Benson and Nilsson's arbitrator also has the property that it will take time out from an urgent goal to work on a less important one that can be completed quickly. In a game environment, pausing during an urgent task could have drastic (even fatal) consequences. One could, of course, make the reward extremely high, but we feel it is more natural to associate each goal with a priority that can optionally be dependent on the time to complete the task.

The need to compute stable nodes also has ramifications for the syntax of rules.

Program A

```
if      guard present AND (have ruby OR have emerald)
then    bribe the guard
```

and

Program B

```
if      have ruby AND unicorn present
then    give ruby to unicorn
```

FIGURE 3.2: Rules with disjunctions.

For example, consider two rules in two different programs as shown in Figure 3.2.

Assuming that both a guard and a unicorn are present, running the rule in *Program B* will make the condition of the rule in *Program A* false only if the agent does not have an emerald. Whether or not the agent has an emerald cannot be determined until run time. There are two possible solutions to this. One is to simply mark the condition as unstable in all cases, which will prevent the agent from taking advantage of the circumstances where it is stable. The second solution is to split the disjunction into two separate rules; however this increases the number of rules that must be processed. [1].

The use of stable nodes also means that the agent must operate in a deterministic environment. Computing the stable nodes requires a list, for each action, of conditions that are falsified by that action which in turn requires the ability to predict the effects of actions in the environment. The agent programmer has the long and tedious task of listing every possible action, in every possible situation, with every possible consequence. Listing every

[1]Disjunctive conditions are not discussed in (Nilsson, 1994), (Benson and Nilsson, 1995) or (Benson, 1996), however both (Benson and Nilsson, 1995) and (Benson, 1996) contain examples with disjunctive conditions. Most likely Benson and Nilsson simply marked such conditions as unstable.

possible consequence may be intractable especially in multi-agent applications, where for instance the result of an action may depend on the actions of another agent (which are unpredictable).

Finally, Benson and Nilsson do not provide a mechanism for generating goals. While they allow agents to have predefined goals (Benson and Nilsson, 1995), in addition to goals provided by human users, they do not say how these goals are represented or prioritized within the system.

3.2 GRUE: A New Architecture

We have developed a new teleo-reactive architecture, GRUE (Goal and Resource Using Entity), which overcomes these limitations:

- it provides support for goal generators, allowing an agent to generate new top-level goals and adjust the priorities of existing goals in response to the current environment;
- it allows the agent to make progress towards multiple goals in environments with few stable nodes; and
- it allows true parallelism rather than co-routineing, with multiple programs running actions in parallel during each cycle where this is possible.

Our architecture contains four main components: a memory, a set of goal generators, a program selector and the arbitrator (see Figure 3.3).

The agent operates in a world which could be a game engine, the physical world (for a robot), or some other environment. The world interface takes information from the

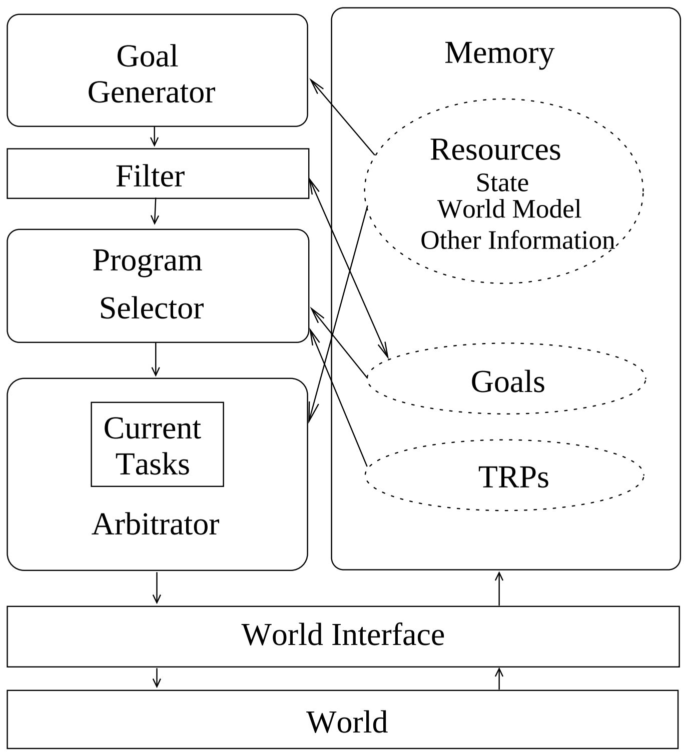

FIGURE 3.3: GRUE

environment, in our case a game engine, does any necessary pre-processing, e.g. computing the distance between game objects and the agent, and stores the results as *resources* in the memory module. The information obtained through the world interface may include information about the agent, if it is obtained from the agent's sensors. Information may additionally be placed in memory by the agent's programs; the distinction is that information processed by the world interface is information about the agent's interaction with the world. Data produced by the agent's programs is purely the result of internal processing. The memory module stores all information in a single database. This allows easy access to all types of information. The ellipses in Figure 3.3 indicate the different types of information and their uses, but are not meant to suggest that they are stored separately. Goal generators are triggered by the presence of particular information in memory. They create appropriate goals, computing the priority values as necessary and putting the goal into memory. For example, a goal generator might be triggered when the agent's health is below a particular value. It would then generate a goal to regain health, with a priority that is inversely proportional to the agent's current health value. Note that the agent can only generate goals for which goal generators have been provided. While the set of possible goals is predefined, a goal does not exist in memory until it is generated. Once created, goals pass through a filter which eliminates low priority goals. Adjusting the filter threshold can adjust the number of goals entering the arbitrator. The program selector is a simple look-up function, which appends the appropriate teleo-reactive program to each goal to create a task. It is provided mainly to allow for possible future additions of planning or learning components. The overall aim of the arbitrator is to run as many tasks as possible, subject to resource constraints and giving priority to tasks which achieve more important goals. Each task run by the arbitrator is allowed to request resources, and if the appropriate resources are available the task proposes one or more actions to be executed. The arbitrator checks the list of proposed actions for conflicts before actually executing them. When a conflict occurs, the action proposed by the lower priority task will be rejected [2].

During each execution cycle, the world interface processes any new sensory input and places it in memory. Next the goal generators generate new goals based on the sensory data, and update the priority of tasks already in the arbitrator. Then the filter removes any goals or tasks with priorities below the threshold. The remaining goals are made into tasks

[2] A similar method for resolving conflicts between tasks is also used in Blumberg's ethologically inspired architecture for autonomous agents (Blumberg, 1997)

by the program selector, and these new tasks enter the arbitrator. The arbitrator then allows each task to propose an action, discards tasks that cannot run, and eliminates conflicts from the action list. Finally, the actions are executed. Actions may be directly executed in the world, or they may be processed by the world interface before being executed. At the end of each cycle, all of the resources used during the cycle are released for use in the next cycle.

We make a distinction between internal actions and external actions. Internal actions modify the agent's memory, for example to store information about the progress toward a goal. External actions are those that are done in the world. We provide a flag that disables the agent's ability to execute multiple external actions. In this case the agent still executes all of its internal actions, but only runs the highest priority external action. This flag is mainly for experimental purposes (see Chapter 5 for details); we expect that ordinarily all actions will be executed. The distinction between internal and external actions was made because some of the tasks in our agent need to add or modify several items in memory during a single step, and removing the ability to do so paralyzes the agent completely. More generally we think it does not make sense to limit the agent's ability to modify its own memory, so limitations on action execution only apply to external actions.

The world interface, the goal generators, and the set of teleo-reactive programs are all application-dependent. When building a GRUE agent, the programmer must write these components. The remainder of the GRUE architecture, in particular the arbitrator, are the same for all agents.

In the remainder of this section, we discuss the main data structures used by the program and their associated algorithms in more detail. We begin by outlining the contents of the agent's memory.

3.2.1 Resources

A key component of GRUE which distinguishes it from similar architectures is the way in which TRPs can obtain exclusive access to items in memory, indirectly precluding the execution of competing TRPs. This exclusive access is implemented using resources and resource variables.

Informally, a *resource* is anything necessary for a rule in a program to run successfully. Objects in the agent's environment are the most obvious example, but other more abstract things like facts, properties of the agent, or time periods can also be regarded as resources. More precisely, a resource consists of a unique identifier naming the resource together with a set of resource properties. Each property is an attribute value set, consisting of a property name and at least one property value. The set of relevant properties will depend on the application, but would typically include those features of the agent and its environment that are relevant to the agent achieving its goals.

Resources are output by the world interface (e.g., the agent's sensors) and are stored in the agent's memory. Each resource is represented as a list consisting of an identifier (a string) followed by one or more name-value sets. The identifier is actually the value of a property called ID which exists for all resources, and is treated like any other property. We list it separately in this chapter to make the notation easier to read. Property names are labels (strings) and property values are constants (strings or numbers). For clarity, we write property names in uppercase. For example, a health item might be represented by the structure

```
(HealthPack101 [TYPE HealthPack] [HEALTH-PTS 20])
```

which indicates that the item `HealthPack101` is of type `HealthPack`, and will restore 20 health points to the agent.

Properties can be multi-valued. A property which is *multi-valued* has more than one value for the same property name. For example, more than one category may be listed for the `TYPE` field; a pickaxe might be represented as

```
(Pickaxe102 [TYPE pickaxe weapon])
```

which indicates that the resource `Pickaxe102` is a pickaxe (a digging tool) and also a weapon.

A property which is declared to be *divisible* can be split, with part being used for one task and part being using for another. This is often convenient when tasks require a number of identical resources such as moments of time, rounds of ammunition or units of money. For example, we could represent 10 coins as 10 different resources or as a single resource with a divisible property `AMOUNT`. Divisible resources are treated specially by the binding algorithm (see below). If a numeric property value is prefixed by the `DIV` keyword, the binding algorithm treats the property value as a divisible quantity. For example, the resource

```
(Gold101 [TYPE money] [AMOUNT DIV 53])
```

can be used to buy several things, as long as the total cost is less or equal to 53. When a resource variable binds part of a resource, the remainder is put back into the list of available resources so it can be used by another program. Note that we must use the DIV keyword and then split the resource during the binding process, as until then there is no way to tell how many portions the resource should be divided into. At the end of the cycle, all divided properties are recombined as they may be divided differently during the next cycle.

3.2.2 Resource Variables

Nilsson's teleo-reactive programs contain free variables which are bound when the program runs (Nilsson, 1994). In GRUE TRPs, we allow only a particular type of variable which is bound to a resource when the TRP runs. A *resource variable* represents a resource which must be available throughout the execution of a durative action. Resource variables are place-holders for a resource in the condition of a rule in a TRP. Resource variables are matched against the resources in the agent's memory. Matching can be constrained so as to specify that only resources with particular properties are selected. When writing GRUE TRPs, we specify conditions in terms of properties of resources, and require that rules only run when the resource variables in its condition can be bound. Keep in mind that a rule in a TRP proposes the same actions regardless of which resource is bound to the resource variable. A resource variable specifies properties which are common to all resources that can be used for the action being proposed.

A *resource variable* is a 4-tuple containing:

- an identifier for this variable;
- a flag indicating whether the variable requires exclusive access to the resource;
- a set of required properties; and
- a set of preferred properties.

The identifier is the variable name (a string), which is bound to a resource. It can subsequently be used to access any of the properties of the resource. By convention, resource variable identifiers in this thesis begin with a '?' character. The mutual exclusion flag determines whether resource variables in other programs can bind to a resource bound by this variable. If the mutual exclusion flag has the value "`SHARED`", then the resource can be bound by other programs. For example, a rule that simply checks the existence of a resource, or which extracts attribute values from a resource which represents information about the environment can share the resource with rules in other programs. However in cases where a resource is deleted or modified by a rule, the flag should have the value "`EXCLUSIVE`". The following example matches a piece of information previously stored by a running program:

```
(?BaseLocation107 SHARED
 ([TYPE location])
 ())
```

and allows other programs to bind the resource.

During the evaluation of our implemented agents, it became clear that shared bindings have a requirement which needs to be stated here. Shared bindings may be used by other TRPs, but that use is limited to read-only access. Once a shared binding is made, other programs may not change or remove the bound resource.

Required properties are those that are necessary for the execution of a rule's actions or the maintenance of a goal condition (for maintenance goals). Preferred properties are used to choose between resources when more than one resource matches all the required properties. Programmers can use preferred properties to give agents different behaviour by specifying preferences for different kinds of otherwise equivalent resources. For example, a rule condition in an attack program might contain the following resource variable:

```
(?Gun104 EXCLUSIVE
 ([TYPE gun] [AMMUNITION 20])
 ())
```

which specifies a weapon with 20 rounds of ammunition and no preferred properties. It also specifies that the rule requires `EXCLUSIVE` access to the resource (e.g. another program cannot simultaneously sell or give the gun to another agent while it is being used by the attack program). If a particular kind of weapon is preferred, we can specify this using a preferred property:

```
(?Gun106 EXCLUSIVE
 ([TYPE Gun] [AMMUNITION 20])
 ([TYPE RocketLauncher]))
```

The matching of numerical values is handled slightly differently. It is often useful to be able to specify that the value of a particular property should lie within a particular range and, additionally, to specify an ordering over values within that range. For example, a program might require an object as close to the agent as possible (smallest distance value) or to prefer a weapon with more ammunition. We use the following notation to specify such additional constraints on resource variable matching.

A *value range* consists of brackets containing two numbers representing the extent of the range and, optionally, a utility arrow. We define two types of brackets: '#|' and '|#'

denote a range with firm lower and upper bounds and the brackets '#:' and ':#' denote a range with soft lower and upper bounds. A firm bound indicates that a number must be within the range to be considered to match. A soft bound indicates that numbers within the range have the highest utility, but values outside the indicated range are still considered to match. All range boundaries are inclusive, so a range with a firm lower bound of 5 would match 5 or more. For example, the following resource variable matches any monster between 1 and 10 units away inclusive:

```
(?Monster1 EXCLUSIVE ([TYPE monster] [DISTANCE #|1 10|#]) ())
```

The two types of brackets can be mixed, to represent ranges with one firm bound and one soft bound. When one bound of a range is soft, we allow the corresponding number to be omitted. If the soft bound is on the right, we assume that the range ends at $+\infty$, and if on the left at $-\infty$. (We expect that it most cases it will be clearer to specify the lower end of the range explicitly.) For example, the following resource variable asks for an amount of money with a value of at least 10 and no upper limit:

```
(?Gold102 EXCLUSIVE ([TYPE money] [AMOUNT #|10 :# ]) ())
```

The arrows, –> and <– are used to indicate a utility ordering over matching values. So if we want to specify that closer monsters are preferred, then we can add an arrow:

```
(?Monster1 EXCLUSIVE ([TYPE monster] [DISTANCE #|1 <- 10|#]) ())
```

Similarly, the following resource variable requires a minimum of 10 units of money, but specifies that more is better:

```
(?Gold103 EXCLUSIVE ([TYPE money] [AMOUNT #|10 -> :# ]) () )
```

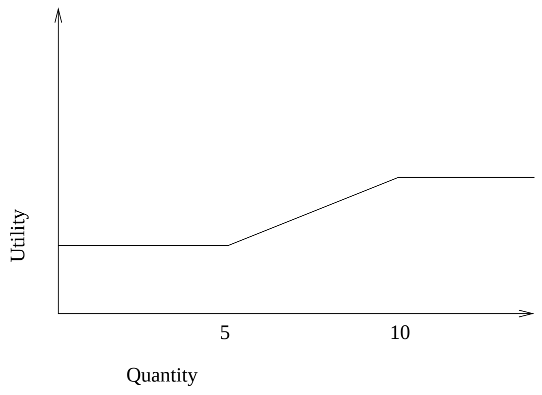

FIGURE 3.4: Graph of utility for the value range #:5–>10:#. Numerical values are not actually assigned a utility value; rather the range is used to compare a group of values and rate them as having higher or lower utilities relative to each other. Thus the actual slope of the lines in these graphs is not important, only the distinctions between increasing utility, decreasing utility and flat utility values matter.

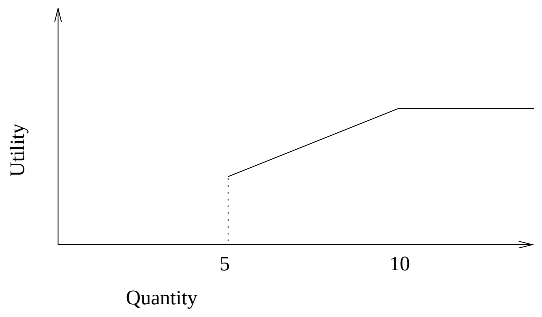

FIGURE 3.5: Graph of utility for the value range #|5–>10:#

Figure 3.4 illustrates the relative utility of property values in the soft-bounded range #:5–>10:#. Contrast this to Figure 3.5, which shows a range with one firm bound and one soft bound. Numerical values are not actually assigned a utility value; rather the range is used to compare a group of values and rate them as having higher or lower utilities relative to each other. Thus the actual slope of the lines in these graphs is not important, only the distinctions between increasing utility, decreasing utility and flat utility values matter.

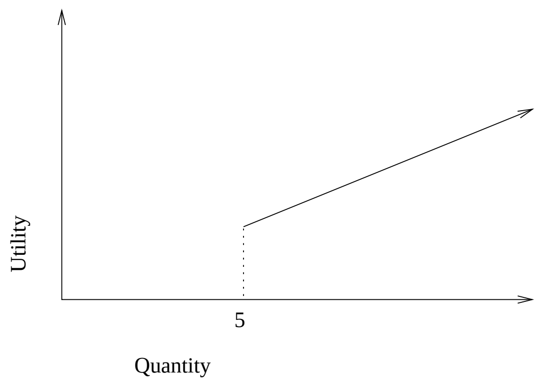

FIGURE 3.6: Graph of utility for the value range #|5–>:#

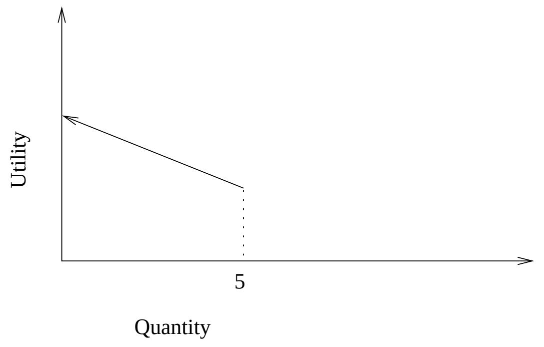

FIGURE 3.7: Graph of utility for the value range #:<–5|#

Figure 3.6 specifies a minimum of 5 and an upper limit of $+\infty$. Figure 3.7 specifies a maximum of 5 and a lower limit of $-\infty$. Note that the utility ordering can go in either direction. The range #| 5 <– :# indicates a maximum utility at 5, and a minimum utility at $+\infty$

Finally, `*` can be used as a special wildcard symbol. It is used in situations where we want to require that a resource has a particular property listed but we do not care about the value. For example, the resource variable:

```
(?Box115
([TYPE Container] [LOCATION *])
())
```

allows us to ask for a container for which we know the location, without requiring a particular value for that location.

The resource variables in each rule in a program are matched against the resources stored in memory. A resource variable only matches those resources which have all of the properties listed in its required properties list. If there is more than one resource which matches the required properties, the resource variable matches the resource with the largest number of preferred properties. If two or more resources have all the required properties and the same number of preferred properties, then one resource will be chosen arbitrarily.

If the resource variable specifies a property value range for a required or preferred property, binding works slightly differently. Required properties are checked as normal, with a numerical quantity matching a range if it is within the range. (For ranges with soft lower and upper bounds, any number matches.) For preferred properties, all resources with the specified required properties are checked to see whether the value of the preferred property lies within the range. Then all the matching property values are evaluated to find the best match according to the utility ordering specified by the range. For example, if the range is #| 5 –> 10 |#, then a value of 9 is better than a value of 6. The resource with the highest utility is considered to match the preferred property, and all the other resources are considered to not match.

We specify a function called *property* to retrieve information from a resource that is bound to a resource variable. If the resource variable

```
(?Gun116 EXCLUSIVE
 ([TYPE gun])
 ())
```

is bound to the resource

```
(Revolver116 [TYPE gun] [AMMUNITION 20])
```

we can then ask how much ammunition `?Gun116` has:

```
property(?Gun116, AMMUNITION) = [20].
```

If the requested property is not present in the resource, the property function will return false.

3.2.3 Goals

A *goal* is a list consisting of an identifier (a string), a priority (an integer), a type (`ACHIEVEMENT` or `MAINTENANCE`), and the name of a program that will achieve or maintain the goal condition (a string).

Achievement goals are straightforward. For example, we can write the goals for our Pac-Man agent as follows:

```
(GetAwayFromGhostsGoal 90 ACHIEVEMENT EscapeFromGhosts)

(EatBlueGhostsGoal 50 ACHIEVEMENT EatBlueGhost)

(EatFruitGoal 50 ACHIEVEMENT EatFruit)

(EatDotsGoal 20 ACHIEVEMENT EatDots)
```

The priorities are set appropriately by the programmer. In this case, we have judged escaping from ghosts to have the highest priority, while the next two goals are of equal priority. In general the goal generators should set priorities according to the current environment. For example, a robot might generate a goal to recharge its battery with a low priority when the battery is nearly full and a high priority when the battery is nearly empty.

In addition to setting the priority when goals are generated, the goal generators update the priority of tasks in the arbitrator. These updates are done continuously. The current implementation runs the goal generators every cycle.

Maintenance goals, where the agent is attempting to maintain a condition, are a special case. At first glance, it seems that a teleo-reactive program which achieves a goal will maintain the goal state automatically—if a necessary condition stops being true the program will automatically try to make it true again. However, teleo-reactive programs are normally removed from the arbitrator (see below) when their goal condition is achieved. The goal condition must be violated for the goal to be regenerated and the corresponding maintenance task to re-enter the arbitrator.[3] This can result in one or more goal conditions being achieved intermittently, rather than maintained. For example, a player may sell her weapon to get money to buy a potion, notice that she is without a weapon (a violation of a maintenance goal) and then buy it right back again! Tasks achieving maintenance goals therefore persist in the arbitrator, even when the goal condition is (currently) achieved. Only the top rule in the TRP will fire, producing a null action, but the rule can bind

[3]Assuming that the goal generators never fire when the condition of the goal they generate is already true.

those resources necessary to maintain the condition. Maintenance goals should have an appropriate priority so they can be used to prevent the character from disposing of necessary items or "forgetting" to maintain a crucial condition. Removal of maintenance goals from the arbitrator must be done by other TRPs.

The type field in the goal data structure is used to distinguish between maintenance goals and ordinary goals, for example:

```
(MaintainHealthPointsGoal 95 MAINTENANCE RegainHealthPoints)
```

3.2.4 Programs

GRUE Programs are 'standard' teleo-reactive programs as defined in (Nilsson, 1994) which use resource variables instead of the free variables allowed by Nilsson. A *GRUE TRP* consists of an identifier (a string), and one or more rules. Each rule consists of a condition and one or more actions to be executed. We allow other TRPs to be "called" in an action as if they were functions. These sub-TRPs may require arguments. Top-level TRPs (those run directly by the arbitrator) do not take arguments. In this section, we first formally define the syntax of rules, then we give some examples illustrating the definition.

rule: Zero or one **condition expressions** followed by $\Longrightarrow$ followed by one or more **actions**.

condition expression: A **condition** or the logical operator NOT followed by a **condition expression** or a **condition expression** followed by the logical operator OR followed by a **condition expression** or a **condition expression** followed by the logical operator AND followed by a **condition expression**.

condition: A resource variable or p($A_1 ... A_n$) where p() is a predicate.

A_i: Either property(resource variable, property name) or f($A_1 ... A_n$) where f() is a valid function in the implementation language (which may be user-defined).

action: The null action or propose(action name, arguments) or a TRP followed by a list of arguments.

The rules are evaluated in order, with the first rule whose condition evaluates to true proposing an action to execute. A resource variable evaluates to true when it successfully binds to a resource. A negated resource variable evaluates to true if no binding exists. (Binding of resource variables is discussed in more detail Section 3.2.6.) Conditions are always evaluated with respect to the agent's memory, which corresponds to an agent's beliefs about the world. These beliefs are not guaranteed to be correct, e.g. if the agent's sensors return partial information about the environment or the world changes between observations.

As an example, Figure 3.8 shows a GRUE TRP for our theoretical Pac-Man agent which achieves the *GetAwayFromGhosts* goal.

The first rule in this program matches if all the ghosts are more than ten units away or if all the ghosts are blue. To express this in terms of resource variables, we must use the negation of a resource variable. So the rule runs when the binding algorithm can't locate a resource of type ghost with a distance of less than ten and also can't locate a ghost resource with a value of false for the *BLUE* property. The second rule matches when a ghost that isn't blue is nearby. It requires both a value of false for the *BLUE* property and a distance of less than ten units. We have ignored the case where a blue ghost is less than

```
(GetAwayFromGhosts
  NOT(?Ghost1 SHARED
             ([TYPE ghost] [DISTANCE #|1 <- 10|#])
             ()) OR
  NOT(?Ghost1 SHARED
             ([TYPE ghost] [BLUE false])
             ())
  ⟹
  null

 (?Ghost1 SHARED
         ([TYPE ghost] [BLUE false] [DISTANCE #|1 <- 10 |#])
         ())
  ⟹
  propose(move-away-from, ?Ghost1)
)
```

FIGURE 3.8: A GRUE TRP for a Pac-Man agent.

ten units away. In the example we gave in Section 3.1.1, that case was handled by the *Eat Blue Ghosts* program.

When a rule condition evaluates to true, the rule's actions are added to a pending actions list for possible execution. Rule actions typically change the state of the environment or the agent, and take as inputs resources or properties of resources bound in the rule condition. In addition, the action of a rule can invoke another TRP. This allows the agent designer to write generic programs for common sub-tasks which can then be used in multiple places. TRP invocations are done immediately, as if the rules in the invoked TRP were part of the calling TRP. TRPs that are called by other TRPs have in addition to their identifiers a list of arguments passed in by the top-level TRP that makes the call.

As an example, here is a very simple program which moves an agent to a location or object. This assumes the existence of a `move-to` action which starts the agent moving in a particular direction. The GOTO program takes an argument specifying the target location of the agent, and continuously checks the direction to that target.

```
(GOTO (?Target)
   (?AgentLocation SHARED ([VALUE *]) ())  AND
   equal(property(?AgentLocation, VALUE),property(?Target, VALUE))
   ⟹
   null

   (?AgentLocation SHARED ([VALUE *]) ())  AND
   NOT(equal(property(?AgentLocation, VALUE)property(?Target, VALUE)))
   ⟹
   propose(move-to, get-direction(?Target))
)
```

The calling program passes a resource that specifies the target location as an argument to the GOTO program from its own resource context when it makes the call. The first rule says that if the current location of the agent matches the location property of the `?Target` resource (extracted from the resource structure by the property function) then there is nothing left to do. The second rule says that if the agent is not at the target location, then get the direction of the target, and move the agent in that direction. A call to GOTO might look like:

```
   (?Enemy1 SHARED ([TYPE monster]) ())
   ⟹
   GOTO (?Enemy1)
```

This provides the bound resource variable `?Enemy1` as the argument to GOTO. Top-level teleo-reactive programs do not take arguments.

Disjunctions in GRUE TRPs

We mentioned in section 3.1.2 that the use of stable nodes for goal arbitration has ramifications for the use of disjunctions in rule conditions, because some disjunctive conditions may be stable in some circumstances and not in others. GRUE in general handles a disjunction just like any other condition; however there are some interesting properties of resource variables relating to disjunctions that are worth exploring.

Some disjunctions can be eliminated by the judicious use of resource variables. If an agent needs one of two items for a particular task, the developer can use a single resource variable that requires the property needed for the task. So, for instance, an attack program might use any resource of type weapon instead of specifying “a sword or a dagger or a battle axe”. This resource variable will match either of the two objects, implicitly encoding a disjunction. Be aware that this only works when there is an attack action that will work with any of the specified weapon types. If each weapon type used a different action, each would have to be matched by a different rule.

There is also a special circumstance where a disjunction of resource property values can be captured by a single resource variable. Assume we are writing a GRUE TRP for an agent in a computer game that involves weapons. The weapons are stored as resources of type gun, but each has a subtype listing the specific type of gun. Each weapon resource has only a single subtype. The TRP contains the following resource variable:

```
(?Gun116 EXCLUSIVE
 ([TYPE gun])
 ([SUBTYPE machine_gun] [SUBTYPE rocket_launcher))
```

Notice that two conflicting values of the subtype property are listed as preferred properties. No weapon will have both at once, so the resource variable will only ever match one or the other. This is not quite the same as using an exclusive or comparison function, as the resource variable will also match a weapon without either subtype value if the preferred type of weapon is not available. However, this technique can be used to effectively create a less-preferred subset of values when a property has a mutually exclusive group of possible values.

There is one restriction on the use of disjunctions in GRUE TRPs. A disjunction of resource variables cannot be used if either of the resource variables is accessed by a call to the property function or by an action performed by the rule. The problem is that the programmer does not know which of the two resource variables in the disjunction will actually be bound (and this cannot be determined until the rule actually fires). In this one circumstance, the programmer is forced to write a separate rule for each case.

3.2.5 Tasks

Tasks are created from goals by the program selector. The program selector replaces the name of the GRUE TRP specified in the goal structure with the text of the TRP itself. The resulting data structure is a task.

A *task* is a list consisting of an identifier (a string), a priority (an integer), a type specifier (`ACHIEVEMENT` or `MAINTENANCE`) and GRUE TRP. As an example, the following task corresponds to the `GetAwayFromGhosts` goal listed above:

```
(GetAwayFromGhosts 90 ACHIEVEMENT
 (EscapeFromGhosts
     NOT(?Ghost1 SHARED
             ([TYPE ghost] [DISTANCE #|1 <- 10|#])
             ())
     OR
     NOT(?Ghost1 SHARED
             ([TYPE ghost] [BLUE false])
             ())
     ⟹
     null

    (?Ghost1 SHARED
            ([TYPE ghost] [BLUE false] [DISTANCE #|1 <- 10|#])
            ())
    ⟹
    propose(move-away-from, ?Ghost1)
 )
)
```

Notice that the task identifier, priority, and type come from the goal while the remainder of the structure is a TRP.

A task is *runnable* if the condition of a rule in the task evaluates to true. Tasks whose programs have achieved the goal condition (and proposed the null action), or whose programs cannot run because the necessary resources are not available, are removed from the arbitrator.

3.2.6 Goal Arbitration

It is the job of the arbitrator to allocate resources to the tasks, run the rules, and resolve conflicts between the actions. The arbitration process allocates resources to a task by binding resource variables to available resources, then allows the task to propose an action. Resources are always assigned to the tasks in order of highest priority task to lowest priority task, so a lower priority task can never take resources away from a higher priority task. The list of proposed actions is examined for conflicts before the actions are executed, and conflicts are resolved in favor of the higher priority task.

Arbitration consists of five main steps:

1. Sort the tasks according to priority.

2. Starting with the highest priority task, consider the rules in textual order looking for one which has a condition that evaluates to true.

3. If no rule in the program can run, then remove the task from the arbitrator.
4. Repeat steps 2 and 3 until all tasks have been processed or the maximum number of runnable tasks is reached.
5. Execute a compatible subset of the actions proposed by the runnable rules.

A rule condition evaluates to true if its condition expression evaluates to true, which requires all of its resource variables to bind successfully (if they are not negated). The main criteria used for binding resource variables is that a lower priority task may never take resources from a higher priority task. We therefore allow the highest priority task to bind its resources first. If two tasks have the same priority, they will be ordered arbitrarily. This can cause problems (Section 5.1.2 contains an example of such a conflict), and the agent developer is advised to offset priorities wherever possible.

Variable binding in GRUE is a potentially destructive operation which may change the contents of the agent's memory. If a mutually exclusive resource variable matches a resource, the resource is effectively consumed and is not available to match resource variables in other programs, though the resource may match other (`EXCLUSIVE` or `SHARED`) resource variables in the same rule. However, in cases where a resource is divisible, the `EXCLUSIVE` flag gives the program containing the resource variable mutually exclusive access only to the portion of the resource that it actually binds. When the property matching a value range has the `DIV` keyword, the resource variable will bind the optimal amount according to the utility ordering in the range. If there is no utility ordering, the minimal amount is bound. Any remainder is treated as a separate resource and returned to memory for other resource variables to bind. Divisions are done based on the required properties, because it is possible for a required property to be satisfied only when the resource is divided. When a resource variable has a `SHARED` flag and the matching resource has the `DIV` keyword, the resource variable will bind the required amount and the remainder will be returned to memory for use by other resource variables as with an `EXCLUSIVE` resource variable. However in this case the portion bound to the `SHARED` resource variable also remains available for other resource variables to use.

Note that no attempt is made to maximize the number of tasks that can run: in particular, the preferred properties of a higher priority task may preempt the resources of a lower priority task, preventing the lower priority task from running. We do require that resource binding is consistent from cycle to cycle: if a resource variable bound during one execution cycle is bound again during the next cycle, it will bind to the same resource if the resource is still available. This is done based on the identifier of the resource variable, so if the same variable name is different rules, then all the rules will use the same resource if possible.

Conversely, no attempt is made to limit the number of tasks the agent will run in parallel. The arbitrator can be limited to processing only a small set of tasks to allow the architecture to execute more quickly. Once the task limit has been reached, the rest of the tasks are simply not run.

Tasks are removed from the arbitrator when the resources required to execute the task do not exist. In general, we would expect this to be a rare occurrence as the goal generators and/or program selector should prevent programs with unfulfilled prerequisites from entering the arbitrator. The exception is the case where a required resource is removed (by the world model or another TRP) while the program is running.

The actions of the first runnable rule in each runnable task are then collected in a proposed actions list. Any task with an `ACHIEVEMENT` type specifier whose top rule is runnable (i.e. which propose a null action indicating that the task has been completed) is removed from the arbitrator. A `MAINTENANCE` type specifier tells the arbitrator that the task should not be removed and should continue using resources even if the goal condition is true. The remaining actions are then checked for conflicts. For example, two tasks might propose moving in opposite directions. Such conflicts are application-dependent—in any given environment, some actions will conflict while others can be executed in parallel. The final stage of the GRUE arbitrator checks the list of proposed actions against a list of conflicting pairs of actions. If a pair of actions in the proposed actions list conflicts, the action proposed by the lower priority task is discarded. The remaining actions are then executed, changing the state of the environment and/or the agent, and the whole cycle starts over.

3.3 Correcting Deficiencies of the Architecture Design

We implemented two agents using the GRUE architecture, which are described in Chapter 4. During the evaluation of these agents, we discovered two flaws in the architecture, which we describe in detail in Section 6.1. We summarize them here as well so those reading this chapter independently of the rest will be aware of the issues.

The first issue has to do with negated resource variables. When we initially described the architecture, we assumed that negated resource variables could be used to match the case where a resource does not exist. However, what a negated resource actually means is that the resource is not available. When evaluating the implementation, it became clear that there are some cases where the distinction is important. We propose to solve the problem in future versions of the architecture by including a predicate *exists(x)* which takes a resource variable, and returns true if a matching resource exists, even if it is already bound. A return value of false would indicate that no such resource exists.

The second issue has to do with the persistence of bindings from cycle to cycle. While we stated this as one of the features supported by GRUE, it later occurred to use that there are cases where this feature is undesirable. We did include the option of disabling the feature for all bindings, but actually there might be some resources for which persistence makes sense and others for which it doesn't. We suggest adding a persistence flag to the resource variable structure. A resource variable would then be a 5-tuple containing an identifier, an exclusive access flag, a persistence flag, a set of required properties and a set of preferred properties. When binding a resource variable with a persistence value of false, the arbitrator would not store the binding for use in the following cycle.

3.4 Comparing GRUE to Similar Systems

In Chapter 2, we described several pieces of work that are similar to GRUE. These systems are Benson and Nilsson's teleo-reactive architecture, on which GRUE is based, Bryson's BOD (Behaviour Oriented Design), and Wright's MINDER1. In this section we consider the similarities and differences between the main components of these architectures. We focus on the plans and goal arbitration methods because these aspects are directly comparable. Chapter 2 provides a more complete description of the systems described by Benson and Nilsson, Bryson and Wright including those aspects that are more specialized and not directly relevant to GRUE. GRUE itself provides facilities for representing resources, including syntax for encoding preferences and knowledge about numerical quantities. We disregard these aspects of GRUE for the purposes of this comparison, as there are no corresponding facilities in the other architectures we are considering. (Facilities provided by one architecture, could of course be added to another. If all of the architectures are equal in expressive power, then it is even possible to create equivalent functionality using any of the architectures, though it may be more difficult in architectures that don't directly support that functionality.)

The first major similarity is the basic plans used by the systems. Both GRUE and Wright's system explicitly used Nilsson's teleo-reactive programs (TRPs). A TRP, we remind you, is a list of rules. Each rule, when executed, causes the condition of a rule higher in the list to become true. When the TRP is continuously evaluated, it achieves a goal. These teleo-reactive programs can be hierarchical; one TRP can activate another TRP as an action in a rule. Wright uses these as specified by Nilsson. We used Nilsson's definition, but replaced Nilsson's free variables with a new *resource variable*. Bryson, in contrast, describes a basic reactive plan which has a few differences from Nilsson's TRPs. Nilsson orders the rules in strict priority order, and evaluates them by simply moving down the list, checking each rule. Bryson tags each rule with a priority value, and allows rules to have the same priority value.

Benson and Nilsson's system includes a mechanism for reducing the amount of computation time spent checking the conditions in a large TRP or TRP hierarchy. Instead of evaluating all of the rules every cycle, they store the most recently executed rule r_i and the rule above it r_j. If r_j becomes true, it replaces r_i. Otherwise, they continue executing r_i. To retain the TRP's ability to take advantage of opportunities, Benson and Nilsson use a separate process to check the rules above r_i. This separate process runs more slowly, reducing the computational demands of the system.

Similarly, each basic reactive plan hierarchy in Bryson's system keeps track of the currently active element. Bryson allows the currently active element to be an entire basic reactive plan (although she specifies that basic reactive plans are expected to be no more than about 7 rules). Bryson does not use a separate process to check the rest of the hierarchy. The result is quite similar to Benson and Nilsson's system. By allowing the current element to be a small set of rules, Bryson's system retains some of its ability to take advantage of opportunities. The entire hierarchy is reconsidered when the active element terminates, or if the active element triggers an element that has not previously been triggered. Both Benson and Nilsson's system and Bryson's slip-stack reduce computation time while retaining the ability to react to opportunities, although Bryson's system gives the programmer less control over the frequency with which the hierarchy is reconsidered.

Neither Wright's system or our own GRUE incorporates this type of mechanism for reducing computation time. In retrospect, we should have included it in GRUE. It would be very easy to add — just include the current active element as part of the task. (We leave the choice between Bryson's mechanism and Benson and Nilsson's mechanism for future work.)

Benson and Nilsson's architecture arbitrates between goals using stable nodes. A stable node is a rule in a program whose condition is guaranteed to remain true if the agent switches to another goal. Stable nodes are safe points where the agent can pause a task without undoing the work it has already done. Their goal arbitration method chooses the goal that has the best reward/time ratio. This allows the agent to take time out from an important goal to work on a less important but much quicker goal. Bryson, in contrast, allows her agents to switch between goals at any point. In Bryson's system, an important goal can easily undo all the work done by a less important goal. Wright also allows his architecture to switch between goals at any point. All three systems only work on one goal during any individual execution cycle.

Our resource-based goal arbitration method is a significant difference between GRUE and the other three systems. Instead of working on the highest priority goal, we work on the set of active goals simultaneously if possible. We manage conflicts between goals in two ways: by managing the resources needed for each goal, and by eliminating conflicts between the actions output by the TRPs. In GRUE, resources are allocated to tasks in order of priority. So a TRP for a higher priority task is allowed to request resources before a TRP for a lower priority task. Resources are requested in terms of properties that are necessary to execute an action, and properties that are desired but not necessary. This system does not always maximize the number of goals that can execute successfully, but we feel it is a reasonable solution. It has the critical property that higher priority goals are never displaced in favor of lower priority goals while avoiding the computational difficulty of assigning resources to maximize the number of goals that can run.

Priorities in our system are set using goal generators which are written by the programmer. If desired, the priority can be generated using a reward/time ratio as in Benson and Nilsson's system. We feel that this is easier than attempting to manipulate reward/time ratios to achieve a desired result. Wright's system, like GRUE, includes a component to generate goals without placing any restrictions on how the priorities are set. Bryson uses a combination of priority and frequency values to control goal arbitration. By setting some goals to run at a certain maximum frequency, Bryson can force these goals to relinquish processor cycles to lower priority goals.

Of the four architectures described here, only Benson and Nilsson's system uses stable nodes. The other three systems all allow high priority tasks to undo progress made toward other goals. The question of which method is better depends on the type of goal encountered by the system. When Benson and Nilsson's system receives an urgent goal, it takes the time to reach a stopping point in its other tasks before working on the new goal. While this may be ok for a delivery robot (the domain used as an example in (Benson and Nilsson, 1995), an agent in a fast-paced game who fails to consider an urgent problem right away can suffer drastic (or even fatal) consequences. (We are also a little worried that Benson and Nilsson's robot might "forget" to recharge its batteries if given a long task with no stable nodes.) If an agent often encounters extremely urgent tasks that have severe consequences when left undone, stable node arbitration is not appropriate. If an agent never suffers severe negative consequences for neglecting a goal, then stable nodes are a good way of preventing the agent from doing extra work.

CHAPTER 4

Implementation

Our prototype implementation of GRUE uses the Sim_Agent toolkit that is described in the next section. In addition we have implemented two complete agents using GRUE. The first agent operates in Tileworld, a simple AI test bed. The second agent operates in Unreal Tournament (Infogrames, PC 1999), a commercial game environment. These two environments have different properties, which allows us to test the performance of GRUE under a wider variety of conditions than we would be able to using a single environment. The Tileworld environment is quite simple, but very configurable. We choose the configuration of Tileworld to allow us to test the use of preferred properties, the use of divisible resources, and the use of value ranges in both preferred and required properties. Unreal Tournament (UT), in contrast, has few opportunities for the use of divisible resources or value ranges but supports more complex behaviour. Our Tileworld agent has only 3 goals, with a maximum of 2 executing concurrently. In contrast, the UT agent has 11 different goals, and up to 8 of them can be in the arbitrator simultaneously. The use of the Unreal Tournament environment allows us to more rigorously test our goal processing features, which are a central part of GRUE. The use of two different environments also allows us to demonstrate that the GRUE components can support the development of agents for widely divergent types of environments, while the use of a commercial game allows us to verify that GRUE could be used in such an application.

4.1 Sim_Agent

Sim_Agent (Sloman and Logan, 1999) is an open source toolkit for developing agents and multi-agent systems. It is based on the Pop11 language, and includes libraries for easy development of graphics as well as agents and objects. The behaviour of agents in Sim_Agent is specified using rulesets. A Sim_Agent ruleset is a list of condition-action rules. Rulesets are highly configurable: they may execute all rules that match the agent's working memory, or they may be set to only run the first rule that matches. The programmer can control how often the ruleset runs, and how many passes are made through the ruleset when it does run.

An agent in Sim_Agent is a class with an associated rulesystem. A rulesystem consists of one or more rulesets, which are grouped together. The agent can be configured to run its rulesets at different relative speeds if necessary. Sim_Agent also includes a grouping called a rulefamily. A rulefamily contains several rulesets, where at most one ruleset can be active at a time. Each ruleset in the rulefamily can switch to another ruleset within the rulefamily, whereupon the new ruleset gains control of the rulefamily until it, in turn, switches to a different ruleset. Rulefamilies can be included in the agent's rulesystem just like rulesets.

It is extremely easy to develop teleo-reactive programs in Sim_Agent. Each program is a ruleset, defined such that only one rule executes per cycle. Following the definition of a teleo-reactive program, the first rule checks for the success condition and the last rule is the most general. Note that Sim_Agent agents must have at least one ruleset, but that that ruleset does not have to be a teleo-reactive program.

4.2 Implementation of GRUE Components

Both the basic components of the GRUE system and the teleo-reactive programs are implemented as Sim_Agent rulesets. The first rulesets to run are those forming the world interface, followed by the goal generators, then the filter, the program selector and the arbitrator. Each TRP is a ruleset that is started by the arbitrator, and returns control to the arbitrator when finished (this is described in more detail below). There is an additional ruleset at the end to undo resource variable bindings and prepare for the next cycle through the system.

We have implemented GRUE agents for two different environments, Tileworld and Unreal Tournament. Each interface requires several rulesets to transform data from the environment into resources. The interfaces are described further later in this chapter.

An agent in Sim_Agent has a database where it can store anything it likes. We use the database to represent the agent's memory. In memory we store goals, the list of tasks, the list of available resources, and the list of bound resources, among other things.

The goal generators are implemented as separate rulesets, one for each teleo-reactive program. Goal generators may contain several generation rules if the goal should be generated in several situations. If the goal is to be updated once it is in the arbitrator, the goal generator has rules to do so. The filter is a simple threshold filter, consisting of a single rule that discards any goal with a priority lower than the threshold. (We allow the filter threshold to be set using a global variable.) The program selector is similarly simple, containing a rule that simply matches goal items in the agent's database, creates the corresponding task structure and deletes the original item.

The arbitrator actually consists of two rulesets and a rulefamily. The first ruleset sorts the tasks in order of priority. A task, as previously described, contains the TRP as well as the priority. The rulefamily runs next. It contains an arbitrator ruleset to control the running TRPs, and each of the TRP rulesets. Each TRP is represented as a Sim_Agent ruleset, where only one rule can run during each cycle. The arbitrator ruleset transfers control to the TRP for the highest priority task in the task list. Each rule in the TRP ruleset attempts to fire, binding resources as necessary and putting actions on a list to be run at the end of the cycle. Once the program has finished running, it transfers control back to the arbitrator which calls the next TRP in the sorted task list. Each rule in the TRP transfers control back to the arbitrator after proposing its actions. TRPs whose rules do not handle all situations are required to include an extra rule to transfer control back to the arbitrator when none of the other rules match. Once all the tasks have been run, the second arbitrator ruleset runs the actions, resolving conflicts by eliminating the lower priority conflicting action.

The binding of resource variables is implemented by a procedure called within the conditions in the TRPs. The procedure returns a binding, which is a pair containing a resource variable and a resource. The binding procedure returns the best binding it can find (as described in Chapter 3), or the empty list if no binding is possible. Rules which need exclusive access to resources call another procedure from within the action part of the rule, which removes the binding from the list of available resources so that it can't be bound to any other resource variables. Procedure calls are done right away; this is different from actions that are proposed by the task and which are run by the arbitrator at the end of the cycle.

4.2.1 Binding Algorithm

In this section, we describe the binding algorithm. Figure 4.1 shows pseudocode for the binding algorithm.

The binding algorithm takes a resource variable and binds it to a resource if an appropriate resource is available. Bindings are normally required to be consistent from cycle to cycle, although it is possible to disable this feature for all bindings. The algorithm therefore begins by checking a list of bindings used during the previous cycle. If the resource variable is in the list, it is bound to the same resource assuming that the resource is available and still matches all of the required properties. Since it is possible for the property values of a resource to change (because they are recomputed by the interface, for example) the resource must be obtained from the current list of available resources instead of simply being obtained from the old binding. If the resource has a divisible property and the resource variable specifies an amount for that property, then it is divided appropriately. The algorithm then returns a binding consisting of the variable and the current, divided version of the resource.

```
If resource variable was bound during last cycle, and resource is still available
{
        take binding from list of previous bindings
        get current version of resource from available list
        make sure the resource still matches the required properties
        if resource is divisible
        {
            divide it if necessary
        }
        construct the binding using the updated resource and return it
}
else
{
        for each available resource
        {
                if all required properties match
                {
                        if resource is divisible
                        {
                            divide it if necessary
                        }
                        put resource on list of possible matches
                }
        }
        for each possible match
        {
                compute its score as the number of non-range preferred properties that match
        }
        for each preferred property containing a range
        {
                check all possible matches and find the one that matches best,
                adding 1 to the score for that match
        }
        if a match was found, construct a binding from the match with the highest score and return it
        else return false
}
```

FIGURE 4.1: Pseudocode for the Binding Algorithm

If the resource variable was not bound during the previous cycle, the previously bound resource is no longer available, or the previously bound resource no longer matches the required properties, then the list of available resources is searched for resources with all of the required properties. If the resource is divisible, it is possible that it only matches a required property once divided so divisions are done when matching the required properties. The amount bound will be chosen according to the utility ordering or the minimum necessary if there is no utility arrow.

If there is more than one match, the resources must be checked to see whether they match the preferred properties. Preferred properties with range values are treated differently from non-range preferred properties, so it is easiest to keep a list of possible matches and check the range and non-range properties separately. Checking the non-range properties are straightforward; each resource receives a number of points equal to the number of matching preferred properties. The range properties are slightly more complicated, as only the resource that is the best match for the range according to the utility ordering receives the point.

The best match is now the resource with the highest preferred properties score. Assuming there is one, it is returned along with the resource variable. If there is more than one best match, one is chosen arbitrarily.

4.2.2 Computational Complexity of GRUE

The computational complexity of GRUE can be analyzed in terms of a number of different factors. In this section we will only consider the algorithms that comprise the GRUE implementation, and will only consider them in terms of data structures used in GRUE. So, we consider the number of goals, tasks, resources etc. in the agent's memory but we will ignore for instance the efficiency of the Sim_Agent rule interpreter, which is independent of GRUE. (See Section 6.5 for why GRUE does not require a rule interpreter.) We will also ignore application-dependent aspects of the agent such as the world interface and the goal generators.

The filter operates on each goal and each task in memory. This operation is linear in the total number of goals and tasks. The program selector likewise operates on each goal in memory and is linear in the number of goals. The two combined are $O(2t)$ where t is the maximum number of goals and tasks.

The arbitrator starts by sorting the task list. This will be as efficient as the sorting algorithm used.

The running of TRPs is bounded by $tnb * O(bind())$, where t is the maximum number of tasks in the task list, n is the number of rules in the TRP and b is the number of calls to the binding algorithm in each rule.

The complexity of the binding algorithm is $3rdl + dl + 2p + 4r$ where r is the number of resources in the agent's memory, d is the number of properties in the resource variable, l is the number of properties in the resource, and p is the number of bindings from the previous cycle. We have simplified this somewhat; r is used as an upper bound for the number of available resources as well as the number of bound resources, and d is an upper bound for the number of properties in both the required and preferred properties lists. We could simplify it even further if we realize that p is also bounded by the total number of resources, so we could rewrite the formula as $3rdl + dl + 6r$.

There is an additional factor if a rule runs successfully. If the rule includes exclusive bindings, those resources must be removed from the list of available resources list, which again will depend on the data structure used. In our implementation it is linear in the number of resources, so it works out to er where e is the number of exclusive bindings in each rule and r is the number of resources.

The arbitrator also runs the actions proposed by the rules; this is $O(a)$ where a is the number of actions in the list. Theoretically, a rule can execute any number of actions, but the length of the list is more likely to be a small constant multiplied by the number of tasks in the task list.

Combining these gives a total complexity of $O(2t(tnb*(3rdl+dl+6r))+er+at)+O(sort)$ where t is the maximum number of tasks that can be running simultaneously (this is also the maximum number of goals that can be proposed), n is the maximum number of rules in a teleo-reactive program, b is the maximum number of resource variables bound in a rule, d is the maximum number of properties in a resource variable (in both lists), l is the maximum number of properties in a resource, r is the maximum number of resources in the agent's memory (both available and unavailable resources), and a is the maximum number of actions proposed by a task during one cycle. $O(sort)$ is the complexity of the sort algorithm called by the arbitrator. This complexity is for the filter and the arbitrator with the teleo-reactive programs, but does not include the complexity of the goal generators, the interface, or the actions.

Note that this complexity is based on our actual implementation (see Appendix D for the code). It may be possible to reduce it by using more efficient data structures.

4.3 The GRUE Tileworld Agent

We have used GRUE to develop an agent that operates in Tileworld (Pollack and Ringuette, 1990), a grid-based world where agents must collect tiles and deposit them in holes, avoiding obstacles while moving around the grid. The agent can only move horizontally and vertically, not diagonally and the environment is dynamic in that tiles and holes appear and disappear.

Several Tileworld features are particularly useful for testing certain GRUE features. First, tiles and holes have shapes in our implementation of Tileworld. The agent receives three points for dropping a tile in a hole when the tile and the hole are the same shape, but only one point if the shapes do not match. Secondly, tiles are found in stacks, and holes have depth. The agent can only carry one tile stack at a time, but it may drop any number of tiles from that stack at one time. If a hole is partially filled, the agent receives a score as normal. If the agent fills a hole, it receives an additional twenty points. These features of the Tileworld environment give us the opportunity to test the use of preferred properties, divisible resources, and numerical ranges in a GRUE agent.

Our Tileworld is very configurable. It is possible to vary the number of objects created in the environment to start with, as well as the rate at which objects vanish and are created. It is also possible to change the size of the Tileworld, the agent's sensor range, the size of tile stacks and the depth of holes. The values we used for our tests are given in Section 5.1 along with the results.

4.3.1 Tileworld Interface

The Sim_Agent sensor system provides sensory information for each item within the agent's sensor range. The first interface ruleset removes objects that have vanished from the list of available resources (i.e. the object ought to be within sensor range but the agent has no sensory information for that object). Additionally separate rulesets exist for each type of object in the Tileworld environment; these rulesets translate the sensory information for new objects into resource structures. Sensory information includes the distance between the agent and the object, and the interface obtains additional information such as shape and age of the object from the Tileworld. If an existing object is sensed, the GRUE interface updates the resource structure to reflect changes in the distance and age of the object. Finally, an additional ruleset calls a procedure to compute the nearest object of each type. A property of the form [NEAREST true] is added to the appropriate resource structure.

An agent in Tileworld can perform only a few actions. They are: move left, right, up or down, pick up a tile stack (on the same square as the agent), and drop one or more tiles into a hole (on the same square as the agent). If the agent attempts to move onto a square containing an obstacle, the action will fail.

4.3.2 Tileworld Goal Generators and TRPs

The Tileworld agent has three TRPs. One causes the agent to acquire a tile stack (*get tile*), one causes the agent to fill a hole (*fill hole*), and one causes the agent to move around obstacles (*avoid obstacle*). When developing the agent, our goal was to implement several interacting behaviours, rather than to create an optimal Tileworld agent. Therefore, our

TRPs create resources to store information which can then be used by the other TRPs. Our obstacle avoidance TRP is designed to take control when one of the other programs is stuck, and uses resources created by the other programs. The actual obstacle avoidance strategy, however, is rather simplistic.

Each behaviour is implemented as a Sim_Agent ruleset, and is executed by GRUE. In the remainder of this section we will first describe the goal generators, then describe each TRP in turn. The TRPs are described in detail here so that it is clear to the reader exactly what each rule does and why it is needed.

Appendix B lists the different types of resources used in Tileworld and shows the properties for each resource type. These resources are used by the goal generators and TRPs described in this section.

4.3.3 Goal Generators

The *get tile* goal is generated when a tile stack is visible, with a priority of 100 divided by the distance to the tile stack. It is also generated when no tile stack is visible, but with a priority of 10. If the *get tile* goal has already been converted into a task and entered the arbitrator, the goal generator updates the priority to 100 divided by the distance to the tile stack (assuming there is one).

The *fill hole* goal is only generated when the agent is holding a tile stack. Otherwise, it is similar to the goal generator for *get tile*, with hole substituted for tile stack.

The *avoid obstacle* goal is only generated if an obstacle is visible, and the priority is 100 divided by the distance to the obstacle. Like the other goal generators, it updates the priority of tasks in the arbitrator according to the current distance to the nearest obstacle.

Each goal is associated with a top-level GRUE TRP which is responsible for choosing the actual object using a resource variable. The goal generator does not have any control over which object will actually be used, and only generates one instance of each goal (no matter how many appropriate objects exist). [1]

4.3.4 Get Tile

The *get tile* TRP chooses, moves toward, and picks up a tile stack. It is considered finished when the agent is holding a tile stack. If the agent cannot see any tiles, the program chooses a direction randomly, stores the direction as a resource in the list of available resources, and moves in that direction. The agent continues to move in this random direction until it can see a tile or until it runs into an obstacle. In either case, it removes the random direction allowing it to compute a new direction the following cycle.

When one or more tile stacks is visible, the agent chooses the best available tile stack. It prefers tile stacks that are nearby, large, and expected to persist for some time. It also computes the direction toward the tile stack and stores the direction as a resource in the database. The agent can only move horizontally or vertically, and it always chooses to move horizontally first. The next cycle, the agent moves in the stored direction if the tile stack is still in that direction. If the tile stack vanishes, the agent removes the stored direction so it can be recomputed the next cycle. This group of rules continues to move the agent toward the tile stack, recomputing the direction when necessary, until the agent is on

[1] Our implementation of the goal generators is such that the priority of a goal does not necessarily correspond to the object being used by the associated TRP. If the density of the Tileworld is low, it is likely that only one object of the appropriate type will be visible, so the priority will be correct. If the density of the Tileworld is high, it will often happen that the priority of a goal will be based on the distance to an object that is not actually being used. This is an implementation decision that was made in order to simply the goal generators, and does not have adverse effects on performance due to the small number of goals in Tileworld.

a tile stack [2].

