Managing Goals and Resources in Dynamic Environments

Elizabeth Gordon, Brian Logan

School of Computer Science
University of Nottingham UK
{esg|bsl}@cs.nott.ac.uk

Abstract

A key problem for agents is responding in a timely and appropriate way to multiple, often conflicting goals in a complex, dynamic environment. In this paper we propose a novel goal processing architecture which allows an agent to arbitrate between multiple conflicting goals. Building on the teleo-reactive programming framework originally developed in robotics, we introduce the notion of a *resource* which represents a condition which must be true for the safe concurrent execution of a durative action. We briefly outline a goal arbitration architecture for teleo-reactive programs with resources which allows an agent to respond flexibly to multiple competing goals with conflicting resource requirements.

Keywords: Agent Architecture, Goals, Resources, Action Selection.

INTRODUCTION

One of the defining characteristics of an autonomous agent is its ability to generate its own goals in response to changes in its environment. At any given time, such an agent will typically have several goals; for example, a package delivery robot may have a goal to deliver a package to a particular office (triggered by a user request), a goal to keep its battery charged (innate), and a goal to avoid colliding with the person who has just stepped out of their office (autonomously generated). A key problem with goal-based architectures is *goal arbitration*, i.e.which goal or goals to work on next. Ideally the agent should work to achieve as many goals as possible, while respecting any priority ordering over goals and the limitations imposed by its environment and effectors. The agent should be able to respond to opportunities and threats as they arise, while continuing to work towards its existing goals (to the extent to which this is possible). However, many existing goal-based agent architectures only allow an agent to work towards a single goal at a time. In this paper, we extend the teleo-reactive framework described in (Benson and Nilsson, 1995) to allow an agent to work towards multiple goals simultaneously. We introduce the notion of a *resource* representing a condition which must be true for the safe concurrent execution of a teleo-reactive program, and present an algorithm for goal arbitration between teleo-reactive programs with resources.

The examples in this chapter and the implementation we describe involve agents which act as characters in a computer game. The domain of computer games is becoming increasingly popular as a research platform for artificial intelligence (see, for example, (Laird and Duchi, 2000; DePristo and Zubek, 2001; Hawes, 2000)). Developing an agent for a computer game is essentially the same as any simulation-based approach to AI. Modern computer games are essentially simulated worlds. While games are simpler than the real world, they provide a range of locations, situations, objects, characters and actions which present game characters with a complex, dynamic environment. Most

computer games are real-time, and the environment can be changed by a human player or other characters. Games therefore present a challenging agent design problem, forcing us to confront issues of real-time action selection in pursuit of multiple goals.

Although our approach has been developed in the context of games, it is completely general and can be applied to any complex dynamic environment or resource-dependent task. It is situated within a body of work which takes agent architectures as central, for example Byrson's work on modular agent designs (2001) and Wright's work on cognitive architectures for emotional agents (1997), both of which are discussed below.

The remainder of the this chapter is organised as follows. In the next section we present a brief introduction to the teleo-reactive framework as described in (Nilsson, 1994; Benson and Nilsson, 1995; Benson, 1996) and highlight some of the problems which form the motivation for our work. We then introduce the notion of a resource, which represents a condition which must be true for the safe concurrent execution of a durative action, and sketch a new goal processing architecture for agents, GRUE, which extends teleo-reactive programs with resources. We briefly describe the main components of the GRUE architecture and present an algorithm for goal arbitration using resources. We conclude with a brief discussion of related work and outline directions for future work.

TELEO-REACTIVE PROGRAMS

The teleo-reactive framework (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 continous feedback necessary to operate effectively (Nilsson, 1994). In this section we discuss the teleo-reactive framework. The next section discusses our own approach which builds on the teleo-reactive framework.

The teleo-reactive (TR) framework comprises a library of plans, an arbitrator, a planner, and a learning system. The plans in the TR framework 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*. Correspondingly, we will refer to conditions that are evaluated continuously as *durative conditions*. 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 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). Plans are either constructed at run time by the planner or selected from a set of existing plans in the agent's plan library.

