

A Goal Processing Architecture for Game Agents

Elizabeth Gordon *
School of Computer Science and IT
University of Nottingham
Nottingham NG8 1BB, UK
esg@cs.nott.ac.uk

Brian Logan
School of Computer Science and IT
University of Nottingham
Nottingham NG8 1BB, UK
bsl@cs.nott.ac.uk

10 November 2002

Abstract

Computer games are becoming increasingly popular as a research platform for agents research and applications. A key problem for game 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 for game agents. Building on the teleo-reactive programming framework originally developed in robotics, we introduce the notion of a *resource*, which can be used to gain exclusive access to specific game objects as well as to represent more abstract things such as properties of the agent. We then describe a goal arbitration architecture for teleo-reactive programs with resources. Our architecture allows an agent to respond flexibly to multiple competing goals, and simplifies the development of game agents by facilitating increased code re-use.

1 Introduction

The domain of computer games is becoming increasingly popular as a research platform for artificial intelligence (eg. [10, 4, 5]). 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. Current game AIs are often implemented as hard coded finite state machines or scripts. While this is appropriate for simple behaviours, it frequently results

*This work is supported by Sony Computer Entertainment Europe.

in rigid, inflexible and unbelievable character behaviour, and the resulting AI code can be hard to develop and debug.

In this position paper, we argue that game environments can be effectively handled by an agent architecture which uses an explicit representation of goals. The use of goal based architectures can result in a higher quality game AI, and allows game designers and developers to share more code between agents, resulting in shorter development times and increased code re-use (see, for example, [12]). A key problem with goal-based architectures is *goal arbitration*, i.e., deciding which goal or goals to work on next. In this paper, we extend the teleo-reactive programs developed in [11] to support goal processing in an architecture for game agents. We introduce a notion of a *resource* and present an algorithm for goal arbitration between teleo-reactive programs with resources. We are currently implementing the system described in this paper in a commercial game environment. While our approach was developed in the context of computer games, we anticipate that our techniques can generalize to other domains which involve a dynamic environment and resource-dependent tasks.

In section 2 we present a brief introduction to the teleo-reactive framework and introduce a simple teleo-reactive agent for the game Pacman which we use as a running example in the remainder of the paper. In section 3 we outline some of the problems of the teleo-reactive framework for game agents, focusing in particular on stable nodes. In section 4 we introduce the notion of a resource, which can be used to gain exclusive access to specific game objects as well as to represent more abstract things such as properties of the agent. In section 5 we sketch a new architecture for game agents, called GRUE, which extends teleo-reactive programs with resources. We briefly describe the main components of the architecture and present an algorithm for goal arbitration using resources. In section 6 we discuss some related work, and in section 7 we conclude and outline directions for future work.

2 Teleo-reactive Programs

Teleo-reactive programs [11, 2] were developed to control agents in dynamic environments. 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 actions can be durative, in which case the action continues as long as its condition is true. The conditions are evaluated continuously — ideally by a circuit, but otherwise continuous evaluation is simulated by executing the smallest time steps practical for the application.

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 [11].

Multiple goals are processed by a component called the *arbitrator*, which determines which program should be allowed to perform an action at each cycle [2, 1]. The arbitrator chooses programs using the concept of stable nodes. A *stable node* is a point in a teleo-reactive program at which execution of the program can safely be suspended.

That is, the condition at that point in the program is stable with respect to the other programs that are running. A condition is stable if running the other programs will not cause the condition to become false. So, for example, a program 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 program that does not require the agent to drop the object. In order to compute stable nodes, each rule is tagged with information about conditions that it makes false. If a condition required by a rule is not made false by any of the other programs, then it is stable. Stable nodes only need to be (re)computed when the set of programs changes. 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. When a TRP achieves its goal(s) and runs the null action, it is removed from the arbitrator.

The teleo-reactive architecture described in [2] and [1] is not completely autonomous. Goals are proposed by a human user who is also responsible for determining the reward for achieving a goal. This allows the use of reinforcement learning algorithms. During each execution cycle, the arbitrator runs the program with the best reward/time ratio. The reward is the expected reward 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. 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.