4.3.5 Get Tile Pseudocode

The first rule simply checks whether the agent is carrying a tile stack.

```
greater_than(number_tile_stacks_carried_by_agent(), 0)
⟹
null
```

The second rule checks whether the agent is on the same square as a tile stack, and if so, picks it up. It removes the stored target and direction, but does not check that the target tile is the one being picked up. This allows the agent to opportunistically pick up tiles. The direction and target are exclusive bindings as they are being removed.

```
(?tgt EXCLUSIVE ([TYPE target]) ()) AND
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ())  AND
(?direction EXCLUSIVE ([TYPE direction] [TOWARD tile_stack]) ()) AND
⟹
propose(pick_up, ?tiles)
propose(remove, ?direction)
propose(remove, ?tgt)
```

The third rule moves the agent toward a target tile stack assuming one exists. The agent always moves horizontally first, then vertically. This means that it may need to recompute the direction once it is lined up with the tile stack, so it needs to check that the direction is still correct.

```
(?tgt SHARED ([TYPE target]) ())  AND
(?tiles SHARED ([ID property(''VALUE'', ?tgt)] [TYPE tile_stack]) ()) AND
(?direction SHARED ([TYPE direction] [TOWARD tile_stack]) ()) AND
equal(property(''VALUE'', ?direction), compute_direction_to_target(?tgt))
⟹
propose(move_in_direction, compute_direction_to_target(?tgt))
```

[2]If the agent attempts to move onto an obstacle, the move action will fail. The *get tile* TRP does not check for obstacles, however.

If the target tile has vanished, then remove the target resource. This rule matches in the case when the direction has not yet been computed.

```
(?tgt EXCLUSIVE ([TYPE target]) ()) AND
NOT(?tiles SHARED ([ID property(''VALUE'', ?tgt)] [TYPE tile_stack]) ()) AND
NOT(?direction SHARED ([TYPE direction] [TOWARD tile_stack]) ())
⟹
propose(remove, ?tgt)
```

If none of the above rules matched, but there is a valid target, then the direction needs to be recomputed. This rule removes the stored direction, allowing the agent to recompute the direction during the next cycle. The direction is an exclusive binding because it is being removed.

```
(?tgt SHARED ([TYPE target]) ()) AND
(?tiles SHARED ([ID property(''VALUE'', ?tgt)] [TYPE tile_stack]) ()) AND
(?direction EXCLUSIVE ([TYPE direction] [TOWARD tile_stack]) ())
⟹
propose(remove, ?direction)
```

If there is a stored target and direction but no tile stack, then the target tile has disappeared. Remove the stored target and direction so the information can be recomputed next cycle. The bindings are exclusive because the resources are being removed.

```
(?tgt EXCLUSIVE ([TYPE target]) ())  AND
(?direction EXCLUSIVE ([TYPE direction] [TOWARD tile_stack]) ())
⟹
propose(remove, ?direction)
propose(remove, ?tgt)
```

If there is a target but no direction, then the direction needs to be recomputed. (We don't need to specifically check for the direction here; if there was a direction one of the above rules would have matched.)

```
(?tgt SHARED ([TYPE target]) ()) AND
(?tiles SHARED ([ID property(''VALUE'', ?tgt)] [TYPE tile_stack]) ())
⟹
propose(add_resource, compute_direction_to_target(?tiles))
propose(move_in_direction, compute_direction_to_target(?tiles))
```

If the agent has stored a random direction but has come in range of a tile stack, then remove the random direction and store a new direction along with the target tile stack.

```
(?tiles SHARED ([TYPE tile_stack]) ([DISTANCE #| 1 <-  :#]
                                    [SIZE #| 1 -> :#]
                                    [AGE young])) AND
(?random EXCLUSIVE ([TYPE direction] [TOWARD random]) ())
⟹
propose(remove, ?random)
propose(add_resource, compute_direction_to_target(?tiles))
propose(add_resource, make_target(?tiles))
propose(move_in_direction, compute_direction_to_target(?tiles))
```

This rule stores a target tile stack and a direction when the agent does not have a stored random direction.

```
(?tiles SHARED ([TYPE tile_stack]) ([DISTANCE #| 1 <-  :#]
                                    [SIZE #| 1 -> :#]
                                    [AGE young]))
⟹
propose(add_resource, compute_direction_to_target(?tiles))
propose(add_resource, make_target(?tiles))
propose(move_in_direction, compute_direction_to_target(?tiles))
```

If the agent has been moving in a random direction and has encountered an obstacle, then remove the random direction allowing it to be recomputed. The random direction is bound exclusively because it is being removed.

```
(?random EXCLUSIVE ([TYPE direction] [TOWARD random]) ())  AND
(?obstacle  SHARED ([TYPE obstacle]
                    [DISTANCE 1]
                    [DIRECTION property(''VALUE'', ?random)]) ())
⟹
propose(remove, ?random)
```

If there is a stored random direction then move in that direction.

```
(?random SHARED ([TYPE direction] [TOWARD random]) ())
⟹
propose(move_in_direction, ?random)
```

And finally, if none of the above rules matched, then pick a random direction and start moving in that direction.

$\Longrightarrow$
```
propose(move_in_direction, generate_random_direction())
propose(add_resource, generate_random_direction())
```

4.3.6 Fill Hole

The *fill hole* program chooses, moves toward, and drops tiles in a hole. It is considered finished when the agent is no longer holding a tile stack. This means if the agent drops part of a tile stack, the program can continue to run. Also, if the tile stack that the agent is holding vanishes before being dropped, the program stops running. If the agent cannot see a hole, the agent starts by choosing a random direction and storing it in the list of available resources. On the next cycle, it moves in the stored direction. It continues this motion until it either sees a hole or runs into an obstacle. In both cases it removes the stored random direction, so the direction can be recomputed.

When the agent can see one or more holes, it chooses the best one. The agent prefers holes that are likely to persist for some time, that are nearby, that are the same shape as the tile stack the agent is holding, and are small (thus being easier to fill). The agent computes the direction to the hole (horizontal directions are always chosen first) and stores it as a resource in the list of available resources. At the next cycle, the agent moves in the stored direction. If the target hole disappears, or if the stored direction is incorrect (because the agent is lined up with the hole), the direction is removed so it can be recomputed during the next cycle. This group of rules continues to move the agent toward the hole, recomputing the direction when necessary until the agent is on the same square as a hole.

Once the agent reaches the hole, it drops a number of tiles which is as large as possible but not greater than the size of the hole. If this uses the entire tile stack, then the program is considered finished. Otherwise, it will continue to run, choosing a new target hole.

4.3.7 Fill Hole Pseudocode

We omit the descriptions for most of the rules here, as this TRP is nearly the same as the TRP for *Get Tile.*

```
less_than(number_tile_stacks_carried_by_agent(), 1)
⟹
null
```

If the agent is on the same square as a hole, it drops as many tiles as possible without wasting any. If the tile stack is bigger than the hole, the hole will be filled and the agent will keep some tiles. Otherwise, the agent will drop all of its tiles and the program will finish. This rule also removes the stored target and direction information, but does not check that the hole it is on is the same as the stored target hole to allow the agent to drop tiles opportunistically. The hole and tile stack are bound exclusively because their sizes must be adjusted when the tiles are dropped. The target and direction will be removed regardless so must be bound exclusively.

```
(?tgt EXCLUSIVE ([TYPE target]) ()) AND
(?h EXCLUSIVE ([TYPE hole] [DISTANCE 0]) ())  AND
(?tiles EXCLUSIVE ([TYPE tile] [DISTANCE 0] [Size #| 1  property(''SIZE'', ?h) |#])
                  ([SIZE #| 1 -> property(''SIZE'', ?h) |#])) AND
(?direction EXCLUSIVE ([TYPE direction] [Toward hole]) ())
⟹
propose(drop, ?tiles)
propose(remove, ?direction)
propose(remove, ?tgt)
```

```
(?tgt SHARED ([TYPE target]) ())  AND
(?h SHARED ([ID property(''VALUE'', ?tgt)] [TYPE hole]) ()) AND
(?tiles SHARED ([TYPE tile_stack] [Distance 0]) ()) AND
(?direction SHARED ([TYPE direction] [Toward hole]) ()) AND
equal(property(''VALUE'', ?direction), compute_direction_to_target(?tgt))
⟹
propose(move_in_direction, property(''VALUE'', ?direction))

(?tgt EXCLUSIVE ([TYPE target]) ()) AND
NOT(?h SHARED ([ID property(''VALUE'', ?tgt)] [TYPE hole]) ()) AND
(?tiles SHARED ([TYPE tile_stack] [Distance 0]) ()) and
NOT(?direction SHARED ([TYPE direction] [Toward hole]) ())
⟹
propose(remove, ?tgt)

(?tgt SHARED ([TYPE target]) ()) AND
(?h SHARED ([ID property(''VALUE'', ?tgt)] [TYPE hole]) ()) AND
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ()) and
(?direction EXCLUSIVE ([TYPE direction] [TOWARD hole]) ())
⟹
propose(remove, ?direction)

(?tgt EXCLUSIVE ([TYPE target]) ()) AND
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ()) AND
(?direction EXCLUSIVE ([TYPE direction] [TOWARD hole]) ())
⟹
propose(remove, ?tgt)
propose(remove, ?direction)

(?tgt SHARED ([TYPE target]) ()) AND
(?h SHARED ([ID property(''VALUE'', ?tgt)] [TYPE hole]) ()) AND
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ())
⟹
propose(add_resource, compute_direction_to_target(?tile_stack))
propose(move_in_direction, compute_direction_to_target(?tile_stack))

(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ()) AND
(?h SHARED ([TYPE hole]) ([SHAPE property(''SHAPE'', ?tiles)]
             [AGE young]
  [DISTANCE #| 1 <- :#]
  [SIZE #| 1 <- :#])) AND
(?random EXCLUSIVE ([TYPE direction] [TOWARD random]) ())
⟹
propose(remove, ?random)
propose(add_resource, compute_direction_to_target(?h))
propose(add_resource, make_target(?h))
propose(move_in_direction, compute_direction_to_target(?h))

(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ()) AND
(?h SHARED ([TYPE hole]) ([SHAPE property(''SHAPE'', ?tiles)]
                          [AGE young]
```

```
                           [DISTANCE #| 1 <- :#]
                           [SIZE #| 1 <- :#])) AND
⟹
propose(add_resource, compute_direction_to_target(?h))
propose(add_resource, make_target(?h))
propose(move_in_direction compute_direction_to_target(?h))
```

```
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ()) AND
(?random EXCLUSIVE ([TYPE direction] [TOWARD random]) ()) AND
(?obstacle SHARED ([TYPE obstacle]
                   [DISTANCE 1]
                   [DIRECTION property(''VALUE'', ?random)]) ())
⟹
propose(remove, ?random)
```

If there is a stored random direction, the agent moves in that direction.

```
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ()) AND
(?random SHARED ([TYPE direction] [TOWARD random]) ())
⟹
propose(move_in_direction, ?random)
```

```
(?tiles SHARED ([TYPE tile_stack] [DISTANCE 0]) ())
⟹
propose(add_resource, generate_random_direction())
propose(move_in_direction, generate_random_direction())
```

4.3.8 Avoid Obstacle

The *avoid obstacle* program runs when the agent is next to an obstacle. It only runs if one of the other programs has stored a direction. If there is no stored direction, or if there is a stored direction but there is no obstacle next to the agent in that direction, then the program is considered finished and stops running. While *avoid obstacle* does not have sole control of the agent, the associated task is the highest priority task when there is an obstacle next to the agent. This means that when there is an obstacle in the direction of travel, *avoid obstacle* will override the actions requested by the other programs.

If there is an obstacle in the direction of travel, the avoid obstacle program computes a perpendicular direction and stores it. On the next cycle, it moves in the stored perpendicular direction. If there is an obstacle blocking the agent in the perpendicular direction, then the program reverses the stored perpendicular direction.

The program keeps a record of the original direction of travel. If at any point there is no longer a direction resource matching the original direction, the program removes the stored perpendicular direction so it can be recomputed. The direction of travel might change in the case that the target object has vanished and a new target has been chosen, or in the case that the agent is lined up with the target and needs to start moving vertically rather than horizontally.

4.3.9 Avoid Obstacle Pseudocode

If is no obstacle in the direction of travel, and there is no stored perpendicular direction, then the program is finished running. The direction resource variable includes preferences which ensure that productive directions are chosen before random directions.

```
NOT(?perpendicular_direction SHARED ([TYPE direction]
                                     [TOWARD around_obstacle]) ()) AND
(?direction SHARED ([TYPE direction]) ([TOWARD hole]
                                       [TOWARD tile_stack])) AND
NOT(?obstacle SHARED ([TYPE obstacle]
      [DISTANCE 1]
                      [DIRECTION property(''VALUE'', ?direction)]) ())
⟹
null
```

If there is no longer a direction resource (because it was deleted by another program), remove the perpendicular direction and stored direction resources.

```
(?perpendicular_direction EXCLUSIVE ([TYPE direction]
                                     [TOWARD around_obstacle]) ()) AND
(?stored_direction EXCLUSIVE ([TYPE stored_direction]) ()) AND
NOT(?direction SHARED ([TYPE direction]
                       [DIRECTION property(''VALUE'', ?stored_direction)]) ())
⟹
```

```
propose(remove, ?perpendicular_direction)
propose(remove, ?stored_direction)
```

If there is no longer an obstacle in the direction of movement, then remove the perpendicular direction and stored direction resources. Also move in the stored direction, which will put the agent next to the obstacle.

```
(?perpendicular_direction EXCLUSIVE ([TYPE direction]
                                     [TOWARD around_obstacle]) ()) AND
(?stored_direction EXCLUSIVE ([TYPE stored_direction]) ()) AND
NOT(?obstacle SHARED ([TYPE obstacle]
                      [DISTANCE 1]
                      [DIRECTION property(''VALUE'', ?stored_direction)]) ())
⟹
propose(remove, ?perpendicular_direction)
propose(remove, ?stored_direction)
propose(move_in_direction property(''VALUE'', ?stored_direction))
```

If the agent is moving in the perpendicular direction and encountered an obstacle, reverse direction.

```
(?perpendicular_direction EXCLUSIVE ([TYPE direction]
                                     [TOWARD around_obstacle]) ()) AND
(?obstacle SHARED ([TYPE obstacle]
   [DISTANCE 1]
   [DIRECTION property(''VALUE'', ?perpendicular_direction)]) ())
⟹
propose(remove, ?perpendicular_direction)
propose(add_resource, reverse_direction(?perpendicular_direction))
```

This rule causes the agent to move in a previously computed direction perpendicular direction.

```
(?perpendicular_direction SHARED ([TYPE direction]
                                  [TOWARD around_obstacle]) ()) AND
(?stored_direction SHARED ([TYPE stored_direction]) ()) AND
(?obstacle SHARED ([TYPE obstacle]
   [DISTANCE 1]
   [DIRECTION property(''VALUE'', ?stored_direction)]) ())
⟹
propose(move_in_direction, property(''VALUE'', ?perpendicular_direction))
```

If there is an obstacle in the direction of movement and none of the above rules matched, then compute a perpendicular direction. The agent chooses randomly between the two possible perpendiculars.

```
(?direction SHARED ([TYPE direction]) ()) AND
(?obstacle  SHARED ([TYPE obstacle]
    [DISTANCE 1]
                     [DIRECTION property(''VALUE'', ?direction)]) ())
⟹
propose(add_resource, change_type(?direction, ''stored_direction''))
propose(add_resource, compute_perpendicular(?direction))
```

4.4 The GRUE Unreal Tournament Agent

Bots aren't just cool. They're trouble.

Andrew Leonard, in Bots: The Origin of New Species

To evaluate GRUE's performance in a commercial game environment, we have developed a GRUE agent for the game Unreal Tournament, which we will abbreviate as UT. UT is a type of game called a first-person shooter. The basic gameplay allows a number of human players and/or AI players (called bots) to interact in one of a number of different environments. Each environment is stocked with weapons, ammunition, armor, and health packs (items which restore health to the player who picks them up). If a player is killed, that player restarts from one of several designated restart points on the map. Several types of gameplay are supported. The simplest is a death match, where the players win points for killing other players and lose points for being killed. The player with the highest number of points wins. Alternatively, there are team games. In a domination game each team tries to keep control of several points on the map. In a capture the flag game, each team has a base with a flag. Players score points for the team by capturing the opposing team's flag returning with it to their own base position. Players can prevent the opposing team from scoring points by preventing them from obtaining the flag, or by killing the player who is carrying the flag.

A number of AI researchers are using UT as a research platform with the help of the *Gamebots* (Kaminka et al., 2002) interface, e.g. (Vu et al., 2003; Munoz-Avila and Fischer, 2004). *Gamebots* allows a bot to connect to the game through a standard sockets interface. The *Gamebots* API provides a set of standard messages that the bot will receive from the server, and a set of commands that the bot can send to the server. Using *Gamebots* allows bots to be developed in any computer programming language. We have developed a GRUE agent for UT using Sim_Agent. We used the standard GRUE components as described above. The new portions of the implementation include the interface which processes *Gamebots* messages, the goal generators, and the TRPs.

Our bot plays capture the flag on one of the simpler levels. This means that it must be able to obtain a weapon, pick up ammunition if it runs out, attack players on the opposing team, navigate to the other team's flag base, pick up the flag, and navigate back to its own flag base. It must also pick up health packs and/or run away when under attack, and pick up its own flag if it is dropped by an enemy player. (Picking up one's own flag automatically returns it to the flag base.)

4.4.1 Unreal Tournament Interface

The interface component of the agent processes *Gamebots* messages and translates them into resources to be used by the TRPs. Types of resources encountered in an Unreal Tournament capture the flag game include players, flags, weapons, ammunition, armor, health packs, and path nodes. Path nodes are invisible objects placed around the level to help bots navigate.

We have chosen to use a simple capture the flag level, which contains no armor. It also has no hazards such as lakes (in which players can drown). It does have a selection of weapons and ammunition for those weapons, and several corridors to be navigated. We are not working on the problem of path finding, so we chose to simplify the problem by requesting paths from the UT server. We store the returned paths as resources, and ignore the path node objects encountered in the level. The rest of the objects encountered are stored as resources. Player resources have team, location, rotation, velocity, and weapon properties. Health packs just have location, while weapons and ammunition specify the weapon type as well as their location. The resources used in UT are listed in Appendix C.

Items such as weapons and health packs are picked up by simply walking into them. The item will reappear after a short period of time (about 30 seconds). Our interface removes items from the list of available resources if they are not visible but ought to be within the agent's field of view, so the bot will consider other goals instead of just waiting for the item to reappear. The exception to this is player resources, which are not removed from the list of available resources when they are not visible. Instead, we use a Visible property which has a value of true or false. This allows the bot to attempt to track down a player who has moved out of sight.

Gamebots sends two main types of messages. Synchronous updates are sent at constant intervals throughout the game, and include information about the current state of the bot (health, location, etc.), information about the game (scores and whether the flags are currently stolen or not), and information about visible players or objects. Asynchronous messages can be received at any time and include messages about events such as bumping into another player, running into the wall, picking up an item, hearing a noise, taking damage, and messages sent by other players. It is also possible for the bot to request information (like a path to a location) from the game server, and the results of those requests are sent as asynchronous messages.

The actions that may be performed by the bot include:

STOP: stops all motion

RUNTO: causes the bot to move to a location or target

TURNTO: causes the bot to turn to face a location or target

SHOOT: starts firing the bot's weapon

STOPSHOOT: stops firing the bot's weapon

GETPATH: requests a path to a location from the game server

ROTATE: turns the bot a specified number of units

Commands are executed by the server in the order in which they are received and repeatedly sending a command has no effect.

Each rule in the bot's TRPs may generate actions to be output to the Gamebots server. We have written a wrapper function used with many of the actions which sends the command to the Gamebots server but also adds it to a command list with a timestamp. If the same command is received by the wrapper function from the same program and the time is still the same as the timestamp, the command is not sent to the server. The system clock used for this functionality returns the time as a number of seconds. Thus the bot will send each command once per second until a different rule triggers. This is a fairly good compromise, preventing the server from having to process many useless messages while still re-sending messages that have been lost. While we implemented this function specifically for Unreal Tournament commands, this type of solution could be used for any real-time application that connects to a server and a higher precision clock could be used to send messages at intervals other than one second.

A number of the actions possible in Unreal Tournament conflict with each other. The list of conflicts used by our bot is as follows:

- RUNTO conflicts with itself
- STOP conflicts with RUNTO
- TURNTO conflicts with itself
- STOP conflicts with TURNTO
- RUNTO conflicts with TURNTO
- STOPSHOOT conflicts with SHOOT
- GETPATH conflicts with RUNTO

Most of these conflicts are self-explanatory. The *GETPATH* command, however, conflicts with *RUNTO* for the reason that the bot is given paths starting from its current location. If the bot moves between requesting the path and receiving it, the path may no longer be valid.

4.4.2 Goals for Capture the Flag

The bot plays on the map CTF-simple, which is available online with the Gamebots package. This level is shown in 4.2.

The bot generates the following set of goals, each of which has a corresponding TRP:

- Return to Base

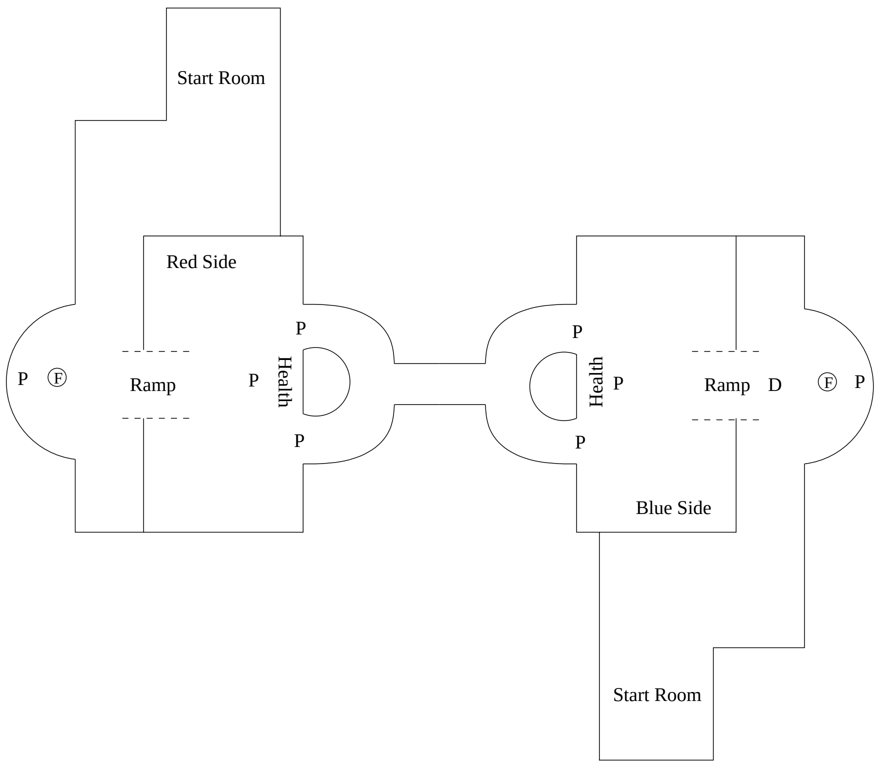

Figure 4.2: The layout of the CTF-simple map. The locations of health packs and the rooms where players start are marked. The circled 'F's represent the flag bases, while 'P' marks a patrol point and 'D' marks the agent's default location. This diagram is not to scale.

- Patrol
- Get Health
- Run Away
- Get Weapon
- Get Ammunition
- Attack Enemy
- Track Enemy

- Capture Enemy Flag
- Score Point
- Return Stolen Flag

Bots initially start the game in a side room. The first goal to be generated is `Return to Base`, which causes the bot to navigate to a spot in front of its flag base. This goal causes the bot to leave the side room when it begins the game (or restarts after being killed) and also combines with `Patrol` to cause the bot to explore the area around its base. `Return to Base` is generated with low priority whenever the bot is away from its base. `Patrol` is generated with a slightly higher priority when the bot is at the base. `Patrol` causes the bot to select a location we call a patrol point and move there. `Patrol` remains active until the bot reaches the patrol point, and `Return to Base` is not generated if `Patrol` is still active. The result is that only one of the goals is active at a time, and two alternating tasks create the bot's default behaviour (which is pre-empted by all other goals).

The bot has two goals to protect itself. `Get Health` causes the bot to pick up a health pack. It is generated when the bot's health is low and a health pack is visible. If the bot's health is below a threshold, `Get Health` is generated with a high priority. Otherwise the priority is inversely proportional to the bot's health. `Run Away` causes the bot to run away from an enemy player. It is generated when the bot's health is low and there is no health pack visible, or when the bot is under attack. `Run Away` has similar priority to `Get Health`, but we make it slightly higher so that it will take priority over `Get Health` in the case that both goals are generated at the same time.

`Get Weapon` and `Get Ammunition` are both generated with high priority if the bot has no weapon/ammunition. They are also generated with a lower priority when the appropriate types of resources are available (so the bot will opportunistically pick up weapons and ammunition).

`Attack` causes the bot to fire on an enemy player. It is generated whenever an enemy player is visible with priority inversely proportional to the distance between the bot and the enemy player or with a high priority if the enemy player is nearby. If the enemy player disappears, the `Track Enemy` goal is generated instead to cause the bot to look for the enemy. `Track Enemy` decreases in priority over time so that if the bot cannot find the enemy player it will eventually give up.

`Return Stolen Flag` causes the bot to pick up its own flag (which returns it to the flag base). It is generated when the flag is visible and not on the flag base, and it is always very high priority.

Finally, there are two goals to allow the bot to score points. `Capture Enemy Flag` is generated when the bot is not holding the enemy flag. It is generated with high priority when the enemy flag is visible. Otherwise, it is generated with medium priority one tenth of the time and low priority the rest of the time. (This allows the patrol behaviour to be expressed.) `Score Point` is generated when the bot is holding the flag, and causes the bot to return to its own base. There is one exception, which is when the bot's own flag has been stolen. A player is not allowed to score a point while their own flag is stolen, so `Score Point` is not generated in this case.

Figure 4.3 shows the relative priorities for the goals in UT. The vertical axis represents the priority value, and the horizontal axis represents the dependent variable for

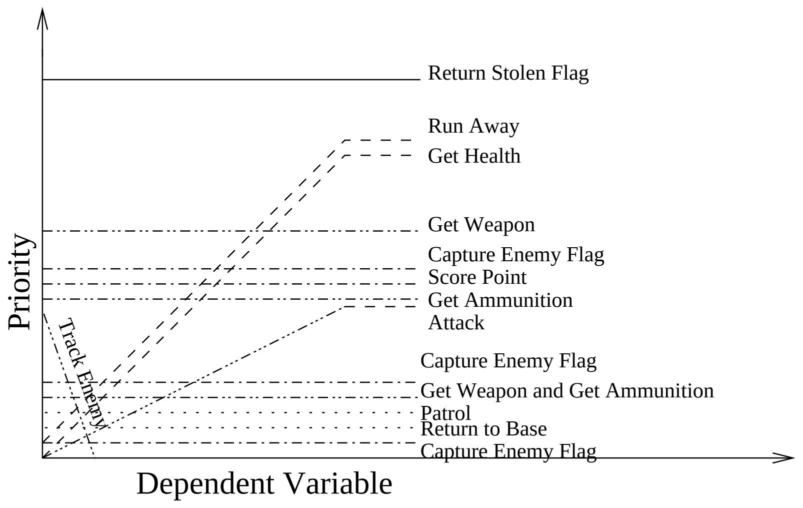

FIGURE 4.3: Relative priorities for goals in Unreal Tournament. The horizontal axis represents the dependent variable, which is different for each goal. Goals which have different priority functions in different circumstances are shown as several lines on the graph.

each goal (i.e. time for `Track Enemy`, the inverse of the bot's health level for `Get Health` and `Run Away`, or the distance to the enemy player for `Attack`). Some of the goals have different values in different circumstances; in these cases we have shown each value as a separate horizontal line on the graph. The higher lines represent urgent cases, and the lower lines represent the priorities in less urgent cases. The priority level of a goal is selected from the available possibilities according to the current conditions when the goal is generated.

4.4.3 Hierarchical TRPs

The UT bot, unlike the Tileworld agent, uses hierarchical TRPs. This technique allows a TRP to "call" a sub-TRP as if it were a function. This has the effect of replacing the rule making the call with the contents of the sub-TRP. We allow TRPs called in this fashion to take arguments (again, like functions).

Calls to sub-TRPs are implemented using Sim_Agent rulefamilies. All of the agent's TRPs are collected into a rulefamily along with a controlling arbitrator ruleset. The arbitrator ruleset transfers control to each top-level TRP required by the current task list. Normally, each TRP transfers control back to the arbitrator at the end of each rule. Rules which call sub-TRPs transfer control to the sub-TRP instead. The rules in the sub-TRP then transfer control back to the arbitrator. Arguments are passed to the sub-TRP by adding a fact to the database. The fact contains the identifier ARGS followed by the name of the sub-TRP, the name of the top-level TRP making the call, and the required arguments. Multiple TRPs can make calls to the same sub-TRP simultaneously. Therefore, each rule in the sub-TRP must check that the ARGS fact being used corresponds to the currently executing task.

4.4.4 Implementation of GOTO

The `GOTO` sub-TRP was the most difficult aspect of the implementation. This sub-TRP is called by nearly every top-level TRP in the Unreal Tournament bot. As a result, it is very likely that multiple copies of `GOTO` will be running simultaneously.

The `GOTO` TRP starts by requesting a path from the server. The server requires an identifier to be used to distinguish between path requests. `GOTO` creates a unique identifier when the path is requested, and stores it in a resource of type "request". The request resource also stores the bot's current location and the destination of the path.

Once the path has been received from the server, the basic behaviour of the `GOTO` program is to run to the first node in the path, then remove the node from the path when the bot is near it. This is complicated by the fact that the bot might have moved off the path (because it was working on a higher priority task, for example). Thus the bot needs to keep track of which node in the path it is currently at, and check that it can get to the next node.

We wrote a function for use in the `GOTO` program that returns the bot's current node in a path. This function will return an error value if the bot is too far off the path. If the bot has moved further along the path in pursuit of another goal, the return value of the function will reflect that fact, preventing the bot from backtracking unnecessarily. (Backtracking can be especially problematic as paths may lead the bot around corners. If the bot has already gone around the corner but then tries to get back the to the first node in the path, it can easily end up running into the wall.)

`GOTO` also checks for a number of error conditions. Some of these are internal book-keeping errors, such as extra path request resources or inaccurate information about the bot's progress along the path. In addition, the server may return an empty path if the bot is very close to its desired destination. `GOTO` causes the bot to run directly to its destination in this case. In the case that the bot is too far off the path, both the path and the path request resources are removed so that the path can be re-requested. Finally, the server only returns paths of lengths up to about 15 nodes. If the required path is too long, the server will return a partial path. Therefore when `GOTO` reaches the last node in the path it must also check to see whether it is at the destination. If it is not at the destination but the destination is nearby, it runs there directly. Otherwise, the path will be removed and re-requested as in the case where the bot leaves the path.

4.4.5 Implementation of top-level TRPs

Each TRP in the bot is implemented as a Sim_Agent ruleset. In this section, we give a brief description of the functionality of each TRP, or group of similar TRPs. We have already given details of the TRPs for the Tileworld agent, so we will not go through each rule in the TRPs for the UT bot.

Patrol and Return to Base

`Patrol` and `Return to Base` interact to produce a patrol behaviour. Return to base is extremely simple, merely calling the `GOTO` sub-TRP when the bot is not located at the default location. The arguments to `GOTO` are the coordinates of the default location. `Patrol` is very similar, except that the first rule chooses a patrol point arbitrarily from a list written by the programmer. These patrol points are selected to move the bot to different sections of the base area, and bring nearby enemies or items into view. If some patrol points are listed several times, the bot will be more likely to choose those points. The patrol behaviour can also be easily modified by changing the set of patrol points off line. (The bot could be provided with a way of adding new patrol points as it explores the level, but this was beyond the scope of our project.)

Once the bot leaves the default location, the `Return to Base` goal will be generated. This goal is lower priority than the `Patrol` goal, so the bot will not return to the base until it has reached the patrol point. These two goals interact to cause the bot to run along lines radiating outward from the default location, returning to the default location in between each pair of patrol points.

Get Weapon / Get Ammunition / Get Health

The `Get Weapon`, `Get Ammunition` and `Get Health` TRPs are extremely similar. Each one binds a resource of the appropriate type and calls `GOTO` with the location of the resource. Once the bot reaches the resource, the TRP removes the resource (the interface should remove it, but removing it in the TRP reduces delays, allowing the agent to switch to another task sooner). This should result in the success condition for the TRP.

Run Away

The `Run Away` TRP is intended to remove the bot from a dangerous situation, and possibly allow it to recover health. To accomplish this, we add two resources to the bot's database when the bot is created. Each resource specifies a "safe point". Since the simple level we are using has two main rooms with a health pack located in each, we have put one "safe point" in the middle of each room. The `Run Away` program causes the bot to choose the safe point that is farthest away from the bots current location, and then navigate to that point by calling the `GOTO` program with the coordinates of the safe point. Ideally, this will cause the bot to move away from any nearby enemies and also put the bot next to a health pack.

Attack

The basic functionality of the `Attack` TRP is to issue the *SHOOT* command when the bot has a weapon, an enemy player is visible, and the enemy player is within range. It also turns toward the enemy player, to attempt to prevent the enemy dodging sideways. Once the player is no longer visible, the TRP issues the *STOPSHOOT* command. If the player is no longer visible and the bot is not shooting, the conditions for the success condition of the TRP are met and the goal leaves the arbitrator. In addition, the TRP has several error checks. If the enemy player is out of range, the bot will move closer to the enemy. (The current implementation uses a global range value, and moves the bot to a point just within range of the enemy player. Ideally, it should check the specific range of the wielded weapon.) Finally, if the enemy player is listed as being visible but is not within a few degrees of the bot's forward direction, then the bot will turn toward the enemy player.

Track Enemy

The `Track Enemy` TRP runs when an enemy player has moved out of view. It starts by attempting to predict the enemy player's new position (based on the player's last observed velocity) and moves there. If the player still isn't visible, the bot turns in a circle hoping to catch sight of it. The TRP succeeds if the player becomes visible again, however the priority of the associated goal gradually decays to enable the bot to give up and do something more productive when it can't locate the enemy.

Return Stolen Flag

The `Return Stolen Flag` TRP runs when the bot sees its own flag and the flag is not at the base. This happens when an enemy steals the flag and then is killed. Usually, the bot has just killed the enemy, and is therefore standing in front of the flag. The TRP assumes that the flag is therefore reachable, and simply runs directly to it. Note that a more complete implementation of this agent would check whether the flag was reachable, and request a path if it were not. Our implementation uses this assumption successfully for two reasons: the environment is quite simple (so there are not many points that are visible but not reachable), and the bot's attack strategy is very basic.

Capture Enemy Flag / Score Point

The `Capture Enemy Flag` goal is generated when the bot is not holding the opposing team's flag. The associated TRP calls `GOTO` with the location of the flag. (In the current implementation this is simply hard-coded by the programmer.) Once the flag is visible, the bot runs directly to it. This achieves the success condition and the TRP leaves the arbitrator. The `Score Point` goal is generated when the bot is carrying the opposing team's flag. The corresponding TRP causes the bot to run back to its own base, scoring a point.

4.5 Properties of TRPs

Nilsson (Nilsson, 1994) defines two properties that may be satisfied by a sequence of teleo-reactive rules. First, the sequence satisfies the regression property if each action in the sequence satisfies a condition that can be found higher in the sequence. Second, the sequence is complete if it is impossible for a situation where no rule can fire to occur (i.e. the rule conditions cover all possibilities). The sequence is said to be universal if it is both complete and satisfies the regression property.

Our Tileworld agent uses TRPs that contain rules for handling error cases. These rules generally remove information computed by the TRP and assume it will be recomputed by a rule lower in the TRP during the next cycle. Thus, our Tileworld TRPs do not satisfy the regression property. They could be made to satisfy the regression property by rewriting the rules such that the information is recomputed in the error handling rule. (This would also make the agent more efficient.)

Our Tileworld TRPs are complete. However, our Unreal Tournament TRPs are not. For example, the TRP that causes the agent to pick up a health pack succeeds when the agent's health is above a threshold. All of the rules in the TRP require the presence of a health pack to pick up. If there is no health pack, the TRP will fail and leave the arbitrator. Because the GRUE agent has goals that apply in all circumstances, the individual TRPs do not need to be complete.

Chapter 5

Evaluation of the GRUE architecture

We state the full list of GRUE features here and their expected effects as a prelude to our evaluation of these features.

Goals are generated only in appropriate situations. Each goal in GRUE is generated by a goal generator. Goal generators can check the current situation of the agent and the world, and generate the goal only if it is appropriate. This keeps the agent from wasting time working on inappropriate goals. We expect that this feature will, in general, increase the performance of the agent when the agent has two or more goals that can potentially interact. We also expect that reducing the set of goals processed during each cycle will allow the architecture to execute more efficiently.

Goal priorities are set according to the current situation. The priority of a goal is set according to the situation when it is generated. A priority might vary according to the state of the agent (if a robot's battery is very low, it is very important that the robot recharge) or according to other aspects of the world (distance from another agent, for example). We expect that setting goal priorities situationally will enhance performance as compared to an agent with only one possible priority value for each goal, assuming that it makes sense for the goals to have different priorities in different situations.

Goal priorities are updated continuously. The priority of a goal may continue to change as the situation in the world changes. This allows the agent to flexibly adjust its behaviour to match the current situation. For example, a robot's goal to recharge its battery would increase in priority as the battery level drops. This feature allows a goal with an initially low priority to increase in importance and preempt other goals once the situation is urgent. This should result in improved performance, assuming it makes sense for the goals to change in importance over time.

Multiple actions can be executed during each execution cycle. There are many situations in the real world where it is possible to do two things at once, for instance one with each hand. This is less common in games and simulations but it is possible in certain situations. For example, it may be possible to operate a tool or a weapon while moving, or it may be possible to send a message to another player while doing other things. Explicit support for this feature allows the agent to operate in the world more efficiently and take advantage of certain situations the way a human would. This should increase the agent's performance, depending on the set of actions available to the agent.

TRPs request resources in terms of required and preferred properties. We represent objects and agents within the world as resources, and they are stored in terms of their properties. Instead of specifying a unique object, GRUE TRPs request a resource in terms of the properties needed. Furthermore, a TRP can specify both required properties which are necessary for the correct execution of the TRP, and preferred properties. Preferred properties can be used to give an agent personality. They can also be used for more practical purposes in situations where two or more resources have similar effects, but one of them is better than the others. In these latter cases, the use of preferred properties should improve the agent's performance.

A TRP can specify that it needs exclusive access to a resource. We allow TRPs to request exclusive access to a resource. When a TRP has exclusive access, no other TRP can use that resource. This is needed in certain situations, such as when the TRP is about to modify or remove the resource. In other cases, a TRP only needs to check the properties of a resource or verify that it exists. In these cases, the TRP allows the resource to be shared by other TRPs. We expect that allowing other programs to use a resource that is being modified or removed will usually result in incorrect behaviour.

A TRP that makes the same request during consecutive cycles will receive the same resource each time. The GRUE architecture keeps a list of resources that were used during a cycle and retains that list during the next cycle. This allows the architecture to assign the same resource to a TRP when the TRP makes the same request repeatedly. The effect of this feature on the performance of the agent depends to some extent on the number and type of available resources in the world. Requiring the agent to continue using the same resource can prevent the agent from taking advantage of the appearance of more desirable resources. However, if there are many equivalent resources, allowing the agent to switch resources every cycle can result in the agent switching resources repeatedly and not making progress toward achieving the goal.

TRPs can use special notation to request numerical values within a range. We have designed special notation for numerical ranges. This allows GRUE to correctly handle items such as money or ammunition which occur in quantities rather than single objects, as well as being useful for handling numerical properties like distance. This should improve the agent's performance in environments that contain quantities, but will have no particular effect otherwise.

Numerical quantities can be divided and used for more than one goal. In cases where a numerical value represents a quantity, it is often the case that a plan requires only part of the available amount. An example of this is when the agent needs to buy an item, but has more than the needed amount of money. In these cases, we allow a TRP to gain exclusive access to only part of the resource, and leave the rest of the resource available to be used by other TRPs. This will increase performance in environments that contain divisible quantities, and will have no effect otherwise.

All of the standard features of the GRUE architecture are optional, and can be disabled or simply not used by the programmer. It is the programmer's decision whether or not to use features that are supported by but not built in to the GRUE architecture components. Such features include preferred properties and dynamic adjustment of goal priorities. Features which are built in to the architecture components, such as the execution of multiple actions during each execution cycle, can be disabled using a flag variable.

We have tested GRUE in both the Tileworld (see Section 4.3) and Unreal Tournament (see Section 4.4) environments. Testing was done by comparing the standard GRUE agent to a GRUE agent with one or more features of GRUE disabled. The tests were designed to discover whether the features we expect to be advantageous actually do confer an advantage, and if so whether that advantage is affected by the properties of the environment. In both types of tests we present data collected in several of the possible of environmental conditions. In Tileworld, we can set both the rate at which objects appear and disappear, and the number of objects in the environment. In Unreal Tournament, we can change the type and number of opponents that compete against the GRUE agent i.e. the opponents can be other GRUE agents or the built-in "bots" that are provided with the game.

5.1 The Tileworld Environment and Experimental Results

Although we described Tileworld (Pollack and Ringuette, 1990) in Section 4.3, we remind the reader that Tileworld is a grid-based world containing tiles, holes, and obstacles. An agent in Tileworld must pick up tiles and drop them in holes while avoiding the obstacles that may block its path. Tiles in our Tileworld are grouped into stacks, and correspondingly, each hole has a depth. Our Tileworld contains only one agent.

We chose to use a standard Tileworld of size 20 by 20, bordered by obstacles. The Tileworld contains obstacles, holes of varying depth, and tile stacks of varying size. The depth of each hole and size of each tile stack is randomly generated at run-time in the range 1 - 10. The agent is allowed to carry only one tile stack at a time, but can drop any portion of that tile stack into a hole. Both tiles and holes have shape. The agent scores 1 for each

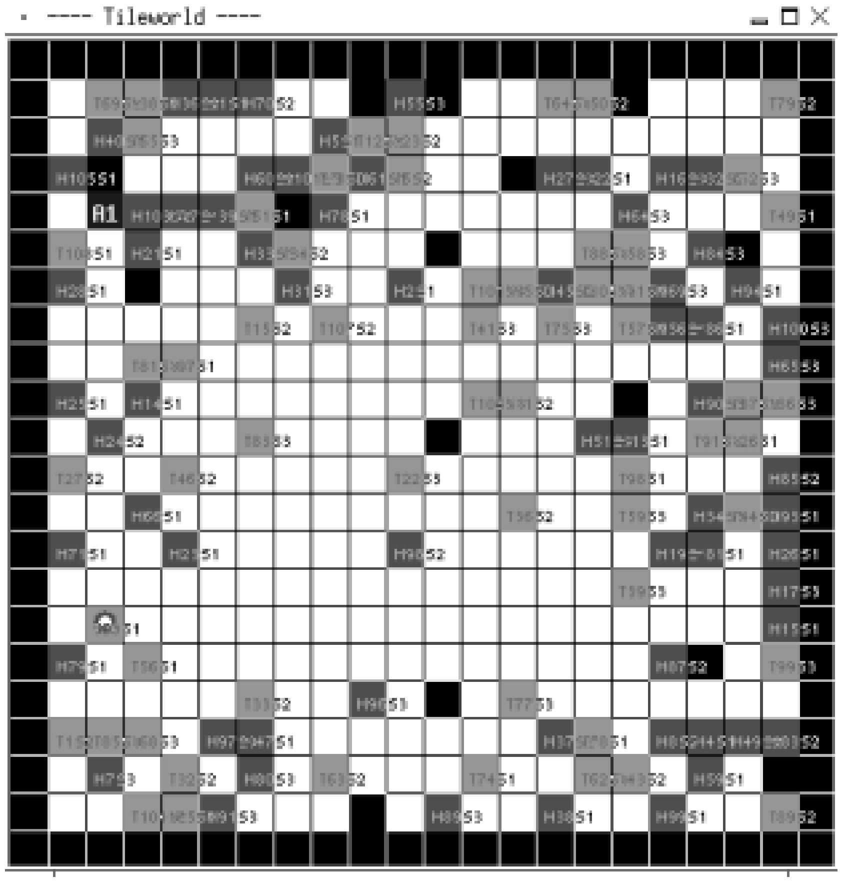

FIGURE 5.1: A screenshot of Tileworld. Solid black squares are obstacles. Squares whose labels start with T are tile stacks, and those whose labels start with H are holes. The agent is labeled A1. The small circle marks the object that the agent is currently moving toward.

tile dropped if the tiles do not match the shape of the hole, and 3 for each tile dropped if they do. If the hole is filled completely, the agent receives an additional 20 points.

The Tileworld agent has a sensor range of five, where distance is measured in terms of the number of moves necessary to reach an object, rather than absolute distance. Two main parameters are used to control the Tileworld environment. The first, density, controls the number of objects in the Tileworld. If the density is set to 10, the Tileworld is initially created with 10 holes, 10 tile stacks and 10 obstacles. The second parameter, which we will call N, controls the rate of change in the environment. On average a tile stack should disappear and a new one be created every N cycles. The probability of creating a new tile stack during each cycle is set to $1/N$ (and the same for holes and obstacles). The probability of deleting a tile stack is also $1/N$; again, it is the same for holes and obstacles. These formulas were designed to result in a steady-state condition — that is, the density in the absence of an agent should on average remain the same. It is important to remember that a higher N value results in an environment that changes more slowly and a lower N value results in an environment that changes more rapidly.

Performance of agents in Tileworld is evaluated primarily by comparing their scores, and those scores will be presented in this chapter. We also collected data on a number of other factors, some of which were used to verify the correct implementation of the Tileworld, and some of which were used to check that our ideas about the agent's behaviour are accurate.

For each of the tests, the agent was run 50 times for 1000 cycles in various different environments. The four values of 10, 40, 70, and 100 were used for both the density and N, giving 16 different environmental conditions. We computed the average score for the 50 runs in each environment. We then normalized the average scores by dividing them by the maximum potential score. Normalization is necessary because the agent has a higher potential score in environments rich in tile stacks and holes than it does in an environment with few tile stacks and holes. The agent also has a higher potential score in environments where the rate of change is slower. It is not meaningful to compare the raw scores for these different environments. Normalizing the agent's scores allows us to compare the agent's performance relative to different environmental conditions; however we were only able to do an approximate normalization so the normalized scores are potentially inaccurate.

The maximum potential score was computed by taking the average number of tiles created and multiplying that value by 3, and then adding the average number of holes created multiplied by 20. This gives the maximum score possible assuming that the agent dropped all generated tiles into holes of matching shape (receiving 3 points for each) and also filled every hole that was generated, receiving the 20 point bonus for each. This of course assumes that the number of tiles and total hole depth generated are equal, and appropriately distributed. We use this approximation of the maximum possible score because computing the actual maximum is prohibitively complicated. The actual score that can be achieved by the agent depends on whether the total number of tiles is the same as the total number of holes, how many of those are also the same shape, and how many of those tiles and holes are in positions where they can be used by the agent. In fact, computing the actual maximum would require computing the agent's optimal target object during each cycle, as well as the minimum number of cycles needed to maneuver around obstacles.

To tell if our comparisons are meaningful, we performed a t-test on the raw score data for each environmental condition. This test tells us whether the difference between the average score for the experimental agent in each environment is significantly different from the average score for the standard GRUE agent in that environment. While we discuss the t-test results where they are relevant in this chapter, we did not want to overwhelm the reader with data. Please see Appendix G for the complete set of t-test data along with a fuller explanation of the t-test. We do provide boxplot comparisons for selected cases in this chapter. While providing boxplots for every case is impractical, we use them to visually verify the information provided by the t-tests for cases specifically mentioned in the text.

5.1.1 Performance of the GRUE Agent

In this section, we discuss the performance of the standard GRUE agent, which includes all the features of the architecture as described above. The data in this section serves as a baseline comparison for the experiments described in the rest of this chapter. We present the average scores for the standard GRUE agent in all of the environmental conditions, the standard deviations and the normalized score values, which allow us to compare the agent's performance in different environmental conditions. Using the raw average scores, we can only make comparisons between different agents in the same environmental conditions.

Density		**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	Average Score	491.12	350.34	327.90	309.26
	Standard Deviation	149.03	99.42	67.43	69.80
	Normalized Average Score	0.12	0.32	0.43	0.43
40	Average Score	1033.22	1100.34	1031.62	1089.32
	Standard Deviation	172.86	126.11	190.01	175.80
	Normalized Average Score	0.21	0.50	0.57	0.59
70	Average Score	1427.12	1236.92	1252.24	1195.88
	Standard Deviation	253.45	339.98	378.83	372.96
	Normalized Average Score	0.23	0.38	0.43	0.41
100	Average Score	1607.78	1321.52	1150.82	1316.02
	Standard Deviation	250.56	414.67	492.51	426.42
	Normalized Average Score	0.22	0.30	0.29	0.33

TABLE 5.1: Scores for the standard GRUE agent in Tileworld.

Using the normalized values in Table 5.1, we see that the standard GRUE agent performs best in an environment with a density of 40 and an N value of 100. Table 5.2 shows the total time for each run. This is the number of seconds it took to complete 1000

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	15.63	16.72	17.02	17.06
40	20.71	21.02	21.70	21.63
70	37.1	39.61	39.5	40.39
100	61.14	65.94	63.87	64.84

Table 5.2: Average number of seconds of processor time used for 1000 cycles of the standard GRUE agent in Tileworld.

cycles, and includes the code for Tileworld as well as the GRUE agent. The Tileworld code is quite simple, consisting mostly of startup and shutdown time, and time spent removing or adding objects. We assume that this is negligible. The total time increases with the density of the Tileworld. With a density of 10, 1000 cycles takes about 17 seconds. This is approximately 60 cycles per second, making it easily fast enough to run in a commercial game environment. With the density set at 100 this drops to 15 cycles per second which is rather on the slow side.

5.1.2 Features of the Goal Generators

GRUE generates goals according to the agent's memory, which is updated based on the agent's environment. It also sets the priority of a goal according to the current environment. Finally, it updates the priority of a goal continuously while that goal is active. To test these features, we have created four different reduced functionality versions of the GRUE agent.

All Goals: All goals are generated all the time, rather than being generated only in when certain conditions are true.

No Updates: Goals and priorities are situation dependent, but priorities are not updated once the goal is generated.

CONSTANT PRIORITIES: Priorities are set to constant values set by the programmer.

ALL GOALS CONSTANT PRIORITIES: All goals are generated all the time, with constant priority values.

The performance of these agents is shown in Table 5.3, Table 5.5, Table 5.7 and Table 5.9.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.09	0.25	0.34	0.30
40	0.17	0.32	0.30	0.27
70	0.21	0.24	0.22	0.23
100	0.20	0.19	0.17	0.17

TABLE 5.3: Normalized average scores for the ALL GOALS agent in Tileworld.

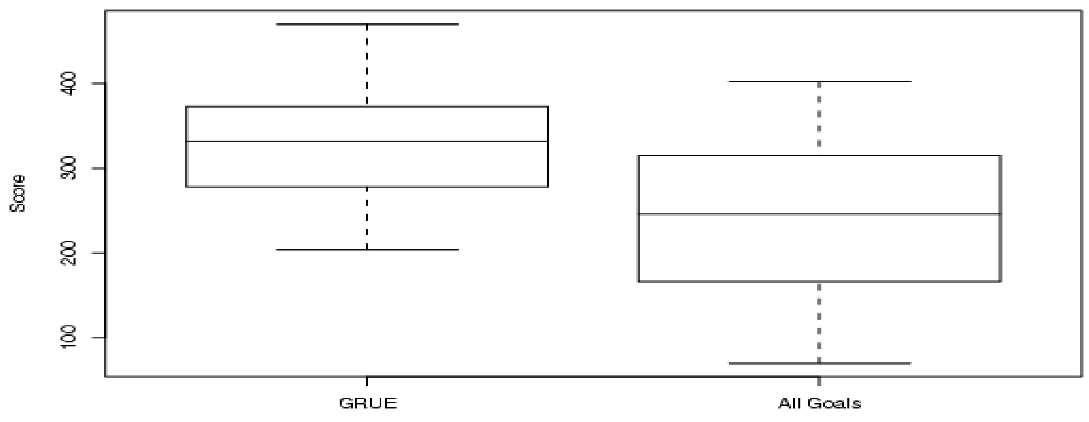

FIGURE 5.2: Boxplot comparison of the raw score data for the ALL GOALS agent and the GRUE agent when N=70 and Density=10. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

From Table 5.3, we see that the ALL GOALS agent does not perform as well as the GRUE agent based on normalized scores. The results of t-tests for these values (see

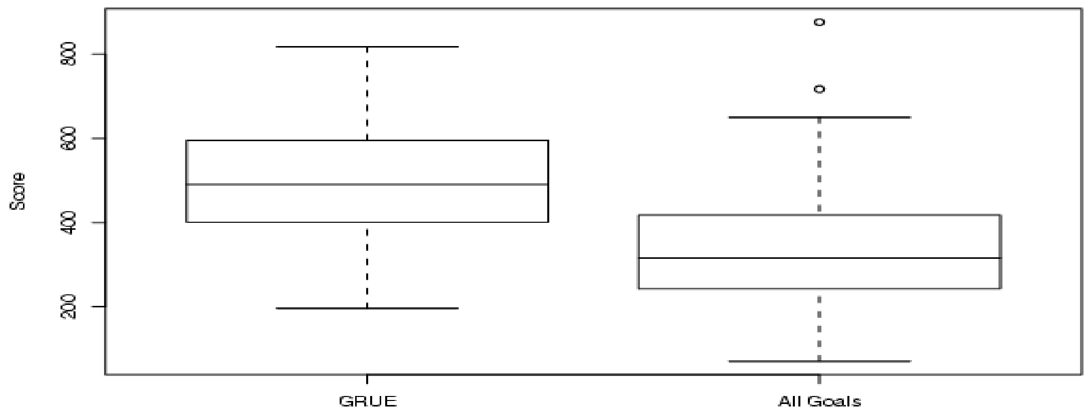

FIGURE 5.3: Boxplot comparison of the raw score data for the ALL GOALS agent and the GRUE agent when N=10 and Density=10. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers. [2]

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	13.71	13.21	12.46	12.68
40	21.17	21.27	20.26	19.99
70	35.34	36.67	37.92	38.49
100	60.20	62.94	58.07	60.04

TABLE 5.4: Average number of seconds of processor time taken for 1000 cycles of the ALL GOALS agent in Tileworld.

G.1) indicate that this difference is significant at the 10% level for all 16 values. The ALL GOALS agent shows its best performance in the low density case where N = 70, although the GRUE agent still outperforms it in that case. It performs worst in the fast low density case, although the scores also drop off towards the slow changing, high density corner. This matches our expectation that generating irrelevant goals will be detrimental in general. Figure 5.2 and Figure 5.3 show boxplots for the ALL GOALS agent's best and worst performances respectively. In both cases, the median of the ALL GOALS agent's scores is below the lower quartile for the GRUE agent, clearly showing that the GRUE agent performs better.

Contrary to our expectations, the processor time used for each cycle does not increase when all the goals are always in the arbitrator. Instead, the All Goals agent actually takes slightly less time to execute than the GRUE agent. This indicates that it takes more time in the current GRUE implementation to check the several goal generation and update cases than it takes to begin running an inappropriate goal, discover that its goal condition is already met, and switch to a different goal. The difference, however, is only significant in about half the environmental conditions (see Table G.2).

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.11	0.31	0.43	0.42
40	0.21	0.50	0.61	0.61
70	0.24	0.44	0.48	0.46
100	0.24	0.37	0.40	0.39

Table 5.5: Normalized average scores for the No Updates agent in Tileworld.

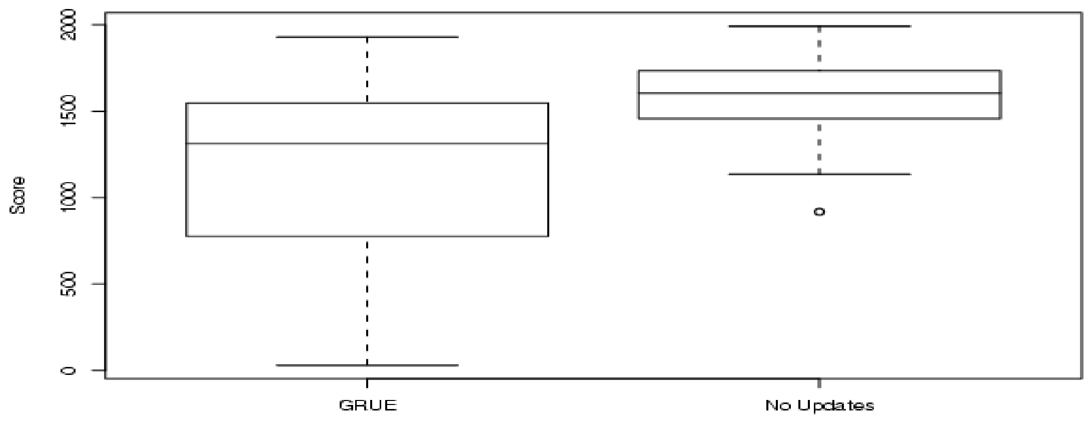

Figure 5.4: Boxplot comparison of the raw score data for the No Updates agent and the GRUE agent when N=70 and Density=100. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	15.02	15.38	15.74	16.26
40	19.47	19.64	19.64	20.02
70	34.44	36.11	37.05	39.36
100	58.45	62.86	64.40	62.92

Table 5.6: Average number of seconds of processor time taken for 1000 cycles of the No Updates agent in Tileworld.