A TR agent attempts to achieve a set of goals which may be pre-specified by the agent designer or provided by a human operator. The *arbitrator* allows a TR agent to work on several goals at once, by determining which plan should be allowed to perform an action

at each cycle (Benson and Nilsson, 1995; Benson, 1996). Plans are chosen using the concept of stable nodes. A *stable node* is a point in a plan at which execution of the program can safely be suspended. That is, the work done up to that point in the plan is stable with respect to the other plans that are running. A condition is stable if running the other plans will not cause the condition to become false. So, for example, a plan 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 plans start running using STRIPS-style delete lists, and only needs to be re-compiled when the set of plans changes.

During each execution cycle, the arbitrator runs the plan with the best expected reward/time ratio. The reward is the reward 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. Stable nodes are safe places to stop programs, so the arbitrator runs each program until it reaches one; it can then switch to another program 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.

Example: Pacman

As an illustration of the TR framework, we present a collection of teleo-reactive programs that might be used to play the game Pacman. We remind the reader that in the game, the player controls a yellow character (Pacman) who moves around a maze eating dots in order to earn points. The maze contains hazards in the form of multi-coloured ghosts. The player starts with three lives, and the game is over when all the lives are gone. If a ghost catches Pacman, the player loses a life and Pacman is placed in a safe position. There are also special dots called power pills which Pacman can eat to make the ghosts vulnerable (and blue coloured). While the ghosts are blue, Pacman 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 Pacman is always eating—it is not possible for Pacman to move over an edible object without eating it. Our Pacman agent will play Pacman 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 Pacman 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 below:

```
PROGRAM: Eat Dots
R1 IF there are no dots
   THEN null
R2 IF there are dots
   THEN move towards a dot

PROGRAM: Escape From Ghosts
R1 IF no ghost is less than 10 units away
      OR all ghosts are blue
   THEN null
R2 IF there is a ghost within 10 units
      AND the ghost is not blue
```

```
   THEN move away from the ghost

PROGRAM: Eat Blue Ghosts
R1 IF there are no blue ghosts
      AND there are no power pills
   THEN null
R2 IF there is a blue ghost
   THEN move towards the ghost
R3 IF there are no blue ghosts
      AND there is a power pill
   THEN move towards the power pill

PROGRAM: Eat Fruit
R1 IF there is no fruit
   THEN null
R2 IF there is a fruit
   THEN move towards the fruit
```

At any given moment, the Pacman agent is running one of the above programs, 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, the arbitrator checks to see if the highest condition which is currently true is stable with respect to other TR programs, and if so switches to the program with the best reward/time ratio (which may be the *Eat Blue Ghosts* program). However, Pacman 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 Pacman to move might cause Pacman to eat the fruit. This is due to the nature of the game—there is no eat command, Pacman simply eats anything it runs into.

Limitations of Teleo-Reactive Programs

Teleo-reactive programs have a number of advantages for controlling agents in dynamic environments like Pacman. 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.

However, the standard teleo-reactive architecture has a number of limitations. One problem is with the approach to goal arbitration which relies on the notion of stable nodes. The presence or absence of stable nodes is determined by the set of actions possible in the environment. In some environments, it is difficult to find stable nodes,

making it impossible to use them to allow pseudo-parallel execution of TRPs. 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 our Pacman 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 Pacman 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.

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. As an example, we built a simple agent that plays the game Unreal Tournament. Unreal Tournament has several game modes; our agent plays one in which each player has a flag and can gain points by stealing their opponent's flag. The game is combat based, so players can use weapons to attack each other. The player's health is measured on a scale of 1 to 100, and there are health items which restore about 20 points. Our agent uses a basic strategy of always guarding its own flag. It has goals for regaining health, attacking an opponent, and returning to its flag. Each goal has a corresponding teleo-reactive program, but only one program can run during each execution cycle. The goal for regaining health has the highest reward, which means that if the agent's health is low, only the program for regaining health will run. The problem is that once a health item has been used it takes some time for another one to appear. The regain health goal is triggered whenever the agent's health is below a threshold. If the agent's health is so low that it is below the threshold even after using a health item, it will simply wait for the health item to reappear. In effect, it stands still in the middle of a room, making itself an easy target. This is not effective or realistic behaviour. More generally, if two programs are the same length 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 not be favoured during arbitration.