2.1 An 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. The maze contains hazards in the form of ghosts. There are also special dots called power pills which Pacman can eat to make the ghosts turn blue. 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:

Eat Dots

r0. if no dots
 then
 null

r1. if there is a dot
 then
 move towards the dot

Escape from Ghosts

r0. if no ghost is less than 10 units away
 OR all ghosts are blue
 then
 null

r1. if there is a ghost within 10 units
 AND the ghost is not blue
 AND there is a power pill nearby
 then
 move towards power pill

r2. if there is a ghost within 10 units
 AND the ghost is not blue
 then
 move away from ghost

Eat Blue Ghosts

r0. if no blue ghosts
 AND no power pills
 then
 null

r1. if there is a blue ghost
 then
 move towards the ghost

r2. if no blue ghosts
 AND there is a power pill
 then
 move towards the power pill

Eat Fruit

```
r0.      if    no fruit
         then
         null

r1.      if    there is a fruit
         then
         move towards the fruit
```

At any given point, 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. In a typical game 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.

Pacman is a fast-paced game, making most conditions unstable. In this set of programs, we can identify a few conditions such as “no fruit is 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 is 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. Conditions such as “all ghosts are blue” are stable with respect to the other programs (Pacman has no way to make the ghosts stop being blue), but not stable in general as ghosts stop being blue after some period of time.

3 Limitations of Teleo-reactive Programs

Teleo-reactive programs have a number of advantages for controlling game agents. 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 several goals simultaneously [2], 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. All top-level goals are given by the user who is also expected to provide rewards when those goals are achieved. This is often inappropriate in a computer game where agents must be truly autonomous; the human players should be playing the game, not providing input to their opponent.

One alternative is to give the agent a constant set of goals with fixed priority values. We used this approach to build 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. 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 priority, 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 agent then wastes time waiting for a health item to appear and ignores approaching enemies. This is not effective or realistic behaviour.

3.1 Beyond Stable Nodes

Using stable nodes also gives the architecture certain limitations. In section 2.1, we presented a set of TRPs for playing Pacman. In our Pacman programs, there were no stable nodes that were not success conditions. Therefore, if we were to try to switch between these programs at stable nodes as in [2], the arbitrator would be unable to run them in parallel. However, it is easy to see that running *Escape from Ghosts* can cause Pacman to eat a power pill, which also makes progress towards the goal of *Eat Blue Ghosts*.

There are other cases in which the use of stable nodes is not feasible. For example, consider the case of a character in a role playing game trying to make money (to pay a fee, or buy a new weapon, for instance). The character has a number of objects they can sell: a pickaxe worth 50 gold, a spellbook worth 100 gold and a magic lamp worth 50 gold. Any combination of objects that generates the appropriate amount of money will satisfy the goal. If the character needs 100 gold, they could sell the spellbook, or the pickaxe and the magic lamp. We could use a stable node that lists all possible combinations of items, but the number of combinations is exponential in the number of items.

Even if the number of combinations is small, using stable nodes requires a list, for each action, of conditions that are falsified by that action. This doesn't work if we allow disjunctive conditions in rules. For example given the TRP fragment:

```
r1.  if    guard present
      AND (have ruby OR have emerald)
      then
      bribe the guard
r2.  if    have ruby
      AND unicorn present
      then
      give ruby to unicorn
```

Assuming that both a guard and a unicorn are present, running the second rule will make the first condition false *only if the agent does not have an emerald*. Whether or not the agent has an emerald cannot be determined until run time. So we need a function that can be given an action and return a list of conditions that are falsified by that action. Which in turn means that we need to be able to predict the consequences

of any action. Of course, actions don't just affect inventory, they can also modify the environment and the effects of an action may depend on the current environment. Now the programmer has the long and tedious task of listing every possible action, in every possible situation, with every possible consequence.

3.2 Solutions

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 in response to the current game situation and assign priorities to goals based on the current situation;
- it includes an arbitration algorithm modified in order to handle situations which are common in computer games;
- it allows multiple programs to run actions in parallel during each cycle where this is possible; and
- where parallel execution of actions is not possible, it allows the programmer to provide rules for combining two or more actions. For example, if one program proposes moving northeast and another program proposes moving northwest, the programmer may wish to specify that the agent should compromise by moving north.

In the remainder of this paper, we outline GRUE. We begin by introducing the idea of resources, which allow us to solve the problems described above.

4 Resources

A *resource* represents a condition that must remain true throughout the execution of a durative action. Rather than specify the conditions that an action makes false, we require that rules whose actions may conflict can run only when they have access to the necessary resources.