The No Updates agent performs very similarly to the GRUE agent, and even outperforms it in the slow changing high density environments. The t-tests (shown in Table G.3) indicate that there is no significant difference between this agent and the GRUE agent in the slow changing low density environments. There is a significant difference in the slow changing high density environments, as well as in all of the cases where N=10. This last is not evident from the normalized score data shown in Table 5.5 indicating that the normalization may have obscured important differences in this case. Figure 5.4 shows a boxplot comparison for the case where N=70 and density is 100; this shows that the No Updates agent clearly outperforms the GRUE agent in this environmental condition. The No Updates agent takes slightly less time to run than the standard GRUE agent, which is expected, as updating the priority of each goal will take a small amount of time. This difference is significant in all except one environmental condition (see Table G.4 for details).

It is understandable that this feature has little effect on the Tileworld agent, due to the fact that the agent has only three programs, one for getting a tile stack, one for filling a hole and one for avoiding obstacles. The agent cannot fill a hole until it has a tile stack, and it can only carry one tile stack at a time. The only time two programs run in parallel is when the avoid obstacle TRP is running. In this case, either the agent is next to an obstacle and the avoid obstacle goal has a very high priority, or else the avoid obstacle executes the null action and leaves the arbitrator. So updating the priority on the other program should not have any effect on the order in which the TRPs are run.

The fact that the NO UPDATES agent outperforms the GRUE agent is due to the way our goal generators are written. The goal generators for the *Get Tile* and *Fill Hole* tasks generate the goal with a priority that increases as the agent approaches an object of the appropriate type. If, however, there are several objects of the appropriate type, the goal generators use the first one. They do not check to see if the object is the current target object. Therefore the agent may end up with the maximum priority for *Get Tile* while it is still several squares away from the target tile. If the agent is also next to an obstacle, it will also have the maximum priority for the obstacle avoidance task, which is the same as the maximum priority for *Get Tile.* This situation is illustrated by Figure 5.5.

If the *Get Tile* task is before the obstacle avoidance task in the tasklist, the agent can get stuck behind the obstacle. When the goal priorities are not updated, it will be much less likely that the *get tile* task will have the maximum priority, reducing this effect. This effect can be compared to the unexpected effect we discovered when testing the effects of including exclusive bindings, discussed in Section 5.1.5, which also occurs when the *Get Tile* task preempts the *Avoid Obstacle* task. (Note that we could easily fix these problems by adding 10 to the priority of avoid obstacle goal.)

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.09	0.31	0.45	0.44
40	0.22	0.50	0.60	0.60
70	0.25	0.45	0.49	0.46
100	0.24	0.37	0.41	0.38

TABLE 5.7: Normalized average scores for the CONSTANT PRIORITIES agent in Tileworld

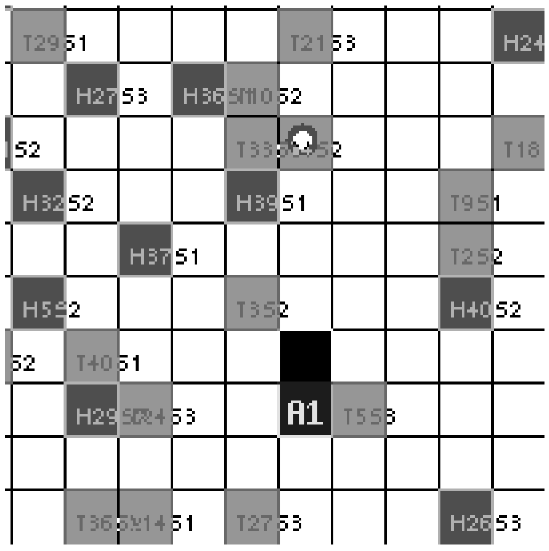

FIGURE 5.5: A case where the *Get Tile*
TRP is targeting a tile several squares away, but the goal generator sets the priority to 100 based on another tile that is next to the agent. The priority of the *Avoid Obstacle* goal is 100 because of the obstacle next to the agent. The two goals have equal priority and *Get Tile* happens to end up first in the task list, so that the move action proposed by *Avoid Obstacle* is discarded. The result is that the agent repeatedly tries to move into the obstacle.

Performance of the CONSTANT PRIORITIES agent is also similar to the GRUE agent, although outperforming it in a few cases scattered through the set of environmental conditions. The t-test results (shown in Table G.5) indicate that this agent does not perform significantly differently from the GRUE agent in the cases where density is 10 and N is 40, 70, or 100. There is also no significant difference in the case were density and N are both 40.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	15.50	16.33	16.34	16.35
40	19.98	19.97	20.67	20.92
70	34.37	38.43	38.77	39.60
100	60.81	63.34	64.19	65.21

TABLE 5.8: Average number of seconds of processor time used for each run of the CONSTANT PRIORITIES agent in Tileworld.

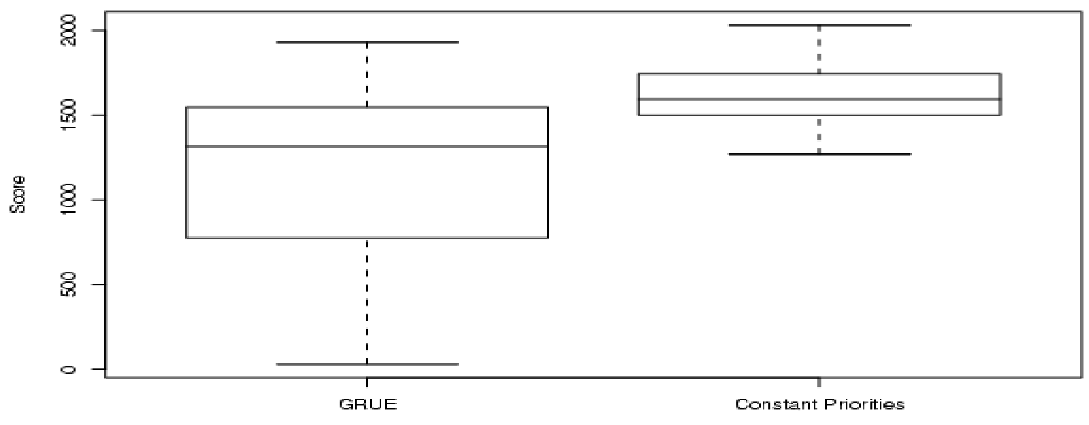

FIGURE 5.6: Boxplot comparison of the raw score data for the CONSTANT PRIORITIES agent and the GRUE agent when N=70 and Density=100. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

The remaining results are significantly different at the 10% confidence level. A boxplot for the high density, N=70 case (Figure 5.6) indicates that the CONSTANT PRIORITIES agent outperforms the GRUE agent in this environment. The average processor time per run is slightly lower than for the GRUE agent in most cases, though not as much lower as for the agent with no goal updates. In the slow changing, high density cases the processor time is actually slightly higher, although the difference is not significant for these cases (see Table G.6).

This agent shows no disadvantages over the GRUE agent for much the same reason as the NO UPDATES agent does not. Since we found that updating the priorities makes little difference to the execution of the TRPs in the standard agent, using constant priorities set by the programmer is just as good as computing those priorities at run time (assuming that the priorities are set appropriately). In this case, the priorities for the CONSTANT

PRIORITIES agent are:

Avoid Obstacle 75

Get Tile 50

Fill Hole 50

Compare the CONSTANT PRIORITIES agent to the standard GRUE agent, which has the following priorities:

Avoid Obstacle 100/(distance to nearest obstacle)

Get Tile 10 if no tile stacks are present, otherwise 100/(distance to nearest tile stack)

Fill Hole 10 if no holes present, otherwise 100/(distance to nearest hole)

In the standard GRUE agent, these priorities do not actually make much difference. The *Get Tile* and *Fill Hole* tasks do not run at the same time, so their relative priorities have no effect. The only important thing about the priorities is that the *Avoid Obstacle* task must have the highest priority when the agent's path is blocked. We set the priority for *Avoid Obstacle* in the CONSTANT PRIORITIES agent to a higher value than the priorities for the other two goals, so this condition is met. The fixed priorities should eliminate the problem described above in the discussion of the NO UPDATES agent, and indeed the CONSTANT PRIORITIES agent does show a performance increase over the GRUE agent in the same environmental conditions as the NO UPDATES agent.

The ALL GOALS CONSTANT PRIORITIES agent demonstrates lower normalized average scores than the GRUE agent, and in many cases the scores are also lower than

Density	N = 10	N = 40	N = 70	N = 100
10	0.08	0.23	0.32	0.31
40	0.16	0.28	0.28	0.25
70	0.17	0.22	0.20	0.17
100	0.16	0.17	0.16	0.16

Table 5.9: Normalized average scores for the All Goals Constant Priorities agent in Tileworld

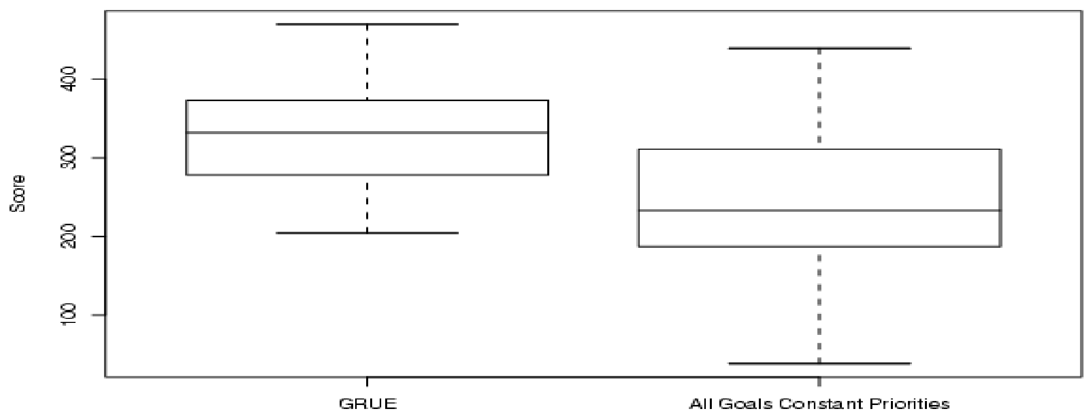

Figure 5.7: Boxplot comparison of the raw score data for the All Goals Constant Priorities agent and the GRUE agent when N=70 and Density=10. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

those of the All Goals agent. The t-tests (shown in Table G.7) indicate that the difference between the All Goals Constant Priorities agent and the GRUE agent is significant in all cases. Figure 5.7 and Figure 5.8 present comparisons between boxplots for this agent and the GRUE agent in two different environmental conditions, both of which show the GRUE agent having a clear advantage over the All Goals Constant Priorities agent.

The average processor time per run is lower than for the GRUE agent in almost all cases since it is not taking the time to update the goals. The difference is significant in most cases, although it is not significant for N values of 10 and 40 with density values of 40 or 70,

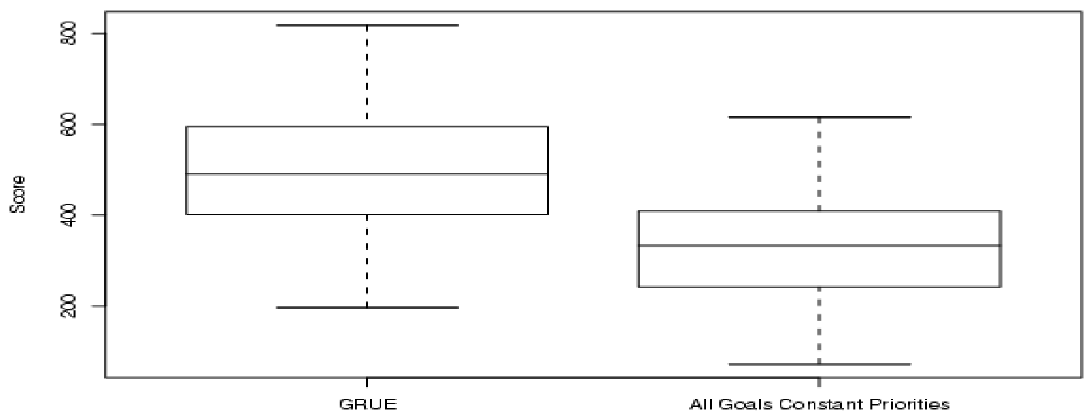

FIGURE 5.8: Boxplot comparison of the raw score data for the ALL GOALS CONSTANT PRIORITIES agent and the GRUE agent when N=10 and Density=10. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	12.73	12.55	12.31	11.38
40	19.83	20.62	20.20	20.06
70	37.80	39.09	35.25	37.03
100	57.70	59.43	60.13	57.58

TABLE 5.10: Average number of seconds of processor time used for each run of the ALL GOALS CONSTANT PRIORITIES agent in Tileworld.

and also for N=70 with density set to 100. The ALL GOALS CONSTANT PRIORITIES agent shows its best performance with a density of 10 and an N value of 70, and in general the chart has a fairly similar profile to the ALL GOALS agent. With all goals always generated, the agent is forced to spend time on inappropriate tasks, which may even interfere with the more urgent goals. With dynamic priorities, some of the inappropriate tasks can be preempted when another task's priority increases, but with constant priorities as in this case, the agent is more inefficient.

5.1.3 Preferred Properties

We evaluated the effectiveness of preferred properties by comparing a standard GRUE agent's performance against the performance of two agents without preferred properties, because there are two ways of eliminating preferred properties from an agent's programs. First, the preferred properties can simply be deleted, leaving an empty list. We will call this agent the Deleted Properties agent. Second, the preferred properties can be moved into the required properties list. We will refer to this second agent as the Required Properties agent.

The main use of preferred properties is in the selection of a target tile stack or a target hole. Tile stacks are preferred to be close, young, and large. This makes it more likely that the tile stack will not vanish before the agent is able to use it, and also maximizes the potential score achievable with the tile stack. Holes are preferred to be close, young, small[3] , and the same shape as the tile stack. This again maximizes the chance that the hole will not vanish before the agent reaches it, and also maximizes the chance of achieving a bonus for matching the shape or filling the hole.

In creating the Required Properties agent, ranges that occur in both the required and preferred properties lists are combined. However, utility orderings in ranges have no effect when used in the required properties list. Also, the only use of preferred properties in the avoid obstacle program uses two contradictory properties, [TOWARD hole] and [TOWARD tile_stack] as in the following rule:

```
NOT(?perpendicular_direction SHARED ([TYPE direction]
```

[3]We might have been able to improve the performance of the agent by instead preferring holes that are no bigger than the size of the agent's tile stack, but otherwise as large as possible. (i.e. #|1 –> size(tile_stack) |#).

```
                                     [TOWARD around_obstacle]) ()) AND
(?direction SHARED ([TYPE direction]) ([TOWARD hole] [TOWARD tile_stack])) AND
NOT(?obstacle SHARED ([TYPE obstacle]
                      [DISTANCE 1]
                      [DIRECTION property(''VALUE'', ?direction)]) ())
⟹
null
```

A direction only has one value of the Toward property, which can be hole, tile_stack, or around_obstacle. This rule causes the agent to "dislike" using directions with the value around_obstacle (i.e. hole and tile_stack are equally preferred values). (This is to cause the *Avoid Obstacle* TRP to prefer to base its movements on directions that are moving the agent toward a useful object.) Thus the properties were simply removed instead of being moved into the required properties list.

We show the performance of each agent for all environments in Table 5.11 and Table 5.12. The Deleted Properties agent performs best with a density of 40 and an N of 100. Interestingly, the Required Properties agent also demonstrates its peak performance when the density is 70, although it occurs with an N value of 100.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.08	0.29	0.38	0.38
40	0.15	0.35	0.41	0.43
70	0.17	0.29	0.30	0.31
100	0.18	0.23	0.25	0.23

Table 5.11: Normalized average scores for the Deleted Properties agent in Tileworld.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.02	0.06	0.06	0.09
40	0.08	0.16	0.22	0.22
70	0.12	0.23	0.26	0.28
100	0.14	0.24	0.25	0.24

Table 5.12: Normalized average scores for the Required Properties agent in Tileworld.

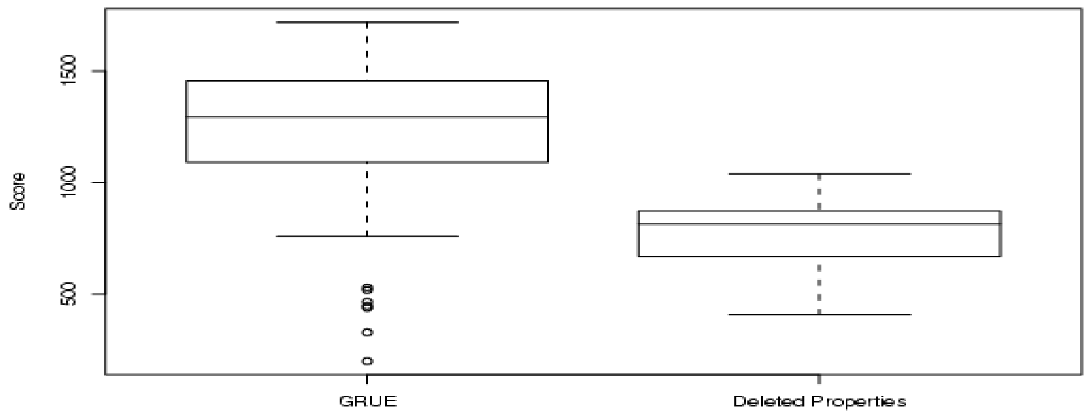

FIGURE 5.9: Boxplot comparison of the raw score data for the DELETED PROPERTIES agent and the GRUE agent when N=100 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outlying data points.

More generally, the DELETED PROPERTIES agent shows better performance in fast, low density environments, while the REQUIRED PROPERTIES agent performs better in slow changing, high density environments. Since there are fewer objects to choose from in a low density environment, it makes sense to let the agent use whatever objects it can sense without any restrictions. In high density environments, the agent can be more discriminatory as the likelihood of an object meeting all of the requirements is much higher. Overall, the DELETED PROPERTIES agent outperforms the REQUIRED PROPERTIES agent.

It is better to relax requirements than to make them more strict. Requiring all of the properties has the result that much of the time the agent is unable to locate an appropriate object and thus wanders randomly. Deleting the properties means that the agent may miss some opportunities for bonuses, but it will nearly always be able to score something.

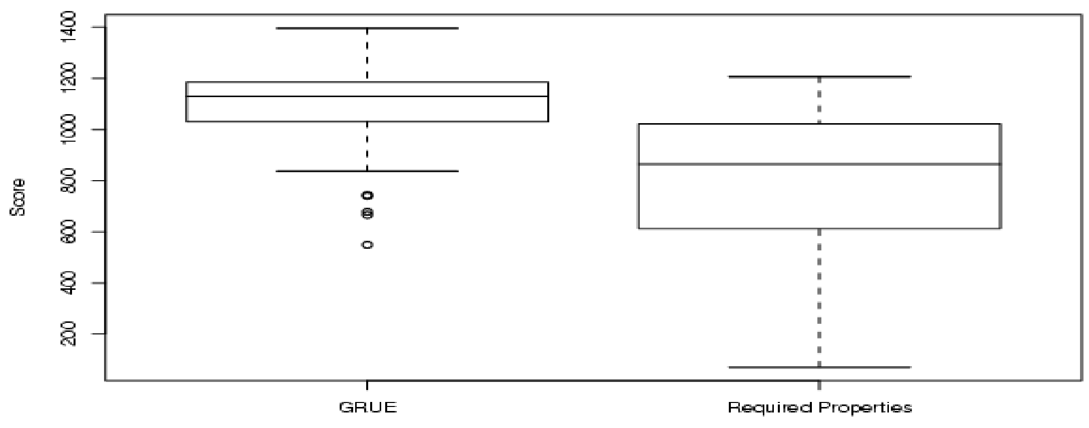

FIGURE 5.10: Boxplot comparison of the raw score data for the REQUIRED PROPERTIES agent and the GRUE agent when N=100 and Density=70. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers.

Based on this data, we recommend against using very stringent combinations of required properties whether or not preferred properties are being used. The one exception to this rule is a case where the environment has a very high density and remains dense over time. In our experiments, the objects are not generated as fast as the agent uses them so the density gradually declines. However, if the density remained constant, the agent would be able to meet the more stringent requirements more often and would therefore waste less time looking for appropriate items.

The GRUE agent outperforms the other two agents in all environments. Results of t-tests (shown in G.9 and G.10) indicate that the difference between these agents and the GRUE agent is significant for all environmental conditions. This matches our expectation that using preferred properties will increase performance in situations where there are a number of resources that can be used for the same purpose and the use of a subset of those resources will result in a higher score. Figure 5.9 shows a boxplot comparison between the Deleted Properties agent and the GRUE agent for the case where density is 40 and N is 100, which is where the Deleted Properties agent performs best. Likewise, Figure 5.10 shows a boxplot comparison between the Required Properties agent and the GRUE agent for the case where N is 100 and density is 70. Both boxplots show that the GRUE agent performs better than the other agent.

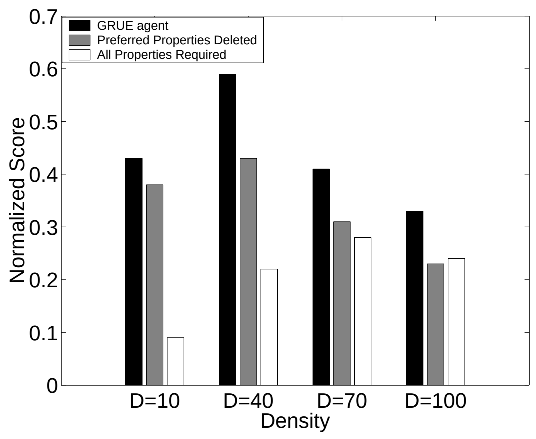

Figure 5.11: The GRUE agent compared to agents without preferred properties for N=100.

Figure 5.11 presents a comparison of the standard GRUE agent to the two agents with no preferred properties for an N value of 100, which includes the peak performance of all three agents.

5.1.4 Mechanisms for Manipulating Numerical Values

GRUE incorporates two mechanisms for handling numerical property values in resources. The first uses the DIV keyword to specify that the resource can be divided into groups. We can test this by simply removing the DIV keyword from any resources that have it. In

Tileworld, the divisible resources are tile stacks. By removing the DIV keyword, we force the agent to drop the entire tile stack, possibly wasting tiles.

The second mechanism is our notation for requesting a value within a range. We created an agent without ranges by removing all range properties from the resource variables in the agent's programs. The Tileworld agent uses ranges in several places. When looking for a tile stack, it uses ranges to specify the largest possible size and the lowest possible distance value. When looking for a hole, it uses ranges to specify the smallest possible size (so it is more likely to fill the hole) and the smallest possible distance. When actually dropping the tiles, it use ranges to bind a number of tiles less than or equal to the depth of the hole (avoiding wasted tiles).

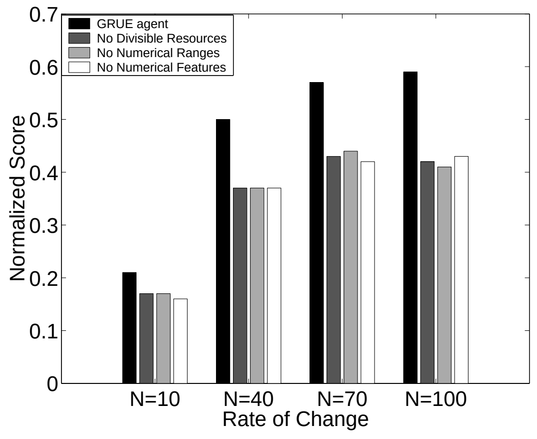

FIGURE 5.12: The GRUE agent compared to agents without numerical features with density set to 40.

We have tested an agent without divisible properties, which we will call the NO DIVISIBLE agent, an agent without ranges, which we will call the NO RANGES agent (for no ranges), and an agent lacking both divisible properties and ranges, which we will call the NO NUMERICAL agent because it uses none of the numerical features included in GRUE.

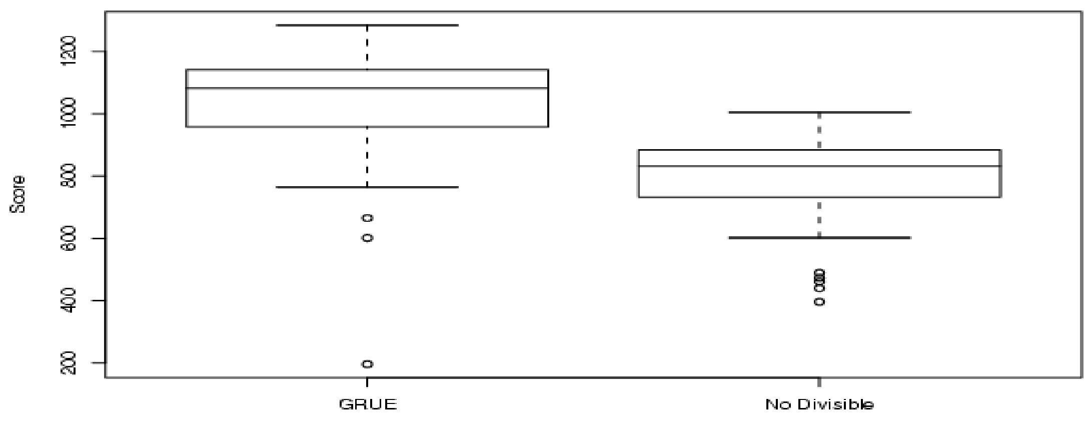

FIGURE 5.13: Boxplot comparison of the raw score data for the NO DIVISIBLE agent and the GRUE agent when N=70 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers.

We compare all three agents to the standard GRUE agent.

The NO RANGES agent cannot use ranges, so all uses of the Distance property are replaced by the Nearest property [4]. Nearest has a value of true, and is present only when the object in question actually is the nearest out of the set of objects of that type that the agent knows about. When filling a hole, the NO RANGES agent specifies the exact size of the hole instead of using a range. This means that the agent is unable to partially fill a hole. The agent must instead choose a hole that is smaller than or of equal size to the tile stack being carried. When an agent with no ranges uses the DIV notation, the agent may choose a hole of smaller size than the tile stack it is carrying and expect the tile stack to be split and the hole filled exactly. If the use of the DIV keyword is also eliminated, then the agent is more restricted and must choose only holes that exactly match the size of the

[4]Our syntax does not allow the use of comparison operators in resource variables. Without using ranges, we are limited to exact value matches except when using divisible resources.

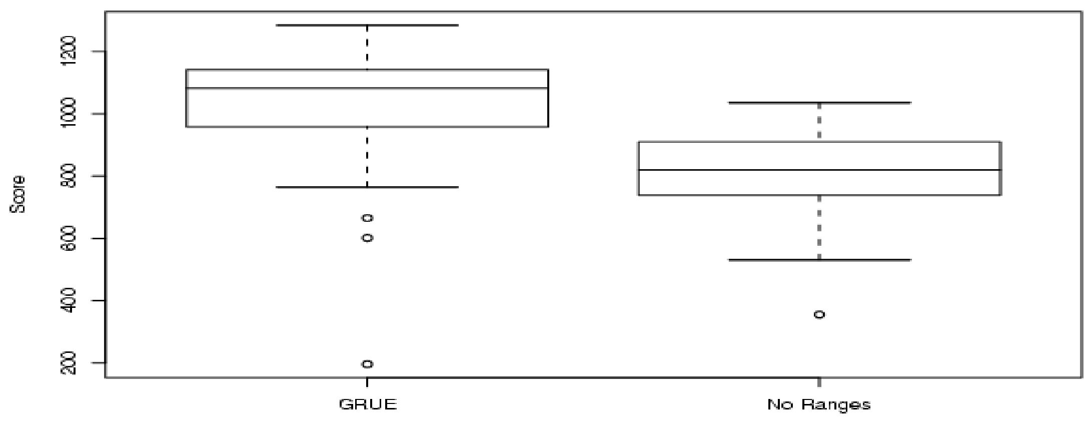

FIGURE 5.14: Boxplot comparison of the raw score data for the NO RANGES agent and the GRUE agent when N=70 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers.

tile stack.

Figure 5.12 shows the performance of the NO RANGES, NO DIVISIBLE, NO NUMERICAL and GRUE agents for density=40. The NO RANGES and NO DIVISIBLE agents both perform less well than the GRUE agent in general, though they perform almost the same as each other. Both have their peak performance at a density of 40 and N value of 70, which is interesting as the GRUE agent peaks with the rate of change equal to 100, as does the NO NUMERICAL agent. The NO NUMERICAL agent also performs quite similarly to the NO RANGES and NO DIVISIBLE agents. The normalized scores for NO RANGES, NO DIVISIBLE and NO NUMERICAL agents in all environments are shown in Table 5.14, Table 5.13, and Table 5.15. All three agents demonstrate significantly different performance from the GRUE agent except in the case where N=70 and density is 100 (see Tables G.11, G.12 and G.13.

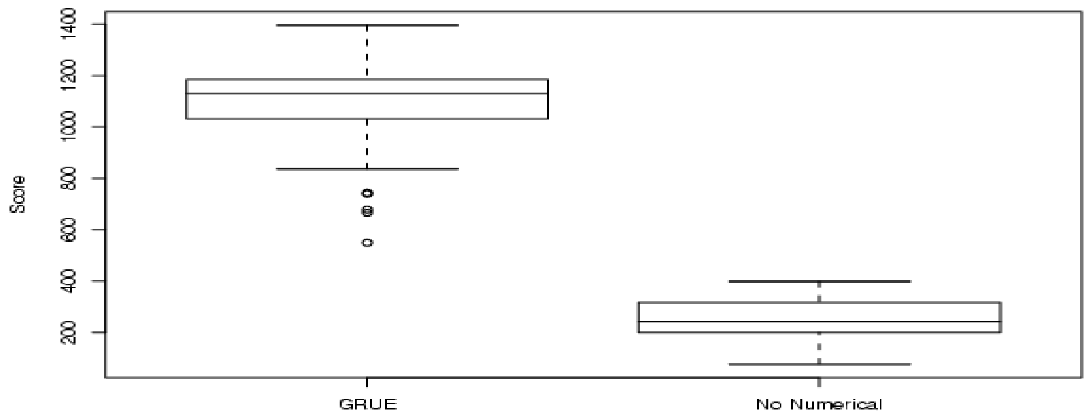

FIGURE 5.15: Boxplot comparison of the raw score data for the NO NUMERICAL agent and the GRUE agent when N=100 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

Figure 5.13 displays a boxplot comparison for the NO DIVISIBLE agent and the GRUE agent in the case where density is 40 and N=70. Figure 5.14 is the same for the NO RANGES agent. Figure 5.15 is a comparison between the boxplots for the NO NUMERICAL agent and the GRUE agent for N=100 and density=40. The boxplots show that the GRUE agent outperforms the other agents, even though these are the environmental conditions where the agents perform best. The NO NUMERICAL agent in particular performs much worse than the GRUE agent; even the GRUE agent's low outliers are higher than the upper range value for the NO NUMERICAL agent.

We draw the conclusion that for our Tileworld agent the use of divisible properties and the representation of ranges are useful, but only produce a beneficial effect when used together. This is due to the fact that the only use of the DIV keyword in the standard Tileworld agent is combined with a range. There are other uses of ranges in the programs, but they have obvious equivalents (such as replacing a distance of #| 0 <− :# with a binary property that specifies whether the object is the closest to the agent). Thus, the three cases are essentially equivalent for the Tileworld agent.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.09	0.25	0.34	0.33
40	0.17	0.37	0.43	0.42
70	0.20	0.32	0.33	0.34
100	0.19	0.24	0.28	0.25

Table 5.13: Normalized average scores for the No Divisible agent in Tileworld.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.09	0.24	0.32	0.33
40	0.17	0.37	0.44	0.41
70	0.20	0.32	0.33	0.36
100	0.20	0.25	0.28	0.29

Table 5.14: Normalized average scores for the No Ranges agent in Tileworld.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.08	0.25	0.33	0.34
40	0.16	0.37	0.42	0.43
70	0.19	0.32	0.32	0.37
100	0.20	0.26	0.27	0.25

Table 5.15: Normalized average scores for the No Numerical agent (no numerical features) in Tileworld.

5.1.5 Shared and Exclusive Binding of Resources

We tested two different aspects of binding. The first is the ability to make exclusive bindings. We will call this agent the Non-exclusive agent. This agent simply skips the creation of exclusive resources, which means that the resources are left available for any other TRPs to use. The second aspect we tested is the architecture's ability to maintain bindings through multiple execution cycles. Normally when a resource variable is bound to a resource, the list of bindings from the previous cycle is checked, and if possible the variable is bound to the same variable as previously. We disabled this feature by simply skipping the check for previous bindings. This affects shared bindings as well as exclusive ones. We will call this second agent NO PERSISTENCE.

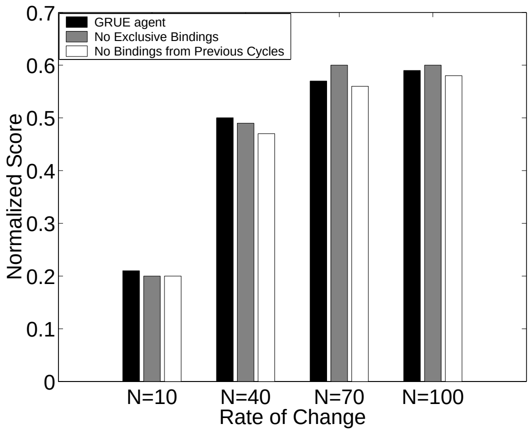

FIGURE 5.16: GRUE agent compared to agents with binding features disabled with density set to 40.

Figure 5.16 shows the data for the standard GRUE agent compared to the NON-EXCLUSIVE and NO PERSISTENCE agents with density=40 (both agents perform best at this density level). Figures 5.17 and 5.18 show boxplot comparisons for two of these four environmental conditions. We can see that disabling the check for previous bindings has no particular effect on the agent's performance, although the boxplots indicate that the GRUE agent has a slight advantage when N=40 and density is 40. For most cases in Tileworld, the agent will use the same resource anyway. In higher density and faster changing environments, the agent will see new choices more often, so it makes sense that we see an increase in performance when we allow the agent to switch resources every cycle. We included this

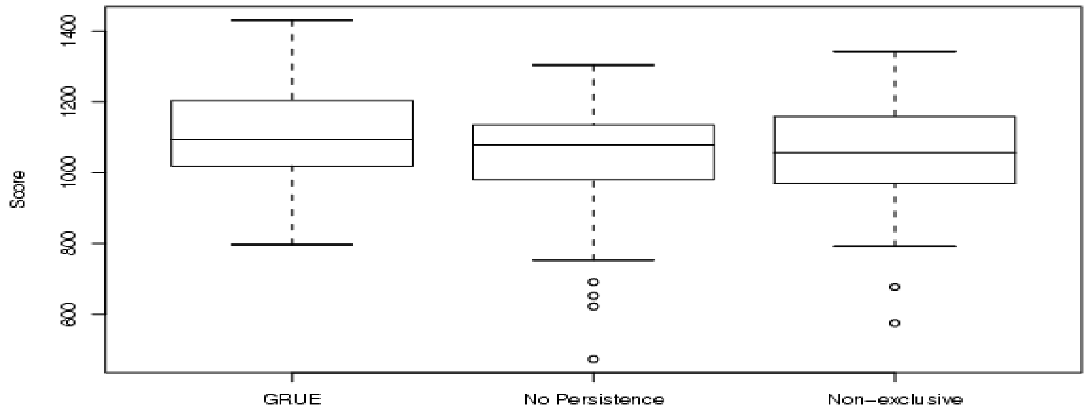

FIGURE 5.17: Boxplot comparison of the raw score data for the NON-EXCLUSIVE agent, the NO PERSISTENCE agent and the GRUE agent when N=40 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outlying data points.

feature mainly for believability reasons; a realistic character ought to pick one thing and stick with it. The experimental data does not show much of an increase in performance without it, and then only under certain environmental conditions.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.11	0.30	0.43	0.42
40	0.20	0.49	0.60	0.60
70	0.24	0.42	0.48	0.47
100	0.24	0.35	0.40	0.38

TABLE 5.16: Normalized average scores for the NON-EXCLUSIVE agent in Tileworld.

Hence in Tileworld the agent can safely use the persistence feature without overly detrimental effects on its performance. Not only that, but allowing persistent bindings causes the agent to run faster, as the run times for the No Persistence agent shown in Table 5.18 are larger than those for the standard GRUE agent, and the difference is significant

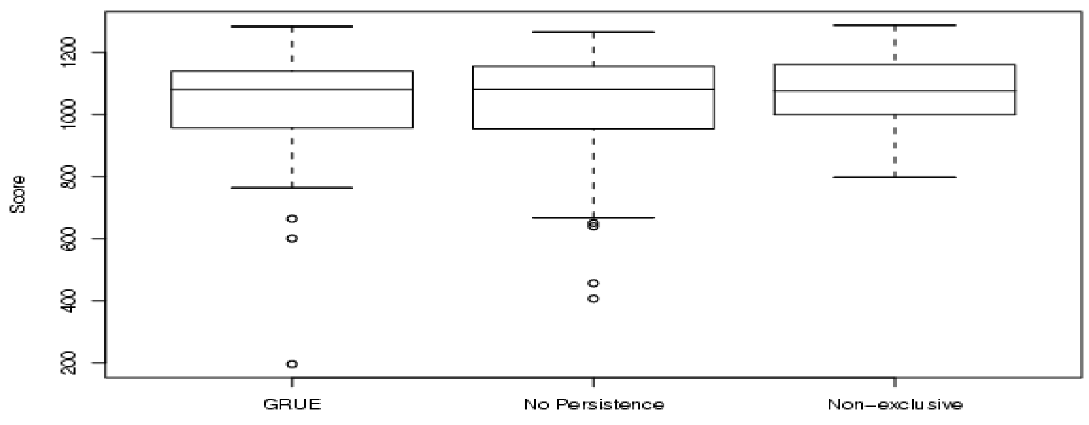

FIGURE 5.18: Boxplot comparison of the raw score data for the NON-EXCLUSIVE agent, the NO PERSISTENCE agent and the GRUE agent when N=70 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.11	0.31	0.42	0.44
40	0.20	0.47	0.56	0.58
70	0.24	0.39	0.39	0.41
100	0.23	0.31	0.32	0.31

TABLE 5.17: Normalized average scores for the NO PERSISTENCE agent (no persistent bindings) in Tileworld.

in all except one case (see Table G.16). This is because using a binding from the previous cycle allows the binding algorithm to avoid searching the list of all available resources. With persistent bindings disabled, the binding algorithm must search the entire list every time.

Interestingly, in the high density cases the NON-EXCLUSIVE agent performs better than the standard GRUE agent as can be seen in Table 5.16. The results of t-tests on this data indicates that the difference is significant at the 10% confidence level (see Table G.14). During the implementation, we formulated a general rule that resources should be bound exclusively when they are being changed or removed. We therefore expected that removing

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	17.19	18.76	18.47	18.69
40	22.89	23.11	24.44	23.33
70	40.59	42.36	43.79	45.23
100	64.52	66.10	68.66	70.85

TABLE 5.18: Average number of seconds of processor time used for each run of the NO PERSISTENCE agent (no persistent bindings) in Tileworld.

the exclusive binding capability would lead to TRPs attempting to use incorrect resources, resulting in confused behaviour.

Initially, we assumed that we must have incorrectly designated a binding exclusive when it could have been shared. This would lead to a resource variable failing to bind, or binding to a less than optimal resource. However, investigation of the code revealed that there were no resource variables that were failing to bind due to resource being preempted by higher priority TRPs.

This is expected, as the only time two TRPs are running simultaneously in the Tileworld agent is during obstacle avoidance, and the obstacle avoidance TRP generally uses different resources. It does, however, use a resource of type direction, which stores the direction the agent is currently moving. The direction resource is stored when the obstacle avoidance TRP starts, and the TRP stops running if the direction resource is deleted. What actually happens in our GRUE agent is that one of the other TRPs binds the direction resource exclusively in order to remove it and either replace it with another direction or finish running. The avoid obstacle program contains a negated resource variable used to find out if the direction has been removed. Removing the ability to make exclusive bindings allows the negated resource variable to bind, which prevents the rule containing the negated resource variable from running.

Our implementation of TRPs for Tileworld has the property that allowing the negated resource variables to bind to resources that are being changed or removed does not result in confused behaviour. Instead, it actually increases the performance of the agent. The negated resource variables are in the obstacle avoidance TRP. The obstacle avoidance goal is generated whenever an obstacle is visible, but the TRP begins by checking for an obstacle next to the agent. If the obstacle is further away, the TRP simply exits. If the obstacle is right next to the agent, then the priority of the obstacle avoidance task is the maximum amount. The only way that the obstacle avoidance task can be preempted by another task is in the case that one of the other tasks is also at the maximum priority, which occurs when there is a hole or a tile also right next to the agent. The effect of allowing the negated resource variable to bind to a resource is that the obstacle avoidance TRP continues running when otherwise it would have terminated. This initially seems like an adverse effect; changing the direction resource may eliminate the need to avoid the obstacle at all, so continuing to run the *Avoid Obstacle* TRP would seem like a bad idea. In fact, it has a beneficial effect because of the specific situation in which it occurs, shown in Figure 5.19.

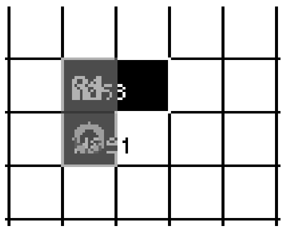

FIGURE 5.19: A case where the GRUE agent has moved horizontally and lined up with its target hole. The hole is right next to the agent, as is an obstacle. The *Fill Hole* program is removing the horizontal direction, and creating a down direction instead. If the *Avoid Obstacle* TRP runs and assumes the agent is still moving to the right, it can have the effect of moving the agent onto the target a cycle early.

Because the agent is right next to its target, attempting to move around the obstacle will cause the agent to move onto its target object 50% of the time. This saves one execution cycle. In the other 50% of cases, it will move the agent further away from the target, costing it one execution cycle. However, moving the agent further away can bring more objects into view. It can also cause the agent to move onto an alternate object which it can use instead of the target. These effects allow the agent to score higher overall. (Note that we could have avoided this entire issue by giving the obstacle avoidance task a slightly higher maximum priority, preventing it from ever losing resources to the other tasks.) This effect can be compared to the unexpected effect of eliminating the goal update feature, which we discussed in Section 5.1.2 and is caused by similar circumstances.

This investigation leads to an important observation. In Chapter 3 we stated that a negated resource variable can be used to specify the condition that a resource does not exist. We did not make a distinction between a resource not existing at all and a resource being unavailable to bind because it is in use by a higher priority task. In the case of the Tileworld agent, we assumed that the negated resource variable in the avoid obstacle program would evaluate to true when the resource does not exist, but actually it is evaluating to true when the resource is already bound. This is essentially an issue with the current implementation of GRUE; it would be possible to change the binding algorithm such that negated resource variables are allowed to bind to resources that are already bound exclusively, however we are not sure that this is a good idea. It may be better to create a separate function to check for the existence of a resource. For our current purposes, we will restate the effect of a negated resource variable. A negated resource variable is equivalent to asking whether a resource is available to be used, and evaluates to true when the resource is not available.

5.1.6 Execution of Multiple Actions During One Cycle

In order to evaluate the utility of performing multiple actions in parallel, we made a distinction between external actions and internal actions. An external action is one that is executed in the external world, and can affect that world accordingly. In contrast, internal actions only affect the agent's memory. Moving to a new square in the Tileworld, picking up a tile stack or dropping tiles are all external actions. Storing or deleting information in memory such as the direction the agent is currently moving are internal actions. Our initial experiments with disabling the execution of multiple actions showed that it is impractical to restrict internal actions this way. The GRUE Tileworld agent often needs to do several internal actions at once in order to keep track of what it is currently doing. Preventing it from executing these actions simply causes it to become confused and stop doing anything. Therefore, we have only disabled the execution of multiple external actions.

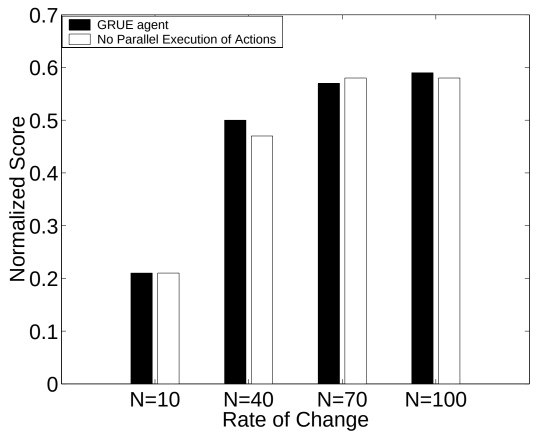

FIGURE 5.20: GRUE compared to an agent with parallel execution of actions disabled with density set to 40.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	0.10	0.28	0.42	0.43
40	0.21	0.47	0.58	0.58
70	0.23	0.36	0.41	0.40
100	0.23	0.30	0.32	0.30

TABLE 5.19: Normalized average scores for an agent that cannot perform more than one external action each cycle in Tileworld.

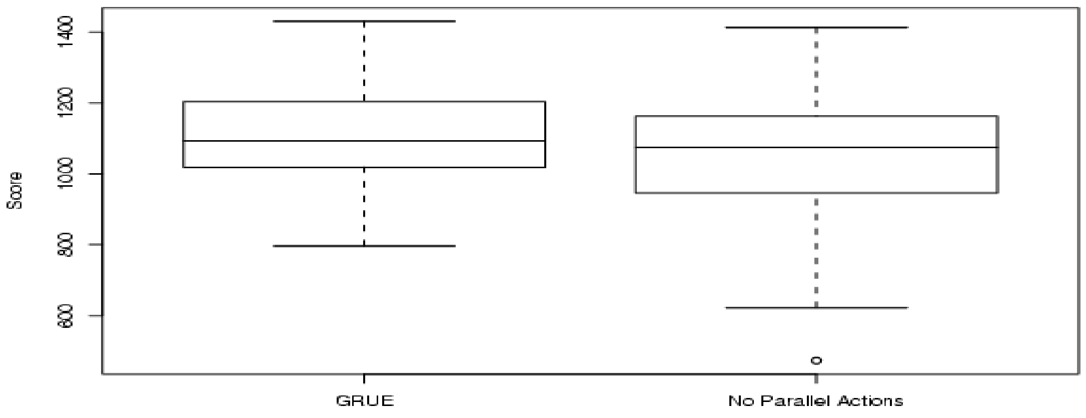

FIGURE 5.21: Boxplot comparison of the raw score data for an agent with parallel execution of actions disabled and the GRUE agent when N=40 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

The Tileworld environment has a limited set of external actions: the agent may move, pick up a tile stack, or drop tiles. These actions conflict with each other — if the agent is picking up tiles it must stay on the same square as the tile stack, and of course it makes no sense to pick up and drop tiles simultaneously. The agent's performance should, therefore, not change when we disable the parallel execution of external actions. Figure 5.20 shows the data for both agents when the density is 40, including the largest normalized scores, and Table 5.19 contains the complete set of normalized average score data for the agent with parallel execution of actions disabled. The performance of this agent is, as expected,

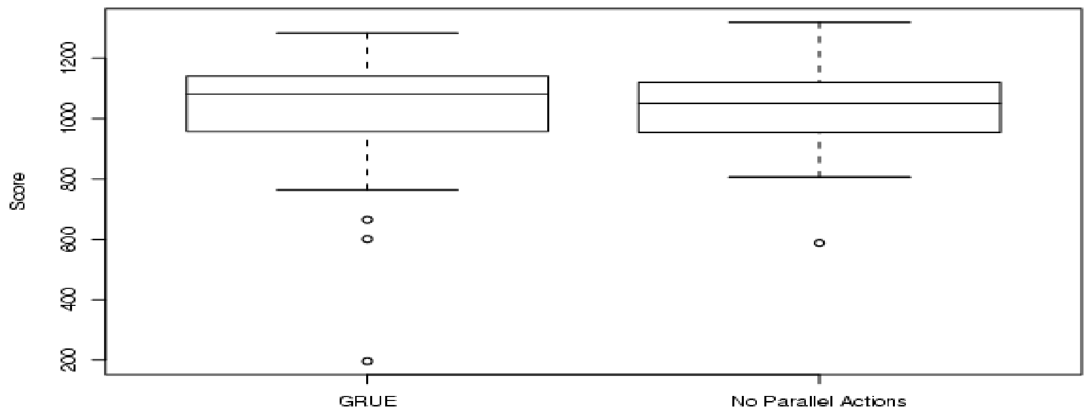

FIGURE 5.22: Boxplot comparison of the raw score data for an agent with parallel execution of actions disabled and the GRUE agent when N=70 and Density=40. The boxplot shows the median, upper quartile, lower quartile, and range of the data. Circles represent outliers.

quite similar to the performance of the standard GRUE agent. Figures 5.21 and 5.22 show boxplot comparisons for this agent and the GRUE agent with density set to 40 for both N=40 and N=70. The first boxplot suggests that the GRUE agent might have a slight advantage, while the second boxplot indicates that the two agents perform very much the same.

The t-tests for this condition indicate that this agent performs significantly worse than the GRUE agent in the low density, fast rate of change corner of the table. In most of the rest of environments there is no significant difference between this agent and the GRUE agent.

5.1.7 Performance of a Non-GRUE Agent in Tileworld

We also tested a simple non-GRUE agent. This agent has no representation of goals at all. Instead it simply uses a set of Sim_Agent rulesets to control its behaviour. It always selects the nearest tile or hole as its current target; having no way of dividing tile_stacks it always drops an entire tile_stack when it reaches a hole. Navigation is done by rating the surrounding squares. A square is given a high score if it is blocked, or if the agent has just moved off that square. Squares that take the agent in the direction of the target are given low scores. The agent moves onto the square with the lowest score. If the agent cannot find a tile or a hole, it will wander randomly until one becomes visible.

From Table 5.20 we see that the non-GRUE agent is not dissimilar to the GRUE agent. However, it is clear that the GRUE agent performs better in low density environments, making more efficient use of the available tiles.

Density		**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	Average Score	428.72	292.96	244.72	223.74
	Standard Deviation	132.90	89.44	52.38	63.25
	Normalized Average Score	0.11	0.26	0.34	0.31
40	Average Score	877.94	836.88	770.72	779.44
	Standard Deviation	138.51	117.40	107.15	85.34
	Normalized Average Score	0.27	0.38	0.42	0.43
70	Average Score	1423.02	1314.58	1244.96	1291.62
	Standard Deviation	177.20	228.25	269.28	217.92
	Normalized Average Score	0.23	0.40	0.43	0.44
100	Average Score	1859.52	1787.24	1729.30	1710.58
	Standard Deviation	195.09	312.71	297.83	386.43
	Normalized Average Score	0.26	0.41	0.44	0.43

TABLE 5.20: Scores for the non-GRUE agent in Tileworld.

In certain of the high density cases, the standard deviation of the scores for the GRUE agent is much higher than for the non-GRUE agent. This indicates that the GRUE agent is performing better than the non-GRUE agent some of the time, but also has a higher percentage of low scores. This is probably due to the GRUE agent spending much time moving around the Tileworld to reach its ideal object. In terms of execution time, the non-GRUE agent is much less affected by density than the GRUE agent because it doesn't store (and search through) all of the objects it has seen. The range is from about 50 cycles per second down to 40 cycles per second, making the non-GRUE agent actually slightly slower than the GRUE agent in low density cases.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	19.54	19.34	19.09	17.97
40	20.98	20.8	20.43	21.09
70	21.74	21.91	21.23	21.94
100	23.06	23.57	23.42	23.62

TABLE 5.21: Average number of seconds of processor time used for 1000 cycles of the non-GRUE agent in Tileworld.

5.1.8 Summary of Evaluation in Tileworld

We compared the performance in Tileworld of a standard GRUE agent to a number of other agents based on the GRUE agent but missing one or more features. The GRUE agent outperformed the agents without preferred properties, without mechanisms for manipulating numbers, and agents which keep all of the goals in the arbitrator at all times. The performance of the standard GRUE agent was similar to that of an agent without the ability to execute multiple actions during each cycle, and an agent in which bindings do not persist to subsequent cycles. All of these results were expected due to the nature of the Tileworld agent.

Unexpectedly, three agents outperformed the GRUE agent in some cases. These were the agent with exclusive bindings disabled, the agent with constant priority goals, and the agent with dynamic updating of goal priorities disabled. These results stemmed from the fact that all three of the goals in Tileworld have the same maximum priority. This can cause problems if a *Get Tile* task or a *Fill Hole* task ends up before the *Avoid Obstacle* task in the tasklist. We gave a full analysis of these cases above.

Finally, we considered a non-GRUE agent that uses a different strategy. The GRUE agent outperforms the non-GRUE agent in low density environments. The GRUE agent does have larger standard deviations than the non-GRUE agent, indicating that it shows more variation in its scores.

Patterns in the Data

Using the difference in scores in conjunction with the results of the t-tests, it is possible to identify some patterns in the data. Here we restate the list of features of the architecture along with patterns associated with those features.

Goals are generated only in appropriate situations.

> Eliminating this feature results in reduced performance in all environmental conditions.

Goal priorities are updated continuously.

> For our Tileworld agent, this feature has little or no effect in low density, fast rate of change conditions. When the density increases or the rate of change is slowed, eliminating this feature actually increases performance. However, we think this result is due to the way our goal generators are written and may not be generally applicable to other agents.

Goal priorities are set according to the current situation.

Eliminating this feature by using constant, preset priorities increases the performance of the Tileworld agent in higher density conditions. However, this feature also eliminates the updates of goal priorities. As a result, this result is attributable to the same issue with our goal generators that affected the evaluation of the goal priority updates feature, so we think it is probably not generally applicable to other agents.

Plans request resources in terms of required and preferred properties.

Both possible ways of eliminating this feature reduce the performance of the agent in all environmental conditions that we tested.

A plan can specify that it needs exclusive access to a resource.

Eliminating this feature made no little or no difference in lower density environments, but increased the performance of the agent in high density environments. Our analysis shows that this is due to an unexpected interaction between our TRPs.

A plan that makes the same request during consecutive cycles will receive the same resource each time.

It is difficult to identify a pattern in this dataset. However, we do see a few cases where eliminating this property increases the performance of the agent which occur in the high density, quickly changing environments. We see a reduced performance or no change for eliminating this feature in low density or slowly changing environments.

If this is a valid pattern, then we can perhaps make a general rule that this feature is not useful for the Tileworld in the fast changing high density cases. We should note that the design of our Tileworld agent means that it will opportunistically use non-optimal resources that it happens to walk over. We suspect that in low density environments switching between resources will result in the agent walking back and forth between resources without actually making progress, an effect that is reduced by the opportunistic properties of our TRPs in high density environments. Therefore the utility of this feature is likely to be highly dependent on the properties of the TRPs used.

Numerical quantities can be divided and used for more than one goal.

Eliminating this property reduced the performance of the Tileworld agent in all of the environmental conditions except for the specific case N=70 with density=100, where we see no significant difference. This may indicate an interesting result for densities greater than 100.

Plans can use special notation to request numerical values within a range.

This case is similar to the last case: eliminating this property reduced the performance of the Tileworld agent in all of the environmental conditions except for the specific case N=70 with D=100, where we see no significant difference. Again, this may indicate an interesting result for densities greater than 100.

Multiple actions can be executed during each execution cycle.

Eliminating this feature reduces the performance of the agent in low density, quickly changing environments. The effect is not so clear in the rest of the environmental cases. In two of the high density cases, it actually increases performance while it has no significant effect in most of the remaining cases.

Finally, we can see that the strategy used by the non-GRUE agent results in lower average scores for the lower two density values, but higher average scores with the highest density value. The standard deviations for this agent are smaller, indicating that this strategy is more consistent than the strategy used by GRUE.

Other Observations

This evaluation did lead to several interesting observations. It is always best to use as few required properties as possible. Including too many requirements leads to an inefficient agent which will waste time searching for a resource that meets all of the requirements instead of using the resources that are actually present. Secondly, in our implementation of GRUE a negated resource variable will evaluate to true whenever there is no matching resource available. We assumed when writing our TRPs that this would only happen when the resource does not exist. However, it also occurs when a resource has been bound by a higher priority program. Future implementations should include an additional mechanism for checking the existence of a resource because merely checking the availability may be insufficient. Finally, we observe that much depends on the actual implementation of the TRPs. Some of our unexpected interactions could have been avoided had we chosen to give the *Avoid Obstacle* goal a higher maximum priority than the other two goals. Furthermore, some of the features of GRUE had no effect due to the simple nature of the Tileworld environment. It is not necessary to allow multiple actions to be executed in parallel if it does not make sense to do so. It is also unnecessary to include situational generation and dynamic updating of goal priorities if the set of goals is small and has an obvious priority hierarchy. Furthermore, the processor time taken by the different versions of the agent indicates that a substantial percentage of that processor time is spent in the goal generators.

5.2 The Unreal Tournament Environment

FIGURE 5.23: A screenshot from Unreal Tournament, during a Capture the Flag game

Unreal Tournament is a first-person shooter. This genre of games features weapons-based gameplay, with several different types of game typically included. Our tests were performed in a capture-the-flag game. In this game type, each player is assigned either to the red team or the blue team. Each team has a flag base containing a flag. Players on the red team score points for their team by capturing the blue team's flag and returning it to the red base and vice versa for players on the blue team. There are various useful objects scattered around the level, including weapons and health packs. Each player has a maximum health level of 100, and health packs restore about 20 points to this. A player whose health level drops to zero is killed and "respawns" from a different point on the level, dropping anything they were holding in the process. Items in Unreal Tournament are acquired by simply walking over them, so we do not have a separate pick-up action.