Using stable nodes requires a list, for each action, of conditions that are falsified by that action. To compute the set of stable nodes we therefore need to be able to predict the effects of actions in the environment. Since the effects of an action may depend on the current environment, this in turn requires that we either have a complete domain model, or the agent programmer has the long and tedious task of listing every possible action, in every possible situation, with every possible consequence.

Even in those cases where a domain model is available, the need to compute stable nodes places restrictions on the syntax of rules. In particular, we cannot have disjunctive conditions in rules. For example, given two rules in two different programs:

```
Program: A
Ri IF there is a guard present
      AND we have a ruby OR we have an emerald
   THEN bribe the guard

Program: B
Rj IF we have a ruby
      AND there is a unicorn present
   THEN give the ruby to the unicorn
```

Assuming that both a guard and a unicorn are present, running rule in *Program B* will make the condition of 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.

More generally, the teleo-reactive architecture described in (Benson and Nilsson, 1995) and (Benson, 1996) is not completely autonomous. Goals and the rewards for achieving them are either predefined by the developer or are given by the user. The former is inflexible and the latter is often inappropriate; for example, in a computer game where agents must be truly autonomous, the human players should be playing the game, not providing input to their opponent.

GRUE: A NEW ARCHITECTURE

We have therefore developed a new teleo-reactive architecture, GRUE (Goal and Resource Using architecturE), 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-routining, 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 1). The system runs in cycles. During each cycle, information from the environment is processed by the world interface and then placed in memory. Goal generators are triggered by the information in memory, then associated with programs by the program selector, and then executed by the arbitrator.

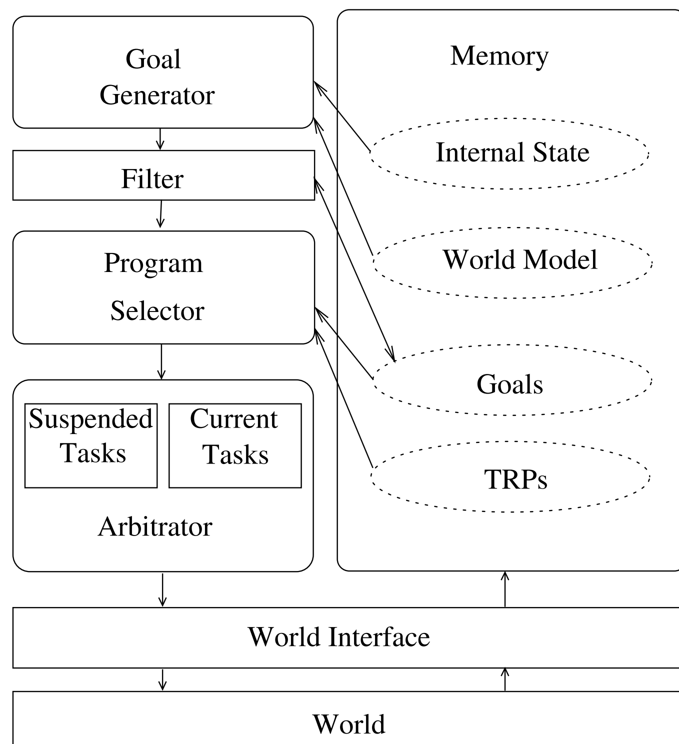

Figure 1: GRUE

The agent operates in a world which could be a game engine, the physical world (for a robot), or any other environment in which the agent operates. The world interface takes information from the environment, in this case a game engine, does any necessary pre-processing, for example, 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 1 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. 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. The agent designer can choose not to run the goal generators at every cycle, trading reduced execution time for an increase in the time it takes for a character to respond to events in the environment.[2] 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 arbitrator manages a list of current tasks, and decides which task(s) to run at the current cycle. 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.

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.