Variables in TRPs can be instantiated during execution with resources meeting particular requirements. This is subject to the constraint that a program should never use resources required by a higher priority program. Resources make it easier to handle multiple goals. A task can only be run if the necessary resources are available (i.e. present and not in use by a higher priority program). If a resource becomes unavailable, for example by being pre-empted by a higher priority program that requires the same resource, programs that depend on it can be suspended until the resource is available again.

Informally, a resource is anything necessary for a rule in a program to run successfully. In the above example, the ruby and the emerald would each be a resource. Game objects 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.

Resources are stored in the agent's world model and represented as lists of lists, where each sublist contains a label and one or more values associated with that label. Each of these sublists represents a property. The following properties are required for all resources:

- ID: a unique identifier;
- TIME: the amount of time for which this resource is available; and
- TYPE: the type(s) of this resource.

In addition, resources may list additional properties as required by the game. For example, a health pack might be represented by the structure

```
([ID Health1] [TIME Infinite] [TYPE Health]
 [HealthPts 20])
```

which indicates that the item `Health1` is not time limited, is of type `Health`, and will restore 20 health points to the agent. When we use the value `Infinite` for the `TIME` property, we mean that the item will not disappear after a limited time. It might still become unavailable due to an unpredictable event (such as another agent picking it up). More than one category may be listed for the `TYPE` field, e.g., a pickaxe might be represented as

```
([ID Pickaxe1] [TIME Infinite] [TYPE pickaxe weapon]
 [Cursed false])
```

which indicates that `Pickaxe1` is a pickaxe that can also be used as a weapon, and is not cursed. We can also list multiple values for other properties (except `TIME` and `ID`) if it makes sense to do so.

A more abstract example is magical power with which the character can be “charged” for some length of time, like the `eco` in the game *Jak & Daxter*. When `Jak` is charged with blue `eco`, he can activate special items, and yellow `eco` gives him a magical attack:

```
([ID YellowEco] [TIME 30] [TYPE eco]
 [Colour yellow])
```

The *YellowEco* resource will only last for 30 seconds. Resources can also be used to represent other aspects of a game such as the locations of characters or objects or the time to complete a task or level.

Some resources have properties which are divisible. Time is one example, but other common examples in a game are ammunition and money. For these types of resources, it may be possible to use the same resource for more than one purpose simultaneously. For example, an agent with 20 gold pieces can buy an apple costing 5 gold and a hat costing 15 gold.

We also define a function called `property`. This function takes two arguments, the name of a property and a resource, and returns a list of property values, eg:

```
property(Colour, ([ID YellowEco] [TIME 30]
 [TYPE eco] [Colour yellow])) = [yellow]
```

4.1 Resource Variables

A *resource variable* is used by a rule in a TRP to specify a required resource. A *resource variable* is a 3-tuple containing:

- an identifier for this variable;
- a set of required properties; and
- a set of preferred properties.

The identifier is the variable name, which is bound to a resource. It can subsequently be used to access any of the fields of the resource structure. The required properties are those that are necessary to run the rule. The preferred properties are used to make decisions when several resources have the required properties. Programmers can use preferred properties to give agents different behaviour without rewriting programs. This can be used to give agents the appearance of different personalities. (See also section 5.4.1 below).

For example, an attack program might use the following resource variable:

```
(?Gun1 [[TYPE gun] [Ammunition 20]] [])
```

which specifies a weapon with at least 20 rounds of ammunition.

Divisible properties are handled specially. Some games group identical items into one listing in the players inventory, with the number indicated. For example, “2 silver arrows” instead of “a silver arrow, a silver arrow”. A rule might require a specific number of items, which could be less than the number in the group. We can use the properties list to handle such situations gracefully. Given the resource:

```
([ID Gold1] [TIME Infinite] [TYPE money] [Number 53])
```

and the resource variables

```
(?Money1 [[TYPE money] [Number 10]] [])  
(?Money2 [[TYPE money] [Number 20]] [])
```

the arbitrator will match both resource variables with the resource because the number specified by the resource variable is less than the number listed in the resources. This works with any numerical value, not just the property called “Number”.

The special value `Infinite` can be used with any numerical property. A “Number” property of `Infinite` would allow any number of variables to bind to a resource. This is useful for representing knowledge, eg:

```
([ID Stairs1] [TIME Infinite] [TYPE knowledge]  
 [Number Infinite] [Locaton 45 50])
```

which specifies the x and y coordinates of a staircase in a two-dimensional game.