All of our tests were done using the Unreal Tournament map called CTF-simple which is available online with the Gamebots package. This is a small map, shaped approx-

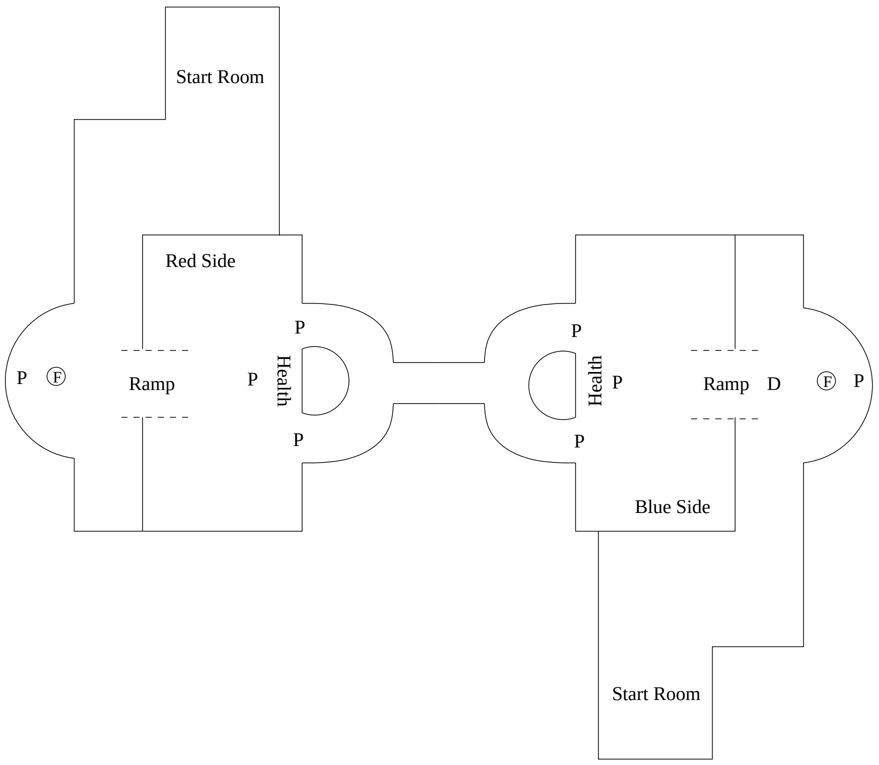

FIGURE 5.24: The layout of the CTF-simple map. The locations of health packs and the rooms where players start are marked. The circled 'F's represent the flag bases, while 'P' marks a patrol point and 'D' marks the agent's default location. This diagram is not to scale.

imately like a figure of eight. Players start (and respawn) from a room to the side of their flag base. Each flag base is raised, and can be reached from the rest of the level only by moving up a ramp. At the bottom of the ramp is a large room containing a health pack. The exits from this large room are two corridors which converge in the middle of the figure of eight before splitting up to form two entrances to the next room. The two halves of the level are exactly identical, except that one flag base is red and the other is blue.

The Unreal Tournament environment does not include resources that can sensibly be divided between tasks. The level we chose is sparse, featuring few choices of items for the agent to acquire. Due to the nature of the Unreal Tournament, we chose not to test the use of preferred properties, divisible resources, or value ranges in this environment. Instead, we focus on GRUE's goal arbitration features. Our Unreal Tournament agent can generate 11 goals, and up to 8 of those simultaneously. This is as compared to only 3 goals that can be generated in Tileworld. Figure 5.25, which was also shown in Section 4.4 shows the possible priority levels of the UT agent's goals.

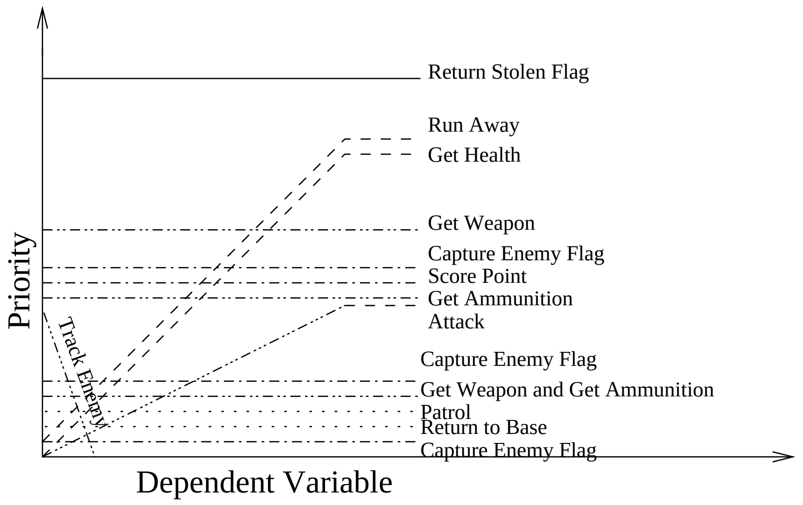

FIGURE 5.25: Relative priorities for goals in Unreal Tournament. The horizontal axis represents the dependent variable, which is different for each goal. Goals which have different priority functions in different circumstances are shown as several lines on the graph.

Like our tests in Tileworld, our tests in Unreal Tournament show the standard GRUE agent and a number of reduced-functionality versions of the agent. The agents we test in this environment are the ALL GOALS agent (all goals are always generated), the NO UPDATES agent (no goal priority updates), the CONSTANT PRIORITIES agent (goals are generated with preset priority values), the ALL GOALS CONSTANT PRIORITIES agent (all goals are generated in all conditions with preset priorities), the NON-EXCLUSIVE agent (no exclusive bindings allowed), the NO PERSISTENCE agent (the arbitrator does not attempt to use bindings from the previous cycle), and an agent with parallel execution of actions disabled.

In our discussion of the data from Tileworld, we used t-tests to give an indication of whether the difference between the experimental agents and the GRUE agent is significant. Unlike our Tileworld data, the sample sets collected in Unreal Tournament are so small that a t-test will not be very accurate (Moore and McCabe, 1993). We have instead provided standard deviations to give an indication of how consistent the results were. We also use boxplots to give a visual summary of the data.

Also unlike our Tileworld discussion, we do not provide a comparison to a non-GRUE agent for the Unreal Tournament environment. The game Unreal Tournament does come with computer players; however the Unreal Tournament has a large range of possible actions. Computer players for this type of game use a wide variety of strategies and may use only a subset of the available actions. We felt that any comparison between our agents and computer players implemented by someone else would be meaningless due to the difference in strategies used. The computer players provided with the game are particularly problematic as we have no information about how work. The agents we know of that have been developed by other researchers were designed to solve such completely different problems that no comparison is possible. (For instance, (Hawes, 2003) deals with anytime planning, and (Vu et al., 2003) with multi-agent strategies.)

5.2.1 Games with No Opponent

We tested the performance of the standard GRUE agent and different reduced functionality versions of the GRUE agent in an empty game. With no opponent, the goals that can be generated are `Patrol`, `Return to Base`, `Capture Enemy Flag`, `Score Point`, `Get`

`Ammunition` and `Get Weapon`. Since the agent will never use its ammunition, the `Get Ammunition` and `Get Weapon` programs should always be generated with low priority values except for the case at the beginning of the game when the agent has not yet picked up a weapon. The empty game also means that the agent will never encounter a situation where it needs to update a goal. (Most goal updates are based on changes in health levels or the proximity of an opponent.) For each of these tests the agent played 15 games, each of which lasted 5 minutes. This gives us some baseline data about how many goals each agent is capable of scoring in 5 minutes. Table 5.22 shows the results for the standard GRUE agent. Note that most of these values are presented per cycle of the GRUE architecture. We collected the average per-cycle values in each of the 15 games, and then took the mean of the averages to get the results displayed here.

We present the number of times the agent scored a point for the team by capturing the opposing team's flag and touching it to the agent's base, which we have described as a goal in the following data tables. We also present the average number of resources in the agent's memory during each cycle, the average number of tasks in the arbitrator during each cycle, the average number of new goals generated during each cycle, the average number of goals in the arbitrator that were updated during each cycle, and the average amount of processor time used for each cycle (measured in seconds).

Table 5.23 shows the results for an agent will all goals generated in all situations. This will cause `Get Ammunition` and `Get Weapon` goals to be generated when the agent cannot actually see any weapons or ammunition. It will also cause the `Score Point` goal to be generated when the bot does not have the flag. These TRPs will start running and

Standard GRUE Agent		
	Average	Standard Deviation
Goals Scored	2.67	0.72
Resources per Cycle	31.73	5.23
Tasks per Cycle	3.54	0.17
Goals Generated per Cycle	1.49	0.01
Seconds of Processor Time per Cycle	0.06	0.03

Table 5.22: Average values for the standard GRUE agent in Unreal Tournament with no opponent, rounded to 2 decimal places.

All Goals Agent		
	Average	Standard Deviation
Goals Scored	2.87	0.74
Resources per Cycle	34.96	5.58
Tasks per Cycle	6.04	0.07
Goals Generated per Cycle	4.90	0.07
Seconds of Processor Time per Cycle	0.08	0.01

Table 5.23: Average values for an agent with all goals generated in all situations in Unreal Tournament with no opponent, rounded to 2 decimal places.

immediately exit as their success conditions are already met. This agent will also always generate the `Patrol` goal, which is normally only generated when the bot is at its default location, but again the TRP will leave the arbitrator right away. Note that while this agent does not check the current situation before generating a goal, it does check that the goal has not already been generated. If a goal is already in the arbitrator, it won't be generated again. The data shows that the agent does indeed generate more goals than the standard GRUE agent, and has more tasks in the arbitrator during each cycle. The average processor time per cycle is higher than for the GRUE agent, but only very slightly.

In Tileworld and in games where the agent plays against an opponent in Unreal Tournament we also tested versions of the GRUE agent with dynamic updating of goal priorities disabled, with constant goal priorities, and with all goals generated in all situations with constant priorities. We did not test these versions in an empty game as they will perform almost identically to the standard GRUE agent. Updates of goal priorities in UT are triggered by events involving other agents. The `Attack`, `Get Health`, and `Run Away` goals are examples. Of the goals that can be generated in an empty game, `Return to Base` and `Patrol` always have constant priorities. `Get Weapon` and `Get Ammunition` can be generated with high priorities if the agent does not have a weapon or is out of ammunition, but this will only occur right at the start of an empty game. `Capture Enemy Flag` has a higher priority when the flag is nearby, but as the agent is not doing anything except getting the flag, it will make no difference to the agent's behaviour.

No Persistence Agent		
	Average	Standard Deviation
Goals Scored	3.07	0.26
Resources per Cycle	32.20	3.65
Tasks per Cycle	3.50	0.14
Goals Generated per Cycle	1.49	0.01
Seconds of Processor Time per Cycle	0.15	0.02

TABLE 5.24: Average values for an agent without persistent bindings in Unreal Tournament with no opponent, rounded to 2 decimal places.

Non-exclusive Agent		
	Average	Standard Deviation
Goals Scored	2.73	0.8
Resources per Cycle	31.61	6.18
Tasks per Cycle	3.43	0.12
Goals Generated per Cycle	1.49	0.01
Seconds of Processor Time per Cycle	0.07	0.01

TABLE 5.25: Average values for an agent without exclusive bindings in Unreal Tournament with no opponent, rounded to 2 decimal places.

Table 5.24 shows the data for an agent without persistent bindings in an empty Unreal Tournament game, while Table 5.25 shows data for an agent with exclusive bindings disabled. There are no cases in our Unreal Tournament implementation where two programs use the same resource. There are also very few cases where an agent might have a choice between two resources (unless there are several other players in the game) because the environment is quite sparse. As a result, disabling these resource binding features has no effect on the performance of the agent. It is worth noting, however, that the average cycle time for the agent without persistent bindings is twice as long as for the other agents tested. This is due to the fact that using a binding from the previous cycle enables the binding algorithm to avoid searching the complete list of resources to find an appropriate binding.

Agent with Parallel Execution of Actions Disabled		
	Average	Standard Deviation
Goals Scored	3.80	0.41
Resources per Cycle	32.74	6.90
Tasks per Cycle	3.66	0.16
Goals Generated per Cycle	1.49	0.01
Seconds of Processor Time per Cycle	0.05	0.01

TABLE 5.26: Average values for an agent that can only execute one external action per cycle in Unreal Tournament with no opponent, rounded to 2 decimal places.

The performance of an agent with parallel execution of external actions disabled is shown in Table 5.26. We notice that the average number of goals scored in the 5 minute game is 3.8, as compared to only 2.66 for the standard GRUE agent. In an empty game situation, there are few possibilities for parallel action execution. In most of the cases where actions are executed in parallel, the actions in question are requests for paths from the server. One explanation for the data is that the paths requested by lower priority tasks can be used by those tasks during cycles when higher priority programs are not executing movement commands. This seems unlikely, as the `Score Point` and `Capture Enemy Flag` goals will have the highest priorities in almost all situations, and they execute commands

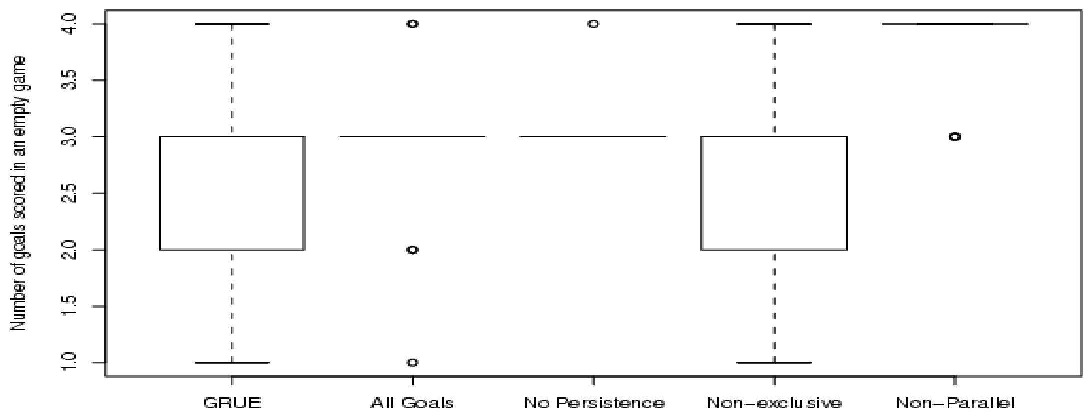

FIGURE 5.26: Boxplot comparison of the number of goals scored by agents in an Unreal Tournament game with no opponent. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

that will conflict with lower priority movement commands except for one or two cycles at the end when they are cleaning up stored information. On the other hand, a single *RUNTO* command in an unproductive direction could slow the agent down (depending on how long it takes for the agent to execute another command and for the game server to receive and process the new command).

It is also possible that the standard GRUE agent actually performs better than reflected in this data. In a second trial of 9 runs, the standard GRUE agent scored 4 points in 8 out of the 9 games for an average of 3.89. Another possibility is that other processes running on the machine used for collecting the data prevented the standard GRUE agent from running as many cycles as it might have done otherwise.

The standard deviations for the data collected in games with no opponent follows the same pattern for all the agents. The number of goals generated per cycle has a small standard deviation, showing that this number is fairly consistent. The standard deviation for the amount of processor time used is also small, but as the average value itself is small we can see that actually this value varies considerably. The number of tasks per cycle is fairly consistent, while the number of goals scored varies more (this value might have been more consistent if we had allowed the agents to play longer games). The number of resources also varies quite a bit, although this is by far the largest value. If we look at the standard deviation as a percentage of the average value for the standard GRUE agent, we see that the amount of processor time used varies the most, followed by the goals scored and the number of resources generated each cycle.

Figure 5.26 shows a boxplot comparison for the number of goals scored in an empty game. Four of the five agents have a median value of 3, the exception being the agent with parallel actions disabled which has a median of 4. The boxplots for the NON-EXCLUSIVE agent looks the same as the boxplot for the standard GRUE agent. Both of these agents had a range of scores from 1 up to 4. Of the other three agents, the agent with parallel actions disabled and the NO PERSISTENCE agent have no scores below three, while the ALL GOALS agent only scored below three twice. While the sample size is too small to say whether these results are significant, the boxplots suggest that for the empty game task the NON-EXCLUSIVE agent is equivalent to the GRUE agent and that the agent with parallel actions disabled has an advantage in this environment. There is not enough data to say anything about the NO PERSISTENCE and ALL GOALS agents; they could be the same as the GRUE agent for this task, or they could perform slightly better.

Game Number	1	2	3	4	5	6	7	8	9	10	**Avg.**	**Std. Dev.**
Team Scores												
First Agent	0	0	1	0	0	0	3	0	1	2	0.70	1.06
Second Agent	0	1	0	0	2	0	0	0	1	3	0.70	1.06
Difference	0	1	1	3	1	-1	-1	-1	-1	-1	0	1.33
Individual Scores												
First Agent	5	9	8	1	1	0	21	3	18	25	9.10	9.09
Second Agent	11	16	0	1	31	4	1	14	13	13	10.40	9.48
Difference	6	7	-8	0	30	4	-20	11	-5	-12	1.30	13.90
Deaths												
First Agent	5	6	1	1	5	1	2	6	3	2	3.20	2.10
Second Agent	7	10	2	1	1	2	1	3	3	7	3.70	3.16
Difference	2	4	1	0	-4	1	-1	-3	0	5	0.50	2.80

TABLE 5.27: Results for two standard GRUE agents playing a Capture the Flag game in Unreal Tournament.

5.2.2 Two GRUE Agents Opposing Each Other

This section presents the results from 10 games with one standard GRUE agent on each team. Table 5.27 displays the team scores, individual scores, and number of deaths for each agent. The team score is equal to the number of successful flag captures (points scored). An agent's individual score is increased if it successfully scores a point for the team, or kills another player. The agent's individual score is decreased if the agent kills itself.

Predictably since the two agents are identical, the team scores are fairly even. The agents won approximately equal numbers of games, and the difference between the team scores for each game is small. The differences in individual scores is quite large in some cases, but we don't see cases where both agents have high scores. We conjecture that once an agent falls behind in points it is difficult for that agent to catch up. The score combines the number of times the agent kills the other agent and the number of times the agent scores a point. If an agent is killed, it will be restarted in the start room. This can cause the agent to lose time which its opponent can use to collect health and ammunition or make progress toward scoring goals. Our conjecture is supported by the number of deaths for each agent, which are even in some cases while others have quite a large difference.

5.2.3 The No Updates Agent Opposed by the Standard GRUE Agent

Game Number	1	2	3	4	5	6	7	8	9	10	**Avg.**	**Std. Dev.**
Team Scores												
GRUE	0	0	0	1	0	1	1	1	1	1	0.60	0.52
No Updates	0	1	1	4	1	0	0	0	0	0	0.70	1.25
Difference	0	1	1	3	1	-1	-1	-1	-1	-1	0.10	1.37
Individual Scores												
GRUE	1	0	6	8	0	12	15	13	7	20	8.20	6.80
No Updates	1	8	9	3	20	10	1	4	7	2	6.50	5.80
Difference	0	8	3	-5	20	-2	-14	-9	0	-18	-1.70	10.92
Deaths												
GRUE	1	1	2	4	5	2	1	6	3	2	2.70	1.77
No Updates	1	1	2	1	2	1	5	8	1	6	2.80	2.57
Difference	0	0	0	-3	-3	-1	4	2	-2	4	0.10	2.56

Table 5.28: Results for the No Updates agent playing a Capture the Flag game in Unreal Tournament against the standard GRUE agent.

Table 5.28 shows the results for a standard GRUE agent playing against an agent with dynamic updating of goal priorities disabled. Interestingly, the two agents seem to be evenly matched. From the individual scores, it seems that the GRUE agent might have a slight advantage over the other agent, but not a huge one. The number of deaths for each agent are again fairly even.

We suspect that the lack of goal priority updates results in the No Updates agent effectively pursuing a different strategy. The standard GRUE agent will notice as its health decreases during a fight. The priority of the `Get Health` goal is increased to its maximum value as soon as the agent's health drops below a threshold. This results in the standard GRUE agent leaving a fight when its health drops. The No Updates agent, on the other hand, will not update the priority of the `Get Health` goal. It effectively doesn't notice that its health is low. To see why ignoring a low health level might be an advantage, consider a case where the standard GRUE agent and the No Updates agent are shooting at each other. The GRUE agent notices its health is low and turns away from the fight to get a health pack. The No Updates agent continues shooting at the GRUE agent, killing it. Once the GRUE agent has been killed, it restarts from the other side of the level which gives the No Updates agent some time to pick up the health pack and pursue other goals like capturing the flag or scoring a point.

The strategy used by the standard GRUE agent can also be advantageous in some circumstances. Consider a case where the GRUE agent spots the No Updates agent and shoots at it from behind. The No Updates agent ignores its falling health level, and continues moving in the same direction with the result that the GRUE agent kills it.

We conclude that in a first-person shooter game, it can be better in some circumstances to disable one's opponent before looking for health items. The fact that the GRUE agent does not use this strategy is due mainly to our inexperience with this type of game.

5.2.4 The Constant Priorities Agent Opposed by the Standard GRUE Agent

We constructed the Constant Priorities version of the Unreal Tournament agent by modifying the goal generators so that all goals are always generated at what would otherwise

Game Number	1	2	3	4	5	6	7	8	9	10	**Avg.**	**Std. Dev.**
Team Scores												
GRUE	0	0	0	1	0	0	0	1	2	0	0.40	0.70
Constant Priorities	2	0	0	4	0	0	2	1	0	0	0.90	1.37
Difference	2	0	0	3	0	0	2	0	-2	0	0.50	1.43
Individual Scores												
GRUE	1	10	0	5	0	7	1	15	15	5	5.90	5.80
Constant Priorities	31	9	1	11	8	17	29	22	2	1	13.10	11.21
Difference	30	-1	1	6	8	10	28	7	-13	-4	7.20	13.34
Deaths												
GRUE	5	7	2	4	4	5	8	7	2	2	4.60	2.22
Constant Priorities	1	8	1	2	0	2	1	4	1	1	2.10	2.33
Difference	-4	1	-1	-2	-4	-3	-7	-3	-1	-1	-2.50	2.22

Table 5.29: Results for the Constant Priorities agent playing a Capture the Flag game in Unreal Tournament against the standard GRUE agent.

be their maximum priority values. The team scores in Table 5.29 shows that the Constant Priorities agent usually ties the standard GRUE agent when they play against each other. The individual scores and number of deaths indicate that the Constant Priorities agent is more likely to kill the standard GRUE agent than to be killed itself.

Two things may contribute to the Constant Priorities agents ability to kill the standard GRUE agent. One is the fact that the `Get Ammunition` goal is always generated with high priority, even when the agent is not about to run out of ammunition. While the Constant Priorities agent might break off in the middle of a fight to pick up ammunition, it will be less likely to run out of ammunition. This behaviour also prevents the Constant Priorities agent's opponent from using the ammunition. The second thing is that the `Track Enemy` goal is always generated with high priority. In the standard GRUE agent, the `Track Enemy` goal decreases in priority over time, allowing the agent to eventually give up and do something else. It is possible that by continuing to track until the enemy is visible again, the CONSTANT PRIORITIES agent is more likely to be able to find and kill its opponent.

The CONSTANT PRIORITIES agent also has some disadvantages over the standard GRUE agent. It will break off attacks and run away sooner than the standard GRUE agent if it detects that it is taking damage. It will never stop to pick up ammunition on its way to get the opposing team's flag or on its way back to its own flag base with the other team's flag. It will also continue following a path to the opposing team's flag even when it cannot see the flag but can see an enemy.

It appears that the advantages that the CONSTANT PRIORITIES exhibits in certain circumstances outweigh the disadvantages of this agent as far as individual scores go. In terms of goal scoring ability, the CONSTANT PRIORITIES agent's advantages balance its disadvantages causing it to tie with the standard GRUE agent.

5.2.5 The ALL GOALS Agent Opposed by the Standard GRUE Agent

The ALL GOALS agent is created by modifying the goal generators such that they do not check for appropriate conditions before generating goals. The priorities of the goals are still set according to the current situation. Any very low priority goals will be filtered out before reaching the arbitrator. Inappropriate goals that do reach the arbitrator will cause the associated TRPs to run. However, most of the goals in the standard agent are only generated when the success conditions of the associated TRPs are not true. In the ALL GOALS agent, inappropriate goals have TRPs whose success conditions are already satisfied,

Game Number	1	2	3	4	5	6	7	8	9	10	**Avg.**	**Std. Dev.**
Team Scores												
GRUE	1	0	1	2	2	0	1	0	0	1	0.80	0.79
All Goals	0	1	0	0	0	0	0	2	0	2	0.50	0.85
Difference	-1	1	-1	-2	-2	0	-1	2	0	1	-0.30	1.34
Individual Scores												
GRUE	9	7	13	19	28	5	18	1	0	18	11.80	8.98
All Goals	2	7	0	5	0	18	0	15	10	19	7.60	7.53
Difference	-7	0	-13	-14	-28	13	-18	14	10	1	-4.20	14.19
Deaths												
GRUE	1	1	0	1	1	4	0	1	3	1	1.30	1.25
All Goals	1	2	2	0	7	7	4	1	0	3	2.70	2.58
Difference	0	1	2	-1	6	3	4	0	-3	2	1.40	2.59

Table 5.30: Results for the All Goals agent playing a Capture the Flag game in Unreal Tournament against the standard GRUE agent.

so they will simply execute the null action and leave the arbitrator. The result is that the All Goals agent does the same things as the standard GRUE agent.

Table 5.30 shows the results for the All Goals agent playing against the standard GRUE agent. The All Goals agent won 3 games and tied 2, while the standard GRUE agent won 5 games. This looks as though the standard GRUE agent might be doing slightly better, but due to the small sample set we think it is more likely that they are about equal. The individual scores and the deaths in are similar, with the standard GRUE agent clearly doing better in about 5 of the 10 cases.

5.2.6 The All Goals Constant Priorities Agent Opposed by the Standard GRUE Agent

The All Goals Constant Priorities generates all goals without checking whether appropriate conditions are true and also generates them with constant priorities. Like the

Game Number	1	2	3	4	5	6	7	8	9	10	**Avg.**	**Std. Dev.**
Team Scores												
GRUE	1	3	1	1	0	1	0	0	1	2	1	0.94
All Goals Constant Priorities	0	0	0	0	1	0	1	0	0	0	0.20	0.42
Difference	-1	-3	-1	-1	1	-1	1	0	-1	-2	-0.80	1.23
Individual Scores												
GRUE	25	22	10	19	0	9	12	1	21	31	15	10.26
All Goals Constant Priorities	5	0	10	5	17	-1	24	0	0	0	6	8.54
Difference	-20	-22	0	-14	17	-10	12	-1	-21	-31	-9	15.66
Deaths												
GRUE	2	0	2	2	2	3	9	1	1	0	2.20	2.57
All Goals Constant Priorities	7	1	8	3	0	6	4	2	6	5	4.20	2.66
Difference	5	1	6	1	-2	3	-5	1	5	5	2	3.53

Table 5.31: Results for the All Goals Constant Priorities agent playing a Capture the Flag game in Unreal Tournament against the standard GRUE agent.

Constant Priorities agent, the goals are generated with priorities that are equal to the maximum values possible in the standard GRUE agent. The team data in Table 5.31 shows that the All Goals Constant Priorities loses to the standard GRUE agent in a majority of cases. From the individual scores and deaths in the same table, we can see that the All Goals Constant Priorities agent also has much lower individual scores than the standard GRUE agent, and is killed much more often.

At first glance, this seems odd because the All Goals agent scores about the same as the standard GRUE agent and the Constant Priorities agent actually tends to get higher individual scores than the standard GRUE agent. However, the combination of goals that are always generated and priorities that are always high can lead to undesirable interactions that do not occur when either condition doesn't hold. In the standard GRUE agent, the `Track Enemy` goal is generated when an enemy under attack is no longer visible. In the All Goals Constant Priorities and All Goals agents the `Track Enemy` goal is generated all the time. This is not a problem for the All Goals agent because the priority of the `Track Enemy` goal drops over time. In the All Goals Constant Priorities agent, however, `Track Enemy` stays at its maximum priority constantly. If there is no enemy visible, the `Track Enemy` TRP will assume that the last enemy the agent attacked has vanished, and attempt to track it. This may cause the All Goals Constant Priorities agent to run to strange locations when other tasks are not executing conflicting actions, and can confuse other TRPs when the agent is not in the expected location. The `Track Enemy` TRP also executes *ROTATE* actions, which are not listed as conflicting with anything. Therefore the agent may also rotate strangely in the middle of paths.

5.2.7 Summary of Unreal Tournament Evaluation

Some researchers have evaluated computer game agents by having their performance judged by human players (Norling, 2003; Laird and Duchi, 2000). Certain features, such as accurate aiming of weapons have a large effect on the agent's believability and performance in the game (Laird and Duchi, 2000). We have not attempted this type of evaluation, partly because we have not specifically worked on those features that most affect the judgment of human observers, and also due in part to some difficulty developing agents with the Gamebots interface. Our agent exhibits a number of intermittent quirks, which we believe are not due to flaws in the architecture but which we have not managed to localize to either the game specific portions of our own code or to the Gamebots server. These include behaviours like walking into a wall repeatedly, pausing, and turning unexpectedly. The agent is supposed to be notified by the game server when it hits a wall, but we have seen this behaviour when the agent has no such notification. The most serious problem with our agent is that it takes quite a long time to receive and process messages from the server. Thus, it sometimes walks past an opponent without attacking, walks away from a dropped flag that it could have picked up, doesn't get a health pack when its health is low, tries to score when its own flag is stolen, and other obviously unbelievable behaviours. We suspect that subjecting our agent to human evaluation at this stage of development would merely result in a list of issues like these instead of comments that reflect on the features of the architecture.

We tested a standard GRUE agent and a number of variants of the GRUE agent in a Capture the Flag game, using the commercial game Unreal Tournament. Our Unreal Tournament agent can generate 11 separate goals, making this a useful environment for testing goal arbitration.

In our first set of tests, each agent played 15 games that did not include an opponent. This reduces the set of goals that may be generated, and demonstrates that the agent can maneuver to the other side of the level, retrieve the opposing team's flag, and return that flag to its own flag base. We find that generating goals that are not appropriate for the current situation has no effect on the agent's performance. In this test, tasks associated with inappropriate goals simply leave the arbitrator immediately. The time taken for the tasks to enter the arbitrator and then leave it again does not increase the amount of processor time used for each cycle of the architecture much if at all. We did not test the No Updates, Constant Priorities, or All Goals Constant Priorities agents in the empty game environment, as the empty game does not contain sufficient goal interactions to be informative for these agents. We did test the Non-exclusive agent, which as expected exhibited similar performance to the standard GRUE agent. We also tested the No Persistence agent which scored similar numbers of goals to the standard GRUE agent, but took about twice as much processor time for each cycle. The data for the agent with parallel execution of actions disabled suggests that the standard GRUE agent, by allowing multiple tasks to request paths from the game server simultaneously, may occasionally enable a low priority task to execute a *RUNTO* command when it would be better not to. This could be seen as being due to the goals generators not sufficiently evaluating the current situation before generating the lower priority goals, or due to an incomplete list of conflicting actions.

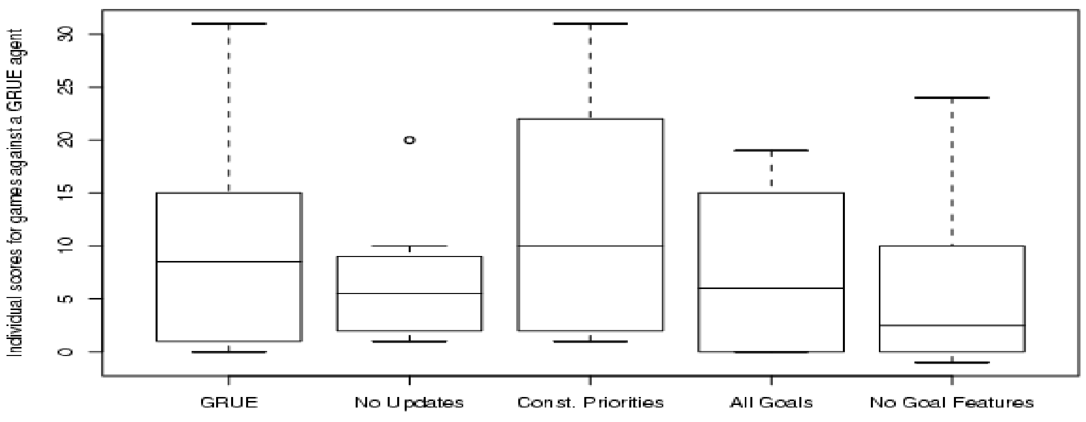

Figure 5.27: Boxplot comparison of the individual scores achieved by agents in an Unreal Tournament game with no opponent. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

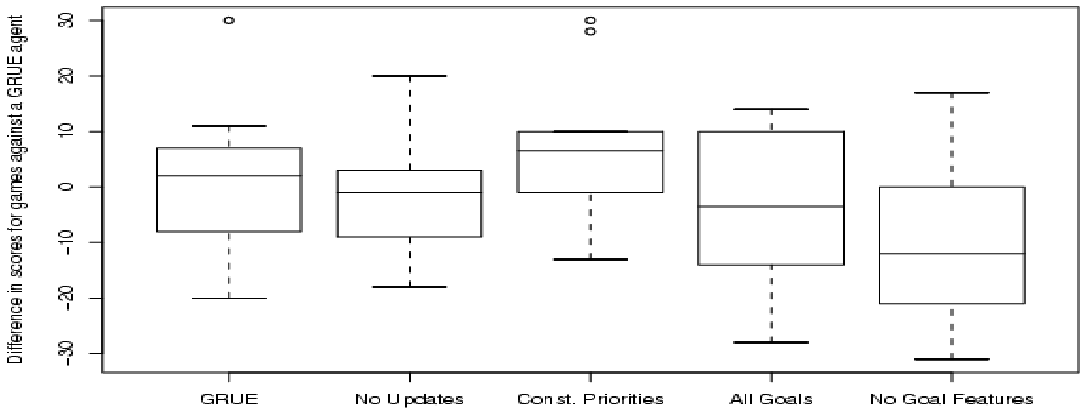

FIGURE 5.28: Boxplot comparison of the difference between the scores achieved by each agent and the standard GRUE agent's score in an Unreal Tournament game. Positive values indicate that the agent did better than the GRUE agent. The first box shows the score differential between two standard GRUE agents. The boxplot shows the median, upper quartile, lower quartile, and range of the data.

We also tested the standard GRUE agent, the ALL GOALS agent, the NO UPDATES agent, and the ALL GOALS CONSTANT PRIORITIES agent in a game with another player. We used the standard GRUE agent as the opponent for all cases. A boxplot comparison for the individual scores achieved by each agent is shown in Figure 5.27, while a boxplot comparison of the difference between the scores of the two agents in each game is shown in Figure 5.28. The boxplot for each agent shows positive values when the agent scored higher than the standard GRUE agent, and negative values when the GRUE agent scored higher. Notice that the first boxplot shows data for two GRUE agents playing against each other.

Unexpectedly, we found that disabling dynamic updating of goal priorities resulted in an agent that was fairly evenly matched against the standard GRUE agent. The boxplot in Figure 5.28 shows that the NO UPDATES was very slightly more likely to lose than win (where lose is defined as having a lower individual score than the opponent), but that when it does win it wins by a lot. An analysis of the goal priorities shows that disabling goal updates can result in an agent which continues attacking even when its health is low. This aggressive attacking strategy can be effective, giving the No Updates agent an advantage in some situations while it has disadvantages in others. On the other hand, Figure 5.27 shows that the No Updates agent clearly scores lower when playing against a GRUE agent than the standard GRUE agent does when playing against a copy of itself.

The Constant Priorities agent also compares favorably to the standard GRUE agent. From the boxplot in Figure 5.28, it seems to win most of the time. An analysis of this agent shows that it will pick up ammunition sooner, denying the GRUE agent the use of the ammunition. The Constant Priorities agent is also more persistent about pursuing an opponent who runs away from a fight.

The performance of the All Goals agent is fairly similar to the standard GRUE agent, because inappropriate tasks that enter the arbitrator simply leave it again. However, the boxplot in Figure 5.28 shows that the median score difference is below zero, and the range extends further on the negative side than the positive side, indicating that the standard GRUE agent does have a slight advantage over this agent.

Finally, the All Goals Constant Priorities agent performed much worse than the standard GRUE agent. Figure 5.28 shows that the median score difference is about 10 in favor of the GRUE agent. The All Goals Constant Priorities agent is much more likely to lose than win, and its biggest wins are smaller than its biggest losses. Figure 5.27 shows that the median score value for this agent playing against a standard GRUE agent is lower than the median for any other agent we tested in this case, although the range of scores is actually larger than for some of the other agents.

Our analysis of these tests reveals several interesting things. It is possible for an appropriate goal priority setting to prevent an inappropriate goal from interfering with other tasks in the arbitrator, and the use of constant goal priorities can be compensated for by only generating goals in appropriate situations. Generating goals in inappropriate situations with constant priorities can however result in severely degraded performance due to inappropriate goal interactions. Secondly, we observe that failing to check environmental conditions when generating goals does not affect performance if the same environmental conditions are tested in the actual TRP. If the goal generator checks conditions that are not tested by the TRP, then removing those checks can result in the TRP running inappropriately. Finally, three out of our four agents with reduced functionality showed similar performance to the standard GRUE agent. Detailed analysis of the goal interactions in these agents shows that some of our implementation choices result in the standard GRUE agent using ineffective strategies. Were we to consult an expert Unreal Tournament player and re-implement our agent using the most effective strategies, our experimental results would be more informative.

Finally, a thought about a more effective navigation strategy leads to an observation about the architecture design. One of the most difficult aspects of developing the Unreal Tournament agent was making sure that different tasks did not issue conflicting movement commands, while still allowing tasks to take advantage of movements made by other tasks. Our reliance on the game server to provide paths was a bit of a handicap, and it is difficult to capture all of the potential problems within a list of conflicting actions.

We think one way to handle conflicts more effectively is to store the agent's location as a resource and allow TRPs to bind it exclusively to prevent it from changing. This strategy, however, contradicts our previous rule that resources should only be bound exclusively when they are being changed or removed. In fact the location resource needs to be shared, but we need to assume that once a shared binding has been made, other TRPs may not change or remove the resource. This assumption was made implicitly in the description of the GRUE architecture, but as we have placed the burden of determining whether a resource is being changed entirely on the programmer we thought it best to state it explicitly here.

Chapter 6

Extensions to the GRUE Architecture

In this chapter, we discuss some possible extensions and optimizations of the GRUE architecture. In some cases, our experience with the implementation made it clear that an extension would be useful, although it is not included as part of the specification of the architecture. In other cases, the architectural design was deliberately constructed with possible extensions in mind. While we have not attempted to optimize the architecture, the computational constraints of the computer game domain are well known, and we provide some suggestions for ways to make GRUE agents run more efficiently. We also suggest some possible directions for future research.

6.1 Flaws in the GRUE architecture

The implementation and testing of GRUE led to the discovery of some issues with the architecture specification. None of them are serious, but they should to be rectified in future implementations of GRUE. The first of these was discussed in Section 5.1.5. It has to do with the use of negated resource variables. When implementing the TRPs for

Tileworld, we used a negated resource variable to mean the non-existence of the resource. In actuality, a negated resource variable is true when the resource is unavailable, which may be due to it being bound by another resource variable rather than being not non-existent. Both interpretations are conditions that might need to be checked in a TRP. An appropriate solution would be to include a predicate for testing the existence of a resource without regard to whether it is bound or not.

The second issue has to do with the way that bindings persist through consecutive cycles. In the Tileworld agent, this feature is important, especially in environmental conditions where objects appear and disappear rapidly. If the agent were free to select the best binding during each cycle, it could easily start approaching one tile stack, find a better one, start approaching the new tile stack, find a better one, and so on. The Unreal Tournament environment, in contrast, has a much lower density. Objects only disappear when picked up by an agent, and do not reappear for about 30 seconds. In this environment, the persistence of bindings is not really useful, except perhaps for ensuring that the agent targets a single opponent rather than switching back and forth. However, it is the case that the agent might want to switch from one resource to a better one. For example, if the bot was using one weapon to attack and happened to pick up a better weapon in the midst of the attack it would be better to switch to the new weapon. There are of course many similar situations in which an agent might wish to switch to a new and better resource in the middle of a task. The current implementation of GRUE includes a flag which can be used to disable the persistence of all bindings. A better solution would be to include a persistence flag in the resource variable structure. This would allow the agent developer to control which resources can be swapped for better ones. We expect that in cases where the agent is moving toward a resource, it will be better to stick with the initial selection of resource. In contrast, tasks where the agent is actually using a resource for something may benefit from the ability to switch to a better resource when one is presented.

6.2 Garbage Collection

During the implementation phase of this project, we discovered that it is often necessary to store "book-keeping" information. This is information about the progress of a TRP that cannot be derived from the world, and therefore must be stored in the form of one or more resources. One of the major tasks during the development of an agent is debugging the storage of these internal book-keeping resources. It is not uncommon for cases to arise where a TRP requires several rules to match different combinations of these resources so that they can be deleted at the end of the program.

One such case occurs in the `GOTO` program in the Unreal Tournament agent. This program must request a path from the game server. Since the server can receive any number of different path requests, it requires that each request have an identification string used for matching a request to a path. Thus, the TRP must create and store the identification string when it sends the request. The path resource is created by the world interface, but must be modified and ultimately deleted by the `GOTO` TRP. The TRP also stores the agent's current node in the path. When the TRP finishes running, it must remove all three of these resources.

One way to reduce the burden on the programmer would be to include a garbage collector. This would periodically check all such informational resources in memory, and remove those created by tasks that are no longer in the arbitrator. One way to implement this would be to require that all resources created by a TRP contain a `CREATOR` property whose value would be the name of the task[1] that created the resource. The garbage collector could then check the value of this property against the set of tasks in the arbitrator. While this does require an extra property not actually used in the TRP, it would vastly simplify the construction of TRPs. If the garbage collector ran at the end of every execution cycle (after tasks are removed from the arbitrator but before new tasks are added), the programmer could avoid writing rules to delete internal resources. They would all be automatically removed during garbage collection. (Resources not actually created by the TRP might still need to be removed in the TRP.)

6.3 Task Suspension

Initially, our design called for tasks to be suspended when the necessary resources were no longer available. A suspended task would remain in the arbitrator but not execute until the resources become available. During development, it became clear that this did not save any computational resources when compared to simply regenerating the goal. Therefore, our final description of the GRUE architecture omits this feature. Intuitively, it seems as though it might be useful to make a distinction between tasks that are no longer appropriate due to changing circumstances, and tasks that have lost resources to higher priority tasks. One way this might be useful is if suspended tasks continued to use any resources that

[1]We specify task rather than TRP to make the point that this refers to top-level TRPs only. For TRPs like `GOTO` that are called from several tasks, the value of the `CREATOR` property would be the name of the task that called the TRP.

haven't been preempted until the complete set of resources is available again. While we have not included task suspension in our specification of GRUE, it could be easily added as an extension.

6.4 Combining Actions

The fact that GRUE allows multiple actions to be output during the same execution cycle opens up a possible investigation into combinations of actions. In the current implementation of GRUE, the only processing that takes place on the list of actions is a simple check for pairs of conflicting actions. Tyrrell suggests taking a decision-by-committee approach, allowing each task to vote for and against the different actions and eventually settling on a compromise (Tyrrell, 1993). In particular Tyrrell believes it is very important to allow tasks to specify actions that should not be done, in order to prevent another task from doing something harmful. We have not provided any mechanism for doing this in GRUE, although it could be added. One way to implement it in GRUE would be to make a "harm prevention" type of goal very low priority so that it is run after all the other programs. The TRP for this "harm prevention" goal would process the list of proposed actions, removing any dangerous actions.

It also might be possible to allow the agent designer to specify a list of actions that can be combined into a different action. This approach is advocated by Blumberg, who says that systems should allow two actions to proceed if they are both possible simultaneously (if, for instance, they require different parts of the body), or if one action modulates the other (Blumberg, 1997). An example of modulation in a game environment would be combining `move forward` and `jump` actions into one `jump forward` action. The possibilities for this type of combination are of course largely dependent on the set of actions available in the environment being used.

A different approach is taken by Thorisson, who represents actions according to their function and allows them to change morphology depending on the current situation (Thorisson, 1996). This system was designed to model face-to-face dialogue, and the actions are things like acknowledging what the other person has said. These actions can be expressed in different ways depending on both the current dialogue situation, and the state of the agent's body. This approach, while interesting, is less relevant to GRUE as game environments don't typically provide a rich enough set of behaviours to allow different ways of expressing an action.

6.5 Optimizing the Architecture

We mentioned in Section 3.4 an approach for reducing the amount of computation done when evaluating a TRP or TRP hierarchy that is described in (Benson and Nilsson, 1995) and a similar system described in (Bryson, 2001). This approach requires keeping track of the current rule in the TRP (or hierarchy). During each cycle, the system checks only the rule above the current rule and the current rule. Most of the time, execution of a rule will lead to the condition of the rule above it becoming true [2] so more information is not needed. Benson and Nilsson retain the ability to take advantage of opportunities by occasionally checking the rules above the current rule when there is time to do so. Bryson's system

[2]Some TRPs contain several rules for alternate situations where running any of the rules will lead to the same rule condition becoming true. In order to use this optimization, these TRPs need to be organized into trees where alternate rules appear as nodes on the same level.

actually stores an active element, which may be a group of rules. When the active element completes, or when a new element is activated, the current active element is set to the top of the hierarchy allowing the system to check for opportunities. Either system would be easy to add to GRUE. We propose adding the current active element or rule to the task structure. The arbitrator could then check only that active element (or the rule and the one above it) during each cycle. We could either have the arbitrator check a few of the higher rules during each cycle (possibly depending on the number of tasks being processed), or reset the active element to the top of the TRP (or TRP hierarchy) occasionally.

There are also ways to optimism the implementation of GRUE. One is to use a more efficient programming language. All of the main architecture components could be re-implemented procedurally, with TRPs implemented as (or compiled into) if-then statements. The current implementation relies on a rule interpreter, which would be eliminated by this type of re-implementation. Most features of GRUE depend on searching lists of information. This can be computationally expensive. A way to improve this is to store items in a data structure that is more efficient than a list (like a hash table). Another simple fix is to limit the amount of information to be searched. The longest list in our implementation is the list of available resources. The size of this list could be limited by adding "forgetfulness" to the architecture. There are several possible ways to do this, including removing resources from the list once they have been stored for a particular length of time without being used, or putting a hard limit on the number of resources in the list and causing new resources to simply displace the oldest resource present. This would have to be done carefully in order to avoid causing currently running TRPs to fail unexpectedly.

Another place where it may be useful to include a limit is on the number of tasks in the arbitrator, which could be done by changing the filter so that only a fixed number of tasks can pass through (although if the limit is too small then the benefits of parallel task execution will be negated).

Finally, we anticipate that it will be possible to decouple the components of GRUE, and run them at different speeds. For instance, the world interface might run only once for every five cycles of the arbitrator. This would have to be carefully investigated, and exactly how fast the different components need to run may be dependent on the typical pace of events in the environment. Likewise, the agent designer might decide not to run the goal generators at every cycle. Either of these options would trade reduced execution time for a small increase in the time it takes the agent to respond to events in the world. Note that in some cases small increases in the time taken to respond to events may actually result in increased believability if the agent would otherwise react faster than a normal human (Freed et al., 2000).

6.6 Dynamic Filter adjustment

GRUE contains a simple threshold filter which eliminates low priority goals. This filter could be dynamically adjusted depending on the agent's current set of tasks. Wright attempts to keep the number of active goals at an optimal number by increasing the filter threshold as the number of goals increases, and decreasing the threshold as the number of goals decreases (Wright, 1997). There are other possible uses for dynamic filter adjustment; for example, if the agent is working on a very important goal, it could raise the filter threshold so that only extremely important things would get through. This would, in effect, enable the agent to concentrate on something. Likewise, the threshold could be lowered to achieve the effect of a flighty agent concerned with trivialities. Dynamic adjustment could potentially give the agent moods, and use of a more complicated filter might create even more interesting effects.

6.7 Meta-Management

Sloman breaks the capabilities of intelligent agents down into three categories: reactive behaviour, deliberative behaviour, and meta-management tasks (Sloman and Logan, 2000). Meta-management is the ability to reason about one's own behaviour, and adjust it accordingly. We have not explicitly included any meta-management abilities in GRUE, but we expect that it would be possible to add them. We have placed no restrictions on what TRPs are allowed to do, so the programmer is free to write a TRP to manage or modify the agent's memory or even the architecture as a whole.

One very important meta-management task is the detection of cycles or oscillations in the set of goals. For instance, it might be that working on one goal causes the priority of a second goal to increase. Working on the second goal then increases the priority of a third goal, which then increases the priority of the first goal again. This leads to the agent cycling through the set of goals, never completing any of them. In the current implementation, we have assumed that preventing this type of cycle is the job of the programmer or agent designer. However, designing a set of goal generators to work together correctly may be difficult, especially if the set of goals is large. Giving the agent the ability to notice that it is trapped in a goal cycle and adjust its priorities accordingly would be extremely helpful for the agent designer.

6.8 Extensible Agents

One thing that might be desirable is the ability to build agents that can gain new abilities over time. This might be useful in a game that allows players to add new features to the game (games like Unreal Tournament allow players to make and distribute modifications), online games where the developer may wish to extend the game world, or in non-game applications.

We suggest that one way to do this is by storing goal generators in memory along with the TRPs. New goal generators and corresponding TRPs could then be added to the agent's memory. Goal generators (and TRPs) could be stored as resources (and treated like any other resource), or they could be stored separately. The goal generator component of the architecture would have to be converted into a goal generator manager, which would maintain a list of active goal generators. The goal generator manager would also periodically check for the presence of new goal generators in memory, and activate them. (We can imagine adding the ability to deactivate goal generators as well, for instance to cause degradation of the agent's abilities when injured or confused.) The program selector would search for a TRP matching generated goals as usual.

6.9 Addition of a Reactive Layer

Those working in biological modeling, and those who need agents that respond to events on extremely short timescales, may be bothered by the hybrid approach we have taken with GRUE. GRUE TRPs can encode both reactive and deliberative behaviour. While the teleo-reactive programs do respond to changes in the environment, it is also true that there is no way of bypassing the arbitrator. We have not as yet encountered any situations where bypassing the arbitrator would be necessary, however, it would be possible to add a reactive layer in the form of an additional ruleset that runs all the time. In order to bypass the main components of GRUE, it would need to run as a separate thread or process. Doing this would potentially lead to conflicts between actions, unless the actions are placed in the action list and executed by the arbitrator as normal. This problem could be resolved by requiring that the reactive layer contain only actions that do not conflict with any other actions, or the potential conflicts (and resulting confusion) might be considered acceptable by the agent designer.

6.10 Planning

The program selector is currently a place holder for a full planner. To add a planning system would only require a small change. The main issue with planning is the fact that durative actions may have different effects depending on the amount of time they are allowed to operate. Benson and Nilsson solved the problem by using teleo-reactive operators (TOPs) (Benson and Nilsson, 1995) which could be treated by the planner like discrete actions. They are designed to be similar to STRIPS operators, listing the intended effect of the action as well as possible side effects.

GRUE adds the additional functionality of resource variables to TRPs. This means that preconditions for operators must include the required properties of any necessary resources. If the same action causes different effects when used with different resource types (for instance, if the action always requires property X but is more efficient when the resources also has property Y), then that needs to be encoded somehow. One way is to use a set of operators, each one with a different list properties for its resource variables and the corresponding list of effects. Alternatively, operators could encode preferred properties with the possible effects associated with each combination of properties.

Much work has been done on the use of planning in games and real-time simulations. For examples see the EXCALIBUR project (Nareyek, 1998; Nareyek, 2000), work by Nick Hawes (Hawes, 2000; Hawes, 2003), etc.. It would be very interesting to determine whether any of these planning systems could be merged with GRUE.

6.11 Learning

In (Benson and Nilsson, 1995), the system learned preconditions for actions by observation. A similar approach could be employed with GRUE, but the use of properties makes the learning process potentially complicated. We expect that using a trial and error approach, the agent would need to store all the properties of a resource used to accomplish a particular effect. After using several resources to accomplish the same purpose, the agent would be able to extract the set of properties that are actually required. One way to learn preferred properties is to notice properties which are not necessary to achieve an effect, but are often associated with that effect. It might also be useful to group effects - for instance two effects might be essentially the same but one might be bigger or last longer. In this case, the effects should be grouped and properties associated with the more desirable subset only should be listed as preferred properties. Detecting effects that can be grouped together as essentially the same is far beyond the scope of this thesis, but suggests a promising line of future research.

Chapter 7

Conclusions

In this thesis we proposed general mechanisms to support the development of characters for computer games and other applications with similar properties. Computer games are particularly useful for evaluating intelligent agents as they provide rich, dynamic environments. By their very nature, computer games are challenging enough to engage human players, while still being simplified as compared to the real world. The utility of goal-based agent architectures in dynamic environments is supported by the literature. AI researchers argue that agents in unpredictable environments require *goal autonomy* (Norman and Long, 1996), the ability to choose and prioritize a set of goals, while game developers suggest that goal-based AI can make computer game characters more human-like (O'Brien, 2002; Orkin, 2004).

We have presented a resource-based goal arbitration method to support the development of goal-autonomous agents. We have also developed an architecture, GRUE, that embodies our method of goal arbitration. GRUE agents can generate their own goals, generate goals only when those goals are appropriate, assign priorities to new goals according to the agent's current knowledge of the environment, and adjust the priorities of goals as the environment (or the agent's information about the environment) changes. GRUE supports the ability to work toward multiple goals in parallel, and even execute actions in parallel when those actions do not conflict.

We provide notation for representing resources in terms of properties. The properties of a resource (and the possible values of those properties) are assigned by the agent designer and can be different in different environments, or for different resources within the same environment. The notation we use is flexible enough to allow almost anything to be represented, including objects, quantities, other agents within the world, and information. A GRUE agent uses plans containing resource variables, which have values that are assigned when the plan executes. Resource variables may specify both required properties and preferred properties. Required properties are those that are necessary for the plan to execute correctly, while a resource that has preferred property values may allow a plan to execute more efficiently or effectively.

We also provide special notation for representing value ranges. A value range is used to specify constraints on numerical property values. Value ranges may have either soft boundaries or firm boundaries, and may contain a utility arrow to specify a preference ordering for values within a range. This notation is expressive, and can be used to represent a large variety of resource requirements. Resources with numerical property values may be designated as divisible. Divisible resources represent quantities, such as money or fuel. A divisible resource may be divided among several plans.

The main requirement for resource-based goal arbitration is that a plan for achieving a low priority goal may never take resources from a plan for achieving a higher priority goal. The algorithms described in Chapter 3 meet this requirement by always allowing plans for high priority tasks to select resources first.

Our work has similarities to other work in intelligent agents. Soar (Laird et al., 1993) and the BDI framework (Rao and Georgeff, 1995) have been applied to the computer games domain (Norling, 2003; Laird and Jones, 1998; van Lent et al., 1999). We based GRUE on a system described by Benson and Nilsson (Benson and Nilsson, 1995). Systems described by Bryson (Bryson, 2001) and Wright (Wright, 1997) have marked similarities to Benson and Nilsson's system as well as to GRUE. We described all of these systems in Chapter 2 and gave a detailed comparison between the latter systems and GRUE in Section 3.4. GRUE's resource based goal arbitration method is a significant departure from Benson and Nilsson's system on which it was based; however GRUE's goal arbitration shares some similarities with the methods used by Bryson and Wright.

7.1 Implementation of GRUE Agents

We successfully implemented GRUE agents for Tileworld (Pollack and Ringuette, 1990), an AI test bed, and Unreal Tournament (Infogrames, PC 1999), a commercial computer game. The Tileworld environment includes stacks of tiles that must be distributed into holes. The agent receives a higher score for matching tiles to holes of the same shape. Our agent divides tile stacks when necessary, using part of the stack to fill a hole and keeping the rest for the next hole. The agent prefers to select holes that are small and nearby, thus being easier to fill, and which match the shape of the tiles the agent is carrying. If no such hole is available, the agent will select any hole that it can find.

The Unreal Tournament environment supports more goals than Tileworld, so is better for demonstrating goal autonomy features of the agent. We used the capture the flag game style within Unreal Tournament. Our Unreal Tournament agent has a total of eleven possible goals, which are:

- Return to Base
- Patrol
- Get Health
- Run Away
- Get Weapon
- Get Ammunition
- Attack Enemy
- Track Enemy
- Capture Enemy Flag
- Score Point
- Return Stolen Flag

The Unreal Tournament agent generates goals when they are appropriate. For example, the `Attack Enemy` goal is only generated when an enemy is present to attack, the

`Capture Enemy Flag` goal is not generated if the agent already has the enemy flag, and `Return Stolen Flag` is only generated when the agent can see its own flag on the floor.

Goals are generated with appropriate priorities. `Attack` is generated with high priority when the enemy is nearby, and otherwise the priority is inversely proportional to the distance between the agent and the enemy. `Get Ammunition` is generated with low priority if the agent sees some ammunition but does not really need it, and with high priority when the agent is almost out of ammunition. `Get Enemy Flag` is generated with high priority when the enemy flag is visible, but otherwise with a medium priority value.

The priorities of goals can change as the situation in the environment changes. If there is an enemy far away, or not visible at all, the agent will start working on the `Get Enemy Flag` goal. If the enemy approaches, the priority of `Attack` will increase until it takes precedence over the `Get Enemy Flag` goal. However, if the agent then sees the enemy flag, it will increase the priority of the `Get Enemy Flag` goal, causing it to stop attacking and run toward the flag instead.