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 pair, consisting of a property name and a 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 (for example, 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 a one or more property name–property value pairs. 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 and/or divisible. A property which is *multi-valued* can have more than one value for the same resource.[3] For example, more than one category may be listed for the TYPE field, for example, a pickaxe might be represented as

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

which indicates that the resource Pickaxe102 is a pickaxe and also a weapon.

In general, a resource can only be used by a single task at a time. However 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 a 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.

Resource Variables

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.

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 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 information 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 by a rule, or where a rule's actions require sole access to the resource, the flag should have the value "EXCLUSIVE". Note that mutual exclusion applies between programs—two exclusive variables in the same rule can bind to the same 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 (for example, 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 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.

Numerical values are 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 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 -> :# ]) () )
```

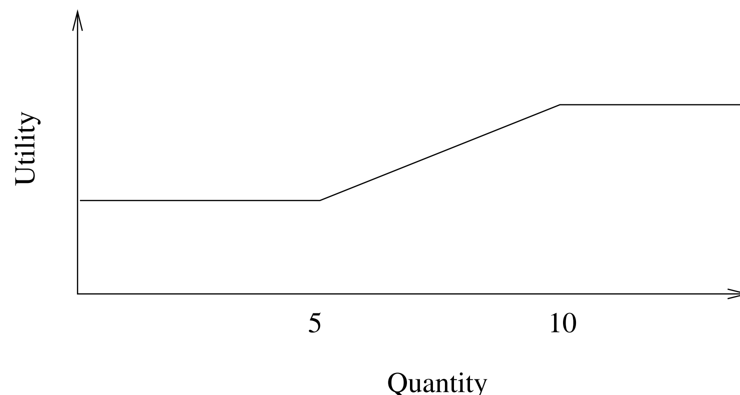

Figure 2: Value range #:5–>10:#

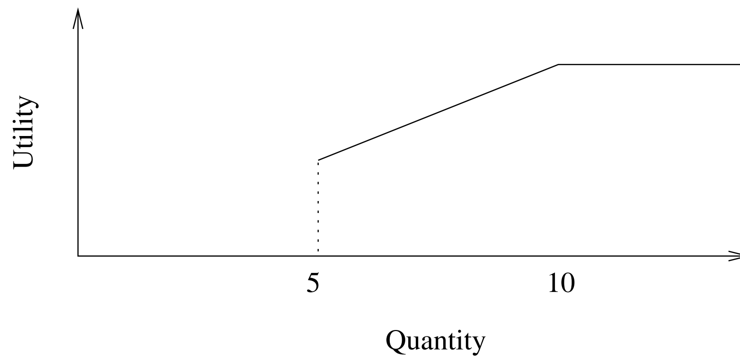

Figure 3: Value range #|5–>10:#

Figure 2 illustrates the relative utility of property values in the soft-bounded range #:5–>10:#. Contrast this to Figure 3, which shows a range with one firm bound and one soft bound.

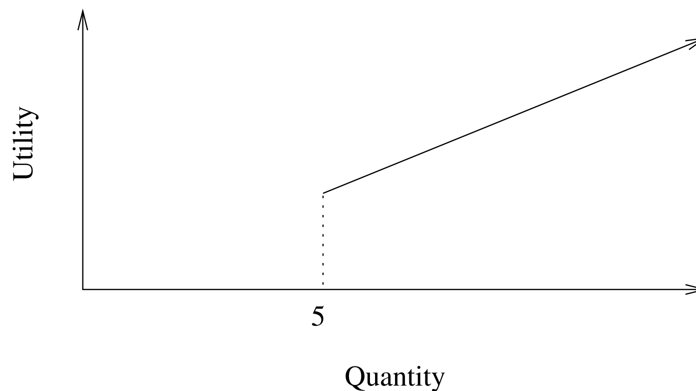

Figure 4: #|5–>:#

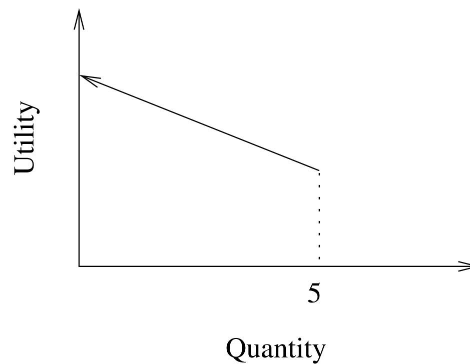

Figure 5: #:<–5|#

Figure 4 specifies a minimum of 5 and an upper limit of $+\infty$. Figure 5 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 then treated as if they did not match.

Once a variable has been bound to a resource, we can use the property function to retrieve additional information. If the resource variable

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

is bound to the resource

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

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

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

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

Pacman agent as follows:

```
(GetAwayFromGhostsGoal 90 ACHIEVEMENT EscapeFromGhosts)

(EatBlueGhostsGoal 50 ACHIEVEMENT EatGhosts)

(EatFruitGoal 50 ACHIEVEMENT EatFruit)

(EatDotsGoal 20 ACHIEVEMENT EatDots)
```

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 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.[4] 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 those resources necessary to maintain the condition. Maintenance goals should have a appropriate priority so they can be used to prevent the character from disposing of necessary items or "forgetting" to maintain a crucial condition.

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

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

Programs

GRUE Programs are 'standard' teleo-reactive programs as defined in (Benson, 1996) extended with resource variables. A *GRUE TRP* is a list consisting of an identifier (a string), a list of input parameters, and one or more rules. A *rule* is a list consisting of an identifier (a string), a condition and a list of actions. The condition of a rule consists of two conjoined parts: a non-empty set of implicitly conjoined resource variables, and a boolean expression of *property tests* of the form $f(x_1, ..., x_n)\ \Theta\ g(y_1, ..., y_m)$, where f,g are functions of the resource variable identifiers appearing in the condition and Θ is a comparison operator, for example, ==, >, and <, etc.. The functions include the `property` function as defined above, as well as user-defined functions implemented directly in the underlying implementation language. User-defined functions can use the `property` function to extract property values from the resource identifier. Resource variables may be negated to express the requirement that no matching resource exists. The resource variable's exclusive access flag is irrelevant in this case and its value has no effect.

The rules are evaluated in order, with the first rule whose condition is true proposing an

action to execute. A condition evaluates to true if both the resource variables and logical expression evaluate to true. A non-negated resource variable evaluates to true when it is successfully bound to a resource. A negated resource variable evaluates to true if no binding exists. (Binding of resource variables is discussed in more detail below.) Logical expressions containing user-defined functions evaluate in the usual way. Logical expressions may only test information derived from resource variables. For example, a rule which needs to compare the locations of two objects could contain two resource variables, one to bind each object (represented as resources), and a user-defined predicate to compare the location information extracted from each binding using the property function. If a disjunction of resource variables is required, it should be written as two separate rules.[5] 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, for example, if the agent's sensors return partial information about the environment or the world changes between observations.

As an example, here is GRUE TRP for our Pacman agent which achieves the *Escape From Ghosts* goal:

```
(EscapeFromGhosts
 (Rule0 NOT(?Ghost1 SHARED
            ([TYPE ghost] [DIST #|1 <- 10|#])
            ()) OR
        NOT(?Ghost1 SHARED
            ([TYPE ghost] [BLUE false])
            ())
  =>
  (null))

 (Rule1 (?Ghost1 SHARED
         ([TYPE ghost] [BLUE false] [DIST #|1 <- 10 |#])
         ())
  =>
  (move-away-from ?Ghost1)))
```

When a rule condition evaluates to true, the rules 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 program. This allows the agent designer to write generic programs for common sub-tasks which can then be used in multiple places. As an example, here is a very simple program which moves an agent to a location or object.