The GRUE agent can also work toward several goals in parallel. The `Score Point` goal causes the agent to run back to its own flag base with the enemy flag. The `Return to Base` goal causes the agent to return to a point in front of its flag base. Since the destination locations for the two goals are nearby, moving the agent toward one moves it toward the other in most circumstances. Thus, the agent can work toward both goals simultaneously.

7.2 Summary of Experimental Results

In Section 1.3 we discussed several features of the environment that can affect the utility of features included in an agent architecture. In Tileworld we evaluated the features of GRUE while varying two features of the environment, the density and the rate of change. Each Tileworld agent was run for 1000 cycles in a 20 by 20 sized Tileworld. The density controlled the number of tiles and holes in the environment. We varied density from a minimum of 10 tiles and 10 holes to a maximum of 100 tiles and 100 holes. Rate of change controls how often tiles, holes, and obstacles are removed from or added to the Tileworld. Our rate of change values ranged from 10, meaning on average one new object of each type would be created and one would be deleted every 10 cycles, up to a maximum of 70. We evaluate individual features by disabling each feature and comparing the performance of the resulting agent to the standard GRUE agent.

In the Unreal Tournament environment, we only used two conditions. The first case was an empty game, with no opponents. We used this to test the number of goals that could be scored by each agent. In the second test, each agent played 20 games against the standard GRUE agent. This tests the performance of the agent when all of its goals are enabled.

Playing a competent game of Unreal Tournament requires an agent with a richer set of goals than succeeding in Tileworld, which has an extremely limited set of possible actions. While our Tileworld agent had three possible goals, our Unreal Tournament agent had a set of eleven, eight of which could be active simultaneously. In the empty game, six possible goals could be triggered, with at most four simultaneously. We can look at

Tileworld, the empty Unreal Tournament game, and the one-on-one Unreal Tournament game as three different values for the property *goal density.*

We summarize the results for each feature of the architecture in the following list.

Goals are generated only in appropriate situations.

Eliminating this feature results in reduced performance in all Tileworld conditions.

Eliminating this feature has no effect on the performance of the agent in the empty Unreal Tournament condition, because inappropriate goals simply leave the arbitrator immediately. This effect occurs when the end condition for a TRP is the same as the precondition for generating the goal, or when in the absence of the precondition none of the TRP's rules can fire. This illustrates that preconditions for goal generation could be implemented in the TRP instead of in the goal generators.

Eliminating this feature in a one-on-one Unreal Tournament game showed a slight disadvantage compared to the standard GRUE agent.

Goal priorities are updated continuously.

For our Tileworld agent, this feature has little or no effect in low density, fast rate of change conditions. When the density increases or the rate of change is slowed, eliminating this feature actually increases performance. However, we believe this result is due to the way our goal generators are written and may not be generally applicable to other agents.

This feature was not tested in the empty Unreal Tournament condition as goals never need to be updated in that environmental condition.

In the one-on-one Unreal Tournament game, eliminating this feature proved to result in an alternate game strategy which has advantages over the standard GRUE agent in some specific game situations. In general, however, it does not perform quite as well as the standard GRUE agent.

Goal priorities are set according to the current situation.

Eliminating this feature by using constant, preset priorities increases the performance of the Tileworld agent in higher density conditions. However, this feature also eliminates the updates of goal priorities. As a result, this result is attributable to the same issue with our goal generators that affected the evaluation of the goal priority updates feature, so we think it is probably not generally applicable to other agents.

This feature was not tested in the empty Unreal Tournament condition, as insufficient situations in which goal priority would be variable.

Eliminating this feature results in an agent with similar performance to the standard GRUE agent in the one-on-one Unreal Tournament game. Analysis of the goal interactions in Unreal Tournament when the goals have constant priorities gives us clues about how to improve the strategy our agent used to play the game.

The combination of goal properties.

Eliminating all three of the above features significantly reduced the performance of the agent in all of the Tileworld conditions.

This combination of features was not tested in the empty Unreal Tournament environment.

Eliminating all three features relating to goals in the one-on-one Unreal Tournament clearly decreases the performance of the agent.

Plans request resources in terms of required and preferred properties.

Both possible ways of eliminating this feature reduce the performance of the Tileworld agent in all environmental conditions that we tested.

We also observed that eliminating preferred properties by deleting them was more effective in low density, fast changing environments while eliminating preferred properties by making them all required is more effective in slow changing, dense environments. This leads to the general observation that fewer restrictions are better when resources are scarce, while large numbers of requirements are acceptable if resources meeting those requirements are going to be relatively easy to find.

We did not test this feature in either Unreal Tournament environment, as there are not enough resources in Unreal Tournament to make it informative.

A plan can specify that it needs exclusive access to a resource.

Eliminating this feature made no little or no difference to the Tileworld agent in lower density environments, but increased the performance of the agent in high density environments. Our analysis shows that this is due to an unexpected interaction between our TRPs.

Eliminating this feature results in similar performance to the standard GRUE agent in the empty Unreal Tournament environment.

We did not test this feature in the one-on-one Unreal Tournament game, as there are not enough exclusive bindings used to make the test informative.

A plan that makes the same request during consecutive cycles will receive the same resource each time.

It is difficult to identify a pattern in the Tileworld dataset. However, we do see a few cases where eliminating this property increases the performance of the agent which occur in the high density, quickly changing environments. We see a reduced performance or no change for eliminating this feature in low density or slowly changing environments.

If this is a valid pattern, then we can perhaps make a general rule that this feature is not useful for the Tileworld agent in the fast changing high density cases. We should note that the design of our Tileworld agent means that it will opportunistically use non-optimal resources that it happens to walk over. We suspect that in low density environments switching between resources will result in the agent walking back and forth between resources without actually making progress, an effect that is reduced by the opportunistic properties of our TRPs in high density environments. Therefore the utility of this feature is likely to be highly dependent on the properties of the TRPs used.

Eliminating this feature in the empty Unreal Tournament condition results in similar performance to the standard GRUE agent, but takes about twice as much processor time.

We did not test this feature in the one-on-one Unreal Tournament game, as it provides no more information regarding this feature than the empty Unreal Tournament game.

Numerical quantities can be divided and used for more than one goal.

Eliminating this property reduced the performance of the Tileworld agent in all of the environmental conditions except for the specific case N=70 with density=100, where we see no significant difference. This may indicate an interesting result for densities greater than 100.

This feature was not used in Unreal Tournament.

Plans can use special notation to request numerical values within a range.

This case is similar to the last case: eliminating this property reduced the performance of the Tileworld agent in all of the environmental conditions except for the specific case N=70 with D=100, where we see no significant difference. Again, this may indicate an interesting result for densities greater than 100.

This feature was not used in Unreal Tournament.

The combination of numerical properties.

Eliminating both of the numerical features has effects in Tileworld similar to eliminating either of them individually.

This feature was not used in Unreal Tournament.

Multiple actions can be executed during each execution cycle.

Eliminating this feature reduces the performance of the Tileworld agent in low density, quickly changing environments. The effect is not so clear in the rest of the environmental cases. In two of the high density cases, it actually increases performance while it has no significant effect in most of the remaining cases.

Eliminating this property in the empty Unreal Tournament game resulted in higher performance than the standard GRUE agents (median score of 4 goals with no goals less than 3, compared to a median value of 3 goals with a minimum score of 1). We conjecture that this could be due to the standard GRUE agent occasionally executing bad actions, but since the sample size is very small (15), more investigation is needed to determine if it is a real effect.

We did not test this feature in the one-on-one Unreal Tournament game as it would have provided no information beyond the results obtained in the empty game.

The results of our experiments caused us to discover a number of interesting interactions between the components in our agents. However, we find that the features we tested are so affected by the specific implementation decisions we made that it is difficult to draw any generally applicable conclusions. We did find that some features of the architecture are redundant, for example the generation of goals in inappropriate situations can be compensated for by putting appropriate conditions on the rules in the corresponding TRP. Generating goals inappropriately can also be compensated for by adjusting priorities, while generating goals only in appropriate situations can compensate for inappropriate priorities. Generating inappropriate goals with constant priorities decreases performance in all of the environments we tested, showing that at least one of these features is generally needed.

The ability to have exclusive access to a resource seems like a necessary property for correct execution of TRPs. Unexpectedly, our Tileworld TRPs interacted in such a way that this feature actually decreased performance in some conditions and otherwise this feature had no effect. The persistence of bindings from cycle to cycle decreased performance in Tileworld in some conditions, while increasing it in others. This is again due to the specific implementation of our TRPs. Persistence of bindings had no effect on performance in the Unreal Tournament test, although it did halve the amount of processor time used.

The use of preferred properties, and the special numerical notation was associated with a performance increase in all of the Tileworld conditions. However, these features are not useful in environments like Unreal Tournament which have few opportunities to use these features. The resource-specific features of GRUE are more suited to domains that are rich in different types of objects.

The execution of multiple external actions each cycle increases performance in quickly changing low density Tileworlds. However, it actually decreases performance in some of the high density cases as well as the empty Unreal Tournament game. We did not expect this feature to have any effect on the Tileworld agent due to the limited number of external actions, and we expected it to be associated with increased performance in the Unreal Tournament environment. Further exploration of this feature is needed to determine in exactly what circumstances it is beneficial.

7.3 Changes to the Architecture Description

The evaluation of our architecture led to the discovery of two minor flaws in our specification. We specified in Chapter 3 that resource variables should bind to the same resource during consecutive execution cycles. This feature can be disabled, but only for the architecture as a whole. Some reflection makes it clear that this feature is not desirable in all circumstances. Clearly, if the agent is considering which of a set of available items to pick up, continuously changing its mind would be undesirable. However, if an agent is forced to use a less-desirable object and a better one becomes available, in most cases the most believable option would be to switch to the new item. There is no point, for example, in continuing to attempt to open a door with an improvised lock pick if someone has handed you a key. This can be fixed by including an persistent binding flag along with the exclusive binding flag in the resource variable. A persistent binding flag value of true would indicate that the binding should persist from cycle to cycle, while a value of false would cause the arbitrator to ignore the previous value each time the variable is bound.

The second flaw in the architecture is the lack of a way to specify that a resource does not exist. It is possible to use a resource variable to require that a resource is not available, but this does not cover all situations. In fact, we found such a case in the implementation of the Tileworld agent. We used a negated resource variable, but found that another program could bind the resource exclusively causing our negated resource variable to return true in a case where it really should have evaluated to false. Specifying that the architecture must include a predicate to test the existence of a resource independent of its binding status would solve this problem.

We also found a need to clarify an implicit assumption in the description of our architecture. If a shared binding is made by a high priority plan, lower priority plans may not change or remove the resource. We have not included any mechanism to enforce this requirement in the GRUE architecture, instead relying on the agent designer to abide by it voluntarily. It might be possible to include some type of mechanism for detecting actions that modify resources and prevent them from operating on resources that are part of a shared binding, or issue a warning to the agent developer. We have also found that our implementation of the binding algorithm is incorrect in that it allows a resource variable to exclusively bind a resource that is already bound non-exclusively.

7.4 Limitations of GRUE

Our architecture design does have certain limitations. While our resource representation is very expressive, there are some limits to the types of preferences that can be expressed. Our value range notation allows an agent to prefer values that fall within a range, and at the same time to prefer bigger or smaller values within the range. Other types of values, however, must be required, preferred, or not specified. All preferred properties are treated as being equivalent, where a resource that matches the largest number of preferred properties is the best match for a resource variable, and resources with the same number of preferred properties are equivalent. This means that resource variables cannot specify complicated preference requirements, or rank property values from highest preference to lowest.

Another limitation of the architecture is that we do not provide deliberative mechanisms such as scheduling, learning or planning. Benson and Nilsson's system, which we used as the basis for GRUE, has the ability to learn preconditions for new actions and then use those actions to construct new plans. Other systems, such as (Drabble and Tate, 1994), take into account the time window during which a goal must be achieved, and the length of time it will take to achieve the goal and use these values to make decisions about the order in which goals should be considered, or to schedule a goal for a future time. The aim of this project was to develop mechanisms for use in real-time dynamic environments, so our goal arbitration method is primarily reactive. We assume goals are generated only at relevant times, and consider them in order of highest priority to lowest priority. GRUE does contain a component we call a program selector, which could easily be replaced with a planner as in (Benson and Nilsson, 1995). However, GRUE is most useful for use in domains that do not require complex deliberative reasoning.

Finally, goal autonomous systems have the potential to enter a goal cycle. This would occur if the agent is working on a set of goals G_1, G_2, through G_n such that working toward G_1 causes the agent to switch to G_2, and working toward G_2 causes the agent to switch to G_3 and so on until working on G_n causes the agent to cycle back to G_1. The GRUE architecture as described here has no way of detecting or breaking out of goal cycles, although we suggested this as a possible extension in Chapter 6. Instead, the burden is on the agent designer to cause the priorities of goals to be set appropriately. One way to avoid goal cycles is to generate goals with priorities at fixed levels instead of using priorities that vary continuously. The levels of the different goals should be offset from each other, so that in each set of concurrently executing goals the goals have a clear priority ordering. We took this approach when designing our UT agent. Of the agent's 11 goals, only 4 vary continuously. Of those 4, one decreases in priority over time which causes it to eventually drop out of the arbitrator. The other three all default to a fixed maximum when they become urgent.

There are also a few issues with the implementation decisions we made. These do not impact the design of the architecture, but affect the observed performance of our agents. First, we stored the resources available to the agent in a list. We iterate over this list several times when binding resource variables, and also in the world interface component. Using a more efficient data structure, perhaps a hash table, could potentially improve the experimental results.

7.5 Wider Applications of Resource-Based Goal Arbitration and GRUE

We have aimed the development of resource-based goal arbitration and the GRUE architecture toward characters in computer games. However, the properties of goal autonomy supported by GRUE are required by unpredictability of the environment. Many other application areas feature dynamic, unpredictable environments. Some possibilities include virtual reality entertainment or training applications, intelligent agents that serve as personal assistants or provide general assistance to humans, software agents that interact with other agents whose behaviour is not reliable, and autonomous robots that interact with people or other robots. In all of these cases, resource-based goal arbitration could be used as long as it is possible to represent information received by the agent as resources.

The GRUE architecture is designed as a core set of application-independent components which include our resource-based goal arbitration method. The remaining components of the architecture, specifically the world interface, the goal generators, and a library of pre-written plans, must be tailored to the agent's environment. This design allows GRUE to be applied to agents in different types of applications with minimum effort. We have shown that the GRUE architecture works in two specific environments which illustrates its portability.

Chapter 6 suggests a number of possible extensions to the GRUE architecture.

Some of these are simple improvements to the implementation, while others are additional capabilities that could be integrated into the architecture. GRUE's applicability to a wide range of possible domains, as well as potential additions such as planning, learning, and meta-management capabilities suggest exciting possibilities for future research.

Appendix A

Summary of Architecture Components and Data Structures

In this appendix, we provide a brief summary of GRUE's components and data structures. For detailed descriptions, please refer to Chapter 3.

Goal

A goal is a list containing an identifier, a priority, a type which is one of the two values MAINTENANCE or ACHEIVEMENT, and the name of a TRP to achieve the goal.

Task

A task is a list containing an identifier, a priority, a type, and a TRP. It is the same as a goal, except that the name of the TRP is replaced with the TRP itself.

TRP

A teleo-reactive program is a list of rules. During each cycle, the first rule in the list whose condition evalutes to true will run. When a rule runs, it proposes one or more actions to be executed by the architecture. The action of a rule, if executed for one or more cycles, should eventually cause a condition higher in the list of rules to become true. The top rule in a TRP contains the success condition for the program, and a null action. TRPs are the plans used to achieve goals in GRUE.

TRPs in GRUE may invoke other TRPs, which take arguments (such as the location for a GOTO program). TRPs taking arguments must not be used as top-level TRPs. The GRUE arbitrator only handles top-level programs; invocations of other programs are treated for arbitration purposes as though they are part of the top-level program.

Resource

A resource is a data structure representing objects in the environment, or information stored by the agent. A resource is a list containing an indentifier and one or more properties. A property consists of a label, the optional keyword DIV, and one or more values (strings or numbers). The DIV keyword may only be used with a numerical value, and indicates that the resource can be divided. For example, the following resource represents an amount of money which could be spent on several things: (GoldCoins [TYPE money] [AMOUNT DIV 30]).

Resource Variable

A resource variable is used in a TRP rule to request a resource that must be available for the rule to execute. It is a list containing an identifier, a type value which can be EXCLUSIVE or SHARED, a set of required properties, and a set of preferred properties. The type indicates whether this TRP needs exclusive access to the resource, or whether it may be used by other TRPs simultaneously. The required properties are those which the resource must have, and the preferred properties are those would be useful, but aren't necessary. For example, the agent might need a weapon and prefer it to be a grenade launcher: (?Weapon ([TYPE weapon]) ([SUBTYPE grenade_launcher])).

If a rule containing a resource variable runs, the resource variable is bound to a resource. The resource variable name can be used in the action part of a rule to refer to the resource.

Resource variables should only use the EXCLUSIVE type when the resource will be changed or removed. If the rule does not change the resource, use SHARED instead.

Value Ranges

Value ranges can be used in resource variables when a non-exact numerical value is needed. The syntax of a range is B1 N A N B2 where B1 is one of the brackets #| or #:, N represents a number or nothing, A is –>, <–, or nothing, and B2 is one of the brackets |# or :#.

Firm boundaries are represented by the brackets #| and |#. Soft boundaries are represented by the brackets #: and :#. A firm boundary indicates that the matched value must be within the values. A soft boundary indicates a preference for values within the bounds, but allows those outside it. The two types of boundaries can be mixed, i.e. a soft upper bound and a firm lower bound.

The two arrows, –> and <– can be used to represent a utility ordering within the range. An arrow to the right indicates that higher values within the range are preferred, and an arrow to the left indicates that lower values are preferred. If the arrow is omitted, then all numbers within the range are assumed to be equally acceptable.

When a soft boundary is used, the corresponding number may be omitted. If the upper bound is omitted, then the bound is assumed to be equal to infinity, and if the lower bound is omitted it is assumed to be negative infinity.

If a value range is found in the required properties section of a resource variable, the only resources considered are those that match the value range. If a value range is found in the preferred properties section then only the resource that best matches the preferred properties, from the list of resources matching the required properties, is considered to match the preferred properties. Best in this case is defined according to the utility ordering of the value range.

When a value range is used as a required property for a divisible resource, the resource will be divided to force it to match the range requirement. The amount actually bound to the variable will be the preferred amount according to the utility ordering, or else the minimum amount if no utility arrow is used.

World Interface

The world interface takes information from the world and converts it to resources for use by GRUE TRPs. Additional processing may be done in the world interface, such as computing the distance between the agent and an object when the world provides the object's location.

Memory

The agent's memory is general storage space. It includes the agent's internal state, a world model in the form of resources obtained by the world interface, the agent's goals, TRPs for achieving those goals, and any other information that needs to be stored.

Goal Generator

A goal generator produces a goal when an appropriate situation exists in the world model. It also updates the priority of tasks in the arbitrator.

Filter

The filter prevents extremely low-priority goals from entering the arbitrator.

Program Selector

The program selector converts a goal into a task by inserting the appropriate TRP. This component exists mainly to allow GRUE to be extended with a planning system.

Arbitrator

The arbitrator binds resource variables to resources, runs the TRPs, and resolves conflicts between actions. It first sorts the tasks according to priority. Then, starting with the highest priority task, all the tasks are allowed to attempt to run a rule. If no rule in a task's TRP can run, then the task is removed from the arbitrator. Once the maximum number of tasks has been processed, the list of proposed actions is checked for conflicts. When a conflict is detected, the lower priority action is discarded. Finally, the set of non-conflicting actions is run.

Appendix B

Resources used in Tileworld

This is a list of the different types of resources available in the Tileworld environment. For each main heading, which is one of the possible values for the TYPE property, we list the other properties associated with that type and their possible values.

tile_stack

Represents a stack of tiles in the Tileworld. Can be divided into individual tiles.

DISTANCE: A numerical value indicating the distance from the agent.

SHAPE: One of three possible shapes (simply indicated by numbers in our implementation).

SIZE: The size of the tile stack. Ranges from 1 to 10, and is preceded by the DIV keyword.

AGE: A boolean value with the value YOUNG or OLD.

DIRECTION: The direction the agent needs to move to reach the tile stack (left, right, up or down).

NEAREST: True if the tile stack is the closest tile stack to the agent, false otherwise.

hole

Represents a hole in the Tileworld.

DISTANCE: A numerical value indicating the distance from the agent.

SHAPE: One of three possible shapes (simply indicated by numbers in our implementation).

SIZE: The depth of the hole. Ranges from 1 to 10.

AGE: A boolean value with the value YOUNG or OLD.

DIRECTION: The direction the agent needs to move to reach the hole (left, right, up or down).

NEAREST: True if the hole is the closest hole to the agent, false otherwise.

obstacle

Represents an obstacle in the Tileworld.

DISTANCE: A numerical value indicating the distance from the agent.

DIRECTION: The direction the agent needs to move to reach the obstacle (left, right, up or down).

NEAREST: True if the obstacle is the closest obstacle to the agent, false otherwise.

direction

Created by a GRUE TRP to represent the direction in which the agent is currently moving (or trying to move).

TOWARD: The type of object that the agent is approaching. One of tile_stack, hole, or random (which indicates that the agent is simply wandering), or around_obstacle.

VALUE: Up, down, left, or right.

stored_direction

The avoid obstacle program stores the direction the agent was originally moving as a resource of type stored_direction.

TOWARD: The type of object that the agent was approaching. This will only be hole or tile_stack for stored_direction resources.

VALUE: Up, down, left, or right (taken from the original direction).

target

Stores the object that the agent is trying to reach.

VALUE: The identifier for the target object.

XCOORD: The x coordinate of the target object.

YCOORD: The y coordinate of the target object.

Appendix C

Resources used in Unreal Tournament

This appendix describes the resources used in the Unreal Tournament environment. For each resource type, we list the other properties for that type and their possible values. Rotations are given in Unreal Tournament units, where there are 65535 units in a full circle.

player

Another player in the game (either an agent or a human).

TEAM: Either red or blue.

LOCATION: A list of three coordinates representing the (x,y,z) location of the player.

ROTATION: A list containing three values representing the rotation about each axis.

VELOCITY: A list of three values representing the three components of the velocity vector.

WEAPON: The name of the weapon wielded by the player.

VISIBLE: A boolean value; true if the player is visible to the agent, false otherwise.

flag

Represents one of the two team flags in the game.

TEAM: Whether the flag belongs to the red team or the blue team.

LOCATION: A list of three coordinates representing the (x,y,z) location of the flag.

weapon

Represents a weapon that the agent can pick up.

LOCATION: A list of three coordinates representing the (x,y,z) location of the weapon.

CLASS: The type of weapon.

ammunition

Represents ammunition that the agent can pick up.

LOCATION: A list of three coordinates representing the (x,y,z) location of the ammunition.

CLASS: The type of weapon that this ammunition is for.

healthpack

An item that restores the agent's health when picked up.

LOCATION: A list of three coordinates representing the (x,y,z) location of the healthpack.

path

When the agent requests a path to a location from the game server, the result is stored as a resource of type path.

CALLER: The TRP that requested this path.

NODES: The value of this property is the list of nodes returned by the server. Each node is a data structure containing the number of the node in the path, the name of the node, and the node's location. If the agent moves from one node to the next until it runs out of nodes, it will then be able to move directly to the destination.

patrol_point

When patrolling, the agent randomly chooses a point from a list of patrol points, and stores it as a resource of type patrol_point.

LOCATION: Three coordinates representing the (x,y,z) location of the patrol point.

time

If the agent needs to time an activity (like a pause before doing the next action), it checks the clock and stores the result as a resource of type time.

CALLER: The TRP that stored this time.

VALUE: The time on the clock when this resource was created.

prediction

Stores information about the movement of other players in the game.

FOR: The name of the player whose movement is being predicted.

BY: The TRP that created this resource.

LOCATION: The predicted position of the player based on the last observed position and velocity of that player.

current_node

Stores the agent's current location in a path, created by the GOTO program.

NODE: The name of the node closest to the agent.

CALLER: The TRP that called GOTO.

request

Used to keep track of information requested from the server.

WHAT: What type of information is requested. Usually has the value path, although it could also be reachable (to check if the agent can move to a location directly).

QUERYID: An identification string used to match this request with the information returned by the server.

CALLER: The TRP requesting the information.

STARTLOCATION: The agent's current location.

ENDLOCATION: The location to which the agent is requesting a path, or for which the agent is requesting a reachability check.

Appendix D

Code for the Components of GRUE

This appendix contains Pop-11 code for those components of GRUE that are not application-specific. The code is divided into four files. The first file contains the rulesets that implement the actual components of the architecture, and the additional three files contain functions that are called from the architecture components or the TRPs. This code can be run in Poplog with the Sim_Agent ruleset installed.

D.1 File: grue_components.p

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_components.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  This file contains the rulesets for the main components
;;;  of GRUE.  (Note that a lot of the work is done in
;;;  functions in other files.)
;;;


;;; Simple threshold filter
define :ruleset grue_agent_filter;
    [DLOCAL
     [prb_allrules = true]
```

```
         [prb_walk = false]
         [prb_walk_fast = false]
         [prb_chatty = false]
         [prb_show_ruleset = false]
         [prb_show_conditions = false]
         [cycle_limit = 1]
         ];

        RULE filter
        [GOAL ?name ?priority ?type ??condition][->>g1]
        [WHERE priority <= filter_threshold]
        ==>
        [DEL ?g1]
        [SAYIF DEBUG filter removed goal ?name with priority <= filter_threshold]


        RULE filter_tasks
        [TASKLIST == ]
        ==>
        [POP11 filter_tasks()]
        [SAYIF DEBUG filtered tasks]

    enddefine;


    ;;; Match on situations that shouldn't happen, and print out debugging info
    ;;; if they do.
    define :ruleset grue_agent_debugger;
        [DLOCAL
         [prb_allrules = false]
         [prb_walk = false]
         [prb_walk_fast = false]
         [prb_chatty = false]
         [prb_show_ruleset = false]
         [prb_show_conditions = false]
         [cycle_limit = 1]
         ];


        RULE hole_discrepancy
        [AVAILABLE == [[ID ?hole] [TYPE hole] == [Distance ?dist] == ] ==]
        [WHERE dist = 0]
        [WHERE sim_x(hole) /= sim_x(sim_myself)]
        [WHERE sim_y(hole) /= sim_y(sim_myself)]
        ==>
        [SAY debugger error case:  ?hole has distance 0 in resource but not at
             same coords as agent]
        [POP11 sim_pr_db(sim_myself);]
        [STOP]
        RULE carried_tiles_check
        [LVARS
         [num = n_tiles(sim_myself)]
         [carried = carried_tiles(sim_myself)]
         ]
        [WHERE length(carried) /= num]
        ==>
        [SAY Inconsistent carried tiles!  n_tiles is ?num and list of carried
             tiles is ?carried]


        RULE bug_catcher
        [UNRUNNABLE fill_hole]
        [NOT TASKLIST == [TASK GetTile ==] ==]
        ==>
        [LVARS
         [tiles = n_tiles(sim_myself)]
         [stacks_grabbed = num_tile_stacks_grabbed(sim_myself)]
         [tiles_dropped = num_tiles_dropped(sim_myself)]
         [tiles_carried = carried_tiles(sim_myself)]
         ]
        [SAY debugger error case:  fill_hole unrunnable and get_tile not generated]
        [SAY agent has ?tiles tiles, has grabbed ?stacks_grabbed stacks and has
             dropped ?tiles_dropped tiles]
        [SAY agent is carrying:  ?tiles_carried]
        [POP11 sim_pr_db(sim_myself);]
        [POP11 sim_stop_scheduler();]

    enddefine;


    ;;; Task builder.  Creates a list of tasks, where each task is a goal
    ;;; with the condition replaced by the name of a TRP ruleset.
    ;;; Note that the list is unsorted, it must be sorted before being used
    ;;; by the arbitrator.
    define :ruleset grue_agent_task_builder;
        [DLOCAL
         [prb_allrules = false]
         [prb_walk = false]
         [prb_walk_fast = false]
         [prb_chatty = false]
         [prb_show_ruleset = false]
         [prb_show_conditions = false]
         [cycle_limit = 0]
         ];

        ;;; Create a tasklist if there isn't one.
        RULE empty_task_list
        [NOT TASKLIST ==]
        [GOAL ?name ?priority ?type ??condition] [->>g1]
        [PROGRAM ?progname ??condition]
        ==>
        [DEL ?g1]
        [TASKLIST [TASK ?name ?priority ?type ?progname]]
        [SAYIF DEBUG created task list]

        ;;; Add a task to the existing tasklist.
        RULE build_task
        [GOAL ?name ?priority1 ?type ??condition] [->>g1]
        [PROGRAM ?progname ??condition]
        [TASKLIST ??tasklist][->>t]
        ==>
        [DEL ?g1]
        [DEL ?t]
        [SAYIF DEBUG adding task ?name to task list ??tasklist]
        [TASKLIST [TASK ?name ?priority1 ?type ?progname] ??tasklist]

        ;;; Print some useful information if debug is on.
        RULE debug
        [PROGRAM ==][->>p]
        [GOAL ==][->>g]
        [[TASK ?name ?priority ?type ?progname]] [->>t]
        ==>
        [SAYIF DEBUG Found program: ?p]
        [SAYIF DEBUG Found goal: ?g]
        [SAYIF DEBUG Found task list: ?t]
    enddefine;


    ;;; Arbitrator
    ;;; The arbitrator runs the TRPs, binding variables as necessary.
    ;;; The arbitrator is made up of several rulesets.  Stage numbers are
    ;;; left over from early papers.

    ;;; This ruleset just sorts the tasklist, and removes any unrunnable tasks.
    define :ruleset grue_agent_arbitrator_stage_two;
        [DLOCAL
         [prb_allrules = false]
         [prb_walk = false]
         [prb_walk_fast = false]
         [prb_chatty = false]
         [prb_show_ruleset = false]
         [prb_show_conditions = false]
         [cycle_limit = 1]
         ];

        ;;; Sort the tasks with POP11 sort facility
        RULE sort
        [TASKLIST ==] [->>task_list]
        ==>
        [LVARS [sorted =cons("TASKLIST", sort_tasks(tl(task_list)))]
               [cleaned = remove_unrunnable(sorted)]]
        [DEL ?task_list]
        [ADD ??cleaned]
        [SAYIF DEBUG Arbitrator sorted task list: ??cleaned]

    enddefine;


    ;;; This ruleset runs the tasks, creating the proposed action list.
    ;;; This must restore each ruleset, then take control back when the
    ;;; program is done.
    ;;; Programs should use 'do' to put actions in an actions list, and
    ;;; all rules must end with POPRULESET.
    define :ruleset grue_agent_arbitrator_stage_three;
        [DLOCAL
         [prb_allrules = false]
         [prb_walk = false]
         [prb_walk_fast = false]
         [prb_chatty = false]
         [prb_show_ruleset = false]
         [prb_show_conditions = false]
         [cycle_limit = 0]
         ];

        ;;; Initialize the 'current task' notation
        RULE init_current
        [NOT CURRENT ==]
        ==>
        [CURRENT temp 42]  ;;; CURRENT is followed by the program name and priority


        ;;; Make a copy of the tasklist
        RULE copy_tasklist
        [TASKLIST ??tasks]
        [NOT TEMPTASKLIST ==]
        ==>
        [TEMPTASKLIST ??tasks]


        ;;; This calls the current program
        RULE call_program
        [TEMPTASKLIST [TASK ?name ?priority ?type ?progname] ??tasks][->>task_list]
        [CURRENT ?n ?p][->>c]
        ==>
        [DEL ?task_list]
        [DEL ?c]
        [TEMPTASKLIST ??tasks]
        [CURRENT ?name ?priority] ;;; keep record of which task is running
        [SAYIF DEBUG Arbitrator: running program ?progname with priority ?priority]
        [PUSHRULESET ?progname]

    enddefine;


    ;;; This ruleset removes conflicting actions from the action list.
    define :ruleset grue_agent_arbitrator_stage_four;
        [DLOCAL
         [prb_allrules = false]
         [prb_walk = false]
         [prb_walk_fast = false]
         [prb_chatty = false]
         [prb_show_ruleset = false]
         [prb_show_conditions = false]
         [cycle_limit = 0]
         ];


        ;;; run the actions stored in the actions list and remove the contents
        ;;; of the list from the database.
       RULE run_actions
       [ACTS ??acts_list] [->>a]
       [CONFLICTS ??conflicts_list] [->>c]
       ==>
       [POP11 run_actions(a, c)] ;;;the run_actions() function checks for conflicts

    enddefine;


    ;;; Runs at the end of the cycle; Cleans up temporary information and
    ;;; prepares for the next cycle.
    define :ruleset clean_up;

       [DLOCAL
        [prb_allrules = true]
        [prb_walk = false]
        [prb_walk_fast = false]
        [prb_chatty = false]
        [prb_show_ruleset = false]
        [prb_show_conditions = false]
        [cycle_limit = 1]
       ];


       RULE clean_up
       [TEMPTASKLIST][->>t]
       ==>
       [DEL ?t]
       RULE store_score
       ==>
       [POP11 (score(sim_myself) :: score_per_cycle(sim_myself)) ->
              score_per_cycle(sim_myself)]


       ;;; move bindings to an old_bindings list so can be used next cycle if
       ;;; possible
       RULE undo_bindings
       [BINDINGS ??b]
       [SHAREDBINDS ??s][->>shared]
       ==>
       [DEL ?shared]
       [NOT OLDBINDS ==]
       [NOT OLDSHAREDBINDS ==]
       [SAYIF CYCLE at end of cycle bindings are ??b]
       [LVARS [tiles = carried_tiles(sim_myself)]
       [ntiles = n_tiles(sim_myself)]]
       [SAYIF CYCLE at end of cycle agent is carrying ?ntiles tiles which are
                    ?tiles]
       [POP11 remove_all_bindings()] ;;; put all bindings back in available
       [POP11 merge_resources()]  ;;; merge together any pieces that were split
       [POP11 prb_flush([NOTE ==])] ;;; clear out all notes
       [OLDBINDS ??b]
       [OLDSHAREDBINDS ??s]  ;;; shared bindings just move to the other list
       [SHAREDBINDS]


    enddefine;
```

D.2 File: grue_binding.p

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_binding.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; This file contains binding functions and related utilities
;;; for the GRUE architecture.
;;;


;;; given a list of bindings, return one that matches variable_name if found
;;; otherwise return the empty list
```

```
define get_resource(variable_name, bindings_list) -> resource;
    lvars variable_name, bindings_list, resource;

    [] -> resource;

    for bound in bindings_list do
        if (not(null(bound))) and (hd(hd(bound)) = variable_name) then
            bound(2) -> resource;
        endif;
    endfor;
enddefine;



;;; Takes a resource, and a list of desired properties and returns the
;;; number of properties which match.
;;; This is used for matching both required and preferred properties.
;;; (Note - could be slightly more efficient for required properties as
;;;  once one thing doesn't match we don't care about the rest.)
define number_matches(resource, desired_props) -> result;
    lvars resource, desired_props, result, dproperty, aproperty, matches_found;

    0 -> matches_found;

    for dproperty in desired_props do
        for aproperty in resource do

            if ((dproperty = aproperty) or ;;; ordinary match
               (range_match(dproperty, aproperty)) or ;;; numerical match
               (div_match(dproperty, aproperty)) or ;;; divisible property
               ;;; or wildcard match
               ((dproperty(1) = aproperty(1)) and (hd(tl(dproperty)) = "*")))
            then
              matches_found + 1 -> matches_found;
              quitloop;
            endif;
        endfor;
    endfor;

    matches_found -> result;

enddefine;


;;; Given a resource that is going to be bound, adjust any values that
;;; need to be changed (i.e. because it's divisible)
;;; The get_binding flag is true if the caller is going to use the resource
;;; for the binding, and false if the caller wants to remainder for the
;;; available list.
define adjust_value(resource, required_list, get_binding) -> nresource;
```

```
lvars resource, required_list, nresource, rproperty, aproperty;
lvars range, value, optimum;

resource -> nresource;

for aproperty in resource do ;;; 1 for
    ;;; If the available property isn't divisible, then don't do
    ;;; anything.
    if (is_div(aproperty)) then  ;;; 1 if

        for rproperty in required_list do  ;;; 2 for

            ;;; See if this is the requested property
            if (aproperty(1) = rproperty(1)) then ;;; 2 if

            ;;; two cases now:  the requested property is a range,
            ;;; and it isn't.

                ;;; range matches.  Only want to divide if the available
                ;;; amount is bigger than the range, so
                if (is_range(rproperty)) then ;;; 3 if
                    expand_range(tl(rproperty)) -> range;
                    optimal_value(range) -> optimum;

                    ;;; if the available amount is not the best we can get
                    if ((optimum /= "inf") and
                       not(optimum > aproperty(3)))  then  ;;; 4 if
                        ;;; Work out the remainder, and update the resource
                        if (get_binding) then
                            optimum -> value
                        else
                            aproperty(3) - optimum -> value;
                        endif;

                        update_property_in_resource(hd(aproperty),
                                                    [DIV ^value],
                                                    resource) -> nresource;
                    endif;  ;;; 4 if

                ;;; Otherwise the request was for an exact number...
                elseif (isnumber(rproperty(2)) and
                       ;;; number requested was less than available amount
                       (rproperty(2) < aproperty(3))) then

                    if (get_binding) then
                        rproperty(2) -> value;
                    else
                        aproperty(3) - rproperty(2) -> value;
                    endif;
```

```
                        update_property_in_resource(hd(aproperty),
                                   [DIV ^value], nresource) -> nresource;
                    endif;  ;;; 3 if
                endif; ;;; if the properties match  ;;; 2 if
            endfor; ;;; rproperty 2 for
        endif;  ;;; is_div(aproperty) 1 if
    endfor; ;;; aproperty  1 for
enddefine;



;;; bound_check() takes a variable, a program name, and a rule number
;;; and checks to see if there's an appropriate resource in the list of
;;; already bound variables.  if so, it makes a note in the database.
;;; (see comments on bind for complete description)
define bound_check(variable, progname, rule_number);
    lvars bound_list, num_required, num_wanted, variable, progname,
    lvars rule_number, binding, old_note, new_note, preempted;

    false -> preempted;

    ;;; pull the bound list out of the database
    prb_in_database([BINDINGS ==]) -> bound_list;

    if not(bound_list) then
        '***Bound_check error:  no bindings list found!' =>
    else
        tl(bound_list) -> bound_list;

        listlength(variable(2)) -> num_required;
        listlength(variable(3)) -> num_wanted;

        prb_in_database([^progname ^rule_number ==]) -> old_note;

        ;;; This is for suspension code, which isn't used.
        if (old_note) then
            ;;;'length of old note is '><length(old_note) =>
        endif;

        ;;; This is still suspension stuff.
        ;;; If the program has already been marked unrunnable, don't bother
        ;;; checking any more.
        ;;; But ignore previous note if checking first rule.
        if ((rule_number < 1) or not(old_note) or length(old_note) < 4) then

            ;;; look in the bound list for an appropriate binding
            for binding in bound_list do

                ;;; we only check required properties as it's not
                ;;; going to actually make the binding  (note - if we wanted
                ;;; to keep track of what it's waiting for, we should
                ;;; collect the whole list of resources and rank order them
                ;;; by number of matching preferred properties.)
                if (number_matches(binding(2), variable(2)) = num_required)
                then
                    [NOTE ^progname ^rule_number] -> new_note;
                    true -> preempted;
                    quitloop;
                endif;
            endfor;

            if (not(preempted)) then
                [NOTE ^progname ^rule_number unrunnable] -> new_note;
            endif;

            ;;; Flush the old note if there is one
            if (old_note) then
                prb_flush(old_note);
            endif;

            ;;; Put the new one in instead
            prb_add(new_note);
        endif;

    endif;  ;;; bindings list ok

enddefine;




;;; suspension_bind() does the same thing as bind, but also collects
;;; information to tell whether a program is unrunnable or whether the
;;; necessary resources are in use by another program.  To do this, it
;;; takes several arguments:  the variable to be bound, the name of the
;;; program, the number of the rule, and a flag indicating whether the
;;; variable is negated.  If the variable cannot be bound and is not negated,
;;; the bound list is checked for appropriate bindings.  If there is an
;;; appropriate resource in the bound list, a note is made in the database.
;;; The note is a pair containing the name of the program and the rule
;;; number.  (Note:  we could include the variable and the resource here
;;; as well, if it turns out to be necessary.)  If a variable is found that
;;; is not negated and cannot bind even when bound resources are included,
;;; the note has the item unrunnable added (so it looks like
;;; [progname rulenum unrunnable].  Once all the rules in the program have
;;; been considered, if any note for that program does not contain unrunnable
;;; then it can be suspended (if suspension is enabled).  Otherwise, the
;;; program is truly unrunnable.
define suspension_bind(variable, progname, rule_number, negated) -> binding;
```

```
lvars variable, binding, highest, best, available_list, num_required;
lvars old_resource, old_bindings_list, num_required, num_wanted, name;
lvars progname, rule_number, negated, possibles;

-1 -> highest;
[] -> best;
[] -> possibles;


;;; Fetch the available list
prb_in_database([AVAILABLE ==]) -> available_list;
if not(available_list) then
    '***Bind error:  no available resources list found!' =>
else

;;; Debug extra:  run bound_check to see if there's an alternative
;;; that's already bound
bound_check(variable, progname, rule_number);

;;; Pull the tag off the available list
tl(available_list) -> available_list;

listlength(variable(2)) -> num_required;
listlength(variable(3)) -> num_wanted;

;;; First get resource from old_bindings list
prb_in_database([OLDBINDS ==]) -> old_bindings_list;
tl(old_bindings_list) -> old_bindings_list;
get_resource(variable(1), old_bindings_list) -> old_resource;

;;; Then check if it's in the available list, and if so get
;;; the current version
if (old_resource /= []) then
    available(old_resource) -> old_resource;
else
    ;;; check the shared bindings list instead
    prb_in_database([OLDSHAREDBINDS == ]) -> old_bindings_list;
    tl(old_bindings_list) -> old_bindings_list;
    get_resource(variable(1), old_bindings_list) -> old_resource;
    if (old_resource /= []) then
    available(old_resource) -> old_resource;
    endif;
endif;


;;; if variable bound in old bindings list and the resource is
;;; still available, then bind it.
if (persistent_binding and
   (old_resource /= false) and
   (not(null(old_resource))) and
   (number_matches(old_resource, variable(2)) = num_required)) then
        adjust_value(old_resource, variable(2), true)-> old_resource;
        [^variable ^old_resource] -> binding;
else
    ;;; look in the available list for an appropriate binding
    for resource in available_list do
        ;;; check required properties.
        if (number_matches(resource, variable(2)) = num_required) then
            ;;; Work out if the match relied on the available resource
            ;;; being divisible, and if so, update the appropriate
            ;;; property or properties in resource.
            adjust_value(resource, variable(2), true)-> resource;

            ;;; Put resource on a list of possible matches
            (resource :: possibles) -> possibles;
        endif;
    endfor;

    ;;; Now find the best preferred properties match
    lvars non_range_props, match_list, range_props, list, num_matches;

    [] -> non_range_props;
    [] -> match_list;
    [] -> range_props;
    [] -> list;

    get_non_range_properties(variable(3)) -> non_range_props;
    get_range_properties(variable(3)) -> range_props;

    ;;; Second loop for preferred properties
    ;;; Non-range properties only.  Store the number of matches
    ;;; in the match list.
    for resource in possibles do
        number_matches(resource, non_range_props) -> num_matches;
        ([^resource ^num_matches] :: match_list) -> match_list;
    endfor;

    ;;; This computes the resource that matches each range best, and
    ;;; updates the number of matches for those resources appropriately
    ;;; A resource only counts as a match for preferred range
    ;;; properties when it is the best match for that range.
    match_ranges(match_list, range_props) -> match_list;

    ;;; Go through the match list and find the resource with the
    ;;; highest number of preferred matches
    for list in match_list do
        if list(2) >= highest then
            list(2) -> highest;
            list(1) -> best;
        endif;
        endfor;

        ;;; Now see if there are any with the same number and pick
        ;;; one randomly if so.
        lvars same_list = [];
        for list in match_list do
            if (list(2) = highest) then
                (list(1) :: same_list) -> same_list;
            endif;
        endfor;

        if (length(same_list) > 1) then
            lvars ran = random(length(same_list));
            same_list(ran) -> best;
        endif;

        ;;; check whether there's a binding and return it if so.
        if (best /= []) then
            hd(variable) -> name;
            [^variable ^best] -> binding;

            if (not(negated)) then
                ;;;'bound '><variable><' to '><best =>
            endif;

        else
            ;;; no binding was found in the available list
            [] -> binding;

            ;;; need to check the bound list here and make a note if
            ;;; appropriate. (only if the variable is actually supposed
            ;;; to be bound)
            ;;; Again, this is for the suspension code which isn't used.
            if (not(negated)) then
                ;;;bound_check(variable, progname, rule_number);
            endif;

            endif;
        endif;
    endif;
enddefine;


;;; Given a program name, return true if it is suspendable and false if it is
;;; unrunnable.  (Doesn't check for runnable, so don't call it with a runnable
;;; program.)
define suspendable(progname) -> result;
    lvars progname, result;

    if prb_in_database([progname = unrunnable]) then
```

```
        false -> result;
    else
        true -> result;
    endif;
enddefine;


;;; Given a binding [variable resource] make it permanent by removing
;;; the resource from the available resources list and adding the binding
;;; to the BINDINGS list.
define make_permanent_binding(binding);
    lvars available_list, binding, bindings_list, resource;
    lvars variable, adjusted_resource;

    ;;; If permanent_binding is false, then don't bother with any of this.
    ;;; (Set in GRUE_control_script.p)
    if (permanent_binding) then

        prb_in_database([AVAILABLE ==]) -> available_list;
        if not(available_list) then
            '***Error making binding permanent - no available resource list!'
            =>
        else

            ;;; Pull tag off head of available_list
            tl(available_list) -> available_list;

            hd(binding) -> variable;

            ;;; Loop through available list and find a resource that matches
            ;;; the ID of binding(2).
            for resource in available_list do
                if (hd(property_from_binding("ID", binding)) =
                    hd(property_from_resource("ID", resource))) then

                        ;;; remove resource from available_list
                        delete(resource, available_list) -> available_list;

                        ;;; ONLY PUT THE ADJUSTED VALUE BACK IF IT HAS CHANGED
                        adjust_value(resource, variable(2), false) ->
                             adjusted_resource;

                        if (resource /= adjusted_resource) then
                            (adjusted_resource :: available_list) ->
                                available_list;
                        endif;

                endif;
            endfor;
            ;;; remove old copy of available list
            prb_flush([AVAILABLE ==]);

            ;;; put modified copy back
            ("AVAILABLE" :: available_list) -> available_list;
            prb_add(available_list);

            ;;; Modify bindings list in database
            prb_in_database([BINDINGS ==]) -> bindings_list;
            if (bindings_list) then
                tl(bindings_list) -> bindings_list;
                [BINDINGS ^binding ^^bindings_list] -> bindings_list;
            else
                ;;; create bindings list if there isn't one
                [BINDINGS ^binding] -> bindings_list;
            endif;

            prb_flush([BINDINGS ==]);
            prb_add(bindings_list);

        endif;
    endif;
enddefine;


;;; Store a shared binding on a list so the variable can bind to the same
;;; resource next cycle
define make_shared_binding(binding);
    lvars bindings_list;

    ;;; Modify bindings list in database
    prb_in_database([SHAREDBINDS ==]) -> bindings_list;
    if (bindings_list) then
        tl(bindings_list) -> bindings_list;

        [SHAREDBINDS ^binding ^^bindings_list] -> bindings_list;
    else
        ;;; create bindings list if there isn't one
        [SHAREDBINDS ^binding] -> bindings_list;
    endif;

    prb_flush([SHAREDBINDS ==]);
    prb_add(bindings_list);

enddefine;


;;; Given a variable name, remove the corresponding binding from the
;;; BINDINGS list and move the resource back to the available resources list
define remove_binding(var_name);
```

```
    lvars available_list, bindings, bound, variable_name, resource;

    prb_in_database([AVAILABLE ==]) -> available_list;
    prb_in_database([BINDINGS ==]) -> bindings;
    tl(bindings) -> bindings;

    false -> value;

    for bound in bindings do

        if (hd(hd(bound)) = variable_name) then

            bound(2) -> resource;

            ;;; remove from bindings list
            delete(bound, bindings) -> bindings;
            ("BINDINGS" :: bindings) -> bindings;

            prb_add(bindings);

            tl(available_list) -> available_list;
            (resource :: available_list) -> available_list;
            ("AVAILABLE"::available_list) -> available_list;

            prb_add(available_list);

            quitloop;
        endif;
    endfor;
enddefine;



;;; This was for the suspension code which isn't used.
;;;define remove_notes();
;;;    prb_flush([NOTE ==]);
;;;enddefine;



;;; Remove all bindings from the BINDINGS list and move them back to the
;;; available resources list.
define remove_all_bindings();
    lvars available_list, bindings, value, bound, resource;

    prb_in_database([AVAILABLE ==]) -> available_list;
    tl(available_list) -> available_list;

    prb_in_database([BINDINGS ==]) -> bindings;
    tl(bindings) -> bindings;
    false -> value;

    ;;; for all bindings, pull out the resource and stick it on available_list
    for bound in bindings do
        bound(2) -> resource;
        (resource :: available_list) -> available_list;
    endfor;

    prb_flush([BINDINGS ==]);

    ;;; put modified bindings and available lists back.
    [BINDINGS] -> bindings;
    prb_add(bindings);

    prb_flush([AVAILABLE ==]);
    ("AVAILABLE"::available_list) -> available_list;
    prb_add(available_list);

enddefine;
```

D.3 File: grue_property_utilities.p

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_property_utils.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  This file contains functions for inspecting the
;;;  properties of resources.
;;;


;;; Given a variable name and the name of a property, get the value of
;;; that property out of the resource bound to the variable name.
;;; (Returns a list of values.)
define property_from_global(prop_name, variable_name) -> value;
    lvars prop_name, variable_name, value, bindings, bound, prop, properties;

    ;;; Get the bindings list.
    prb_in_database([BINDINGS ==]) -> bindings;

    tl(bindings) -> bindings;

    false -> value;
    ;;; Loop through the bindings list looking for the correct binding.
    for bound in bindings do
        if (hd(bound) = variable_name) then
            bound(2) -> properties;
            for prop in properties do
                if (hd(prop) = prop_name) then
                    tl(prop) -> value;
                endif;
            endfor;
        endif;
    endfor;

enddefine;


;;; Given a property name and a binding, get the value of
;;; that property out of the resource in the binding.  (Returns a list.)
define property_from_binding(prop_name, binding) -> value;
    lvars prop_name, binding, value, resource, prop;

    binding(2) -> resource;

    false -> value;

    for prop in resource do
        if (hd(prop) = prop_name) then
            tl(prop) -> value;
        endif;
    endfor;
enddefine;


;;; Given a property name and a resource, get the value of
;;; that property out of the resource.  Returns a list.
define property_from_resource(prop_name, resource) -> value;
    lvars prop_name, resource, value, prop;

    false -> value;

    for prop in resource do
        if (hd(prop) = prop_name) then
            tl(prop) -> value;
        endif;
    endfor;

enddefine;


;;; Given a resource, a property name, and a value, change the value of the
;;; given property to be the given value.  This is a recursive procedure!
;;; Value should be a list containing all the values, including the DIV tag
;;; if needed.
define update_property_in_resource(prop_name, value, resource) -> nresource;
    lvars prop_name, value, resource, nresource;

    if (not(null(resource))) then

        ;;; If we have the right property, then stick the property name on
        ;;; the value and add it back to the list.  Recursing on the remainder
        ;;; of the list shouldn't change the rest of the list unless there
        ;;; happens to be another instance of the property in the list (which
        ;;; there shouldn't be, really.)
        if (hd(hd(resource)) = prop_name) then
            ((prop_name::value) :: (update_property_in_resource(prop_name,
                                     value, tl(resource)))) -> nresource;

        ;;; Otherwise, just put the property back on the list and recurse
        ;;; on the remainder.
        elseif (not(null(resource))) then
            (hd(resource) :: (update_property_in_resource(prop_name,
                                  value, tl(resource))))  -> nresource;
        endif;

    else ;;; If we have reached the end of the list, then stop the recursion.
        [] -> nresource;
    endif;
enddefine;
```

D.4 File: grue_number_utilities.p

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_number_utilities.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  This file contains functions for handling the
;;;  special range notation.
;;;


uses fmatches;

;;; Compare two matches where a match is of the format
;;; [name [resource, number]].  NO LONGER USED
define better_match(match1, match2) -> answer;
```

```
    lvars match1, match2, answer;

    if (hd(tl(tl(match1))) >= hd(tl(tl(match2)))) then
        true -> answer;
    else
        false -> answer;
    endif;
enddefine;


;;; Compare two lists where the first thing in each list is a number.
;;; Returns true if the head of thing1 is greater than or equal to
;;; the head of thing2.  (Appropriate for use with syssort).
define head_compare(thing1, thing2) -> result;
    lvars thing1, thing2, result;

    if (hd(thing1) >= hd(thing2)) then
        true -> result;
    else
        false -> result;
    endif;
enddefine;


;;; Expands a numerical range, since programmers are allowed to use
;;; shortened notation.
;;; An expanded range consists of two brackets containing a lower bound,
;;; an arrow and a lower bound.  In the shortened range, the arrow is omitted
;;; if no preference ordering is required, and the numbers may be omitted if
;;; on the same side as a soft boundary.  So |# 5 :# specifies a lower bound
;;; of 5, and an implicit upper bound of infinity.  Note that if both
;;; boundaries are soft, it's impossible to tell which side a single number
;;; should go on, so both numbers must be specified.

;;; The range argument should contain only the range information without
;;; names, etc.
define expand_range(range) -> expanded_range;
    lvars range, expanded_range;
    lvars open, close, low, high, arrow;

    ;;; Default values of things that might be omitted.
    "inf" -> low;
    "inf" -> high;
    "--" -> arrow;

    ;;; This uses the pop11 fmatches functionality.  This allows us to
    ;;; specify a pattern that the range must match.  Each piece of the
    ;;; pattern can be specified explicitly or read into a variable.  If
    ;;; using variables, we can specify the type of item to be matched.
    ;;; got everything except arrow, so just use default for arrow
    if (range fmatches [?open ?low:isnumber ?high:isnumber ?close]) then

    ;;; got everything, so done
    elseif (range fmatches [?open ?low:isnumber ?arrow ?high:isnumber ?close])
        then

    ;;; soft boundary on the left means the high number must be specified
    elseif (range fmatches [#: ?high:isnumber |#]) then
        "#:" -> open;
        "|#" -> close;
        "--" -> arrow;
        "inf" -> low;

    ;;; soft boundary on the right means the low number must be specified
    elseif (range fmatches [#| ?low:isnumber :#]) then
        "#|" -> open;
        ":#" -> close;
        "--" -> arrow;
        "inf" -> high;

    ;;; soft boundary on the left with an arrow
    elseif (range fmatches [#: ?arrow ?high:isnumber |#]) then
        "#:" -> open;
        "|#" -> close;
        "inf" -> low;

    ;;; soft boundary on the right with an arrow
    elseif (range fmatches [#| ?low:isnumber ?arrow :#]) then
        "#|" -> open;
        ":#" -> close;
        "inf" -> high;
    endif;


    ;;; Now put all the pieces of the range together
    [^open ^low ^arrow ^high ^close] -> expanded_range;

enddefine;



;;; Check whether a number is within a range.  The range must be an expanded
;;; range as returned by expand_range above.
define grue_in_range(number, range) -> result;
    lvars number, range, result;
    lvars open, close, arrow, high, low;

    false -> result;
    ;;; Check that the range contains all the right things.
    if (not(range fmatches
        [?open ?low ?arrow ?high ?close])) then
        'Error:  grue_in_range got bad range!  '><range =>
    else
        ;;; Check for each possible combination, taking into account
        ;;; hard boundaries, soft boundaries, and inf values

        ;;; If the range is -infinity to +infinity then it matches anything
        if ((high = "inf") and (low = "inf")) then
            true -> result;

        ;;; The next two cases have infinity on one side or the other
        elseif ((high = "inf") and (isnumber(low)) and (number >= low)) then
            true -> result;
        elseif ((low = "inf") and (isnumber(high)) and (number <= high)) then
            true -> result;

        ;;; If the boundaries are numbers, check numerically
        elseif ((isnumber(high)) and (number <= high) and
                (isnumber(low)) and (number >= low)) then
            true -> result;

        ;;; If using soft boundaries, it might match even if the last
        ;;; case failed.
        elseif ((isnumber(high)) and (number <= high) and (open = "#:")) then
            true -> result;
        elseif ((isnumber(low)) and (number >= low) and (close = ":#")) then
            true -> result;

        ;;; If there are soft boundaries on both sides, then it matches
        ;;; anything
        elseif ((open = "#:") and (close = ":#")) then true -> result;

        ;;; These cases were used for debugging.
        elseif ((open = "#:")) then
            ;;;'open is soft' =>
        elseif ((close == ":#")) then
            ;;;'close is soft' =>
        else
            ;;;' no cases matched!' =>
        endif;
    endif;

enddefine;



;;; Given a range, return it's optimal value (highest utility value)
;;; If there is a range of values with equally optimal utility values,
;;; return the minimum one to conserve resources.
;;; The range is assumed to be expanded.  If the optimal value is positive
;;; infinity the return value will be "inf".  If the optimal value is negative
;;; infinity, the return value will be 0.
define optimal_value(range) -> value;
    lvars range, value, low, high, arrow;

    range(2) -> low;
    range(3) -> arrow;
    range(4) -> high;

    ;;; high optimal value
    if (arrow = "->") then
        high -> value;
    elseif (low = "inf") then  ;;; low optimal or minimum, returning 0
        0 -> value;
    else  ;;; low optimal or minimum
        low -> value;
    endif;
enddefine;


;;; A simple test to check if a property is a range property.
define is_range(property) -> result;
    lvars property, result;

    false -> result;

    ;;; Check for presence of a range bracket
    if ((property(2) = "#|") or (property(2) = "#:")) then
        true -> result;
    endif;

enddefine;



;;; Matches ranges for one variable.
;;; Takes a list of matches, which consists of a list of lists
;;; containing a resource and the number of non-range matches.  Also takes
;;; a list of range properties.  Returns the updated match_list
;;; (with the number of matching properties updated to include range matches)
;;; Essentially the match list contains a score for each resource, where the
;;; highest scoring resource is bound to the variable.

;;; Range properties should be the whole property, with the name included.
define match_ranges(match_list, prop_list) -> new_match_list;
    lvars match_list, prop_list, new_match_list;
    lvars prop, expanded_prop_list, range, match, temp_match_list;
    lvars best_match, number;
```

```
[] -> expanded_prop_list;
[] -> temp_match_list;

;;; First expand the ranges.
for prop in prop_list do
    if (not(is_range(prop))) then
        'Error - match_ranges got non-range property.' =>
    else
        (hd(prop) :: expand_range(tl(prop))) ::
                     expanded_prop_list -> expanded_prop_list;
    endif;
endfor;


;;; Then for each range, sort all possible matches. (Throw out anything
;;; that doesn't match)
for range in expanded_prop_list do

    ;;; check whether the resources have this property
    ;;; (might be unnecessary, as required properties have already been
    ;;; checked.)
    for match in match_list do
        property_from_resource(hd(range), hd(match)) -> number;

        ;;; check for div cases
        if (hd(number) = "DIV") then
            number(2) -> number;
        else
            hd(number) -> number;
        endif;

        ;;; Then see if it matches
        if (grue_in_range(number, tl(range))) then
            ((number :: match)::temp_match_list) -> temp_match_list;
        endif;
    endfor;

    ;;; Sort the match list
    syssort(temp_match_list, true, head_compare) -> temp_match_list;


    ;;; Pick either the highest match or the lowest depending which
    ;;; way the arrow goes.
    if ((range(4) = "->") and not(null(temp_match_list))) then
        hd(temp_match_list) -> best_match;
    elseif not(null(temp_match_list)) then
        last(temp_match_list) -> best_match;
    else
        [] -> best_match;
        endif;

        ;;; pull the name off the front of best_match
        if (not(null(best_match))) then
            tl(best_match) -> best_match;

            ;;; and increment the match, updating the real match_list
            ;;; instead of the temporary one.
            delete(best_match, match_list) -> match_list;
            conspair(hd(tl(best_match)) + 1, []) -> temp;
            hd(best_match) :: temp -> best_match;
            (best_match::match_list) -> match_list;

        endif;
    endfor;

    match_list -> new_match_list;
enddefine;


;;; Returns true if the given property is a divisible one, and false
;;; otherwise
define is_div(property) -> value;
    lvars property, value;

    false -> value;

    if (length(property) >= 2) then
        if (property(2) = "DIV") then
            true -> value;
        ;;;else
            ;;;false -> value;
        endif;
    endif;
enddefine;


;;; Returns true if the given resource has a DIV property.
define has_div(resource) -> result;
    lvars resource, result, property;

    false -> result;

    for property in resource do
        if is_div(property) then
            true -> result;
        endif;
    endfor;
enddefine;
```

```
;;; Takes a requested property from a variable and a property from an
;;; available resource, and returns true if the available property value is
;;; a number which falls within a range specified by the requested property.
;;; Calls grue_in_range()
define range_match(requested, available) -> result;
    lvars requested, available, result;
    lvars range, value;

    false -> value;
    false -> result;

    if (requested(1) = available(1)) then

        ;;; Check whether we've got a range
        if (is_range(requested)) then

            ;;; If so, then make sure we're matching it against a number
            ;;; and ignore DIV if it's there
            if (isnumber(available(2))) then
                available(2) -> value;
            elseif (is_div(available)) then
                available(3) -> value;
            else
                ;;;'Bad value for range_match '><available=>
            endif;

            ;;; Expand the range and check whether the
            ;;; value is in the range
            if (isnumber(value)) then
                expand_range(tl(requested)) -> range;
                grue_in_range(value, range) -> result;
            endif;
        endif;
    endif;
enddefine;


;;; Returns true if the requested property (dproperty) matches the
;;; available property (aproperty).  This procedure only handles cases
;;; where the available property is divisible, but it does handle ranges
;;; for those cases.  (Unlike range_match, above, which handles ordinary
;;; range cases but not DIV cases.)
define div_match(dproperty, aproperty) -> result;
    lvars dproperty, aproperty, result, avalue, dvalue, ovalue;

    false -> result;
    if (aproperty(1) = dproperty(1)) then
        if (is_div(aproperty)) then
```

```
            if (is_range(dproperty)) then
                expand_range(tl(dproperty)) -> range;

                ;;; Work out the optimal value for the range, and see if we
                ;;; have enough to bind that amount.
                optimal_value(range) -> ovalue;
                aproperty(3) -> avalue;

                if (ovalue = "inf") then
                    true -> result;
                elseif (avalue >= ovalue) then
                    true -> result;
                endif;

            ;;; if not a range, then it's a match as long as the available
            ;;; amount is more than enough.
            else
                dproperty(2) -> dvalue;
                aproperty(3) -> avalue;

                if (isnumber(dvalue) and isnumber(avalue)) then
                    if (avalue >= dvalue) then
                        true -> result;
                    endif;
                endif;
            endif; ;;; if is_range
        endif; ;;; if is_div
    endif; ;;; if property names match
enddefine;


;;; Given an list of properties from a variable, return a list of the ones
;;; that aren't range properties.
define get_non_range_properties(props_list) -> non_range_props;
    lvars props_list, non_range_props, property;

    [] -> non_range_props;

    for property in props_list do
        if (not(is_range(property))) then
            (property::non_range_props) -> non_range_props;
        endif;
    endfor;

enddefine;



;;; Given an list of properties from a variable, return a list of the ones
;;; that are range properties.
```

```
define get_range_properties(props_list) -> range_props;
    lvars props_list, range_props, property;

    [] -> range_props;

    for property in props_list do
        if (is_range(property)) then
            (property::range_props) -> range_props;
        endif;
    endfor;

enddefine;


;;; Get all properties from a resource that have the DIV keyword
define get_div_properties(resource) -> list;
    lvars resource, list, prop;

    [] -> list;

    for prop in resource do
        if (prop(2) = "DIV") then
            (prop::list) -> list;
        endif;
    endfor;

enddefine;
```

D.5 File: grue_utilities.p

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_utilities.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; This file contains miscellaneous utilities.
;;;


;;; Given a string and a list, construct a new list containing the string
;;; followed by each element of the list separated by spaces.
define print_list_with_header(header, list) -> printstring;
    lvars header, list, printstring, item;

    header -> printstring;
    for item in list do
        printstring><' '><item -> printstring;
    endfor;
enddefine;


;;; Generate a unique id for use in resources, etc.
0-> id_generator;
define generate_id() -> id;
    lvars id;

    id_generator + 1 -> id_generator;

    id_generator -> id;
enddefine;


;;; compare two tasks using priority values
;;; tasks have the format [TASK <name> <priority> <type> <program>]
;;; (works on any list where the third thing is a number)
define higher(task1, task2) -> answer;
    lvars task1, task2, answer;

    if (task1(3) >= task2(3)) then
         true -> answer;
    else
         false -> answer;
    endif;
enddefine;


;;; Opposite of higher.  Works on any list where the third item is a number
define lower(task1, task2) -> answer;
    lvars task1, task2, answer;

    if (task1(3) <= task2(3)) then
         true -> answer;
    else
         false -> answer;
    endif;
enddefine;


;;; Tasks are sorted highest to lowest for arbitration
define sort_tasks(task_list) ->c;
    lvars task_list, c;
    syssort(task_list, true, higher) -> c;
enddefine;
```

```
;;; Actions are sorted lowest to highest for conflict check.
define sort_actions(acts_list) -> c;
    lvars acts_list, c;
    syssort(acts_list, true, lower) -> c;
enddefine;


;;; Removes unrunnable tasks from the task list.  Task list is assumed to
;;; start with TASKLIST.  UNRUNNABLE lines are removed from the database
;;; where relevant.
define remove_unrunnable(tasklist) -> runnable_tasks;
    lvars tasklist, runnable_tasks, task, progname, unrun;

    ;;; end case for recursion
    if (tl(tasklist) = []) then
        tasklist -> runnable_tasks;
    else
        ;;; first thing in tasklist should be TASKLIST
        tasklist(2) -> task;
        task(5) -> progname;

        sim_in_database([UNRUNNABLE ^progname], sim_myself) -> unrun;
        if (unrun) then
            sim_delete_data(unrun, sim_get_data(sim_myself));
            remove_unrunnable(tasklist(1) :: tl(tl(tasklist))) ->
                 runnable_tasks;
        else
            tasklist(1) :: (tasklist(2) ::
                           tl(remove_unrunnable(tasklist(1) ::
                           tl(tl(tasklist))))) -> runnable_tasks;
        endif;
    endif;
enddefine;


;;; special version of division for use in goal generators.  highest priority
;;; possible is 100
define gdiv(a, b) -> c;
    lvars a, b, c;

    if (not(b = 0)) then
        a / b -> c;
    else
        100 -> c;
    endif;
enddefine;



;;; Given a task name and a list of actions, add the actions to an
;;; ACTIONS list in the database for later processing.
define grue_do_action(actions);
    lvars actions, action_list, task_info, task_name, to_be_added, priority;
    lvars act;

    ;;; Get the action list out of the database
    sim_in_database([ACTS ==], sim_myself) -> action_list;


    ;;; If no action list in database (there should always be one)
    ;;; make an empty one
    if not(action_list) then
        [ACTS] -> action_list;
    endif;

    ;;; take the ACTS tag off the action_list
    tl(action_list) -> action_list;

    ;;; Get the current task out of the database
    ;;; If none found, then create an empty one.
    sim_in_database([CURRENT ==], sim_myself) -> task_info;
    if not (task_info) then
         [CURRENT Unknown -42] -> task_info;
    endif;


    task_info(2) -> task_name;
    task_info(3) -> priority;

    for act in actions do

        ;;; Construct the list to be added to the action list.  It consists of
        ;;; the name of the task followed by the list of actions passed to
        ;;; this procedure.
        [^task_name ^act ^priority] -> to_be_added;

        ;;; Add the new list to the tail of the action list
        (to_be_added :: action_list) -> action_list;

    endfor;

    ;;; Stick the ACTS tag back on
    ("ACTS" :: action_list) -> action_list;

    ;;; Delete the old line from the database, and put the new one in.
    sim_delete_data([ACTS ==], sim_get_data(sim_myself));
    sim_add_data(action_list, sim_get_data(sim_myself));

enddefine;
```

```
;;; Pull the list of available resources out and update the nearest object of
;;; the specified type.
define update_nearest(type_value);
    lvars available_list, nearest, temp, current_nearest, resource, value;
    lvars dist, current_nearest, new_nearest;

    prb_in_database([AVAILABLE ==]) -> available_list;

    tl(available_list) -> available_list;

    20 -> nearest;   ;;; nothing should be further away than 10 (sensor limit)
    [] -> temp;  ;;; store the resource temporarily so can add the Nearest tag

    [] -> current_nearest; ;;; the resource that currently has the Nearest tag

    for resource in available_list do
        hd(property_from_resource("TYPE", resource)) -> value;

        ;;; if the resource is the right type
        if (value = type_value) then

            ;;; check whether it's closer than the closest found so far
            hd(property_from_resource("Distance", resource)) -> dist;

            if (dist < nearest) then
                dist -> nearest;
                resource -> temp;
            endif;

            ;;; if this is the current Nearest, then keep track of it
            if property_from_resource("Nearest", resource) then
                resource -> current_nearest;
            endif;

        endif; ;;; if right type
    endfor; ;;; for all resources

    ;;; Assuming the nearest has changed, then modify the available list
    if not(null(temp)) and not(null(current_nearest)) and
        (current_nearest /= temp) then

        ;;; remove old versions of resources
        delete(current_nearest, available_list) -> available_list;
        delete(temp, available_list) -> available_list;

        ;;; remove the nearest tag
        delete([Nearest true], current_nearest) -> current_nearest;

        ;;; add Nearest tag to temp
```

```
            explode(temp);
            [Nearest true];
            listlength(temp) + 1;
            conslist() -> new_nearest;

            ;;; now add modified current_nearest and the new list new_nearest
            ;;; back into the available resources list

            (new_nearest :: available_list) -> available_list;
            (current_nearest :: available_list) -> available_list;
        else
            ;;; Previously used for debugging
        endif;

        ;;; put AVAILABLE tag back on list
        ("AVAILABLE" :: available_list) -> available_list;

        ;;; put it back in the database after getting rid of the old one
        prb_flush([AVAILABLE ==]);
        prb_add(available_list);

enddefine;


;;; Go through the available list, and remove all objects for which there
;;; is no new_sense_data.
;;; This function had to be modified for the Unreal Tournament agent, and was
;;; moved into a different file.  However, it must exist for each agent.
define remove_vanished_objects;
   lvars available_list, resource, id, updated_avail_list;
   lvars sense_data;

   [] -> updated_avail_list;

   prb_in_database([AVAILABLE ==]) -> available_list;
   tl(available_list) -> available_list;

   for resource in available_list do
       hd(property_from_resource("ID", resource)) -> id;

       if (istile(id) or ishole(id) or isobstacle(id)) then
            ;;; check database for a new_sense_data line - if there is one,
            ;;; then put the resource back
            prb_in_database([new_sense_data ^id ==]) -> sense_data;

            if (sense_data or not(in_range(sim_myself, id))) then
                (resource :: updated_avail_list) -> updated_avail_list;
            else
                if (istile(id)) then
                    if (member(id, carried_tiles(sim_myself))) then
```

```
                        tiles_minus_one(sim_myself);
                        delete(id, carried_tiles(sim_myself)) ->
                           carried_tiles(sim_myself);
                        lost_tiles(sim_myself) + 1 -> lost_tiles(sim_myself);
                    else
                        ;;; Previously used for debugging
                    endif;
                else
                    ;;; Previously used for debugging
                endif;
            endif;
       else
            ;;; if not a tile, hole, or obstacle we don't care so put it back
            (resource :: updated_avail_list) -> updated_avail_list;
       endif;
   endfor;

   ;;; put AVAILABLE tag back on list
   ("AVAILABLE" :: updated_avail_list) -> updated_avail_list;

   ;;; put it back in the database after getting rid of the old one
   prb_flush([AVAILABLE ==]);
   prb_add(updated_avail_list);
enddefine;



;;; Takes an action list and a conflict list.  The ACTS and CONFLICTS
;;; tags should be removed, and the action list should be sorted from
;;; lowest priority to highest priority.
define run_actions_recursive(actionlist, conflicts);
    lvars actionlist, conflicts, act1, act2, action_type1, action_type2,
     ok_to_run;

    ;;; quit if no actions
    if (actionlist /= []) then
        hd(actionlist) -> act1;

        act1(2) -> action_type1;
        action_type1(2) -> action_type1;

        ;;; check act1 against each other action
        for act2 in tl(actionlist) do

            act2(2) -> action_type2;
            action_type2(2) -> action_type2;

            if (member([^action_type1 ^action_type2], conflicts)) then
                false -> ok_to_run;
            endif;
        endfor;

        if ok_to_run then
            prb_eval(act1(2));
        endif;

        ;;; recurse on the remainder of the list
        run_actions_recursive(tl(actionlist), conflicts);
    endif;

enddefine;


;;; Called by arbitrator to run actions in action list.  Eliminates the
;;; lower priority action if the pair of action types is found in the
;;; CONFLICTS list.
;;; [CONFLICTS [type1 type2] [type3 type4] ...]
;;; [ACTS [taskname action priority] ...]
;;; where action = [do action_type arg1 ...]
define run_actions(actionlist, conflicts);
    lvars act1, actionlist, conflicts;

    ;;; need to pull off ACTS and CONFLICTS tags
    tl(actionlist) -> actionlist;
    tl(conflicts) -> conflicts;

    ;;; sort actions lowest to highest
    sort_actions(actionlist) -> actionlist;

    ;;; Check whether multiple_actions flag is on
    ;;; (set in GRUE_control_script.p).  Otherwise just run the highest
    ;;; priority action.
    if (multiple_actions) then
        run_actions_recursive(actionlist, conflicts);
        [] -> actionlist;
    elseif (not(null(actionlist))) then
        ;;; Go through actionlist and execute any actions that are not
        ;;; world actions (i.e. add/remove resource are allowed, go_to is
        ;;; not)
        lvars action, external_actionlist;
        [] -> external_actionlist;
        for action in actionlist do
            if ((action(2))(2) = "add_resource") or
                ((action(2))(2) = "remove_resource") then
             prb_eval(action(2));
            else
                (action::external_actionlist) -> external_actionlist;
            endif;
        endfor;
        ;;; note that sort_actions leaves the highest priority action
        ;;; at the end of the list, but the above loop reverses it.

        if (not(null(external_actionlist))) then
            hd(external_actionlist) -> act1;
            prb_eval(act1(2));

            [] -> actionlist;
        endif;

    endif;


    ;;; Put action list back in database
    ("ACTS" :: actionlist) -> actionlist;

    ;;; put it back in the database after getting rid of the old one
    prb_flush([ACTS ==]);
    prb_add(actionlist);
enddefine;


;;; If the given resource is in the available list, returns the version from
;;; the available list; false
;;; otherwise.  Only checks the ID property.  Assumes that IDs are unique.
;;; (This means that a resource will be treated as available even if it has
;;; a DIV property with a different size than that given.)  Note also that
;;; this procedure assumes every resource has an ID.  Two resources that do
;;; not have ID properties will always match.
define available(resource) -> result;
    lvars resource, id, available_list, available_resource, result;

    false -> result;

    ;;; Pull out the available list and get rid of the tag
    prb_in_database([AVAILABLE ==]) -> available_list;
    if (available_list) then
        tl(available_list) -> available_list;
    endif;

    ;;; Get the ID of the resource we're looking for
    hd(property_from_resource("ID", resource)) -> id;

    for available_resource in available_list do
        ;;; If the available resource matches the one we're looking for
        if id = hd(property_from_resource("ID", available_resource)) then
            available_resource -> result;
        endif;
    endfor;
```

```
enddefine;


;;; Locate any resources that are parts of the same thing (that is the IDs
;;; must be the same, but the DIV amounts different) and merge them together.
;;; Should be run at the beginning of a cycle, after bound resources are
;;; made available again.
define merge_resources();
    lvars available_list, to_be_merged, available_resource, id;
    lvars amount, property, property_two, merged_resource;
    lvars merged_ids;  ;;; list of ids that have already been merged

    [] -> merged_ids;

    ;;; Pull out the available list and get rid of the tag
    prb_in_database([AVAILABLE ==]) -> available_list;
    if (available_list) then
        tl(available_list) -> available_list;
    endif;

    ;;; check for things that need to be merged
    for to_be_merged in available_list do
      if (has_div(to_be_merged)) then
        property_from_resource("ID", to_be_merged) -> id;

        ;;; If this item is already in the merged list, then skip it.
        ;;; (checked items get deleted in the inner loop, but the outer
        ;;; loop appears to work with a static version of the list from
        ;;; before the deletions.)
        if not(member(id, merged_ids)) then
            to_be_merged -> merged_resource;
            (id::merged_ids) -> merged_ids;

            ;;; find things to merge it with
            for available_resource in available_list do
                if (available_resource /= to_be_merged) then
                    if id = property_from_resource("ID", available_resource)
                    then
                        for property in merged_resource do
                            if is_div(property) then
                                property_from_resource(hd(property),
                                       available_resource) -> property_two;
                                property(3) + property_two(2) -> amount;
                                update_property_in_resource(hd(property),
                                       [DIV ^amount], merged_resource) ->
                                   merged_resource;
                            endif;
                        endfor;

                        ;;; need to delete available_resource
                        delete(available_resource, available_list) ->
                            available_list;
                    endif;
                endif;
            endfor;

            ;;; After updating div properties, get rid of old to_be_merged,
            ;;; and replace it with merged_resource
            delete(to_be_merged, available_list) -> available_list;
            (merged_resource::available_list) -> available_list;

        endif;
      endif;
    endfor;

    ;;; put AVAILABLE tag back on list
    ("AVAILABLE" :: available_list) -> available_list;

    ;;; put it back in the database after getting rid of the old one
    prb_flush([AVAILABLE ==]);
    prb_add(available_list);
enddefine;


;;; remove low priority tasks
define filter_tasks;
   lvars task_list, task, id, updated_task_list;

   [] -> updated_task_list;

   prb_in_database([TASKLIST ==]) -> task_list;
   tl(task_list) -> task_list;

   for task in task_list do
       if (task(3) > filter_threshold) then
           (task::updated_task_list) -> updated_task_list;
       endif;
   endfor;

   ;;; get rid of old task list and put updated one back.
   prb_flush([TASKLIST ==]);

   ("TASKLIST"::updated_task_list) -> updated_task_list;
   prb_add(updated_task_list);
enddefine;
```

Appendix E

Tileworld Specific Code

This appendix contains some of the code necessary to run a Tileworld agent. The code consists of the goal generators, the actions that can be executed by the agent and the TRPs used by this agent.

E.1 Tileworld Goal Generators

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      goal_generators.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  This file contains the goal generator rulesets for
;;;  Tileworld agent.
;;;


;;; Generate the get tile goal or update the priority of an existing task.
define :ruleset grue_agent_generate_get_tile;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
```

```
 ];

;;; For the case where a tile is visible.
;;; Generate get tile with priority proportional to the distance of the
;;; tile.
RULE generate_get_tile
[LVARS [nts = n_tiles(sim_myself)]]
[NOT GOAL GetTile ==]
[AVAILABLE ??res_start [[ID ?tile]
                        [TYPE tile]
                        [TIME Infinite]
                        [Distance ?dist] ==
                        [Timestamp ?time] ==
                        [Nearest true]] ??res_end]
[NOT TASKLIST == [TASK GetTile ==] ==]
[WHERE nts < 1]
==>
[LVARS [priority=gdiv(100.0,dist)]]
[TESTADD GOAL GetTile ?priority  NonMaintenance HAVE_TILE]
[SAYIF DEBUG generated goal GetTile with priority ?priority]


;;; For cases where no tiles are visible.  Generate get tile anyway, but
;;; with a low priority
RULE generate_get_tile2
[LVARS [nts = n_tiles(sim_myself)]]
[NOT GOAL GetTile ==]
[NOT AVAILABLE ==  [[ID ?tile]
                    [TYPE tile]
                    [TIME Infinite]
                    [Distance ?dist] ==
                    [Timestamp ?time] == ] ==]
[NOT TASKLIST == [TASK GetTile ==] ==]
[WHERE nts < 1]
==>
[TESTADD GOAL GetTile 10.0 NonMaintenance HAVE_TILE]
[SAYIF DEBUG generated goal GetTile with priority 10.0]


;;; If there's an existing GetTile task, update the priority according to
;;; the distance to the tile.
RULE update_get_tile
[LVARS [nts = n_tiles(sim_myself)]]
[AVAILABLE ??res_start [[ID ?tile] [TYPE tile] [TIME Infinite]
                        [Distance ?dist] == [Timestamp ?time] ==
                        [Nearest true]] ??res_end]
[TASKLIST ??start
          [TASK GetTile ?oldpriority NonMaintenance ?progname]
          ??end][->>t]
[WHERE nts < 1]
```

```
    ==>
    [LVARS [priority=gdiv(100.0,dist)]]
    [DEL ?t]
    [TASKLIST ??start [TASK GetTile ?priority NonMaintenance ?progname] ??end]
    [SAYIF DEBUG update GetTile priority to ?priority]

enddefine;



;;; Generate the avoid obstacle goal, or update the priority of an existing
;;; task.
define :ruleset grue_agent_generate_avoid_obstacle;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

    ;;; Generate the goal, with priority proportional to distance
    ;;; of obstacle from agent.  Priority should be highest when obstacle is
    ;;; next to the agent.
    RULE generate_avoid_obstacle
    [NOT GOAL Avoid ==]
    [AVAILABLE ??res_start [[ID ?obstacle] [TYPE obstacle] ==
                            [Distance ?dist] == [Timestamp ?time] ==
                            [Nearest true] ==] ??res_end]
    [NOT TASKLIST == [TASK Avoid ==] ==]
    ==>
    [LVARS [priority=gdiv(100.0,dist)]]
    [SAYIF DEBUG generating avoid obstacle goal - priority is ?priority]
    [TESTADD GOAL Avoid ?priority  NonMaintenance NO_OBSTACLE]


    ;;; Update the priority of an existing goal, with priority proportional
    ;;; to distance of obstacle from agent.
    RULE update_avoid_obstacle
    [AVAILABLE ??res_start [[ID ?obstacle] [TYPE obstacle] ==
                            [Distance ?dist] == [Timestamp ?time] ==
                            [Nearest true] ==] ??res_end]
    [TASKLIST ??start [TASK Avoid ?oldpriority NonMaintenance ?progname]
              ??end][->>t]
    ==>
    [LVARS [priority=gdiv(100.0,dist)]]
    [DEL ?t]
    [SAYIF DEBUG updating avoid obstacle goal - dist is ?dist new priority
```

```
                ?priority]
    [TASKLIST ??start [TASK Avoid ?priority NonMaintenance ?progname] ??end]

enddefine;


;;; Generate the fill hole goal or update the priority of an existing task.
define :ruleset grue_agent_generate_fill_hole;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

    ;;; Generate fill_hole goal if the agent has a tile.  Priority is
    ;;; proportional to distance from hole to agent.
    RULE generate_fill_hole
    [LVARS [nts = n_tiles(sim_myself)]]
    [NOT GOAL FillHole ==]
    [AVAILABLE ??res_start  [[ID ?hole] [TYPE hole] [TIME Infinite]
                             [Distance ?dist] == [Timestamp ?time] ==
                             [Nearest true]] ??res_end]
    [NOT TASKLIST == [TASK FillHole ==] ==]
    [WHERE nts > 0]
    ==>
    [LVARS [priority=gdiv(100.0,dist)]]
    [TESTADD GOAL FillHole ?priority  NonMaintenance NOT(HAVE_TILE)]
    [SAYIF DEBUG generated goal FillHole with priority ?priority]


    ;;; Generate fill_hole with a low priority if the agent
    ;;; has a tile and there is no hole visible.
    RULE generate_fill_hole2
    [LVARS [nts = n_tiles(sim_myself)]]
    [NOT GOAL FillHole ==]
    [NOT AVAILABLE == [[ID ?hole] [TYPE hole] [TIME Infinite]
                       [Distance ?dist] == [Timestamp ?time] ==] == ]
    [NOT TASKLIST == [TASK FillHole ==] ==]
    [WHERE nts > 0]
    ==>
    [TESTADD GOAL FillHole 10.0 NonMaintenance NOT(HAVE_TILE)]
    [SAYIF DEBUG generated goal FillHole with priority 10.0]


    ;;; Update the priority of an existing fill_hole task.
    RULE update_fill_hole
    [LVARS [nts = n_tiles(sim_myself)]]
    [AVAILABLE ??res_start [[ID ?hole] [TYPE hole] [TIME Infinite]
                            [Distance ?dist] == [Timestamp ?time] ==
                            [Nearest true]] ??res_end]
    [TASKLIST ??start
              [TASK FillHole ?oldpriority NonMaintenance ?progname]
              ??end] [->>t]
    [WHERE nts > 0]
    ==>
    [LVARS [priority=gdiv(100.0,dist)]]
    [DEL ?t]
    [TASKLIST ??start [TASK FillHole ?priority NonMaintenance ?progname] ??end]
    [SAYIF DEBUG update FillHole priority to ?priority]

enddefine;
```

E.2 Tileworld Actions

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_tile_actions.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This file contains the actions that can be
;;; executed by the Tileworld agent.



;;; This method moves the agent, and also any tiles being carried by the
;;; agent.  The agent moves one square in the specified direction.
define :method go_to(agt:tile_agent, dir);
    lvars dx =0, dy=0, obs_dir=8, tile;

    ;;; Loop through the list of tile IDs carried by the agent and
    ;;; move them.
    if (n_tiles(agt) > 0) then ;;; there's a tile
        for tile in carried_tiles(agt) do
            if (same_location(agt, tile)) then ;;; probably unnecessary
                move_to(tile, dir);  ;;; move the tile
            else
                'Go_to error: Carried tile not at same location as agent!' =>
            endif;
         endfor;
    else
         ;;;'n_tiles <= 0' =>
    endif;
    move_to(agt, dir);  ;;; move the agent

enddefine;



;;; Given a resource, add it to the available resources list.
define :method add_resource(agt:grue_tile_agent, resource);
    lvars avail_list;

    sim_in_database([AVAILABLE ==], agt) -> avail_list;

    (resource :: tl(avail_list))-> avail_list;
    ("AVAILABLE" :: avail_list)->avail_list;

    sim_flush([AVAILABLE ==], agt);
    sim_add_data(avail_list, sim_get_data(agt));
enddefine;


;;; Remove a resource from the available resource list, given it's ID.
;;; If the variable is not available, but is bound, then remove the binding
;;; from the BINDINGS list.  (Otherwise, the resource will be put back in
;;; the available list at the end of the cycle.)
;;; Note that this requires knowing it's ID.
define :method remove_resource(agt:grue_tile_agent, id);
    lvars avail_list, done, bindings;

    sim_in_database([AVAILABLE ==], agt) -> avail_list;
    tl(avail_list) -> avail_list;

    false -> done;

    for resource in avail_list do
        ;;; If this is the correct resource, then delete it
        if (hd(resource))(2) = id then
            delete(resource, avail_list) -> avail_list;
            true -> done;
            quitloop;
        endif;
    endfor;

    ;;; If the resource wasn't in the available list, then check the bindings
    ;;; list.
    if not(done) then
        sim_in_database([BINDINGS ==], agt) -> bindings;
        tl(bindings) -> bindings;

        for bound in bindings do
```

```
            bound(2) -> resource;
            if (hd(resource))(2) = id then
                delete(bound, bindings) -> bindings;
                true -> done;
                quitloop;
            endif;
        endfor;

        sim_flush([BINDINGS ==], agt);
        ("BINDINGS" :: bindings) -> bindings;
        sim_add_data(bindings, sim_get_data(agt));
    endif;

    sim_flush([AVAILABLE ==], agt);
    ("AVAILABLE" :: avail_list) -> avail_list;
    sim_add_data(avail_list, sim_get_data(agt));
enddefine;


;;; Increase the number of carried tiles by one.
define :method tiles_plus_one(agt:tile_agent);
    n_tiles(agt) + 1 -> n_tiles(agt);
enddefine;


;;; Decrease the number of carried tiles by one.
define :method tiles_minus_one(agt:tile_agent);
    n_tiles(agt) - 1 -> n_tiles(agt);
enddefine;


;;; Pick up a tile (or tile stack), increasing the number of carried tiles
;;; and storing the tile ID.
define :method grue_grab_tile(agt:tile_agent, tile);

    ;;; Check that the tile is actually in the same location as the agent
    if (same_location(agt, tile)) then

        ;;; add the tile ID to the agent's list
        (tile::(carried_tiles(agt))) -> carried_tiles(agt);

        ;;; Increase the number of carried tiles (or tile stacks)
        tiles_plus_one(agt);

        num_tile_stacks_grabbed(agt) + 1 -> num_tile_stacks_grabbed(agt);
        num_tiles_grabbed(agt) + size(tile) -> num_tiles_grabbed(agt);
    else
        '** grue_grab_tile error - tile not in same location as agent!' ->
    endif;
```

```
enddefine;


;;; Drop a tile stack, decreasing the number of carried tiles, removing the
;;; agent's stored tile ID, and removing the tile and hole objects
;;; (if the tile filled a hole).  If the stack was dropped in a hole, but did
;;; not fill it, then recompute the size of the hole.
;;; Compute the score as appropriate.
define :method grue_drop_tile(agt:tile_agent, tile, num, hole);
    lvars agt, tile, num, hole, sc;

    0 -> sc;


    ;;; check the tile and hole are in the right place
    if (same_location(tile, hole) and (same_location(agt, tile))) then  ;;; 1

        if (not(istile(tile)) or not(ishole(hole))) then   ;;; 2
            'Drop tile error - wrong type of object!' =>
        else ;;; 2
            ;;; num is number of tiles to drop.  If num is <1, then
            ;;; don't do anything.  If num >= size of the hole, then
            ;;; the hole will be filled.

            ;;; error check
            if (num > size(tile)) then ;;; 4
                'Drop tile: tried to drop '><num><' tiles but only had '><
                                             size(tile) =>
                size(tile) -> num;
            endif; ;;; 3


            ;;; If the number of tiles is enough to fill the hole
            if (num >= size(hole)) then
                (size(tile) - num) -> size(tile);

                ;;; compute score - 1 for dropping a tile that doesn't match
                ;;; the shape of the hole, or 3 if it matches the hole
                if (shape(tile) = shape(hole)) then  ;;; 4
                    (size(hole) * 3) + sc -> sc;
                    (num_shape_matched(agt) + size(hole))
                                                -> num_shape_matched(agt);
                else
                    size(hole) + sc -> sc;
                endif;  ;;; 3

                if (num > size(hole)) then
                    ((num - size(hole)) + num_wasted(agt)) -> num_wasted(agt);
                endif;
                ;;; hole is filled.  remove the hole and add a 20 point bonus.
                sc + 20 -> sc;
                remove_object(agt, hole, TW);

                ;;; store information about this experiment
                (num_holes_filled(agt) + 1) -> num_holes_filled(agt);
                num + num_tiles_dropped(agt) -> num_tiles_dropped(agt);


                ;;; If using whole tile stack, then remove the tile stack
                ;;; and recompute n_tiles
                if (0 >= size(tile)) then
                    remove_object(agt, tile, TW);
                    tiles_minus_one(agt);
                    delete(tile, carried_tiles(agt)) -> carried_tiles(agt);
                endif;


            ;;; Otherwise the hole isn't filled
            elseif (num > 0) then

                ;;; recompute size of hole
                (size(hole) - num) -> size(hole);
                (size(tile) - num) -> size(tile);

                num + num_tiles_dropped(agt) -> num_tiles_dropped(agt);

                ;;; compute score
                if (shape(tile) = shape(hole)) then ;;; 4
                    (num * 3) + sc -> sc;
                    (num_shape_matched(agt) + num) -> num_shape_matched(agt);
                else
                    num + sc -> sc;
                endif; ;;; 3

                ;;; This value isn't actually used.
                score_so_far(hole) + sc -> score_so_far(hole);

                ;;; If using whole tile stack, then remove the tile stack
                ;;; and recompute n_tiles
                if (0 >= size(tile)) then ;;; 4
                    remove_object(agt, tile, TW);
                    tiles_minus_one(agt);
                    delete(tile, carried_tiles(agt)) -> carried_tiles(agt);
                endif;

            else
                'Drop tile: tried to drop '><num><' tiles!'=>
            endif;
```

```
        endif;
    else
        ;;;'Drop tile error - tile and agent not in same location!'=>
    endif; ;;; 0

    score(agt) + sc -> score(agt);

    (sc :: individual_scores(agt)) -> individual_scores(agt);

enddefine;
```

E.3 Get Tile TRP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      get_tile.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The get_tile TRP for the GRUE Tileworld agent
;;;
;;;


;;; The get_tile program acquires a tile if there is one visible.  If the
;;; tile disappears, the program will switch to a different tile if possible.
;;; If no tiles are visible, the agent will move in a random direction each
;;; cycle until it sees a tile.
define :ruleset get_tile;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

    ;;; Done if the agent is carrying a tile.
    RULE done
    [LVARS [tiles = n_tiles(sim_myself)]]
    [WHERE tiles > 0]
    ==>
    ;;; No change to bindings in this rule, and no actions either
    [SAYIF DEBUG Get_Tile is done]
    [UNRUNNABLE get_tile]
```

```
[POPRULESET]


;;; Agent picks the tile up when it is on the same square as the tile.
RULE at_tile
[LVARS [tile_binding=suspension_bind([get_tile_tile [[TYPE tile]
                                                      [Distance 0]] []],
                                      get_tile, 1, false)]]
[WHERE tile_binding /= []]
[LVARS [dir_binding = suspension_bind([get_tile_dir [[TYPE direction]
                                                      [Toward tile]] []],
                                       get_tile, 1, false)]
]
[WHERE dir_binding /= []]
;;; Match the target here so it can be deleted.
[LVARS [target_binding=suspension_bind([get_tile_target
                                        [[TYPE target]] []],
                                        get_tile, 1, false)]]
[WHERE n_tiles(sim_myself) < 1]
==>
[POP11 make_shared_binding(tile_binding)]
[POP11 make_permanent_binding(dir_binding)]  ;;; being removed
[POP11 if (not(null(target_binding))) then
         make_permanent_binding(target_binding); endif;]  ;;; being removed
[LVARS [dir_id = hd(property_from_binding("ID", dir_binding))]
    [tile_id = hd(property_from_binding("ID", tile_binding))]]
[POP11 grue_do_action([[do grue_grab_tile ^tile_id ]
                       [do remove_resource ^dir_id]])]
[POP11 if (drawgraphics(TW)) then
        (-2, -2) -> sim_coords(target(sim_myself)) endif;]
;;; This is a cheat to avoid writing a second rule (which would be exactly
;;; like this one, but not match the target to allow opportunistic
;;; acquisition of tiles before the target is created).
[POP11 if not(null(target_binding)) then
        hd(property_from_binding("ID", target_binding)) -> temp;
        grue_do_action([[do remove_resource ^temp]]); endif;]
[SAYIF DEBUG Get_Tile: agent is at the tile]
[POPRULESET]


;;; Move in the specified direction
RULE move
;;; Find the target we're heading for.
[LVARS
[target_binding = suspension_bind([get_tile_target [[TYPE target]] []],
                                  get_tile, 2, false)]]
[WHERE target_binding /= []]
;;; Compute the direction to the target, and check whether the target tile
;;; is still there
[LVARS
```

```
 [target_tile = hd(property_from_binding("Value", target_binding))]
 [tile_binding = suspension_bind([get_tile_tile [[ID ^target_tile]
                                                 [TYPE tile]] [] ],
                                 get_tile, 2, false)]
]
[WHERE tile_binding /= []]
;;; Now check whether there's a direction, and whether it's the correct
;;; direction to get to the target.
[LVARS
 [dir_binding = suspension_bind([get_tile_dir
                                 [[TYPE direction] [Toward tile]] []],
                                get_tile, 2, false)]
]
[WHERE dir_binding /= []]
[LVARS
 [[targetx targety] = sim_coords(target_tile)]
 [target_dir = get_direction(sim_x(sim_myself), sim_y(sim_myself),
                             targetx, targety)]
 [dir_value = hd(property_from_binding("Value", dir_binding))]
]
[WHERE dir_value = target_dir]
==>
[POP11 make_shared_binding(tile_binding)]
[POP11 make_shared_binding(target_binding)]
;;; Not changing dir binding, so allow avoid_obstacle to use it too
[POP11 make_shared_binding(dir_binding)]
[POP11 grue_do_action([[do move_to ^dir_value] ])]
[LVARS
 [tile_id = hd(property_from_binding("ID", tile_binding))]
 [[tilex tiley] = sim_coords(tile_id)]
]
[POP11 if (drawgraphics(TW)) then
          (tilex, tiley) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Get_Tile: agent moved towards tile in direction ?dir_value]
[POPRULESET]


;;; This rule removes a bad target, in the case where the tile has
;;; vanished and no direction has been computed.
RULE remove_bad_target_no_direction
[LVARS
 [target_binding = suspension_bind([get_tile_target
                                    [[TYPE target]] []],
                                   get_tile, 3, false)]
 [dir_binding = suspension_bind([get_tile_dir
                                 [[TYPE direction] [Toward tile]] []],
                                get_tile, 3, true)]
]
[WHERE not(null(target_binding))]
[WHERE null(dir_binding)]
```

```
[LVARS
 [tile_id = hd(property_from_binding("Value", target_binding))]
 [tile_binding = suspension_bind([get_tile_tile [[TYPE tile]
                                                 [ID ^tile_id]] []],
                                  get_tile, 3, true)]
]
[WHERE null(tile_binding)]
==>
[POP11 make_shared_binding(tile_binding)]
[POP11 make_permanent_binding(target_binding)] ;;; being removed
[LVARS [target_id = hd(property_from_binding("ID", target_binding))]]
[POP11 grue_do_action([[do remove_resource ^target_id]])]
[POP11 if (drawgraphics(TW)) then
        (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Get_Tile: removed bad target ?target_id]
[POPRULESET]


;;; This rule removes a bad direction when the target is still valid.
RULE remove_bad_direction
[LVARS
 [target_binding = suspension_bind([get_tile_target
                                    [[TYPE target]] []],
                                   get_tile, 4, false)]
 [dir_binding = suspension_bind([get_tile_dir
                                 [[TYPE direction] [Toward tile]] []],
                                get_tile, 4, false)]
]
[WHERE not(null(target_binding))]
[WHERE not(null(dir_binding))]
[LVARS
 [tile = hd(property_from_binding("Value", target_binding))]
 [tile_binding = suspension_bind([get_tile_tile
                                  [[TYPE tile] [ID ^tile]] []],
                                 get_tile, 4, false)]
]
[WHERE not(null(tile_binding))]
==>
[POP11 make_shared_binding(tile_binding)]
[POP11 make_shared_binding(target_binding)]
[POP11 make_permanent_binding(dir_binding)] ;;; being removed
[LVARS [target_id = hd(property_from_binding("ID", target_binding))]
[dir_id = hd(property_from_binding("ID", dir_binding))]]
[POP11 grue_do_action([[do remove_resource ^dir_id]])]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Get_Tile: removed bad target direction ?dir_id]
[POPRULESET]
```

```
;;; This rule removes a bad target (for the case where the target tile has
;;; vanished.)  Also removes the direction, as it will have to be
;;; recomputed for the new target.  Don't have to do much in this rule -
;;; if the target were good one of the above rules would have matched.
RULE remove_bad_target
[LVARS
 [target_binding = suspension_bind([get_tile_target
                                   [[TYPE target]] []],
                                   get_tile, 5, false)]
 [dir_binding = suspension_bind([get_tile_dir
                                [[TYPE direction] [Toward tile]] []],
                               get_tile, 5, false)]
]
[WHERE not(null(target_binding))]
[WHERE not(null(dir_binding))]
[LVARS
 [tile_id = hd(property_from_binding("Value", target_binding))]
 [tile_binding = suspension_bind([get_tile_tile
                                 [[TYPE tile] [ID ^tile_id]] []],
                                get_tile, 5, true)]
]
[WHERE null(tile_binding)]
==>
[POP11 make_permanent_binding(target_binding)] ;;; being removed
[POP11 make_permanent_binding(dir_binding)]    ;;; being removed
[LVARS [target_id = hd(property_from_binding("ID", target_binding))]
[dir_id = hd(property_from_binding("ID", dir_binding))]]
[POP11 grue_do_action([[do remove_resource ^target_id]
                       [do remove_resource ^dir_id]])]
[POP11 if (drawgraphics(TW)) then
        (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Get_Tile: removed bad target ?target_id and direction ?dir_id]
[POPRULESET]



;;; This recomputes the direction toward the target tile
RULE pick_direction_to_target
[LVARS
 [target_binding = suspension_bind([get_tile_target [[TYPE target]] []],
                                   get_tile, 6, false)]
]
[WHERE target_binding /= []]
[LVARS
 [tile_id = hd(property_from_binding("Value", target_binding))]
 ;;; prefer tile stacks that are young, nearby, and big.
 [tile_binding = suspension_bind([get_tile_tile
                                 [[ID ^tile_id] [TYPE tile]] []],
                                get_tile, 6, false)]
]
```

```
[WHERE tile_binding /= []]
[LVARS
 [[tilex tiley] = sim_coords(tile_id)]
 [agentx = sim_x(sim_myself)]
 [agenty = sim_y(sim_myself)]
 [dir = get_direction(agentx, agenty, tilex, tiley)]
 [dir_id = generate_id()]
]
[WHERE dir > 0]
==>
;;; Neither the tile nor the target is being changed.
[POP11 make_shared_binding(tile_binding)]
[POP11 make_shared_binding(target_binding)]
[POP11 grue_do_action([[do add_resource [[ID ^dir_id]
                                         [TYPE direction]
                                         [TIME Infinite]
                                         [Value ^dir] [Toward tile]]]
                       [do go_to ^dir]])]
[POP11 if (drawgraphics(TW)) then
         (tilex, tiley) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Get_Tile: agent picked direction ?dir, already had target
             ?tile_id]
[POPRULESET]


;;; This chooses a direction to move toward the tile, and removes a
;;; random direction.
RULE pick_dir_and_remove_random
[LVARS
;;; prefer tile stacks that are young, nearby and big
[tile_binding = suspension_bind([get_tile_tile
                                 [[TYPE tile]] [[Distance #| 1 <- 10 :#]
                                  [Size #| 1 -> :#] [Age Young]]],
                                get_tile, 7, false)]
[random_binding = suspension_bind([get_tile_random_dir
                                   [[TYPE direction] [Toward random]] []],
                                  get_tile, 7, false)]
]
[WHERE not(null(tile_binding))]
[WHERE not(null(random_binding))]
[LVARS
 [tile_id = hd(property_from_binding("ID", tile_binding))]
 [random_id = hd(property_from_binding("ID", random_binding))]
 [[tilex tiley] = sim_coords(tile_id)]
 [agentx = sim_x(sim_myself)]
 [agenty = sim_y(sim_myself)]
 [dir = get_direction(agentx, agenty, tilex, tiley)]
]
[WHERE dir > 0]
==>
```

```
[LVARS
 [dir_id = generate_id()]
 [target_id = generate_id()]
]
;;; This is not being changed
[POP11 make_shared_binding(tile_binding)]
[POP11 make_permanent_binding(random_binding)]  ;;; being removed
[POP11 if (drawgraphics(TW)) then
         (tilex, tiley) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do add_resource
                          [[ID ^dir_id] [TYPE direction] [TIME Infinite]
                           [Value ^dir] [Toward tile]]]
                        [do remove_resource ^random_id]
                        [do add_resource [[ID ^target_id] [TYPE target]
                                          [Value ^tile_id] [Xcoord ^tilex]
                                          [Ycoord ^tiley]] ]
                        [do move_to ^dir] ])]
[SAYIF DEBUG Get_Tile: agent picked direction ?dir and
                        target ?tile_id and removed random dir]
[POPRULESET]


;;; This chooses a direction to move toward the tile
;;; It makes a direction resource which is consumed in the move rule
;;; above.
RULE pick_direction
[LVARS
 ;;; prefer tile stacks that are young, nearby, and big.
 [tile_binding = suspension_bind([get_tile_tile
                               [[TYPE tile]] [[Distance #| 1 <- 10 :#]
                                [Size #| 1 -> :#] [Age Young]]],
                               get_tile, 8, false)]
]
[WHERE tile_binding /= []]
[LVARS
    [tile_id = hd(property_from_binding("ID", tile_binding))]
]
[LVARS
    [[tilex tiley] = sim_coords(tile_id)]
    [agentx = sim_x(sim_myself)]
    [agenty = sim_y(sim_myself)]
    [dir = get_direction(agentx, agenty, tilex, tiley)]
    [dir_id = generate_id()]
    [target_id = generate_id()]
]
[WHERE dir > 0]
==>
;;; The tile is not being changed.
[POP11 make_shared_binding(tile_binding)]
[POP11 grue_do_action([[do add_resource
                            [[ID ^dir_id] [TYPE direction] [TIME Infinite]
                             [Value ^dir] [Toward tile]]]
                        [do add_resource
                            [[ID ^target_id] [TYPE target]
                             [Value ^tile_id] [Xcoord ^tilex]
                                              [Ycoord ^tiley]] ]
                        [do move_to ^dir] ])]
[POP11 if (drawgraphics(TW)) then
         (tilex, tiley) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Get_Tile: agent picked direction ?dir and picked target
                       ?tile_id]
[POPRULESET]


;;; Remove the random direction if the agent has hit an obstacle.  It will
;;; be recomputed by the rule below.
RULE change_random
[LVARS
 [random_binding = suspension_bind([get_tile_random_dir
                                  [[TYPE direction] [Toward random]][]],
                                  get_tile, 9, false)]
]
[WHERE not(null(random_binding))]
[LVARS
 [dir = hd(property_from_binding("Value", random_binding))]
 [obstacle_binding = suspension_bind([get_tile_obstacle
                                    [[TYPE obstacle] [Distance 1]
                                    [Direction ^dir]] []], get_tile, 7,
                                    false)]
]
[WHERE not(null(obstacle_binding))]
==>
[POP11 make_permanent_binding(random_binding)]
[POP11 make_shared_binding(obstacle_binding)]
[LVARS
 [id = hd(property_from_binding("ID", random_binding))]
]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do remove_resource ^id]])]
[SAYIF DEBUG get_tile:  Hit an obstacle and removed random direction]
[POPRULESET]


;;; Move in a random direction
RULE random_move
[LVARS
 [random_binding = suspension_bind([get_tile_random_dir
                                  [[TYPE direction] [Toward random]] []],
                                  get_tile, 10, false)]
```

```
    ]
    [WHERE random_binding /= []]
    ==>
    [LVARS
     [dir = hd(property_from_binding("Value", random_binding))]
     [id = hd(property_from_binding("ID", random_binding))]
    ]
    [POP11 make_shared_binding(random_binding)]
    [POP11 grue_do_action([[do move_to ^dir]])]
    [POP11 if (drawgraphics(TW)) then
             (-2, -2) -> sim_coords(target(sim_myself)) endif;]
    [SAYIF DEBUG Get_Tile:  agent moving in random direction ?dir]
    [POPRULESET]


    ;;; Pick a random direction
    RULE random_pick
    ==>
    [LVARS [dir=random(4)]
     [id = generate_id()]]
    [POP11 if (drawgraphics(TW)) then
             (-2, -2) -> sim_coords(target(sim_myself)) endif;]
    [POP11 grue_do_action([[do add_resource [[ID ^id] [TYPE direction]
                                            [TIME Infinite] [Value ^dir]
                                            [Toward random]]]
                           [do move_to ^dir]])]
    [SAYIF DEBUG Get_Tile: agent picked random direction ?dir]
    [POPRULESET]


    ;;; This should never happen...and is not actually implemented anyway.
    RULE suspend
    [WHERE suspendable(get_tile)]
    ==>
    [SAYIF DEBUG get_tile is suspendable]
    [UNRUNNABLE get_tile]
    [POPRULESET]


    ;;; This really shouldn't happen.
    RULE kill
    ==>
    [SAYIF DEBUG get_tile is not suspendable]
    [UNRUNNABLE get_tile]
    [POPRULESET]

enddefine;
```

E.4 Fill Hole TRP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      fill_hole.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The fill hole TRP for the GRUE Tileworld agent.
;;;
;;;


;;; Basically the same as get_tile, but finds holes instead.  Only runs
;;; if the agent is carrying a tile, and if the tile disappears this program
;;; will be stopped.
define :ruleset fill_hole;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


    RULE done
    ;;; Not holding a tile.
    [LVARS [tilestacks = n_tiles(sim_myself)]]
    [WHERE tilestacks < 1]
    ;;; Find any old direction and target information.
    [LVARS [direction = suspension_bind([fill_hole_dir
                                        [[TYPE direction] [Toward hole]] []],
                                        fill_hole, 10, false)]
    [target = suspension_bind([fill_hole_target
                              [[TYPE htarget]] []], fill_hole, 10, false)]]
    ==>
    ;;; Clean up old information if it's still there (This is a cheat to avoid
    ;;; writing lots of extra rules)
    [POP11 if not(null(direction)) then
            hd(property_from_binding("ID", direction)) -> id;
            make_permanent_binding(direction);
            grue_do_action([[do remove_resource ^id]]); endif; ]
    [POP11 if not(null(target)) then
            hd(property_from_binding("ID", target)) -> id;
            make_permanent_binding(target);
            grue_do_action([[do remove_resource ^id]]); endif; ]
    [SAYIF DEBUG Fill_Hole is done]
```

```
[UNRUNNABLE fill_hole]
[POPRULESET]


;;; If the agent is at the hole and has a tile, then drop the tile in the
;;; hole.
RULE at_hole
;;; If the agent is on a hole
[LVARS [num_tiles = n_tiles(sim_myself)]]
[WHERE num_tiles > 0]
[LVARS [hole = suspension_bind([fill_hole_hole
                               [[TYPE hole] [Distance 0]] []],
                              fill_hole, 1, false)]  ]
[WHERE hole /= []]
[LVARS [hole_size = hd(property_from_binding("Size", hole))]]
;;; Make sure the agent still has a tile, and bind an amount less than or
;;; equal to the size of the hole.
;;; THIS NEEDS TO BE A CARRIED TILE -- BUT IF TWO ARE EQUIVALENT AND BIND
;;; RETURNS WRONG ONE, THIS WILL FAIL.  SO NEED TO GET TILE ID OUT OF
;;; CARRIED LIST.
[LVARS
 [tile_id = hd(carried_tiles(sim_myself))]
 [tile = suspension_bind([fill_hole_tile [[ID ^tile_id] [TYPE tile]
                                          [Distance 0]
                                          [Size #| 1 -> ^hole_size |#]]
                                          [[Size #| 1 -> ^hole_size |#]]],
                         fill_hole, 1, false)] ]
[WHERE tile /= []]
[LVARS
 [dir_binding = suspension_bind([fill_hole_dir
                                 [[TYPE direction] [Toward hole]] []],
                                fill_hole, 1, false)]
]
[LVARS [target_binding=suspension_bind([fill_hole_target
                                        [[TYPE htarget]] []],
                                       fill_hole, 1, false)]]
==>
[POP11 make_permanent_binding(tile)] ;;;size will change; could be removed
[POP11 make_permanent_binding(hole)] ;;;size will change; could be removed
;;; Another cheat to reduce the number of rules
[POP11 if (not(null(target_binding))) then
         make_permanent_binding(target_binding); endif;]
[POP11 if (not(null(dir_binding))) then
         make_permanent_binding(dir_binding); endif;]
[LVARS [tile_id = hd(property_from_binding("ID", tile))]
       [hole_id = hd(property_from_binding("ID", hole))]
       [stack_size = hd(tl(property_from_binding("Size", tile)))]
]
;;; Once the tile and hole are removed from the environment, the world
;;; interface layer should take them out of the available resources list
;;; (next cycle).
[POP11 grue_do_action([[do grue_drop_tile ^tile_id ^stack_size ^hole_id]])]
;;;[POP11 grue_do_action([[do remove_resource ^dir_id]])]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[POP11 if not(null(target_binding)) then
         hd(property_from_binding("ID", target_binding)) -> temp;
         grue_do_action([[do remove_resource ^temp]]); endif;]
[POP11 if not(null(dir_binding)) then
         hd(property_from_binding("ID", dir_binding)) -> temp;
         grue_do_action([[do remove_resource ^temp]]); endif;]
[SAYIF DEBUG Fill_Hole: at hole, dropping tiles]
[POPRULESET]


;;; Move in the specified direction.  Check for target hole, then make
;;; sure that stored direction is still the correct direction to get to
;;; the target hole.
RULE move
[LVARS
 ;;; Get target
 [target_binding = suspension_bind([fill_hole_target
                                    [[TYPE htarget]] []],
                                    fill_hole, 2, false)]]
[WHERE target_binding /= []]
[LVARS
 ;;; Check that hole still exists
 [target_hole = hd(property_from_binding("Value", target_binding))]
 [hole_binding = suspension_bind([fill_hole_hole
                                  [[ID ^target_hole] [TYPE hole]] [] ],
                                  fill_hole, 2, false)]
]
[WHERE hole_binding /= []]
[LVARS
 ;;; Bind direction, and make sure the agent is still carrying a tile
 [dir_binding = suspension_bind([fill_hole_dir
                                 [[TYPE direction] [Toward hole]] []],
                                 fill_hole, 2, false)]
 [tile_id = hd(carried_tiles(sim_myself))]
 [tile_binding = suspension_bind([fill_hole_tile
                                  [[TYPE tile] [Distance 0]
                                   [ID ^tile_id]] []],
                                  fill_hole, 2, false)]
]
[WHERE dir_binding /= []]
[WHERE tile_binding /= []]
;;; Now need to compute direction to hole and make sure it matches val
[LVARS
 [val = hd(property_from_binding("Value", dir_binding))]
 [target_dir = get_direction(sim_x(sim_myself), sim_y(sim_myself),
                             sim_x(target_hole), sim_y(target_hole))]
]
;;; Direction to target should be the same as the value of the stored
;;; direction
[WHERE target_dir = val]
==>
[POP11 make_shared_binding(hole_binding)]
[POP11 make_shared_binding(target_binding)]
[POP11 make_shared_binding(tile_binding)]
[POP11 make_shared_binding(dir_binding)]
[LVARS [tile_id = hd(property_from_binding("ID", tile_binding))]
 [dir_id = hd(property_from_binding("ID", dir_binding))]
]
[POP11 grue_do_action([[do go_to ^val]])]
[LVARS
 [hole_id = hd(property_from_binding("ID", hole_binding))]
 [[holex holey] = sim_coords(hole_id)]
]
;;; Update target (should be unnecessary)
[POP11 if (drawgraphics(TW)) then
         (holex, holey) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG fill_hole: agent moved towards hole in direction ?val
             with tile ?tile_id]
[POPRULESET]