```
(GOTO (?Target)
 (Rule0 (?AgenttLocation SHARED ([Value *]) ()) AND
        (property(?AgentLocation, Value) ==
         property(?Target, Value))
  =>
  (null))
```

```
 (Rule1 (?AgentLocation SHARED ([Value *]) ())  AND
        NOT(property(?AgentLocation, Value) ==
            property(?Target, Value))
  =>
  (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:

```
(Rule4 (?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.

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
  (Rule0 NOT(?Ghost1 SHARED
             ([TYPE ghost] [DIST #|1 <- 10|#])
             ()) OR
         NOT(?Ghost1 SHARED
             ([TYPE ghost] [BLUE false])
             ())
   =>
   (null))

  (Rule2 (?Ghost1 SHARED
          ([TYPE ghost] [BLUE false] [DIST #|1 <- 10|#])
          ())
   =>
```

```
    (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 which cannot run because another (higher priority) task has preempted one or more resources are flagged as *unrunnable* and stay in the arbitrator until they become runnable. Tasks whose programs have finished executing, or whose programs cannot run because the necessary resources are not available, are removed from the arbitrator.

Goal Arbitration

It is the job of the arbitrator to allocate resources to the tasks, giving priority to higher priority 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. The list of proposed actions is examined for conflicts before the actions are executed, and conflicts are resolved in favour 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 check whether the task should be removed from the arbitrator. This requires examining resources that have already been bound, in order to determine whether any of them would allow the task to run. If not, the task is removed.
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 there is a consistent binding of its resource variables and the associated boolean expression of predicates defined on resource property values evaluates to true. 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. Unlike other rule-based languages, 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. 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.

We require that resource binding is consistent from cycle to cycle: a resource variable bound during one execution cycle remains bound to the same resource until the condition of a prior rule in the TRP matches or the resource is no longer available.

Note that no attempt is made to maximise 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. Conversely, no attempt is made to limit the number of tasks the agent will run in parallel. If necessary, the arbitrator can be limited to processing only a small set of tasks. Once the task limit has been reached, the rest of the tasks are simply not run.

Tasks which are runnable in principle, i.e.could run if a higher priority task had not bound some resource(s) are not removed from the arbitrator, and simply wait until they have the necessary resources to run. Tasks are removed from the arbitrator in 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 property is deleted (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.propose a null action indicating that the task has been completed) is removed from the arbitrator. However, 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.

RELATED WORK

While our work has been developed in the context of computer games, it has strong similarities to work done in other problem domains. It extends the telo-reactive programs of Benson and Nilsson (Nilsson, 1994; Benson and Nilsson, 1995), which were developed for use in robotics. We chose TRPs as a starting point in order to take advantage of its mechanism for handling dynamic environments, as this is a key problem in both real and simulated worlds. Our work also has similarities to the cognitive architecture developed by Wright (1997), and to the BOD approach developed by Bryson (2001). In this section we first briefly outline the differences between our work and the teleo-reactive framework proposed by Benson and Nilsson. We then discuss other similar approaches such as Bryson's and Wright's.

The main difference in our use of TRPs is that we have written them in terms of resources. This gives us two advantages over the approach described in (Benson and Nilsson, 1995). First, disjunctive conditions can be used to represent situations where

there are several alternate ways of achieving the same goal. As discussed above, there are potential problems when using stable nodes with disjunctive conditions.[6] GRUE eliminates these problems by not using stable nodes and by disallowing disjunctions involving resource variables. Instead, resource variables implicitly encode disjunctions through the use of preferences. The second advantage of GRUE is that using resources allows us to use the same basic programs for several agents, but differentiate them by giving them preferences for different types of items. This property is particularly useful in entertainment applications like games.

GRUE also differs from (Benson and Nilsson, 1995; Benson, 1996) in that the arbitrator does not use any idea of reward or time estimates. Instead, goals in GRUE are given a priority value by the goal generator. A disadvantage of using a reward/time ratio is that the programmer cannot force the agent to work exclusively on a high priority goal. If a less important goal can be completed in a sufficiently short time, it will always be completed. By contrast, GRUE's arbitrator will execute the appropriate actions simultaneously if possible and otherwise focus on the highest priority goal. GRUE can also be made to exhibit similar behaviour to that described in (Benson and Nilsson, 1995) by creating goal generators which use a reward/time ratio to compute the priority. We do not have a human user to provide rewards, but changes to the environment and the agent state could be regarded as rewards (and represented as such by the world interface).

The architecture which is probably most similar to ours is Wright's MINDER1 (Wright, 1997). MINDER1 uses a library of teleo-reactive programs and generates and manages motives, which seem to be the functional equivalent of goals. Each motive contains a condition to be made true, an insistence value which represents the importance of the goal, and a flag indicating whether or not the motive has passed through an attention filter. The filter uses a simple threshold function, allowing motives through when their insistence is above the threshold. MINDER1 is based on a three-layer architecture developed as part of the Cognition & Affect project by Sloman and Beaudoin (Wright et al., 1996). As such, it includes both a management layer and a meta-management layer. The management layer can suspend tasks, schedule tasks, and expand a motive into a plan. The meta-management layer is responsible for monitoring the management layer and making adjustments as necessary. To make this clearer, we will explicitly compare the functions of Wright's architecture to those of GRUE. First, Wright includes an interface to the agent's sensors, and a belief maintenance system which is equivalent to the world model. Next, there is a collection of "generactivators" which produce motives. Motives serve the same purpose in Wright's architecture as goals do in ours, and the generactivators are equivalent to our goal generators. Motives that pass the filter are processed by the management layer. The management layer has three tasks. The first is to decide whether or not the motives that have passed the filter should continue being processed by the management layer. The second is to determine which of the motives should be activated. Only one motive is activated at a time. Finally, it expands the motive into a complete plan by retrieving a plan from a plan library. The management layer in MINDER1 incorporates the function of our program selector and some of the functionality of the arbitrator. However, MINDER1 can only execute one plan at a time. The GRUE arbitrator has the ability to run multiple tasks at once, balanced with a mechanism for allocating resources based on dynamic priorities. Finally, MINDER1 includes a meta-management layer. This layer has two functions within the architecture. First, it can adjust the threshold in the filter. Second, it can detect perturbant states in which resources are continually diverted to particular, less useful goals. Wright's purpose in constructing MINDER1 was to investigate emotional states, some of which manifest as perturbances. We have not included a meta-management layer in GRUE; however it

would be possible to change the filter threshold dynamically. We are not attempting to model emotional states, and do not anticipate problems with perturbances.

Bryson's work (Bryson, 2001) is closely related to teleo-reactive programs. Bryson describes an approach to building behaviour-based agents called Behavior-Oriented Design. She discusses both a development process and a modular architecture which includes Basic Reactive Plans as one of the core components. Basic Reactive Plans are identical to a teleo-reactive programs, with the exception that the regression property is not required. She has used BOD systems for a number of applications, including a robot control system, modelling primate behaviour, and characters in an interactive virtual world. One significant difference between Bryson's system and ours is that her agents are controlled by drive collections. A drive collection is just a Basic Reactive Plan which contains a list of tasks the agent might want to do. Because these are similar to teleo-reactive programs, tasks always have the same relative importance. For example, if an agent using a drive collection played Pacman, it might list *Escape from Ghosts*, then *Gain Points*. The goals would always be considered in that order. In contrast, a GRUE agent playing Pacman could make the importance of *Escape from Ghosts* proportional to the distance of the nearest ghost. This means that the *Gain Points* goal might be generated with a higher priority than *Escape from Ghosts*. This allows for greater flexibility in decision making.

Our architecture is not intended to be a cognitive model, but it is worth noting that some capabilities included in GRUE, particularly arbitration between multiple competing goals, is not found in most cognitive models. Both Soar and Act-R use only a single goal hierarchy(Johnson, 1997). Nonetheless, much work in games and and real-time simulations has been done using the Soar architecture (eg. (Laird and Duchi, 2000; Laird, 2000; van Lent et al., 1999)). 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).

DePristo and Zubek (DePristo and Zubek, 2001) describe a hybrid architecture used for an agent in a role-playing game. This type of game typically involves tasks like buying items, and choosing which items should be kept in a limited inventory. The architecture included a deliberative truth maintenance and reasoning component along with a reactive layer capable of handling urgent situations without input from the deliberative layer. The system had difficulty representing quantities like amounts of gold, and because the system was focused on facts there were some problems handling goals. In contrast to DePristo and Zubek's architecture, GRUE is primarily focused on goals and resources rather than facts. Furthermore, we have designed our data structures specifically to allow reasoning about quantities and properties of objects.