;;; This rule removes a bad target, in the case where the hole has
;;; vanished and no direction has been computed
RULE remove_bad_target_no_direction
[LVARS
 [target_binding = suspension_bind([fill_hole_target
                                    [[TYPE htarget]] []],
                                    fill_hole, 3, false)]
 [dir_binding = suspension_bind([fill_hole_dir
                                 [[TYPE direction] [Toward hole]] []],
                                 fill_hole, 3, true)]
]
[WHERE not(null(target_binding))]
[WHERE null(dir_binding)]
[LVARS
 [hole = hd(property_from_binding("Value", target_binding))]
 [hole_binding = suspension_bind([fill_hole_hole
                                  [[TYPE hole] [ID ^hole]] []],
                                  fill_hole, 3, true)]
 [tile_binding = suspension_bind([fill_hole_tile
                                 [[TYPE tile] [Distance 0]] []],
                                 fill_hole, 3, false)]
]
[WHERE null(hole_binding)]
[WHERE not(null(tile_binding))]
```

```
==>
[POP11 make_permanent_binding(target_binding)] ;;; being removed
[POP11 make_shared_binding(tile_binding)]
[LVARS [target_id = hd(property_from_binding("ID", target_binding))]]
[POP11 grue_do_action([[do remove_resource ^target_id]])]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Fill_Hole: removed bad target, no direction]
[POPRULESET]


;;; This rule removes a bad direction when the target is still valid.
RULE remove_bad_direction
[LVARS
 [target_binding = suspension_bind([fill_hole_target
                                    [[TYPE htarget]] []],
                                   fill_hole, 4, false)]
 [dir_binding = suspension_bind([fill_hole_dir
                                 [[TYPE direction] [Toward hole]] []],
                                fill_hole, 4, false)]
[WHERE not(null(target_binding))]
[WHERE not(null(dir_binding))]
[LVARS
 [hole = hd(property_from_binding("Value", target_binding))]
 [hole_binding = suspension_bind([fill_hole_hole
                                 [[TYPE hole] [ID ^hole]] []],
                                fill_hole, 4, false)]
 [tile_binding = suspension_bind([fill_hole_tile
                                 [[TYPE tile] [Distance 0]] []],
                                fill_hole, 4, false)]
]
[WHERE not(null(hole_binding))]
[WHERE not(null(tile_binding))]
==>
[POP11 make_shared_binding(target_binding)]
[POP11 make_permanent_binding(dir_binding)]  ;;; being removed
[POP11 make_shared_binding(hole_binding)]
[POP11 make_shared_binding(tile_binding)]
[LVARS
 [dir_id = hd(property_from_binding("ID", dir_binding))]]
[POP11 grue_do_action([[do remove_resource ^dir_id]])]
[SAYIF DEBUG Fill_Hole: removed bad direction, still using target hole
       ?hole]
[POPRULESET]


;;; This rule removes a bad target (for the case where the target hole has
;;; vanished.)  Also removes the direction, as it will have to be
;;; recomputed for the new target.  Don't have to do much in this rule -
;;; if the target were good one of the above rules would have matched.
```

```
RULE remove_bad_target
[LVARS
 [target_binding = suspension_bind([fill_hole_target
                                    [[TYPE htarget]] []],
                                   fill_hole, 5, false)]
 [dir_binding = suspension_bind([fill_hole_dir
                                 [[TYPE direction] [Toward hole]] []],
                                fill_hole, 5, false)]
 [tile_binding = suspension_bind([fill_hole_tile
                                  [[TYPE tile] [Distance 0]] []],
                                  fill_hole, 5, false)]
]
[WHERE not(null(target_binding))]
[WHERE not(null(dir_binding))]
[WHERE not(null(tile_binding))]
[LVARS
 [hole = hd(property_from_binding("Value", target_binding))]
 [hole_binding = suspension_bind([fill_hole_hole
                                  [[TYPE hole] [ID ^hole]] []],
                                 fill_hole, 5, true)]
]
[WHERE null(hole_binding)]
==>
[POP11 make_permanent_binding(target_binding)]  ;;; being removed
[POP11 make_permanent_binding(dir_binding)]     ;;; being removed
[POP11 make_shared_binding(tile_binding)]
[LVARS [target_id = hd(property_from_binding("ID", target_binding))]
 [dir_id = hd(property_from_binding("ID", dir_binding))]]
[POP11 grue_do_action([[do remove_resource ^target_id]
                        [do remove_resource ^dir_id]])]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[SAYIF DEBUG Fill_Hole: removed bad target]
[POPRULESET]


;;; Re-compute the direction to the target hole
RULE compute_direction_to_target
[LVARS [tile = suspension_bind([fill_hole_tile
                                [[TYPE tile] [Distance 0]] []],
                                fill_hole, 6, false)]
 [target_binding = suspension_bind([fill_hole_target [[TYPE htarget]] []],
                                    fill_hole, 6, false)]]
[WHERE tile /= []]
[WHERE target_binding /= []]
[LVARS
 [hole = hd(property_from_binding("Value", target_binding))]
 [hole_binding = suspension_bind([fill_hole_hole
                                  [[TYPE hole] [ID ^hole]] []],
                                  fill_hole, 6, false)]
```

```
]
[WHERE hole_binding /= []]
[LVARS
 [[holex holey] = sim_coords(hole)]
 [agentx = sim_x(sim_myself)]
 [agenty = sim_y(sim_myself)]
 [dir = get_direction(agentx, agenty, holex, holey)]
 [id = generate_id()]
]
[WHERE dir > 0]
==>
[POP11 make_shared_binding(tile)]
[POP11 make_shared_binding(target_binding)]
[POP11 make_shared_binding(hole_binding)]
[POP11 if (drawgraphics(TW)) then
         (holex, holey) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do add_resource
                           [[ID ^id] [TYPE direction] [TIME Infinite]
                           [Value ^dir] [Toward hole]]] [do go_to ^dir]])]
[SAYIF DEBUG  Fill_hole:  agent picked direction ?dir, already had target
                          ?hole]
[POPRULESET]



;;; Pick direction toward hole and get rid of stored random direction
RULE pick_direction_and_remove_random
;;; Make sure agent still has a tile
[LVARS [tile = suspension_bind([fill_hole_tile
                                [[TYPE tile] [Distance 0]] []],
                                fill_hole, 6, false)] ]
[WHERE tile /= []]
[LVARS [st = hd(property_from_binding("Shape", tile))]]
;;; Find a hole to put it in, preferably the nearest one
[LVARS [hole = suspension_bind([fill_hole_hole
                                [[TYPE hole]] [[Shape ^st] [Age Young]
                                 [Distance #| 1 <- :#]
                                 [Size #| 1 <- :#]]],
                               fill_hole, 5, false)] ]
[WHERE hole /= []]
[LVARS
 [random_binding = suspension_bind([fill_hole_dir [[TYPE direction]
                                                   [Toward random]] []],
                                    fill_hole, 6, false)]
]
[WHERE not(null(random_binding))]
;;; Get the direction and make sure it's valid
[LVARS
 [h = hd(property_from_binding("ID",hole))]
 [t = hd(property_from_binding("ID", tile))]
 [random_id = hd(property_from_binding("ID", random_binding))]
 [[holex holey] = sim_coords(h)]
 [agentx = sim_x(sim_myself)]
 [agenty = sim_y(sim_myself)]
 [dir = get_direction(agentx, agenty, holex, holey)]
 [id = generate_id()]
 [target_id = generate_id()]
]
[WHERE dir > 0]
==>
[POP11 make_shared_binding(tile)]
[POP11 make_shared_binding(hole)]
[POP11 make_permanent_binding(random_binding)]  ;;; being removed
[POP11 if (drawgraphics(TW)) then
        (holex, holey) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do add_resource [[ID ^id] [TYPE direction]
                                         [TIME Infinite] [Value ^dir]
                                         [Toward hole]]]
                                         [do remove_resource ^random_id]
                      [do add_resource [[ID ^target_id] [TYPE htarget]
                                         [Value ^h] [Xcoord ^holex]
                                         [Ycoord ^holey]] ]
                      [do go_to ^dir]])]
[SAYIF DEBUG Fill_Hole: agent picked direction ?dir and removed random]
[SAYIF DEBUG target hole is ?hole]
[POPRULESET]


;;; If there's a tile and a hole, then work out the direction to the hole
;;; and store it.
RULE pick_direction
;;; Make sure agent still has a tile
[LVARS [tile = suspension_bind([fill_hole_tile
                                 [[TYPE tile] [Distance 0]] []],
                                fill_hole, 7, false)] ]
[WHERE tile /= []]
[LVARS [st = hd(property_from_binding("Shape", tile))]]
;;; Find a hole to put it in, preferably a young one that matches the
;;; shape of the tile, and is nearby, and is small (so easier to fill for
;;; the bonus)
[LVARS [hole = suspension_bind([fill_hole_hole
                                 [[TYPE hole]] [[Shape ^st] [Age Young]
                                  [Distance #| 0 <- :#]
                                  [Size |# 1 <- :#] ]],
                                fill_hole, 7, false)] ]
[WHERE hole /= []]
;;; Get the direction and make sure it's valid
[LVARS
 [h = hd(property_from_binding("ID",hole))]
 [t = hd(property_from_binding("ID", tile))]
 [[holex holey] = sim_coords(h)]
 [agentx = sim_x(sim_myself)]
 [agenty = sim_y(sim_myself)]
 [dir = get_direction(agentx, agenty, holex, holey)]
 [id = generate_id()]
 [target_id = generate_id()]
]
[WHERE dir > 0]
==>
[POP11 make_shared_binding(tile)]
[POP11 make_shared_binding(hole)]
[POP11 if (drawgraphics(TW)) then
         (holex, holey) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do add_resource [[ID ^id] [TYPE direction]
                                         [TIME Infinite] [Value ^dir]
                                         [Toward hole]]]
        [do add_resource [[ID ^target_id] [TYPE htarget] [Value ^h]
                          [Xcoord ^holex] [Ycoord ^holey]] ]
        [do go_to ^dir]])]
[SAYIF DEBUG Fill_Hole: agent picked direction ?dir]
[SAYIF DEBUG target hole is ?hole]
[POPRULESET]


;;; Remove the random direction if the agent has hit an obstacle.  It will
;;; be recomputed by the rule below.
RULE change_random
[LVARS
 [random_binding = suspension_bind([fill_hole_random_dir
                                    [[TYPE direction] [Toward random]] []],
                                   fill_hole, 8, false)]
]
[WHERE not(null(random_binding))]
[LVARS
 [dir = hd(property_from_binding("Value", random_binding))]
 [obstacle_binding = suspension_bind([fill_hole_obstacle
                                      [[TYPE obstacle] [Distance 1]
                                      [Direction ^dir]] []],
                                     fill_hole, 8, false)]
 [tile_binding = suspension_bind([fill_hole_tile
                                 [[TYPE tile] [Distance 0]] []],
                                fill_hole, 8, false)]
]
[WHERE not(null(obstacle_binding))]
[WHERE not(null(tile_binding))]
==>
[POP11 make_shared_binding(tile_binding)]
[POP11 make_shared_binding(obstacle_binding)]
[POP11 make_permanent_binding(random_binding)]  ;;; being removed
[LVARS
```

```
 [id = hd(property_from_binding("ID", random_binding))]
]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do remove_resource ^id]])]
[SAYIF DEBUG fill_hole:  Hit an obstacle and removed random direction]
[POPRULESET]


;;; Move in a previously stored random direction
RULE random_move
[LVARS
[tile_binding = suspension_bind([fill_hole_tile
                                 [[TYPE tile] [Distance 0]] [] ],
                                fill_hole, 9, false)]
[dir_binding = suspension_bind([fill_hole_dir
                                 [[TYPE direction] [Toward random]] [] ],
                               fill_hole, 9, false)]
]
[WHERE tile_binding /= []]
[WHERE dir_binding /= []]
==>
[POP11 make_shared_binding(tile_binding)]
[POP11 make_shared_binding(dir_binding)]
[LVARS
 [dir = hd(property_from_binding("Value", dir_binding))]
 [dir_id = hd(property_from_binding("ID", dir_binding))]
]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do go_to ^dir]])]
[SAYIF DEBUG fill_hole moved randomly]
[POPRULESET]


;;; If no hole visible and no direction resource, then pick a random
;;; direction and store it.
RULE random_pick
[LVARS [binding = suspension_bind([fill_hole_tile
                                   [[TYPE tile] [Distance 0]] []],
                                  fill_hole, 10, false)] ]
[WHERE binding /= []]
==>
[POP11 make_shared_binding(binding)]
[LVARS [dir=random(4)]
 [id = generate_id()]
]
[POP11 if (drawgraphics(TW)) then
         (-2, -2) -> sim_coords(target(sim_myself)) endif;]
[POP11 grue_do_action([[do add_resource
                                [[ID ^id] [TYPE direction] [TIME Infinite]
                                [Value ^dir] [Toward random]]]
                                [do go_to ^dir]])]
    [SAYIF DEBUG fill_hole picked a random direction]
    [POPRULESET]


    ;;; Suspend the program.
    ;;; (This functionality isn't actually used.)
    RULE suspend
    [WHERE suspendable(fill_hole)]
    ==>
    [SAYIF DEBUG fill_hole is suspendable]
    [UNRUNNABLE fill_hole]
    [POPRULESET]


    ;;; To take care of the case where this ruleset gets called even though
    ;;; the tile has disappeared
    RULE bail_out
    ==>
    [UNRUNNABLE fill_hole]
    [SAYIF DEBUG fill_hole is unrunnable]
    [POPRULESET]

enddefine;
```

E.5 Avoid Obstacle TRP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      avoid_obstacle.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The avoid obstacle TRP for the GRUE Tileworld agent.
;;;
;;;


;;; A really simple avoid obstacle behaviour.
define :ruleset avoid_obstacle;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
```

```
 [prb_show_conditions = false]
 [cycle_limit = 1]
 ];

;;; If there's no direction resource, and there's no direction stored
;;; by avoid obstacle, then the program doesn't need to run.
;;; (If there is a stored perpendicular direction, then we might need
;;; to clean it up.)
RULE really_really_done
[LVARS [dir_binding = suspension_bind([dir [[TYPE direction]] []],
                                      avoid_obstacle, 0, true)]
       [perpdir_binding = suspension_bind([perp_dir
                                           [[TYPE direction]
                                           [Toward around_obstacle]] []],
                                          avoid_obstacle, 0, true)]]
[WHERE null(dir_binding)]
[WHERE null(perpdir_binding)]
==>
[SAYIF DEBUG avoid obstacle not running - no direction stored]
[UNRUNNABLE avoid_obstacle]
[POPRULESET]


;;; Done if there's a direction but no obstacle in that direction
;;; and we've already deleted the perpendicular direction
RULE done
[LVARS [perpdir_binding = suspension_bind([perp_dir
                                           [[TYPE direction]
                                           [Toward around_obstacle]] []],
                                          avoid_obstacle, 1, true)]]
[WHERE null(perpdir_binding)]
[LVARS [dir_binding = suspension_bind([avoid_dir
                                       [[TYPE direction]]
                                       [[Toward hole] [Toward tile]]],
                                      avoid_obstacle, 1, false)]
        ;;; always check a productive direction before a random one
 ]
[WHERE not(null(dir_binding))]
[LVARS
 [val = hd(property_from_binding("Value", dir_binding))]
 [obs_binding = suspension_bind([avoid_obstacle
                                 [[TYPE obstacle] [Distance 1]
                                  [Direction ^val]] []],
                                avoid_obstacle, 1, true)]
 ]
[WHERE null(obs_binding)]
==>
[POP11 make_shared_binding(dir_binding)]
[SAYIF DEBUG avoid obstacle done, no obstacle in direction ?val]
[UNRUNNABLE avoid_obstacle]
```

```
[POPRULESET]


;;; If there's a perpendicular direction and a stored direction, but
;;; no longer a direction that matches the stored direction (i.e. because
;;; whichever program created it deleted it again) then delete the stored
;;; direction and perpendicular direction so they can be re-computed if
;;; necessary.
RULE direction_changed
[LVARS
 [perpdir_binding = suspension_bind([perp_dir [[TYPE direction]
                                               [Toward around_obstacle]]
                                              []],
                                     avoid_obstacle, 2, false)]
 [olddir_binding = suspension_bind([stored_dir [[TYPE stored_direction]]
                                               []],
                                    avoid_obstacle, 2, false)]
 ]
[WHERE not(null(perpdir_binding))]
[WHERE not(null(olddir_binding))]
[LVARS
 [old_dir = hd(property_from_binding("Value", olddir_binding))]
 [dir_binding = suspension_bind([dir [[TYPE direction] [Value ^old_dir]]
                                     []],
                                 avoid_obstacle, 2, true)]
 ]
[WHERE null(dir_binding)]
==>
[POP11 make_permanent_binding(perpdir_binding)] ;;; removing this
[POP11 make_permanent_binding(olddir_binding)]  ;;; removing this
[LVARS
 [olddir_id = hd(property_from_binding("ID", olddir_binding))]
 [perpdir_id = hd(property_from_binding("ID", perpdir_binding))]
 ]
[POP11 grue_do_action([[do remove_resource ^olddir_id]
           [do remove_resource ^perpdir_id]])]
[SAYIF DEBUG avoid obstacle deleting old information due to direction
             change]
[POPRULESET]


;;; If no obstacle in the stored direction,  then remove the perpendicular
;;; direction and the stored direction.
RULE reset_direction
[LVARS
 ;;; first look for stored perpendicular direction, and the original dir
       [perpdir_binding = suspension_bind([perp_dir
                                           [[TYPE direction]
                                            [Toward around_obstacle]] []],
                                          avoid_obstacle, 3, false)]
```

```
   [olddir_binding = suspension_bind([stored_dir
                                      [[TYPE stored_direction]] []],
                                     avoid_obstacle, 3, false)]
   ]
[WHERE perpdir_binding /= []]
[WHERE olddir_binding /= []]
[LVARS
 [val = hd(property_from_binding("Value", olddir_binding))]
 [obs_binding = suspension_bind([obstacle
                                 [[TYPE obstacle] [Distance 1]
                                  [Direction ^val]] [] ],
                                 avoid_obstacle, 3, true)]
 ]
[WHERE null(obs_binding)] ;;; better not be an obstacle in direction val
==>
[POP11 make_permanent_binding(perpdir_binding)] ;;; removing this
[POP11 make_permanent_binding(olddir_binding)]  ;;; removing this
[LVARS [perp_id = hd(property_from_binding("ID", perpdir_binding))]
       [old_id = hd(property_from_binding("ID", olddir_binding))]
   ]
[POP11 grue_do_action([[do remove_resource ^old_id]
                       [do remove_resource ^perp_id]
                       [do go_to ^val]
                      ] ) ]
[SAYIF DEBUG avoid obstacle done, removing stored info,
             no obstacle in direction ?val]
[POPRULESET]


;;; If there's an obstacle in the perpendicular direction, then reverse
;;; directions.  (Still won't help if the agent's in a u-shape, but
;;; should prevent it getting stuck in corners.)
RULE reverse_perpendicular
[LVARS
 ;;; first look for stored perpendicular direction
 [perpdir_binding = suspension_bind([perp_dir
                                     [[TYPE direction]
                                     [Toward around_obstacle]] []],
                                    avoid_obstacle, 4, false)]
 ]
[WHERE perpdir_binding /= []]
[LVARS
 [val = hd(property_from_binding("Value", perpdir_binding))]
 ;;; see if there's an obstacle in the perpendicular direction
 [obs_binding = suspension_bind([avoid_obstacle [[TYPE obstacle]
                                                 [Distance 1]
                                                 [Direction ^val]]
                                                [] ],
                                avoid_obstacle, 4, false)]
 ]
```

```
[WHERE not(null(obs_binding))]
==>
[POP11 make_permanent_binding(perpdir_binding)] ;;; removing this
[POP11 make_shared_binding(obs_binding)]
[LVARS
 [perp_id = hd(property_from_binding("ID", perpdir_binding))]
 [new_id = generate_id()]
 [new_dir = reverse_direction(val)]
 ]
[SAYIF DEBUG avoid obstacle reversing perpendicular direction,
             obstacle in previous direction ?val]
[POP11 grue_do_action([[do add_resource
                            [[ID ^new_id] [TYPE direction]
                             [Value ^new_dir] [Toward around_obstacle]]]
                       [do remove_resource ^perp_id] [do go_to ^new_dir]])]
[POPRULESET]


;;; Move in the perpendicular direction
RULE move_perpendicular
[LVARS [perp_binding=suspension_bind([perp_dir
                                      [[TYPE direction]
                                       [Toward around_obstacle]] []],
                                     avoid_obstacle, 5, false)]]
[WHERE perp_binding /= []]
[LVARS [olddir_binding = suspension_bind([stored_dir
                                          [[TYPE stored_direction]]
                                          []],
                                         avoid_obstacle, 5, false)]]
[WHERE olddir_binding /= []]
[LVARS
 [olddir = hd(property_from_binding("Value", olddir_binding))]
 [obs_binding=suspension_bind([obstacle [[TYPE obstacle] [Distance 1]
                                         [Direction ^olddir]] []],
                              avoid_obstacle, 5, false)]
 ]
[WHERE obs_binding /= []]
 ==>
 [POP11 make_shared_binding(perp_binding)]   ;;; not changing this
 [POP11 make_shared_binding(olddir_binding)]
 [POP11 make_shared_binding(obs_binding)]
 [LVARS [dir = hd(property_from_binding("Value", perp_binding))]]
 [POP11 grue_do_action([[do go_to ^dir]])]
 [SAYIF DEBUG avoid obstacle moving perpendicular;
             ?obs_binding ?olddir_binding ?perp_binding]
 [POPRULESET]


 ;;; If there's an obstacle in front of the agent, then change
 ;;; to a perpendicular direction while storing the old one so
 ;;; we can tell when we've gone past the obstacle
RULE obstacle_present
[LVARS
 [dir_binding = suspension_bind([avoid_dir
                                  [[TYPE direction]] [] ],
                                avoid_obstacle, 6, false)]
 ]
[WHERE dir_binding /= []]
[LVARS
 [old_dir = hd(property_from_binding("Value", dir_binding))]
 ;;; [old_dir_id = hd(property_from_binding("ID", dir_binding))]
 [old_to = hd(property_from_binding("Toward", dir_binding))]
[obs_binding = suspension_bind([obstacle
                                [[TYPE obstacle] [Distance 1]
                                 [Direction ^old_dir]] [] ],
                               avoid_obstacle, 5, false)]]
[WHERE obs_binding /= []]  ;;; check bindings are valid
==>
[POP11 make_shared_binding(obs_binding)]
[POP11 make_shared_binding(dir_binding)]
[LVARS [perp_dir = get_perpendicular_direction(old_dir)]
    [perp_id = generate_id()]
[stored_id = generate_id()]]
[SAYIF DEBUG  avoid obstacle computing perpendicular direction,
              found obstacle in direction ?old_dir]
[POP11 grue_do_action([[do add_resource [[ID ^perp_id] [TYPE direction]
                                          [Value ^perp_dir]
                                          [Toward around_obstacle]]]
                       [do add_resource [[ID ^stored_id]
                                          [TYPE stored_direction]
                                          [Value ^old_dir]
                                          [Toward ^old_to]]]
                       [do go_to ^perp_dir]])]
[POPRULESET]


;;; This functionality isn't actually used.
RULE suspendable
[WHERE suspendable(avoid_obstacle)]
==>
;;; actually suspend the program here if suspension is turned on.
[SAYIF DEBUG avoid_obstacle is suspendable]
[UNRUNNABLE avoid_obstacle]
[POPRULESET]


;;; Just in case this program doesn't run, pop back to arbitrator
RULE avoid_obstacle
==>
[UNRUNNABLE avoid_obstacle]
```

```
    [SAYIF DEBUG avoid_obstacle is unrunnable]
    [POPRULESET]

enddefine;
```

Appendix F

Unreal Tournament Specific Code

This appendix contains the goal generators and the TRPs used by the Unreal Tournament agent.

F.1 Unreal Tournament Goal Generators

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      ut_goal_generators.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This file contains goal generators for the GRUE ut bot

define :ruleset generate_heal;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; Get healthpacks sometimes if not doing anything else
RULE generate_heal_low
[WHERE sim_myself.healthlevel >= HEALTH_THRESHOLD]
[AVAILABLE == [== [TYPE HEALTHPACK] == ]==]
```

```
[NOT TASKLIST == [TASK Heal ==] ==]
[NOT GOAL Heal ==][NOT GOAL Heal ==]
==>
[LVARS [priority = 4 * (100.0 - sim_myself.healthlevel)]]
[TESTADD GOAL Heal ?priority  NonMaintenance HEALTHY]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal Heal with priority ?priority]


;;; If health below threshold, then heal goes to maximum priority
RULE generate_heal_high
[WHERE sim_myself.healthlevel < HEALTH_THRESHOLD]
[NOT TASKLIST == [TASK Heal ==] ==]
[AVAILABLE == [== [TYPE HEALTHPACK] == ]==]
[NOT GOAL Heal ==]
==>
[TESTADD GOAL Heal 400  NonMaintenance HEALTHY]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal Heal with priority 400]


;;; Update the priority when it should be low.
RULE update_heal_low
[WHERE sim_myself.healthlevel >= HEALTH_THRESHOLD]
[TASKLIST ??beg [TASK Heal ?p ??end_condition ] ??end][->>tasklist]
[LVARS [priority = 4 * (100.0 - sim_myself.healthlevel)]]
[WHERE p /= priority]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK Heal ?priority ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal Heal to new priority ?priority]


;;; Update the priority when it should be high.
RULE update_heal_high
[WHERE sim_myself.healthlevel < HEALTH_THRESHOLD]
[TASKLIST ??beg [TASK Heal ?p ??end_condition ] ??end][->>tasklist]
[WHERE p /= 400]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK Heal 400 ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal Heal to new priority 400]

enddefine;



define :ruleset generate_attack;
```

```
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; Generate attack with high priority (near enemy player)
RULE attack_high
[AVAILABLE ??res_start [[ID ?id] [TYPE PLAYER] [Team ?tm] [Location ?loc]
                        [Velocity ?vel] [Rotation ?rot] [Weapon ?weap]
                        [Timestamp ?t] [Visible true]] ??res_end]
[NOT TASKLIST == [TASK attack ==] ==]
[NOT GOAL attack ==]
[WHERE tm /= team(sim_myself)]
[WHERE sim_myself.weapstring /= "None"]
[WHERE sim_myself.ammunition > 0]
[WHERE nearby(sim_myself.location, loc)]
==>
[TESTADD GOAL attack 200  NonMaintenance PLAYER_GONE]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal Attack with priority 200]


;;; Generate attack with high priority (enemy player has flag)
RULE attack_high_flag
[ENEMY HAS FLAG]
[AVAILABLE ??res_start [[ID ?id] [TYPE PLAYER] [Team ?tm] [Location ?loc]
                        [Velocity ?vel] [Rotation ?rot] [Weapon ?weap]
                        [Timestamp ?t] [Visible true]] ??res_end]
[NOT TASKLIST == [TASK attack ==] ==]
[NOT GOAL attack ==]
[WHERE sim_myself.weapstring /= "None"]
[WHERE sim_myself.ammunition > 0]
[WHERE tm /= team(sim_myself)]
==>
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[TESTADD GOAL attack 200  NonMaintenance PLAYER_GONE]
[SAYIF DEBUG generated goal Attack with priority 200]


;;; Generate attack with low priority
;;; (priority depends on distance, range 75-200)
RULE attack_low
[AVAILABLE ??res_start [[ID ?id] [TYPE PLAYER] [Team ?tm] [Location ?loc]
                        [Velocity ?vel] [Rotation ?rot] [Weapon ?weap]
                        [Timestamp ?t] [Visible true]] ??res_end]
```

```
[NOT TASKLIST == [TASK attack ==] ==]
[NOT GOAL attack ==]
[WHERE tm /= team(sim_myself)]
[WHERE sim_myself.weapstring /= "None"]
[WHERE sim_myself.ammunition > 0]
[NOT ENEMY HAS FLAG]
==>
[LVARS
 [dist = threeDDistance(location(sim_myself), loc)]
 [priority= ((125.0 - gdiv(dist,10.0)) + 75)]
]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[TESTADD GOAL attack ?priority  NonMaintenance PLAYER_GONE]
[SAYIF DEBUG generated goal Attack with priority ?priority]


;;; Update attack when the priority should be low
RULE update_attack
[AVAILABLE ??res_start [[ID ?id] [TYPE PLAYER] [Team ?tm] [Location ?loc]
                        [Velocity ?vel] [Rotation ?rot] [Weapon ?weap]
                        [Timestamp ?t] [Visible true]] ??res_end]
[TASKLIST ??beg [TASK attack ?p ??taskend] ??end][->>tasklist]
[WHERE tm /= team(sim_myself)]
[NOT ENEMY HAS FLAG]
[WHERE not(nearby(sim_myself.location, loc))]
[LVARS
 [dist = threeDDistance(location(sim_myself), loc)]
 [priority= ((125.0 - gdiv(dist,10.0)) + 75)]
]
[WHERE p /= priority]
==>
[DEL ?tasklist]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[TASKLIST ??beg [TASK attack ?priority ??taskend] ??end]
[SAYIF DEBUG updated task Attack to priority ?priority]


;;; Update when the priority should be high. This is for the case where the
;;; enemy has the flag.
RULE update_high1
[AVAILABLE ??res_start [[ID ?id] [TYPE PLAYER] [Team ?tm] [Location ?loc]
                        [Velocity ?vel] [Rotation ?rot] [Weapon ?weap]
                        [Timestamp ?t] [Visible true]] ??res_end]
[TASKLIST ??beg [TASK attack ?p ??taskend] ??end][->>tasklist]
[WHERE tm /= team(sim_myself)]
[ENEMY HAS FLAG]
[WHERE p /= 200]
==>
[DEL ?tasklist]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
```

```
[TASKLIST ??beg [TASK attack 200 ??taskend] ??end]
[SAYIF DEBUG updated task Attack to priority 200]


;;; Update the priority when the enemy player is nearby.
RULE update_high2
[AVAILABLE ??res_start [[ID ?id] [TYPE PLAYER] [Team ?tm] [Location ?loc]
                        [Velocity ?vel] [Rotation ?rot] [Weapon ?weap]
                        [Timestamp ?t] [Visible true]] ??res_end]
[TASKLIST ??beg [TASK attack ?p ??taskend] ??end][->>tasklist]
[WHERE tm /= team(sim_myself)]
[WHERE nearby(sim_myself.location, loc)]
[WHERE p /= 200]
==>
[DEL ?tasklist]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[TASKLIST ??beg [TASK attack 200 ??taskend] ??end]
[SAYIF DEBUG updated task Attack to priority 200]

enddefine;



;;; Generate a goal to return to the default location at the top of the ramp
;;; by the base.
define :ruleset generate_return_to_base;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


RULE default
;;; Don't create duplicates of this task in the task list
[NOT TASKLIST == [TASK return_to_base == ] == ]
;;; Don't generate this goal if the bot is patrolling, as the two
;;; programs conflict.
[NOT TASKLIST == [TASK patrol == ] ==]
;;; Only generate the goal if the bot is not already there.
[WHERE not(near(bot.location, bot.default_location))]
[NOT GOAL return_to_base ==]
==>
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[TESTADD GOAL return_to_base 50  NonMaintenance AT_BASE]
[SAYIF DEBUG generated goal ReturnToBase with priority 15]
```

```
enddefine;



;;; Patrol the local area when not doing anything else.
define :ruleset generate_patrol;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


RULE default
[NOT TASKLIST == [TASK patrol == ] == ]
[NOT GOAL patrol ==]
[WHERE near(bot.location, bot.default_location)]
==>
[TESTADD GOAL patrol 55 NonMaintenance PATROL_COMPLETE]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal patrol with priority 55]


enddefine;


;;; get_weapon and get_ammo both have priority 300 if urgent and 85 if not.
;;; Urgent = lacking weaponry.  get_own_flag, heal, and run_away all take
;;; precedence
define :ruleset generate_get_weapon;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


RULE generate_get_weapon
[WHERE sim_myself.weapstring = "None"]
[AVAILABLE == [== [TYPE WEAPON] == ] == ]
[NOT TASKLIST == [TASK GetWeapon ==] ==]
```

```
[NOT GOAL GetWeapon ==]
==>
[TESTADD GOAL GetWeapon 300  NonMaintenance ARMED]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetWeapon with priority 300]


;;; Generate it anyway when the bot has a weapon but sees a new one.
;;; Relatively low priority so it doesn't preempt anything else.
RULE generate_get_weapon2
[AVAILABLE == [== [TYPE WEAPON] == ] == ]
[NOT TASKLIST == [TASK GetWeapon ==] ==]
[NOT GOAL GetWeapon ==]
==>
[TESTADD GOAL GetWeapon 85 NonMaintenance ARMED]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetWeapon with priority 85]


;;; Update the priority in urgent situations.
RULE update_get_weapon
[TASKLIST ??beg [TASK GetWeapon ?p ??end_condition] ??end][->>tasklist]
[WHERE sim_myself.weapstring = "None"]
[WHERE p /= 300]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK GetWeapon 300 ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal GetWeapon with priority 300]


enddefine;


;;; This is how to score a goal...
define :ruleset generate_get_own_flag;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; This goal has the highest possible priority.  Note that the goal should
;;; only be generated when the flag is nearby, so it should be ok to get
;;; the flag before doing anything else.  (Well, it might get the bot
;;; killed in certain circumstances, but the bot would probably get killed
;;; anyway.)
RULE generate_get_own_flag
[AVAILABLE == [== [TYPE FLAG] == [Location ?loc] == [Team ?tm] ==] ==]
[WHERE tm = team(bot)]
[WHERE not(near(loc, sim_myself.base_loc))]
[NOT TASKLIST == [TASK GetOwnFlag ==] ==]
[NOT GOAL GetOwnFlag ==]
==>
[TESTADD GOAL GetOwnFlag 500 NonMaintenance HAVE_FLAG]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetOwnFlag with priority 500]

enddefine;


;;; The get_ammo goal has priority 300 if high priority and 85 if not
define :ruleset generate_get_ammo;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; high priority case
RULE generate_get_ammo
[NOT TASKLIST == [TASK GetAmmo ==] ==]
[AVAILABLE == [==[TYPE AMMO]==] == ]
[NOT GOAL GetAmmo ==]
[WHERE sim_myself.ammunition <= 3]
==>
[TESTADD GOAL GetAmmo 210  NonMaintenance HAVE_AMMO]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetAmmo with priority 210]


;;; high priority case (weapons come with ammo)
RULE generate_get_ammo2
[NOT TASKLIST == [TASK GetAmmo ==] ==]
[NOT GOAL GetAmmo ==]
[AVAILABLE == [==[TYPE WEAPON]==] == ]
[WHERE sim_myself.ammunition <= 3]
==>
[TESTADD GOAL GetAmmo 210  NonMaintenance HAVE_AMMO]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetAmmo with priority 210]
```

```
;;; low priority case (grab ammo if it's there)
RULE generate_get_ammo_low
[NOT TASKLIST == [TASK GetAmmo ==] ==]
[NOT GOAL GetAmmo ==]
[AVAILABLE == [== [TYPE AMMO]==] == ]
==>
[TESTADD GOAL GetAmmo 85  NonMaintenance HAVE_AMMO]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetAmmo with priority 85]


;;; If bot's used up its ammo, bump the priority up to 300
RULE update_get_ammo
[TASKLIST ??beg [TASK GetAmmo ?p ??end_condition ] ??end][->>tasklist]
[NOT GOAL GetAmmo]
[WHERE sim_myself.ammunition <= 3]
[WHERE p /= 210]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK GetAmmo 210 ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal GetAmmo to new priority 210]

enddefine;


;;; Run away if health low and no health available
;;; or if under attack (the under_attack flag is set
;;; to true if the bot gets notified that it's taking
;;; damage)
define :ruleset generate_run_away;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; Low priority when health is above threshold.
RULE generate_run_away_low
[WHERE sim_myself.healthlevel >= HEALTH_THRESHOLD]
[NOT TASKLIST == [TASK RunAway ==] ==]
[NOT AVAILABLE == [== [TYPE HEALTHPACK] ==] ==]
[NOT GOAL RunAway ==]
==>
[LVARS [priority = (4*(100.0 - sim_myself.healthlevel)) + 10]]
```

```
[TESTADD GOAL RunAway ?priority  NonMaintenance SAFE]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal RunAway with priority ?priority]


;;; Also generate run_away when under attack.  The priority is the
;;; amount of damage the bot has taken so far.
RULE generate_run_away2
[WHERE sim_myself.healthlevel >= HEALTH_THRESHOLD]
[WHERE sim_myself.under_attack]
[NOT GOAL RunAway ==]
[NOT TASKLIST == [TASK RunAway ==] ==]
==>
[LVARS [priority = 4* ((sim_myself.damage_amount) +
                   (100 - sim_myself.healthlevel))]]
[TESTADD GOAL RunAway ?priority  NonMaintenance SAFE]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal RunAway with priority ?priority]


;;; high priority when health is really low
RULE generate_run_away_high
[WHERE sim_myself.healthlevel < HEALTH_THRESHOLD]
[NOT TASKLIST == [TASK RunAway ==] ==]
[NOT GOAL RunAway ==]
[NOT AVAILABLE == [== [TYPE HEALTHPACK] ==] ==]
==>
[TESTADD GOAL RunAway 410  NonMaintenance SAFE]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal RunAway with priority 410]


;;; Update when the priority should be low.  This case is when there's no
;;; healthpack.
RULE update_run_away
[WHERE sim_myself.healthlevel >= HEALTH_THRESHOLD]
[TASKLIST ??beg [TASK RunAway ?p ??end_condition ] ??end][->>tasklist]
[NOT AVAILABLE == [== [TYPE HEALTHPACK] == ] ==]
[LVARS [priority = 4*((100.0 - sim_myself.healthlevel)) + 10]]
[WHERE p /= priority]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK RunAway ?priority ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal RunAway to new priority ?priority]


;;; Update if the priority should be low and an enemy is visible.
RULE update_run_away2
[WHERE sim_myself.healthlevel >= HEALTH_THRESHOLD]
```

```
[TASKLIST ??beg [TASK RunAway ?p ??end_condition ] ??end][->>tasklist]
[AVAILABLE == [== [TYPE PLAYER] [Team ?tm] == [Visible true] ==] ==]
[WHERE tm /= bot.team]
[LVARS [priority = ((100.0 - sim_myself.healthlevel) + 5)]]
[WHERE p /= priority]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK RunAway ?priority ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal RunAway to new priority ?priority]


;;; Update when the priority should be high.
RULE update_high
[WHERE sim_myself.healthlevel < HEALTH_THRESHOLD]
[TASKLIST ??beg [TASK RunAway ?p ??end_condition ] ??end][->>tasklist]
[WHERE p /= 410]
==>
[DEL ?tasklist]
[TASKLIST ??beg [TASK RunAway 410 ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal RunAway to new priority 410]

enddefine;



;;; Generated with high priority if not at the default location and no enemies
;;; present, or if there is an enemy but really close to flag.
;;; If away from base and an enemy is present then medium priority.
;;; If at the base, then 1 in 20 chance of beating the filter.
define :ruleset generate_get_enemy_flag;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; If the enemy flag is available, generate the goal with max priority.
;;; This case is for when there's no enemy player around.
RULE generate_get_enemy_flag_high
[NOT TASKLIST == [TASK GetEnemyFlag ==] ==]
[NOT GOAL GetEnemyFlag ==]
[NOT HAVE FLAG ==]
[LVARS
 [tm = sim_myself.enemy_team]
]
[AVAILABLE == [== [TYPE FLAG] [Location ?loc] [Team ^tm] ==] ==]
[NOT AVAILABLE == [== [TYPE PLAYER] == [Team ^tm] == [Visible true] ==] ==]
[WHERE not(nearNode(sim_myself.location, sim_myself.default_location))]
==>
[TESTADD GOAL GetEnemyFlag 230 NonMaintenance HAVE_ENEMY_FLAG]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetEnemyFlag with priority 230]


;;; Not at base, but enemy nearby.  Generate at high priority if flag is
;;; nearby.
RULE generate_get_enemy
[NOT TASKLIST == [TASK GetEnemyFlag ==] ==]
[NOT GOAL GetEnemyFlag ==]
[NOT HAVE FLAG ==]
[LVARS
 [tm = sim_myself.enemy_team]
]
[AVAILABLE == [== [TYPE FLAG] [Location ?loc] [Team ^tm] ==] ==]
[AVAILABLE == [== [TYPE PLAYER] == [Team ^tm] == ] ==]
[WHERE not(nearNode(sim_myself.location, sim_myself.default_location))]
[WHERE nearby(sim_myself.location, loc)]
==>
[TESTADD GOAL GetEnemyFlag 230 NonMaintenance HAVE_ENEMY_FLAG]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetEnemyFlag with priority 230]


;;; Enemy present, flag not nearby. medium priority
RULE generate_get_enemy_2
[NOT TASKLIST == [TASK GetEnemyFlag ==] ==]
[NOT GOAL GetEnemyFlag ==]
[NOT HAVE FLAG ==]
[LVARS
 [tm = sim_myself.enemy_team]
]
[AVAILABLE == [== [TYPE FLAG] [Location ?loc] [Team ^tm] ==] ==]
[AVAILABLE == [== [TYPE PLAYER] == [Team ^tm] == ] ==]
[WHERE not(nearNode(sim_myself.location, sim_myself.default_location))]
[WHERE not(nearby(sim_myself.location, loc))]
==>
[TESTADD GOAL GetEnemyFlag 100 NonMaintenance HAVE_ENEMY_FLAG]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetEnemyFlag with priority 100]


;;; At the base, generate with random priority
RULE generate_get_enemy_flag_random
[NOT TASKLIST == [TASK GetEnemyFlag ==] ==]
```

```
[NOT GOAL GetEnemyFlag ==]
[NOT HAVE FLAG ==]
[NOT AVAILABLE == [== [TYPE FLAG] [Location ?loc]
                      [Team ^sim_myself.enemy_team] ==] ==]
[WHERE nearNode(sim_myself.location, sim_myself.default_location)]
==>
[LVARS
 [chance = random(20)]
[priority = 10]
]
[POP11 if chance < 2 then 100 -> priority; endif;]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[TESTADD GOAL GetEnemyFlag ?priority NonMaintenance HAVE_ENEMY_FLAG]
[SAYIF DEBUG generated goal GetEnemyFlag with priority ?priority]


;;; If the flag's not available and the bot's not at the default location,
;;; generate this goal anyway.
RULE generate_get_enemy_flag_catchall
[NOT TASKLIST == [TASK GetEnemyFlag ==] ==]
[NOT GOAL GetEnemyFlag ==]
[NOT HAVE FLAG ==]
[NOT AVAILABLE == [== [TYPE FLAG] [Location ?loc]
                      [Team ^sim_myself.enemy_team] ==] ==]
[NOT GOAL GetEnemyFlag ==]
==>
[TESTADD GOAL GetEnemyFlag 100 NonMaintenance HAVE_ENEMY_FLAG]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal GetEnemyFlag with priority 100]


;;; don't bother updating this goal

enddefine;



;;; Generated only when the bot has the enemy's flag.
define :ruleset generate_score_point;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];
```

```
;;; If near the flag base, scoring is high priority.
RULE generate_score_point_high_near
[HAVE FLAG ==]
[NOT GOAL ScorePoint ==]
;;; can't score when the enemy has the flag
[NOT ENEMY HAS FLAG ==]
[NOT TASKLIST == [TASK ScorePoint ==] ==]
[WHERE nearby(sim_myself.location,  sim_myself.base_loc)]
==>
[TESTADD GOAL ScorePoint 220 NonMaintenance SCORED]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal ScorePoint with priority 220]


;;; If no enemies to deal with, scoring is high priority.
RULE generate_score_point_high_no_enemies
[HAVE FLAG ==]
[NOT GOAL ScorePoint ==]
[NOT ENEMY HAS FLAG ==]
[NOT TASKLIST == [TASK ScorePoint ==] ==]
[LVARS
 [tm = sim_myself.enemy_team]
]
[NOT AVAILABLE == [== [TYPE PLAYER] == [Team ^tm] ==] ==]
==>
[TESTADD GOAL ScorePoint 220 NonMaintenance SCORED]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal ScorePoint with priority 220]


;;; In other cases, it's low priority.
RULE generate_score_point_low
[HAVE FLAG ==]
[NOT GOAL ScorePoint ==]
[NOT ENEMY HAS FLAG ==]
[NOT TASKLIST == [TASK ScorePoint ==] ==]
[NOT GOAL ScorePoint ==]  ;;; goal not already generated
==>
[TESTADD GOAL ScorePoint 100 NonMaintenance SCORED]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal ScorePoint with priority 100]


;;; Update when the bot is near the flag base.
RULE update_score_point_high_near
[HAVE FLAG ==]
[NOT ENEMY HAS FLAG ==]
[TASKLIST ??beg [TASK ScorePoint ?p ??end_condition ] ??end][->>tasklist]
[WHERE nearby(sim_myself.location,  sim_myself.base_loc)]
[WHERE p /= 220]
```

```
==>
[DEL ?tasklist]
[TASKLIST ??beg  [TASK ScorePoint  220 ??end_condition] ??end]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[SAYIF DEBUG updated goal ScorePoint with priority 220]


;;; Update when there's no enemies nearby.
RULE update_score_point_high_no_enemies
[HAVE FLAG ==]
[NOT ENEMY HAS FLAG ==]
[TASKLIST ??beg [TASK ScorePoint ?p ??end_condition ] ??end][->>tasklist]
[WHERE p /= 220]
[LVARS
 [tm = sim_myself.enemy_team]
]
[NOT AVAILABLE == [== [TYPE PLAYER] == [Team ^tm] ==] ==]
==>
[DEL ?tasklist]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
[TASKLIST ??beg  [TASK ScorePoint  220 ??end_condition] ??end]
[SAYIF DEBUG updated goal ScorePoint with priority 220]

enddefine;




;;; Generated only when the target enemy has disappeared.
define :ruleset generate_track_enemy;
    [DLOCAL
     [prb_allrules = true]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; Generate this goal with priority of 200 to begin with.
RULE generate_track_enemy
[LVARS
 [enemy = sim_myself.attack_target]
]
[Vanished ?enemy]
[NOT AVAILABLE == [[ID ?enemy] [TYPE PLAYER] == [Visible true]] ==]
;;;[NOT TrackEnemy CycleCounter ==]  ;;; got one of these left from previous
[NOT TASKLIST == [TASK TrackEnemy ==] ==]
```

```
[NOT GOAL TrackEnemy ==]
==>
[TrackEnemy CycleCounter 1]
[TESTADD GOAL TrackEnemy 200 NonMaintenance FOUND]
[POP11 (num_goals_generated + 1) -> num_goals_generated]
[SAYIF DEBUG generated goal TrackEnemy with priority 100]


;;; Reduce the priority over time.
RULE update_track_enemy
[LVARS
 [enemy = sim_myself.attack_target]
]
[Vanished ?enemy]
[NOT AVAILABLE == [[ID ?enemy] [TYPE PLAYER] == [Visible true]] ==]
[TrackEnemy CycleCounter ?count][->>counter]
[TASKLIST ??beg [TASK TrackEnemy ?p ??end_condition] ??end][->>taskl]
[NOT GOAL TrackEnemy ==]
==>
[DEL ?counter ?taskl]
[LVARS
 [priority = (200 - (10*count))] ;;; reduce priority each cycle
 [new_count = (1 + count)]
]
[TrackEnemy CycleCounter ?new_count]
[POP11 (num_goals_updated + 1) -> num_goals_updated]
;;;[TESTADD GOAL TrackEnemy ?priority NonMaintenance FOUND]
[TASKLIST ??beg [TASK TrackEnemy ?priority ??end_condition] ??end]
[SAYIF DEBUG generated goal TrackEnemy with priority ?priority count is
             ?count]


;;; Clean up info when the enemy is no longer marked as having vanished.
RULE remove_cycle_counter
[LVARS
 [enemy = sim_myself.attack_target]
]
[NOT Vanished ?enemy]
[TrackEnemy CycleCounter ?count] [->>counter]
==>
[DEL ?counter]
[SAYIF DEBUG removed cycle counter (no vanished enemy)]


;;; Also clean up info if the goal priority dropped below the filter threshold.
RULE remove_cycle_counter2
;;;[NOT Vanished ==]
[Vanished ==] [->>v]
[TrackEnemy CycleCounter ?count] [->>counter]
[WHERE (200 - (10*count)) < filter_threshold]
```

```
==>
[DEL ?counter ?v]
[SAYIF DEBUG removed cycle counter (goal dropped below filter)]

enddefine;
```

F.2 The go_to TRP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      go_to.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Contains the code for the go_to TRP.
;;;
;;; go_to is not a top-level program.  It requires an ARGS line which should
;;; follow the following format:  [ARGS go_to [x y z] caller rule_number]
;;; x, y, and z are coordinates of the location to go to.  The caller is
;;; the calling program (name for use by suspension_bind) and rule_number
;;; should be an integer which gets added to the rule numbers in go_to for
;;; use in suspension_bind.  (rule_number is the number of the rule in the
;;; calling program which calls go_to.  Any subsequent rules in the caller
;;; should use a base value of rule_number + number of rules in go_to for
;;; calls to suspension_bind.)

define :ruleset go_to;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; At the destination and have removed info.  Delete the ARGS line to
;;; indicate that this program is finished.
RULE done
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num][->> args]
[WHERE near(loc, location(sim_myself))]
[LVARS
 [query = suspension_bind([query [[TYPE Request] [WHAT path]
                                [Caller ^prog_name]] []],
                         prog_name, 0 + rule_num, true)]
]
[WHERE null(query)]
[LVARS
 [path = suspension_bind([path [[TYPE PATH] [Caller ^prog_name]] []],
                         prog_name, 0 + rule_num, true)]
]
[WHERE null(path)]
==>
[DEL ?args]
[SAYIF DEBUGGOTO go_to is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; At the destination, remove info.
RULE delete_info
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[WHERE near(loc, location(sim_myself))]
[LVARS
 [query = suspension_bind([query [[TYPE Request] [WHAT path]
                                  [Caller ^prog_name]] []],
                          prog_name, 1 + rule_num, false)]
]
[WHERE not(null(query))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", query))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                         prog_name, 1 + rule_num, false)]
]
[WHERE not(null(path))]
[LVARS
 [current_node = suspension_bind([node [[TYPE CurrentNode]
                                        [Caller ^prog_name]] []],
                                 prog_name, 1+rule_num, false)]]
[WHERE not(null(current_node))]
==>
[POP11 make_permanent_binding(path)]
[POP11 make_permanent_binding(query)]
[POP11 make_permanent_binding(current_node)]
[SAYIF DEBUGGOTO go_to cleaned up]
[LVARS
 [id = hd(property_from_binding("ID", query))]
 [node_id = hd(property_from_binding("ID", current_node))]
]
[POP11 grue_do_action([[do remove_resource ^id] [do remove_resource ^queryid]
                       [do remove_resource ^current_node]]);]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; At the destination, remove info.  This is for the case where there is
;;; no current node (bot had an empty path).
RULE delete_info2
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[WHERE near(loc, location(sim_myself))]
[LVARS
 [query = suspension_bind([query [[TYPE Request] [WHAT path]
                                  [Caller ^prog_name]] []],
                          prog_name, 1 + rule_num, false)]
]
[WHERE not(null(query))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", query))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                         prog_name, 1 + rule_num, false)]
]
[WHERE not(null(path))]
==>
[POP11 make_permanent_binding(path)]
[POP11 make_permanent_binding(query)]
[SAYIF DEBUGGOTO go_to cleaned up]
[LVARS
 [id = hd(property_from_binding("ID", query))]
]
[POP11 grue_do_action([[do remove_resource ^id]
                       [do remove_resource ^queryid]]);]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; At the destination, remove any spare queries hanging around.
;;; (This is a kludge - really there shouldn't be any spare queries.)
RULE delete_spare_query
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[WHERE near(loc, location(sim_myself))]
[LVARS
 [query = suspension_bind([query [[TYPE Request] [WHAT path]
                                  [Caller ^prog_name]] []],
                          prog_name, 2 + rule_num, false)]
]
[WHERE not(null(query))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", query))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                         prog_name, 2 + rule_num, false)]
]
[WHERE null(path)]
==>
[POP11 make_permanent_binding(query)]
[SAYIF DEBUGGOTO go_to cleaned up a spare query ?query]
```

```
[LVARS
 [id = hd(property_from_binding("ID", query))]
]
[POP11 grue_do_action([[do remove_resource ^id] ]);]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; The server returns an empty list of path nodes when the bot is close
;;; enough to run straight to the destination
RULE empty_path
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [query = suspension_bind([query [[TYPE Request] [WHAT path]
                                  [Caller ^prog_name]] []],
                          prog_name, 4 + rule_num, false)]
]
[WHERE not(null(query))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", query))]
 [path = suspension_bind([path [[TYPE PATH] [ID ^queryid]] []],
                         prog_name, 4 + rule_num, false)]
]
[WHERE not(null(path))]
[LVARS
 [pathnodes = property_from_binding("Nodes", path)]
]
[WHERE null(pathnodes)]
[WHERE threeDDistance(bot.location, loc) <= MAX_SEGMENT_LENGTH]
==>
[POP11 make_shared_binding(path)]
[POP11 make_shared_binding(query)]
[SAYIF DEBUGGOTO go_to got empty pathnode list, running straight to
                 destination]
[POP11 grue_do_action([
    [do continuous_action_step GOTO [RUNTO [LOCATION ^^loc]]] ])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Delete the path if its left over from before the bot was killed
;;; ** The interface should be deleting the path, query and current node
;;;    when the bot is killed.  Keep this rule just in case this situation
;;;    somehow happens, though.  (Like if the bot gets knocked off course?)
RULE bad_path
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [query = suspension_bind([query [[TYPE Request] [WHAT path]
                                  [Caller ^prog_name]] []],
                          prog_name, 3 + rule_num, false)]
]
[WHERE not(null(query))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", query))]
 [path = suspension_bind([path [[TYPE PATH] [ID ^queryid]] []],
                        prog_name, 3 + rule_num, false)]
]
[WHERE not(null(path))]
[LVARS
 [startloc = hd(property_from_binding("StartLocation", query))]
 [endloc = hd(property_from_binding("EndLocation", query))]
 [nodes = property_from_binding("Nodes", path)]
 ]
[WHERE not(isBotOnPath(bot, startloc, endloc, nodes))]
==>
[POP11 make_permanent_binding(path)]
[POP11 make_permanent_binding(query)]
[LVARS
 [path_id = hd(property_from_binding("ID", path))]
 [query_id = hd(property_from_binding("ID", query))]
 [currloc = bot.location]
]
[SAYIF DEBUGGOTO go_to got a bad path, deleting info, bot at ?currloc and
                 path was ?path]
[POP11 grue_do_action([[do remove_resource ^path_id]
                       [do remove_resource ^query_id]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Already at the next node, so remove the node from the node list and
;;; replace the current node.  But check that we're not already at the last
;;; node in the list...
RULE at_next_node
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [request = suspension_bind([request [[TYPE Request] [WHAT path]
                                      [Caller ^prog_name]] []],
                           prog_name, 4 + rule_num, false)]
 ]
[WHERE not(null(request))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", request))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                        prog_name, 5 + rule_num, false)]
 [current_node = suspension_bind([current [[TYPE CurrentNode]
                                           [Caller ^prog_name]] []],
                                prog_name, 5 + rule_num, false)]
]
[WHERE not(null(path))]
```

```
[WHERE not(null(current_node))]
[LVARS
 [nodes = property_from_binding("Nodes", path)]
]
[WHERE not(null(nodes))]
[WHERE length(hd(nodes)) >= 5]   ;;; node has number, name, and 3 coords
[WHERE near(tl(tl(hd(nodes))), location(sim_myself))]
[WHERE length(nodes) > 1]
==>
[POP11 make_permanent_binding(path)]
[POP11 make_shared_binding(request)]
[POP11 make_permanent_binding(current_node)]
[SAYIF DEBUGGOTO go_to removed a node from the list and swapped the
                 current node]
[LVARS
 [old_current_id = hd(property_from_binding("ID", current_node))]
 [new_current_node = hd(nodes)]
 [newnodes = tl(nodes)]
 [current_id = generate_id()]
 [nodeloc = tl(tl(hd(newnodes)))]
]
[POP11 grue_do_action([[do remove_resource ^queryid]
    [do remove_resource ^old_current_id]
    [do add_resource [[ID ^queryid] [TYPE PATH] [Nodes ^^newnodes]]]
    [do add_resource [[ID ^current_id] [TYPE CurrentNode]
                      [Node ^new_current_node]
                      [Caller ^prog_name] ]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Near the last node in the path, but for some reason the last node
;;; in the path is not the target location.
RULE run_to_endpoint
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [request = suspension_bind([request [[TYPE Request] [WHAT path]
                                      [Caller ^prog_name]] []],
                            prog_name, 7 + rule_num, false)]
 ]
[WHERE not(null(request))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", request))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                        prog_name, 7 + rule_num, false)]
 [current_node = suspension_bind([current [[TYPE CurrentNode]
                                           [Caller ^prog_name]] []],
                                 prog_name, 7 + rule_num, false)]
]
[WHERE not(null(path))]
```

```
[WHERE not(null(current_node))]
[LVARS
 [nodes = property_from_binding("Nodes", path)]
 [startloc = hd(property_from_binding("StartLocation", request))]
 [endloc = hd(property_from_binding("EndLocation", request))]
 [currnode = hd(property_from_binding("Node", current_node))]
 [currloc = tl(tl(currnode))]
 ;;; current is the index of the current node in nodes
 [current = isBotOnPath(sim_myself, startloc, endloc, nodes)]
]
[WHERE current_node]
[WHERE current]
[WHERE current >= length(tl(nodes))]
==>
[POP11 make_shared_binding(path)]
[POP11 make_shared_binding(request)]
[POP11 make_shared_binding(current_node)]
[SAYIF DEBUGGOTO go_to is moving to target location ?loc, bot is at node
                 ?current, nodes are ?nodes]
[POP11 grue_do_action([
    [do continuous_action_step GOTO [RUNTO [LOCATION ^^loc]]] ])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Near the path, so run to the next node as determined by isBotOnPath.
;;; Also change the current node.
RULE run_to_node
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [request = suspension_bind([request [[TYPE Request] [WHAT path]
                                      [Caller ^prog_name]] []],
                            prog_name, 7 + rule_num, false)]
 ]
[WHERE not(null(request))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", request))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                         prog_name, 7 + rule_num, false)]
 [last_node = suspension_bind([current [[TYPE CurrentNode]
                                        [Caller ^prog_name]] []],
                              prog_name, 7 + rule_num, false)]
]
[WHERE not(null(path))]
[WHERE not(null(last_node))]
[LVARS
 [nodes = property_from_binding("Nodes", path)]
 [startloc = hd(property_from_binding("StartLocation", request))]
 [endloc = hd(property_from_binding("EndLocation", request))]
 [currnode = hd(property_from_binding("Node", last_node))]
 [currloc = tl(tl(currnode))]
 ;;; currentNode is the index of current node
 [currentNode = isBotOnPath(sim_myself, startloc, endloc, nodes)]
]
[WHERE currentNode]
==>
[POP11 make_shared_binding(path)]
[POP11 make_shared_binding(request)]
[POP11 make_permanent_binding(last_node)]
[SAYIF DEBUGGOTO go_to is moving to next node in list ?nodes]
[LVARS
 [newnodes = tl(nodes)]
 [nodeloc = tl(tl(nodes(currentNode + 1)))]
 [old_current_id = hd(property_from_binding("ID", last_node))]
 [current_id = generate_id()]
]
;;; A hack to handle the case where the current node doesn't need
;;; to be changed.
[POP11 if (currentNode > 0) and (currentNode <= length(nodes)) then
    nodes(currentNode) -> new_current_node;
    else
    currnode -> new_current_node;
    endif;]
[POP11 grue_do_action([
    [do remove_resource ^old_current_id]
    [do add_resource [[ID ^current_id] [TYPE CurrentNode]
                      [Node ^new_current_node]
                      [Caller ^prog_name] ]]
    [do continuous_action_step GOTO [RUNTO [LOCATION ^^nodeloc]]] ])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; This rule is for running to the first node in the path
;;; Theoretically, the bad_path rule should get rid of cases where the
;;; bot isn't on the path, so we don't have to worry about it here.
RULE run_to_first_node
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [request = suspension_bind([request [[TYPE Request] [WHAT path]
                                     [Caller ^prog_name]] []],
                           prog_name, 7 + rule_num, false)]
 ]
[WHERE not(null(request))]
[LVARS
 [queryid = hd(property_from_binding("QUERYID", request))]
 [path = suspension_bind([path [[ID ^queryid] [TYPE PATH]] []],
                         prog_name, 8 + rule_num, false)]
 [currentNode = suspension_bind([current_node [[TYPE CurrentNode]
                                              [Caller ^prog_name]] [] ],
                                 prog_name, 8+rule_num, false)]
]
[WHERE not(null(path))]
[WHERE not(null(currentNode))]
[LVARS
 [nodes = property_from_binding("Nodes", path)]
]
==>
[POP11 make_shared_binding(path)]
[POP11 make_shared_binding(request)]
[POP11 make_shared_binding(currentNode)]
[SAYIF DEBUGGOTO go_to is moving to first node in list ?nodes]
[LVARS
 [newnodes = tl(nodes)]
 [nodeloc = tl(tl(nodes(1)))]
]
[POP11 grue_do_action([
    [do continuous_action_step GOTO [RUNTO [LOCATION ^^nodeloc]]] ])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Have no path, so request the path.  This is the case where we haven't
;;; already made a Request resource.
RULE request_path
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [request = suspension_bind([request [[TYPE Request] [WHAT path]
                                      [Caller ^prog_name]] []],
                            prog_name, 9 + rule_num, true)]
 ]
[WHERE null(request)]
==>
[SAYIF DEBUGGOTO go_to requesting path]
[LVARS
  [queryid = consword('goto'><generate_id())]
  [requestid = generate_id()]
  [currentid = generate_id()]
  [startloc = bot.location]
]
[POP11 grue_do_action([[do add_resource [[ID ^requestid] [TYPE Request]
                                         [WHAT path] [QUERYID ^queryid]
                                         [Caller ^prog_name]
                                         [StartLocation ^startloc]
                                         [EndLocation ^loc]]]
                        [do continuous_action_step GOTO
                                                  [GETPATH [LOCATION ^^loc]
                                                           [ID ^queryid]]]
                        [do add_resource [[ID ^currentid] [TYPE CurrentNode]
                                         [Node [-1 Start ^^startloc]]
```

```
                                          [Caller ^prog_name]]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Have no path, so request the path.
;;; In this case, we already have a Request resource.
RULE request_path_repeat
[CURRENT ?task ?priority ?prog_name]
[ARGS go_to ?loc ?prog_name ?rule_num]
[LVARS
 [request = suspension_bind([request [[TYPE Request] [WHAT path]
                                       [Caller ^prog_name]] []],
                            prog_name, 10 + rule_num, false)]
 ]
[WHERE not(null(request))]
==>
[POP11 make_shared_binding(request)]
[SAYIF DEBUGGOTO go_to requesting path]
[LVARS
  [queryid = hd(property_from_binding("QUERYID", request))]
]
[POP11 grue_do_action([[do continuous_action_step GOTO
                                                 [GETPATH [LOCATION ^^loc]
                                                 [ID ^queryid]]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]



RULE endcase
==>
[SAYIF DEBUGGOTO go_to hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;
```

F.3 Other TRPs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  File:      grue_ut_programs.p
;;;  Author:    Elizabeth Gordon, University of Nottingham
;;;  Copyright 2005 Elizabeth Gordon
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Programs for the GRUE UT agent


;;; Causes the agent to return to its own flag base and turn to face down
;;; the ramp.  Calls go_to.
define :ruleset return_to_base;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;;  If the ARGS line is gone (go_to is finished running) and the bot is
;;; facing the correct direction, then finished.
RULE done
[NOT ARGS go_to ?loc return_to_base 2]
[WHERE loc = sim_myself.default_location]
[WHERE near(sim_myself.location, sim_myself.default_location)]
[WHERE facing(sim_myself.rotation, sim_myself.location, sim_myself.turn_loc)]
==>
[UNRUNNABLE return_to_base]
[SAYIF DEBUGPROGS return_to_base is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the ARGS line is gone, then go_to is finished running.  Turn to the
;;; correct heading.
RULE at_base
[NOT ARGS go_to ?loc return_to_base 2]
[WHERE loc = sim_myself.default_location]
[WHERE nearNode(sim_myself.location, sim_myself.default_Location)]
==>
[SAYIF DEBUGPROGS return_to_base is at the base and turning]
[LVARS [turn_loc = sim_myself.turn_loc]]
[POP11 grue_do_action([[do continuous_action_step
                           RETURN [TURNTO [LOCATION ^^turn_loc]]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
==>
[LVARS [loc = sim_myself.default_location]]
[TESTADD ARGS go_to ?loc return_to_base 2]
[SAYIF DEBUGPROGS return_to_base called go_to]
[RESTORERULESET go_to]

enddefine;
```

```
;;; Causes the agent to fetch a health pack if its health is low.
define :ruleset get_health;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;;  If the bot is healthy, then finish.  (The only way to have health
;;;  restored w/o getting to the healthpack is to be killed, so assume
;;;  respawn_cleanup will get rid of the args line.)
RULE done
[WHERE sim_myself.healthlevel > HEALTH_THRESHOLD]
==>
[UNRUNNABLE get_health]
[SAYIF DEBUGPROGS get_health is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the ARGS line is gone, then go_to is finished running.  Remove the
;;; healthpack resource.
RULE at_healthpack
[NOT ARGS go_to ?loc get_health 2]
[LVARS
 [healthpack = suspension_bind([health [[TYPE HEALTHPACK]] []], get_health, 1,
                               false)]
]
[WHERE not(null(healthpack))]
[LVARS
 [healthloc = hd(property_from_binding("Location", healthpack))]
 [healthid = hd(property_from_binding("ID", healthpack))]
]
[WHERE near(sim_myself.location, healthloc)]
==>
[POP11 make_permanent_binding(healthpack)]
[SAYIF DEBUGPROGS get_health is at the healthpack, removing it ]
[POP11 grue_do_action([[do remove_resource ^healthid]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
[LVARS
    [healthpack = suspension_bind([health [[TYPE HEALTHPACK]] []],
                                  get_health, 2, false)]
]
[WHERE not(null(healthpack))]
==>
[POP11 make_shared_binding(healthpack)]
[LVARS
   [loc = hd(property_from_binding("Location", healthpack))]
]
[TESTADD ARGS go_to ?loc get_health 2]
[SAYIF DEBUGPROGS get_health called go_to with loc ?loc, binding is
                  ?healthpack]
[RESTORERULESET go_to]


;;; If no rules matched, then this TRP is unrunnable.
RULE unrunnable
==>
[UNRUNNABLE get_health]
[SAYIF DEBUGPROGS get_health hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;


;;; Causes the agent to shoot at an opposing player.  Stops shooting when
;;; the player is no longer visible.
define :ruleset attack;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; Done if the target player is killed
RULE done
[WHERE shooting(sim_myself) /= true]
[LVARS
[myteam = team(sim_myself)]
[enemyteam = enemy_team(sim_myself)]
[target = attack_target(sim_myself)]
]
[WHERE target /= '']
[LVARS
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [ID ^target]] []],
                          attack, 0, true)]
]
```

```
[WHERE not(null(enemy))]
[KILLED ?target]
==>
[UNRUNNABLE attack]
[SAYIF DEBUGPROGS attack done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Cause the bot to stop shooting when no enemy is visible.
RULE stop_shooting_and_finish
[WHERE sim_myself.shooting = true]
[LVARS
 [myteam = team(sim_myself)]
 [target = sim_myself.attack_target]
]
[WHERE target /= '']
[LVARS
 [enemy = suspension_bind([enemy [[ID ^target] [TYPE PLAYER] [Visible true]]
                                 []], attack, 1, true)]
]
[WHERE null(enemy)]
==>
[POP11 grue_do_action([[do perform_gamebot_action [STOPSHOOT]]]);]
[POP11 false -> sim_myself.shooting]
[SAYIF DEBUGPROGS attack stopped shooting, enemy is ?enemy]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; This needs to check for a weapon as well as an enemy!!
RULE shoot
[WHERE weapstring(sim_myself) /= "None"]
[WHERE sim_myself.ammunition > 0]
[LVARS
 [myteam = team(sim_myself)]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [Visible true]] []],
                          attack, 2, false)]
 ;;; and holding weapon
]
[WHERE not(null(enemy))]
[WHERE hd(property_from_binding("Team", enemy)) /= myteam]
[LVARS
  [enemyloc = hd(property_from_binding("Location", enemy))]
]
;;; check for in range here
[WHERE in_range(sim_myself.location, enemyloc)]
[WHERE facing(sim_myself.rotation, sim_myself.location, enemyloc)]
==>
[POP11 make_shared_binding(enemy)]
[LVARS
 [enemy_id = hd(property_from_binding("ID", enemy))]
 ]
[POP11 grue_do_action([[do perform_gamebot_action
                           [TURNTO [TARGET ^enemy_id]]]
                       [do perform_gamebot_action  [SHOOT]]]])]
[POP11 true -> sim_myself.shooting]
[POP11 enemy_id -> sim_myself.attack_target]
[SAYIF DEBUGPROGS attack started shooting]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Turn to face the enemy.
RULE face_enemy
[WHERE weapstring(sim_myself) /= "None"]
[WHERE sim_myself.ammunition > 0]
[LVARS
 [myteam = team(sim_myself)]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [Visible true]] []],
                          attack, 3, false)]
]
[WHERE not(null(enemy))]
[WHERE hd(property_from_binding("Team", enemy)) /= myteam]
[LVARS
  [enemyloc = hd(property_from_binding("Location", enemy))]
]
[WHERE in_range(sim_myself.location, enemyloc)]
[WHERE not(facing(sim_myself.rotation, sim_myself.location, enemyloc))]
==>
[POP11 make_shared_binding(enemy)]
[LVARS
 [enemy_id = hd(property_from_binding("ID", enemy))]
 ]
[POP11 grue_do_action([[do perform_gamebot_action
                           [TURNTO [LOCATION ^^enemyloc]]]
                       [do perform_gamebot_action [SHOOT]]]])]
[POP11 true -> sim_myself.shooting]
[POP11 enemy_id -> sim_myself.attack_target]
[SAYIF DEBUGPROGS attack turning to face enemy]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Rule:  approach enemy
;;; Handles the case where enemy is out of range, just does a runto
RULE approach_enemy
[WHERE weapstring(sim_myself) /= "None"]
[WHERE sim_myself.ammunition > 0]
[LVARS
 [myteam = team(sim_myself)]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [Visible true]] []],
                          attack, 4, false)]
]
```

```
[WHERE not(null(enemy))]
[WHERE hd(property_from_binding("Team", enemy)) /= myteam]
[LVARS
  [enemyloc = hd(property_from_binding("Location", enemy))]
]
[WHERE not(in_range(sim_myself.location, enemyloc))]
==>
[POP11 make_shared_binding(enemy)]
[LVARS
 [enemy_id = hd(property_from_binding("ID", enemy))]
 [loc = compute_attack_point(sim_myself.location, enemyloc, WEAPONS_RANGE)]
]
[POP11 grue_do_action([[do perform_gamebot_action  [RUNTO [LOCATION ^^loc]]]
                       [do perform_gamebot_action [SHOOT]]]);]
[POP11 true -> sim_myself.shooting]
[POP11 enemy_id -> sim_myself.attack_target]
[SAYIF DEBUGPROGS attack approaching enemy]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Left over from unused suspension code.
RULE suspend
[WHERE suspendable(attack)]
==>
[UNRUNNABLE attack]
[SAYIF DEBUGPROGS attack is suspendable]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make this program unrunnable if no rules match.
RULE endcase
==>
[UNRUNNABLE attack]
[SAYIF DEBUGPROGS attack unrunnable]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;



;;; Causes the agent to fetch a weapon if it doesn't have one.
define :ruleset get_weapon;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; If the bot has a weapon, then finish
RULE done
[WHERE sim_myself.weapstring /= "None"]
==>
[UNRUNNABLE get_weapon]
[SAYIF DEBUGPROGS get_weapon is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the ARGS line is gone, then go_to is finished running.
RULE at_weapon
[NOT ARGS go_to ?loc get_weapon 2]
[LVARS
 [weapon = suspension_bind([weapon [[TYPE WEAPON]] []], get_weapon, 1, false)]
]
[WHERE not(null(weapon))]
[LVARS
 [weaploc = hd(property_from_binding("Location", weapon))]
 [weapid = hd(property_from_binding("ID", weapon))]
]
[WHERE near(sim_myself.location, weaploc)]
==>
[POP11 make_permanent_binding(weapon)]
[SAYIF DEBUGPROGS get_weapon is at the weapon, removing it ]
[POP11 grue_do_action([[do remove_resource ^weapid]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
[LVARS
    [weapon = suspension_bind([weapon [[TYPE WEAPON]] []],
                              get_weapon, 2, false)]
]
[WHERE not(null(weapon))]
==>
[POP11 make_shared_binding(weapon)]
[LVARS
   [loc = hd(property_from_binding("Location", weapon))]
]
[TESTADD ARGS go_to ?loc get_weapon 2]
[SAYIF DEBUGPROGS get_weapon called go_to with loc ?loc, binding is ?weapon]
[RESTORERULESET go_to]


;;; Make this program unrunnable when no rules match.
RULE unrunnable
```

```
==>
[UNRUNNABLE get_weapon]
[SAYIF DEBUGPROGS get_weapon hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;


;;; Pick a patrol point and walk to it.
define :ruleset patrol;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;;  If the ARGS line is gone (go_to is finished running)
;;;  and patrol point removed then finished running patrol.
RULE done
[NOT ARGS go_to == patrol ==]
[WHERE not(nearby(bot.location, sim_myself.default_location))]
[LVARS
 [patrolpoint = suspension_bind([patrolpoint [[TYPE PatrolPoint]] []],
                                patrol, 0, true)]
]
[WHERE null(patrolpoint)]
==>
[UNRUNNABLE patrol]
[SAYIF DEBUGPROGS patrol is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Remove the stored patrol point resource.
RULE remove_patrol_point
[NOT ARGS go_to == patrol ==]
[LVARS
 [patrolpoint = suspension_bind([patrolpoint [[TYPE PatrolPoint]] []],
                                patrol, 1, false)]
 [timestamp = suspension_bind([time [[TYPE Time] [Caller patrol] [Number two]]
                                    []], patrol, 1, false)]
]
[WHERE not(null(patrolpoint))]
[WHERE not(null(timestamp))]
[LVARS
  [loc = hd(property_from_binding("Location", patrolpoint))]
```

```
]
[WHERE nearby(bot.location, loc)]
[LVARS
   [current_time = sys_real_time()]
   [timestamp_value = hd(property_from_binding("Value", timestamp))]
]
[WHERE current_time > (timestamp_value + 14)]  ;;; wait about 15 secs
==>
[POP11 make_permanent_binding(patrolpoint)]
[POP11 make_permanent_binding(timestamp)]
[LVARS
 [patrolid = hd(property_from_binding("ID", patrolpoint))]
 [time_id = hd(property_from_binding("ID", timestamp))]
]
[POP11 grue_do_action([[do remove_resource ^patrolid]
                       [do remove_resource ^time_id]])]
[SAYIF DEBUGPROGS patrol removed a patrol point]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; At the patrol point, turn around.
RULE rotate
[NOT ARGS go_to == patrol ==]
[LVARS
 [patrolpoint = suspension_bind([patrolpoint [[TYPE PatrolPoint]] []],
                                patrol, 1, false)]
 [timestamp = suspension_bind([time [[TYPE Time] [Caller patrol] [Number two]]
                                    []], patrol, 1, false)]
]
[WHERE not(null(patrolpoint))]
[WHERE not(null(timestamp))]
[LVARS
  [loc = hd(property_from_binding("Location", patrolpoint))]
]
[WHERE nearby(bot.location, loc)]
==>
[POP11 make_shared_binding(timestamp)]
[POP11 grue_do_action([[do perform_gamebot_action [ROTATE [Amount 100]]]]);]
[SAYIF DEBUGPROGS patrol rotating]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Start a timer so the bot can wait/turn around at the patrol point.
RULE start__second_timer
[NOT ARGS go_to == patrol ==]
[LVARS
 [patrolpoint = suspension_bind([patrolpoint [[TYPE PatrolPoint]] []],
                                patrol, 1, false)]
 [timestamp = suspension_bind([time [[TYPE Time] [Caller patrol] [Number one]]
                                    []], patrol, 1, false)]
```

```
]
[WHERE not(null(patrolpoint))]
[WHERE not(null(timestamp))]
[LVARS
  [loc = hd(property_from_binding("Location", patrolpoint))]
]
[WHERE nearby(bot.location, loc)]
==>
[LVARS
  [time = sys_real_time()]  ;;; time in seconds
  [time_id = generate_id()]
  [old_time_id = hd(property_from_binding("ID", timestamp))]
]
[SAYIF DEBUGPROGS patrol started timing]
[POP11 grue_do_action([[do remove_resource ^old_time_id]
                       [do add_resource [[ID ^time_id] [TYPE Time]
                                         [Caller patrol] [Number two]
                                         [Value ^time]]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]



;;; Make a call to go_to, which will pop back to arbitrator when finished.
;;; Uses patrol point selected by previous rule
RULE follow_path
[LVARS
  [patrolpoint = suspension_bind([patrolpoint [[TYPE PatrolPoint]] []],
                                 patrol, 2, false)]
]
[WHERE patrolpoint /= []]
==>
[POP11 make_shared_binding(patrolpoint)]
[LVARS
  [patrol_point = property_from_binding("Location", patrolpoint)]
]
[TESTADD ARGS go_to ??patrol_point patrol 2]
[SAYIF DEBUGPROGS patrol called go_to with location ??patrol_point]
[POP11 bot.location =>]
[RESTORERULESET go_to]


;;; Pick a random patrol point + call go_to
RULE pick_patrol_point
[WHERE near(bot.location, sim_myself.default_location)]
[LVARS
 [timestamp = suspension_bind([time [[TYPE Time] [Caller patrol]] []],
                              patrol, 3, false)]
]
[WHERE not(null(timestamp))]
[LVARS
```

```
   [current_time = sys_real_time()]
   [timestamp_value = hd(property_from_binding("Value", timestamp))]
]
[WHERE current_time > (timestamp_value + 9)]  ;;; wait about 10 secs
==>
[POP11 make_shared_binding(timestamp)]
[LVARS
 [patrol_point = pick_random(sim_myself.patrol_pts)]
 [patrol_id = generate_id()]
]
;;; This is 2 rather than 3 because the main call is in the rule above, this
;;; was added to speed things up
[TESTADD ARGS go_to ?patrol_point patrol 2]
[SAYIF DEBUGPROGS patrol picked ?patrol_point]
[POP11 grue_do_action([[do add_resource [[ID ^patrol_id] [TYPE PatrolPoint]
                                          [Location ^patrol_point]]]])]
[RESTORERULESET go_to]


;;; Wait for a bit at the default location.
RULE wait
[WHERE near(bot.location, sim_myself.default_location)]
[LVARS
 [timestamp = suspension_bind([time [[TYPE Time] [Caller patrol]] []],
                              patrol, 5, false)]
]
[WHERE not(null(timestamp))]
==>
[POP11 make_shared_binding(timestamp)]
[SAYIF DEBUGPROGS patrol still timing]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Start timer to cause the bot to wait at the default location.
RULE start_timer
[WHERE near(bot.location, sim_myself.default_location)]
[LVARS
 [timestamp = suspension_bind([time [[TYPE Time] [Caller patrol]] []],
                              patrol, 4, true)]
]
[WHERE null(timestamp)]
==>
[LVARS
  [time = sys_real_time()]  ;;; time in seconds
  [time_id = generate_id()]
]
[SAYIF DEBUGPROGS patrol started timing]
[POP11 grue_do_action([[do add_resource [[ID ^time_id] [TYPE Time]
                                          [Caller patrol] [Number one]
                                          [Value ^time]]]])]
```

```
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; This program is unrunnable if no rules match.
RULE unrunnable
==>
[SAYIF DEBUGPROGS patrol hit end case]
[UNRUNNABLE patrol]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;



;;; Causes the agent to retrieve its own flag if the flag is not on the
;;; flag base.
define :ruleset get_own_flag;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; If the flag is gone, then finish.  (This will trigger even if the
;;; enemy has the flag.  But, in that case the bot will have to kill
;;; the enemy before it can get the flag.
RULE done
[LVARS
   [flag = suspension_bind([flag [[TYPE FLAG]] []], get_own_flag, 0, true)]
]
[WHERE null(flag)]
==>
[UNRUNNABLE get_own_flag]
[SAYIF DEBUGPROGS get_own_flag is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the ARGS line is gone, then go_to is finished running.  Remove the
;;; flag resource.
RULE at_flag
[LVARS
 [flag = suspension_bind([flag [[TYPE FLAG]] []], get_own_flag, 1, false)]
]
[WHERE not(null(flag))]
[LVARS
```

```
 [flagloc = hd(property_from_binding("Location", flag))]
 [flagid = hd(property_from_binding("ID", flag))]
 [flagteam = hd(property_from_binding("Team", flag))]
]
[WHERE flagteam = bot.team]
[WHERE not(near(flagloc, sim_myself.base_loc))]
[WHERE near(sim_myself.location, flagloc)]
==>
[POP11 make_permanent_binding(flag)]
[SAYIF DEBUGPROGS get_own_flag is at the flag, removing it ]
[POP11 grue_do_action([[do remove_resource ^flagid]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Run straight to the flag on the assumption that if it's visible it's also
;;; reachable.  Possibly not a good assumption, but it'll work most of the
;;; time.
RULE run_to_flag
[LVARS
    [flagteam = bot.team]
    [flag = suspension_bind([flag [[TYPE FLAG] [Team ^flagteam]] []],
                            get_own_flag, 2, false)]
]
[WHERE not(null(flag))]
[LVARS
    [loc = hd(property_from_binding("Location", flag))]
]
[WHERE not(near(loc, sim_myself.base_loc))]
==>
[POP11 make_shared_binding(flag)]
[POP11 grue_do_action([[do perform_gamebot_action [RUNTO [LOCATION ^^loc]]]]);]
[SAYIF DEBUGPROGS get_own_flag ran to loc ?loc, binding is ?flag]
[RESTORERULESET go_to]


;;; Make this program unrunnable if no rules matched.
RULE unrunnable
==>
[UNRUNNABLE get_own_flag]
[SAYIF DEBUGPROGS get_own_flag hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;



;;; Causes the agent to fetch ammo if it doesn't have any.
;;; Will go for a weapon instead if it can't see any ammo
;;; (weapons come with ammo).
define :ruleset get_ammo;
```

```
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; If the bot has ammo, then finish
RULE done
[WHERE sim_myself.ammunition = MAX_AMMO]
==>
[UNRUNNABLE get_ammo]
[SAYIF DEBUGPROGS get_ammo is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the ARGS line is gone, then go_to is finished running.
RULE at_weapon
[NOT ARGS go_to ?loc get_ammo 2]
[LVARS
 [ammo = suspension_bind([ammo [[TYPE AMMO]] []], get_ammo, 1, false)]
]
[WHERE not(null(ammo))]
[LVARS
 [ammoloc = hd(property_from_binding("Location", ammo))]
 [ammoid = hd(property_from_binding("ID", ammo))]
]
[WHERE near(sim_myself.location, ammoloc)]
==>
[POP11 make_permanent_binding(ammo)]
[SAYIF DEBUGPROGS get_ammo is at the ammo, removing it ]
[POP11 grue_do_action([[do remove_resource ^ammoid]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
[LVARS
    [ammo = suspension_bind([ammo [[TYPE AMMO]] []], get_ammo, 2, false)]
]
[WHERE not(null(ammo))]
==>
[POP11 make_shared_binding(ammo)]
[LVARS
   [loc = hd(property_from_binding("Location", ammo))]
]
```

```
[TESTADD ARGS go_to ?loc get_ammo 2]
[SAYIF DEBUGPROGS get_ammo called go_to with loc ?loc, binding is ?ammo]
[RESTORERULESET go_to]


;;; This program is unrunnable if no rules matched.
RULE unrunnable
==>
[UNRUNNABLE get_ammo]
[SAYIF DEBUGPROGS get_ammo hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;



;;; Causes the bot to run to a safe point
define :ruleset run_away;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];

;;; Done if the enemy player is no longer visible and health is over the
;;; threshold.
RULE done
[LVARS
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [Visible true]] []],
                          run_away, 0, true)]
]
[WHERE null(enemy) or (hd(property_from_binding("Team", enemy)) = bot.team) or
 (bot.healthlevel >= HEALTH_THRESHOLD)]
==>
[UNRUNNABLE run_away]
[SAYIF DEBUGPROGS run_away is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
[LVARS
 [safept = suspension_bind([safety [[TYPE SafePoint]] [[Distance #| 0 -> :#]]],
                           run_away, 1, false)]
]
[WHERE safept /= []]
```

```
==>
[POP11 make_shared_binding(safept)]
[LVARS
 [loc = hd(property_from_binding("Location", safept))]
]
[TESTADD ARGS go_to ?loc run_away 1]
[SAYIF DEBUGPROGS run_away called go_to with safept ?safept loc is ?loc]
[RESTORERULESET go_to]


;;; This program is unrunnable if no rules matched.
RULE unrunnable
==>
[UNRUNNABLE run_away]
[SAYIF DEBUGPROGS run_away hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;



;;; Need to get path to enemy flag base.
;;; Run to enemy flag base.
;;; Pick up flag if it's there.
;;; Otherwise, if flag is somewhere else, pick it up.
;;; Done when holding enemy flag.
define :ruleset get_enemy_flag;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; Done if holding the enemy flag
RULE done
[HAVE FLAG =]  ;;; cheating by using Nick's database fact instead of a resource
==>
[UNRUNNABLE get_enemy_flag]
[SAYIF DEBUGPROGS get_enemy_flag is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the flag is visible, run to it and pick it up
RULE see_flag
[LVARS
```

```
 [enemyteam = sim_myself.enemy_team]
 [flag = suspension_bind([flag [[TYPE FLAG] [Team ^enemyteam]] []],
                        get_enemy_flag, 1, false)]
]
[WHERE flag /= []]
[LVARS
 [loc = hd(property_from_binding("Location", flag))]
]
[WHERE nearby(sim_myself.location, loc)]
==>
[POP11 make_shared_binding(flag)]
[POP11 grue_do_action([[do continuous_action_step GET_ENEMY_FLAG
                          [RUNTO [LOCATION ^^loc]]]]);]
[SAYIF DEBUGPROGS get_enemy_flag running to flag]
[RESTORERULESET grue_agent_arbitrator_stage_three]



;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
[WHERE not(nearNode(sim_myself.location, sim_myself.enemy_base))]
==>
[LVARS [loc = sim_myself.enemy_base]]
[TESTADD ARGS go_to ?loc get_enemy_flag 2]
[SAYIF DEBUGPROGS get_enemy_flag called go_to]
[RESTORERULESET go_to]


;;; This happens when the bot is at the enemy flag base but doesn't have the
;;; flag and the flag is not visible
RULE unrunnable
==>
[UNRUNNABLE get_enemy_flag]
[SAYIF DEBUGPROGS get_enemy_flag hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;


;;; Score a point by returning to the flag base.  This is different from
;;; return_to_base, which returns the bot to a spot a little in front of the
;;; flag.
define :ruleset score_point;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; done when the bot doesn't have the flag anymore
RULE done
[NOT HAVE FLAG =]
==>
[UNRUNNABLE score_point]
[SAYIF DEBUGPROGS score_point is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If the ARGS line is gone, then go_to is finished running.
RULE at_base
[LVARS [loc = sim_myself.base_loc]]
[NOT ARGS go_to ?loc score_point 2]
;;; need to check proximity or this will trigger before go_to has been called
[WHERE nearby(sim_myself.location, sim_myself.base_loc)]
[HAVE FLAG =]
[NOT ENEMY HAS FLAG ==]
==>
[SAYIF DEBUGPROGS score_point is finished running go_to]
[POP11 grue_do_action([[do continuous_action_step SCORE
                                                  [RUNTO [LOCATION ^^loc]]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Make a call to go_to, which will pop back to arbitrator when finished.
RULE follow_path
[HAVE FLAG =]
[NOT ENEMY HAS FLAG ==]
==>
[LVARS [loc = sim_myself.base_loc]]
[TESTADD ARGS go_to ?loc score_point 2]
[SAYIF DEBUGPROGS score_point called go_to]
[RESTORERULESET go_to]


;;; This would happen if the enemy got the bot's flag before the bot
;;; manages to score.  In that case, the bot can't score until it returns
;;; it's own flag.
RULE unrunnable
==>
[UNRUNNABLE score_point]
[SAYIF DEBUGPROGS score_point hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;
```

```
;;; Track down an enemy that's gone out of sight.
;;; The priority of this goal drops over time, so it will eventually drop
;;; out of the arbitrator (instead of looking forever.)
;;; Current strategy is to attempt to predict the position of the player
;;; based on last observed position/velocity.  Then go to the predicted
;;; position.  If enemy still not visible, then turn slowly in a circle.
define :ruleset track_enemy;
    [DLOCAL
     [prb_allrules = false]
     [prb_walk = false]
     [prb_walk_fast = false]
     [prb_chatty = false]
     [prb_show_ruleset = false]
     [prb_show_conditions = false]
     [cycle_limit = 1]
     ];


;;; done when the enemy is visible
RULE done
[LVARS
 [enemytarget = sim_myself.attack_target]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [ID ^enemytarget]
                                  [Visible true]] []],
                          track_enemy, 0, false)]
]
[WHERE not(null(enemy))]
[WHERE not(tracking(sim_myself))]
==>
[POP11 make_shared_binding(enemy)]
[POP11 false -> sim_myself.tracking]
[UNRUNNABLE track_enemy]
[SAYIF DEBUGPROGS track_enemy is done]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Clean up stored information if the enemy is visible again.
RULE clean_up
[LVARS
 [enemytarget = sim_myself.attack_target]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [ID ^enemytarget]
                                  [Visible true]] []],
                          track_enemy, 0, false)]
]
[WHERE not(null(enemy))]
[WHERE tracking(sim_myself) = true]
==>
[POP11 make_shared_binding(enemy)]
```

```
[POP11 false -> sim_myself.tracking]
[SAYIF DEBUGPROGS track_enemy is cleaning up]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Turn a small number of degrees.  It has to be small so bot won't turn
;;; past player without seeing it.
RULE turn
[WHERE tracking(sim_myself) = true]
[TrackEnemy CycleCounter ?c]
[WHERE c > TRACKCOUNT]
==>
;;;Turn
[POP11 grue_do_action([[do perform_gamebot_action [ROTATE [Amount 3500]]]]);]
[SAYIF DEBUGPROGS track_enemy is turning]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; If at the predicted position, delete the resource
RULE at_predicted_position
[WHERE tracking(sim_myself) = true]
[LVARS
 [enemytarget = sim_myself.attack_target]
 [predicted_position = suspension_bind([position [[TYPE Prediction]
                                                   [FOR ^enemytarget]
                                                   [BY TrackEnemy]] []],
                                       track_enemy, 2, false)]
]
[WHERE not(null(predicted_position))]
[LVARS
 [loc = property_from_binding("Location", predicted_position)]
 ]
[WHERE near(sim_myself.location, loc)]
==>
[POP11 make_permanent_binding(predicted_position)]
[LVARS
 [id = hd(property_from_binding("ID", predicted_position))]
]
[POP11 grue_do_action([[do remove_resource ^id]])]
[SAYIF DEBUGPROGS track_enemy is at predicted position]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Walk to the predicted position (directly, since it won't be far off)
RULE walk_to_predicted
[WHERE tracking(sim_myself) = true]
[TrackEnemy CycleCounter ?c]
[WHERE c <= TRACKCOUNT]
[LVARS
 [enemytarget = sim_myself.attack_target]
 [predicted_position = suspension_bind([position [[TYPE Prediction]
                                                   [FOR ^enemytarget]
                                                   [BY TrackEnemy]] []],
                                       track_enemy, 3, false)]
]
[WHERE not(null(predicted_position))]
==>
[POP11 make_shared_binding(predicted_position)]
[LVARS
 [loc = property_from_binding("Location", predicted_position)]
]
[POP11 grue_do_action([[do continuous_action_step TRACK_ENEMY
                           [RUNTO [LOCATION ^^loc]]]])]
[SAYIF DEBUGPROGS track_enemy running to predicted position ?loc]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; Predict the player's position and store the predicted position resource
RULE predict_position
[TrackEnemy CycleCounter ?c]
[WHERE c <= TRACKCOUNT]
[WHERE sim_myself.tracking = true]
[LVARS
 [enemytarget = sim_myself.attack_target]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [ID ^enemytarget]
                                   [Visible false]] []],
                          track_enemy, 4, false)]
]
[WHERE not(null(enemy))]
[LVARS
  [loc = hd(property_from_binding("Location", enemy))]
]
[WHERE nearNode(sim_myself.location, loc)]
==>
[POP11 make_shared_binding(enemy)]
[SAYIF DEBUGPROGS predicted position]
[LVARS
 [velocity = hd(property_from_binding("Velocity", enemy))]
 [timestamp = hd(property_from_binding("Timestamp", enemy))]
 [timediff = (sys_real_time() - timestamp)]
 [predicted_position = predictPosition(loc, velocity, timediff)]
 [id = generate_id()]
]
[POP11 true -> sim_myself.tracking]
[POP11 grue_do_action([[do add_resource [[ID ^id] [TYPE Prediction]
                                          [FOR ^enemytarget] [BY TrackEnemy]
                                          [Location ^^predicted_position]]]])]
[RESTORERULESET grue_agent_arbitrator_stage_three]
```

```
;;; RUNTO the enemy's last observed position
RULE run_to_enemy
[LVARS
 [enemytarget = sim_myself.attack_target]
 [enemy = suspension_bind([enemy [[TYPE PLAYER] [ID ^enemytarget]
                                  [Visible false]] []],
                          track_enemy, 5, false)]
]
[WHERE not(null(enemy))]
==>
[POP11 make_shared_binding(enemy)]
[LVARS
 [loc = hd(property_from_binding("Location", enemy))]
 [loc2 = sim_myself.location]
]
[POP11 true -> sim_myself.tracking]
[POP11 grue_do_action([[do continuous_action_step TRACK_ENEMY [RUNTO ^^loc]]])]
[SAYIF DEBUGPROGS running to enemys last observed position ?loc from ?loc2]
[RESTORERULESET grue_agent_arbitrator_stage_three]


;;; The program is unrunnable if no rules matched.
RULE end_case
==>
[UNRUNNABLE track_enemy]
[SAYIF DEBUGPROGS track_enemy hit end case]
[RESTORERULESET grue_agent_arbitrator_stage_three]

enddefine;
```

Appendix G

T-Tests for the Tileworld Agent

This appendix contains the results of applying the Student's t-test to the data from Tileworld. For each experimental version of the agent, we hypothesize that the agent performs identically to the standard GRUE agent. We then apply the t-test to the raw score data in each environmental condition, comparing the data against the mean for the raw scores of the standard GRUE agent in the same environmental condition. A value greater than 1.677 and less than -1.677 shows that there is no evidence at the 10% level of significance that the scores for the experimental agent are drawn from a population with a mean differing from the mean of the standard GRUE agent's scores. Thus, a value outside the range -1.677-1.677 indicates a significant difference from the standard GRUE agent in a particular environmental condition.

We should note that the t-test assumes that both data samples are drawn from normal populations. However, the central limit theorem states that as a sample size increases, the distribution becomes closer to a normal distribution, even if the actual distribution is different. Moore and McCabe (Moore and McCabe, 1993) suggest that the t-test is appropriate for samples sizes of 40 and larger, even if the distribution is significantly skewed. In our case, we know that the sample is drawn from a discrete population, and the boxplots shown in Chapter 5 indicate some degree of skewness. However, all of our samples are of size 50, so we will assume that the t-test is appropriate in this case.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-6.81	-5.87	-6.92	-7.02
40	-5.23	-10.14	-10.94	-13.77
70	-4.20	-7.82	-11.82	-9.41
100	-3.96	-9.19	-7.49	-11.37

Table G.1: Results of 2-tailed t-tests for the scores of the agent with all goals always present.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-4.90	-11.53	-13.33	-9.17
40	0.65	0.36	-2.24	-2.16
70	-1.80	-2.38	-1.24	-1.63
100	-0.57	-1.28	-3.00	-1.77

Table G.2: Results of 2-tailed t-tests for the seconds of processor time used by the agent with all goals always present.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-1.76	-0.82	-1.03	-0.30
40	1.80	-0.39	2.22	1.21
70	1.93	7.95	4.21	4.26
100	6.29	9.55	13.90	6.74

Table G.3: Results of 2-tailed t-tests for the scores of the agent with goal updates disabled.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-1.57	-5.28	-5.61	-3.63
40	-2.63	-3.89	-6.63	-4.63
70	-2.86	-5.64	-4.50	-1.71
100	-2.19	-3.29	0.53	-2.17

Table G.4: Results of 2-tailed t-tests for the seconds of processor time used by the agent with goal updates disabled.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-5.36	-1.06	-0.32	0.87
40	2.83	0.00	2.84	-0.42
70	3.72	9.87	7.45	4.97
100	5.18	8.76	19.32	6.09

Table G.5: Results of 2-tailed t-tests for the scores of the agent with constant goal priorities.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-0.39	-1.26	-2.86	-3.30
40	-1.39	-2.69	-2.78	-1.76
70	-2.53	-1.95	-1.18	-1.50
100	-0.25	-2.26	0.35	0.37

Table G.6: Results of 2-tailed t-tests for the seconds of processor time used by the agent with constant goal priorities.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-8.66	-7.49	-6.64	-6.17
40	-8.77	-15.1	-12.5	-16.02
70	-10.76	-9.8	-13.3	-15.95
100	-10.08	-10.54	-9.47	-11.27

Table G.7: Results of 2-tailed t-tests for the scores of the agent with constant goal priorities and all goals always present.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-8.63	-14.23	-12.5	-16.43
40	-1.38	-0.64	-2.16	-1.98
70	0.68	-0.47	-3.12	-2.06
100	-2.03	-3.39	-1.66	-3.05

Table G.8: Results of 2-tailed t-tests for the seconds of processor time used by the agent with constant goal priorities and all goals always present.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-8.77	-2.36	-4.93	-3.88
40	-13.46	-19.07	-12.00	-15.69
70	-15.03	-9.42	-8.50	-8.11
100	-15.01	-6.58	-2.61	-6.75

Table G.9: Results of 2-tailed t-tests for the scores of the agent with preferred properties deleted.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-51.23	-57.77	-53.51	-37.42
40	-42.93	-47.27	-34.56	-29.21
70	-25.90	-16.48	-11.75	-10.28
100	-23.01	-5.51	-2.60	-6.19

Table G.10: Results of 2-tailed t-tests for the scores of the agent with preferred properties changed to required properties.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-10.21	-7.97	-12.26	-9.00
40	-8.80	-16.89	-11.33	-12.12
70	-7.84	-4.70	-5.64	-4.51
100	-7.17	-4.40	-0.55	-5.52

Table G.11: Results of 2-tailed t-tests for the scores of the agent with no divisible resources.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-7.35	-8.57	-11.54	-7.72
40	-6.39	-13.00	-11.63	-12.26
70	-8.99	-3.47	-5.29	-3.30
100	-2.89	-3.65	0.46	-3.23

Table G.12: Results of 2-tailed t-tests for the scores of the agent with no value ranges.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-8.95	-7.00	-10.16	-5.44
40	-11.14	-17.27	-10.62	-11.36
70	-8.92	-6.12	-5.64	-2.47
100	-4.32	-3.71	-1.15	-5.01

Table G.13: Results of 2-tailed t-tests for the scores of the agent with no numerical features.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-2.34	-2.15	-2.08	-1.01
40	0.87	-2.18	2.51	-0.27
70	4.06	3.81	6.36	4.50
100	5.86	3.90	24.32	5.13

Table G.14: Results of 2-tailed t-tests for the scores of the agent with no exclusive bindings.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-2.56	-0.57	-2.05	0.25
40	0.44	-2.44	-0.05	-1.47
70	2.87	1.24	-1.75	-0.30
100	2.50	0.41	1.74	-0.87

Table G.15: Results of 2-tailed t-tests for the scores of the agent with no persistent bindings.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	6.06	6.48	4.58	4.64
40	3.97	4.12	6.84	4.14
70	3.59	3.40	3.68	6.33
100	2.49	0.10	2.85	4.23

Table G.16: Results of 2-tailed t-tests for the seconds of processor time used by the agent with no persistent bindings.

Density	**N = 10**	**N = 40**	**N = 70**	**N = 100**
10	-5.41	-3.5	-2.40	1.33
40	2.17	-2.47	0.26	-2.09
70	0.97	-1.26	-0.55	-0.21
100	2.60	0.03	1.97	-1.81

Table G.17: Results of 2-tailed t-tests for the scores of the agent with parallel execution of actions disabled.

References

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 268–272.

America's Army (2004). America's army - special forces - homepage. Webpage. Retrieved 28 August 2004 from http://www.americasarmy.com.

Arkin, R. C. (1998). *Behavior-Based Robotics*. MIT Press.

Bates, J. (1992a). The Nature of Character in Interactive Worlds and the Oz Project. Technical Report CMU-CS-92-200, Carnegie Mellon University.

Bates, J. (1992b). Virtual Reality, Art and Entertainment. *Presence: The Journal of Teleoperators and Virtual Environments*, 1(1):133–138.

Beaudoin, L. (1994). *Goal Processing in Autonomous Agents*. PhD thesis, University of Birmingham.

Benson, S. (1996). *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Stanford University.

Benson, S. and Nilsson, N. (1995). Reacting, planning and learning in an autonomous agent. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence*, volume 14, pages 29–64. The Clarendon Press.

Bererton, C. (2004). State Estimation for Game AI using Particle Filters. In *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop, AAAI Technical Report WS-04-04*. AAAI Press.

Blumberg, B. (1994). Action-selection in Hamsterdam: lessons from ethology. In *Proceedings of the third international conference on Simulation of Adaptive Behavior : From Animals to Animats 3*, pages 108–117. MIT Press.

Blumberg, B. (1997). *Old Tricks, New Dogs: Ethology and Interactive Creatures.* PhD thesis, MIT.

Blumberg, B. et al. (1999). (void*): A Cast of Characters. In *Abstracts and Applications, SIGGRAPH '99*. ACM Press.

Blumberg, B., Todd, P., and Maes, P. (1996). No Bad Dogs: Ethological Lessons for Learning in Hamsterdam. In *From Animals to Animats, Proceedings of the Fourth International Conference on the Simulation of Adaptive Behavior*. MIT Press.

Blumberg, B., Tomlinson, B., and Downie, M. (2001). Multiple Conceptions of Character-Based Interactive Installations. In *Computer Graphics International*, pages 5–11.

Brenner, M. (2001). A Formal Model for Planning with Time and Resources in Concurrent Domains. Workshop on Planning with Resources, IJCAI'01.

Brooks, R. A. (1985). A Robust Layered Control System for a Mobile Robot. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory. A.I. Memo 864.

Brooks, R. A. (1990). Elephants Don't Play Chess. *Robotics and Autonomous Systems.*

Brooks, R. A. (1991). Intelligence Without Representation. *Artificial Intelligence Journal*, 47:139–159.

Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.* PhD thesis, MIT, Department of EECS, Cambridge, MA. AI Technical Report 2001-003.

Cavazza, M., Charles, F., and Mead, S. J. (2002). Sex, Lies, and Video Games: an Interactive Storytelling Prototype. In *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium, AAAI Technical Report SS-02-01.* AAAI Press.

DePristo, M. and Zubek, R. (2001). being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment.*

Drabble, B. and Tate, A. (1994). The Use of Optimistic and Pessimistic Resource Profiles to Inform Search in an Activity Based Planner. In Proceedings of the Second International Conference on AI Planning Systems (AIPS-94).

Fikes, R. and Nilsson, N. (1971). STRIPS: a new approach to the application of theorem proving. *Artificial Intelligence*, 2:189–208.

Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University.

Freed, M., Bear, T., Goldman, H., Hyatt, G., Reber, P., and Tauber, J. (2000). Towards More Human-like Computer Opponents. In *AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment, March 2000, AAAI Technical Report SS-00-02.*

Funge, J. D. (1999). *AI for Games and Animation.* A K Peters.

Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (1998). The Belief-Desire-Intention Model of Agency. Proceedings of the 5th International Workshop on Agent Theories, Architectures and Languages (ATAL-98).

Hawes, N. (2000). Real-Time Goal Orientated Behaviour for Computer Game Agents. In *Proceedings of Game-ON 2000, 1st International Conference on Intelligent Games and Simulation*, pages 71–75.

Hawes, N. (2003). *Anytime Deliberation for Computer Game Agents.* PhD thesis, University of Birmingham, School of Computer Science.

Horswill, I. D. and Zubek, R. (1999). Robot Architectures for Believable Game Agents. In *Proceedings of the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games, AAAI Technical Report SS-99-02.*

Humphrys, M. (1996). Action Selection methods using Reinforcement Learning. In et al., P. M., editor, *From Animals to Animats 4: Proceedings of the Fourth Inter-*

national Conference on Simulation of Adaptive Behavior (SAB-96), pages 135–144, Massachusetts, USA.

Johnson, T. R. (1997). Control in Act-R and Soar. In Shafto, M. and Langley, P., editors, *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, pages 343–348. Lawrence Erlbaum Associates.

Jones, R., Laird, J., Tambe, M., and Rosenbloom, P. (1994). Generating Behavior in Response to Interacting Goals. In *Proceedings of the 4th Conference on Computer Generated Forces and Behavioral Representation*.

Kaebling, L. P. and Rosenschein, S. J. (1994). Action and Planning in Embedded Agents. In Maes, P., editor, *Designing Autonomous Agents*. MIT Press.

Kaminka, G. A., Veloso, M. M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S. (2002). Gamebots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45.

Khoo, A., Dunham, G., Trienens, N., and Sood, S. (2002). Efficient, Realistic NPC Control Systems using Behaviour-Based Techniques. In *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium, AAAI Technical Report SS-02-01*. AAAI Press.

Kline, C. and Blumberg, B. (1999). The Art and Science of Synthetic Character Design. In *Proceedings of the AISB 1999 Symposium on AI and Creativity in Entertainment and Visual Art*.

Koehler, J. (1998). Planning Under Resource Constraints. In *Proceedings of the 19th European Conference on Artificial Intelligence*. John Wiley & Sons, Ltd.

Kvarnstrom, J., Doherty, P., and Haslum, P. (2000). Extending TALplanner with Concurrency and Resources. In Proceedings of ECAI-00, Berlin, Germany.

Laird, J. (2000). It Knows What You're Going To Do: Adding Anticipation to a Quakebot. In *AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment, March 2000, AAAI Technical Report SS-00-02.*

Laird, J. and Duchi, J. (2000). Creating Human-like Sythetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. In *AAAI Fall Symposium Seris: Simulating Human Agents.*

Laird, J. E. and Congdon, C. B. (2004). *Soar User's Manual, Version 8.5*, 1 edition.

Laird, J. E., Congdon, C. B., Altman, E., and Doorenbos, R. (1993). *Soar User's Manual, Version 6*, 1 edition.

Laird, J. E. and Jones, R. M. (1998). Building Advanced Autonomous AI Systems for Large Scale Real Time Simulations. Proceedings of the Computer Games Developers Conference.

Laird, J. E. and van Lent, M. (2001). Human-Level AI's Killer Application: Interactive Computer games. *AI Magazine*, 22(2):15–26.

Lehman, J. F., Laird, J. E., and Rosenbloom, P. (1998). A gentle introduction to Soar, an architecture for human cognition. In Sternberg, S. and Scarborough, D., editors, *Invitation to Cognitive Science*, volume 4. MIT Press.

Leonard, A. (1998). *Bots: The Origin of New Species.* Penguin Books.

Lewis, R. (2001). Cognitive Theory, Soar. In *International Encyclopedia of the Social and Behavioral Sciences.* Elsevier Science, Amsterdam.

Logan, B., Fraser, M., Fielding, D., Benford, S., Greenhalgh, C., and Herrero, P. (2002). Keeping in Touch: Agents Reporting from Collaborative Virtual Environments. In *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium, AAAI Technical Report SS-02-01.* AAAI Press.

Loyall, A. B. (1997). *Believable Agents: Building Interactive Personalities.* PhD thesis, Carnegie Mellon University. CMU Technical Report CMU-CS-97-123.

Maes, P. (1994). Situated Agents Can Have Goals. In Maes, P., editor, *Designing Autonomous Agents.* MIT Press.

Magerko, B. and Laird, J. E. (2004). Mediating the Tension between Plot and Interaction. In *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop, AAAI Technical Report WS-04-04.* AAAI Press.

Mateas, M. and Stern, A. (2002). A Behaviour Language for Story-based Believable Agents. In *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium, AAAI Technical Report SS-02-01.* AAAI Press.

Minsky, M. (1988). *The Society of Mind.* Pan Books.

Moore, D. S. and McCabe, G. P. (1993). *Introduction to the Practice of Statistics.* W. H. Freeman and Company, 2nd edition.

Munoz-Avila, H. and Fischer, T. (2004). Strategic Planning for Unreal Tournament©Bots. In *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop, AAAI Technical Report WS-04-04.* AAAI Press.

Nareyek, A. (1998). A Planning Model for Agents in Dynamic and Uncertain RealTime Environments. In *Proceedings of the AIPS-98 Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*, pages 7–14.

Nareyek, A. (2000). Beyond the Plan-Length Criterion. In *Local Search for Planning and Scheduling*, volume 2148 of *Lecture Notes in Artificial Intelligence*, pages 55–78. Springer.

Nilsson, N. (1994). Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research*, 1:139–158.

Norling, E. (2003). Capturing the quake player: Using a BDI Agent to Model Human Behaviour. In *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 03).* ACM Press.

Norman, T. J. and Long, D. (1995). Goal Creation in Motivated Agents. In Wooldridge, M. J. and Jennings, N. R., editors, *Intelligent Agents: Proceedings of the 1994 workshop on Agent Theories, Architectures and Languages (ATAL-94)*, volume 890 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag.

Norman, T. J. and Long, D. (1996). Alarms: An implementation of motivated agency. In Wooldridge, M., Muller, J., and Tambe, M., editors, *Intelligent Agents: Theories, Architectures and Languages (ATAL-96)*, volume 1037 of *Lecture Notes in Artificial Intelligence*, pages 219–234. Springer-Verlag.

O'Brien, J. (2002). A Flexible Goal-Based Planning Architecture. In Rabin, S., editor, *AI Game Programming Wisdom*, pages 375–383. Charles River Media.

Orkin, J. (2004). Symbolic Representation of Game World State: Toward Real-Time Planning in Games. In *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop, AAAI Technical Report WS-04-04*. AAAI Press.

Pollack, M. and Ringuette, M. (1990). Introducing the Tileworld: experimentally evaluating agent architectures. In Dietterich, T. and Swartout, W., editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189, Menlo Park, CA. AAAI Press.

Rao, A. S. and Georgeff, M. P. (1995). BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco.

Reilly, W. S. N. (1996). *Believable Social and Emotional Agents*. PhD thesis, Carnegie Mellon University. CMU Technical Report CMU-CS-96-138.

Rickel, J. and Johnson, W. (1998). Animated agents for procedural training in virtual reality: Perception, cognition, and motor control. In *Applied Artificial Intelligence*.

Robocup (2004). Robocup: Small-Size Robot League Regulations. Webpage. Retrieved 20 September 2004 from http://www.robocup.org/regulations/42.html.

Sengers, P. (1998). *Anti-Boxology: Agent Design in Cultural Context.* PhD thesis, Carnegie Mellon University.

Sloman, A. (2002). Architecture-Based Conceptions of Mind. In Gardenfors, P., Kijania-Placek, K., and Wolenski, J., editors, *In the Scope of Logic, Methodology and Philosophy of Science (Vol II)*, Synthese Library Vol. 316. Kluwer.

Sloman, A. and Chrisley, R. (2003). Virtual Machines and Conciousness. *Journal of Consciousness Studies*, 10(4-5).

Sloman, A. and Logan, B. (1999). Building Cognitively Rich Agents Using the SIM_AGENT Toolkit. *Communications of the ACM*, 42(3):71–77.

Sloman, A. and Logan, B. (2000). Evolvable Architectures for Human-Like Minds. In Hatano, G., editor, *Affective Minds.* Elsevier.

Soar (2004). Soar: Home. Webpage. Retrieved on 22 September from http://sitemaker.umich.edu/soar.

Thorisson, K. R. (1996). *Communicative Humanoids: A Computational Model of Psychosocial Dialogue Skills.* PhD thesis, MIT.

Traum, D., Rickel, J., Gratch, J., and Marsella, S. (2003). Negotiation over Tasks in Hybrid Human-Agent Teams for Simulation-Based Training. In *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 03).* ACM Press.

Tyrrell, T. (1992). Defining the Action Selection Problem. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Society*. Lawrence Erlbaum Associates.

Tyrrell, T. (1993). The use of hierarchies for action selection. In *From Animals to Animats 2*, pages 138–147. MIT Press.

van Lent, M., Carpenter, P., McAlinden, R., and Tan, P. G. (2004). A Tactical and Strategic AI Interface for Real-Time Strategy Games. In *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop, AAAI Technical Report WS-04-04*. AAAI Press.

van Lent, M., Laird, J., Buckman, J., Hartford, J., Houchard, S., Steinkraus, K., and Tedrake, R. (1999). Intelligent Agents in Computer Games. In *Proceedings of the National Conference on Artificial Intelligence*, pages 929–930.

Vu, T., Go, J., Kaminka, G., Veloso, M., and Browning, B. (2003). MONAD: A Flexible Architecture for Multi-Agent Control. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems.*

Wetzel, B. (2004). Step One: Document the Problem. In *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop, AAAI Technical Report WS-04-04*. AAAI Press.

Wright, I. (1997). *Emotional Agents*. PhD thesis, University of Birmingham.

Yoon, S., Blumberg, B., and Schneider, G. (2000). Motivation Driven Learning for Interactive Synthetic Characters. In *Proceedings of Autonomous Agents (Agents 2000)*, pages 365–372.