CONCLUSIONS AND FUTURE WORK

In this chapter we proposed a novel goal processing architecture which allows an agent to arbitrate between multiple conflicting goals. Our architecture, GRUE, is based on teleo-reactive programs, which are designed to handle changes in the environment gracefully. We have shown that the teleo-reactive architecture described in (Benson and Nilsson, 1995) and (Benson, 1996) has several limitations. In environments which have small number of stable nodes it can become 'trapped' by a single high priority goal, with the result that it is unable to respond to changes in the environment affecting the execution of other active goals. In addition, the original teleo-reactive architecture did not address the problem of goal generation and was incapable of executing multiple actions in parallel. To overcome these limitations we introduced the idea of a *resource* representing a condition which must be true for the safe concurrent execution of a durative action, and

briefly outlined a goal arbitration scheme for teleo-reactive programs with resources which allows an agent to respond flexibly to multiple competing goals with conflicting resource requirements.

We have several ideas about possible future directions for this work. In particular, we expect it will be possible to follow Scott Benson's lead in adding planning and learning capabilities to the architecture (Benson, 1996). We also think there are some interesting possibilities in dynamic filter adjustments and the addition of a purely reactive layer.

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 resource variables with required properties. If the same action causes different effects when used with different resources, then there must be a set of operators, each one with a different list of required properties for its resource variables and the corresponding list of effects.

In (Benson and Nilsson, 1995), the system learned TOPs through trial and error. Again, a similar approach could be employed with GRUE. In addition, if the system has a model of emotions then it could be set up to learn preferred properties as well as required properties. Required properties in this case are those that are necessary for an action to have the desired effect, while preferred properties are those that are most often associated with a positive emotional state. Using this method, the agent could develop preferences that, while inconsequential in and of themselves, would give the agent personality.

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. 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.

Those working in biological modeling may be bothered by the hybrid approach we have taken with GRUE. While the teleo-reactive programs do respond to changes in the environment, it is 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, and the overhead is small enough that we do not expect it to be a problem. However, it would be possible to add a pure reactive layer in the form of an additional ruleset that runs all the time. In order to completely eliminate overhead, it would need to bypass the main portions of GRUE by running in a separate thread or process. Doing this would potentially lead to conflicts between actions, as it would also bypass GRUE's conflict resolution mechanism. This problem could be handled 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.

Notes

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

2. Small increases in the time taken to respond to events may actually result in increased believability (Freed et al, 2000).

3. The resulting resource structure is similar to the augmentations of SOAR's working memory elements (Laird et al, 1993), except that here, values can't be other resources, only constants.

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

5. Some disjunctions can be encapsulated in a single resource variable, through the judicious use of preferred properties.

6. Disjunctive conditions are not discussed in (Nilsson, 1994; Benson & Nilsoon 1995; Benson, 1996). However, both Benson & Nilsson (1995) and Benson (1996) contain examples of disjunctive conditions. It is possible that the problems mentioned here were simply not encountered in the envrionments used in (Benson & Nilsson, 1995) and (Benson, 1996).

Acknowledgements

Elizabeth Gordon is a PhD student supported by Sony Computer Entertainment Europe. Thanks also to Jamie Strachan for assistance in proofreading the chapter.

References

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. The Clarendon Press.

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.

DePristo, M. and Zubek, R. (2001). being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment.*

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, Technical Report SS-00-02*.

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.

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.

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, 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., Congdon, C. B., Altman, E., and Doorenbos, R. (1993). *Soar User's Manual, Version 6*, 1 edition.

Nilsson, N. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158.

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.

Wright, I. (1997). *Emotional Agents*. PhD thesis, University of Birmingham.

Wright, I., Sloman, A., and Beaudoin, L. (1996). Towards a design-based analysis of emotional episodes. *Philosophy Psychiatry and Psychology*, 3(2):101–137.