Once a variable has been bound to a resource, we can use the `property` function to retrieve additional information. If `?Gun1` is bound to the resource

```
([ID revolver] [TIME infinite] [TYPE weapon gun]  
 [Ammunition 20])
```

we can then ask what type ?Gun1 is:

```
property(TYPE, ?Gun1) = [weapon gun].
```

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 don't care about the value. The following resource variable

```
(?Box1 [[TYPE container] [Location *]], [])
```

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

5 GRUE: a new architecture

Based on our experience with the prototype described in section 3, we have designed GRUE specifically for situations encountered in games. The 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.

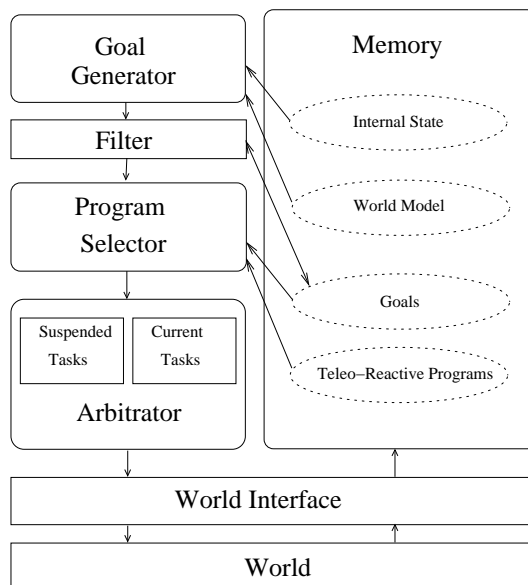


Figure 1: GRUE

The world interface takes information from the environment, in this 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.

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 the execution time for an increase in the time it takes for a character to respond to events in the environment.¹

The program selector is a simple look-up function, which matches the condition specified by a goal to the success condition of a program. The program replaces the condition in the goal structure, creating a task.

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. If a task cannot proceed due to lack of resources, it is suspended rather than discarded as the situation that gave rise to the task may no longer form part of the agent's world model, even though achieving the goal may still be a good idea. For example, an agent might see a staircase and generate a goal to go up to the upper floor. Other goals, such as finding food, may be more important and cause the agent to wander off in search of something to eat. The agent can no longer see the staircase and may have removed it from memory, thus preventing it from re-generating the goal. If we want the agent to remember that it had the goal of going upstairs, we must suspend the task.

In the remainder of this section, we discuss the main data structures used by the program and their associated algorithms in more detail.

5.1 Goals

A goal consists of an identification string, a priority, a type, and a condition to be made true. This condition is the same as the end condition in the program that will achieve the goal.

Achievement goals are straightforward. For example, we can rewrite the goals in our Pacman program as the following achievement goals:

```
[GetAwayFromGhosts 90 Non-maintenance
  [NOT(Ghost1 Infinite [[Type ghost] [Dist < 10]] [])
   OR
   NOT(Ghost1 Infinite [[Type ghost] [Blue false]]
        [])]]

[EatBlueGhosts 50 Non-maintenance
  [NOT(Ghost1 Infinite [[Type ghost] [Blue true]] [])]]

[EatFruit 50 Non-maintenance
  [NOT(Fruit1 Infinite [[Type fruit]] [])]]

[EatDots 20 Non-maintenance
```

¹Small increases in the time taken to respond to events may actually result in increased believability.

```
[NOT(Dot1 Infinite [[Type dot]] [])]]
```

Note that while the first goal is called `GetAwayFromGhosts`, the condition actually specifies that either there are no ghosts within 10 units or else all ghosts are blue. This allows the program selector to match it with the correct program (see below).

Maintenance goals, where the agent is attempting to maintain a condition, are a special case. At first glance, it seems that 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. If the agent’s goals conflict, 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 persist in the arbitrator, even when the goal condition (currently) achieved. As long as the condition is maintained, the rules in the TRP will not fire, but the task can still use resources. These 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 an ordinary goals, for example:

```
[KeepHealthPoints 95 Maintenance [HP > 20]]
```

5.2 Programs

Programs are pre-written teleo-reactive programs extended with resource variables. A TRP is a list, containing an identification string, a list of arguments, and 1 or more rules. The rules are evaluated in order, with the first rule whose condition is true proposing an action to execute.

A rule consists of an identifier, a condition and a list of actions. The condition can contain resource variables and/or user defined predicates. These may be joined using the logical operators AND, OR, NOT, and the relational operators =, >, <, ≤, and ≥. A resource variable evaluates to true when it is successfully bound to a resource. 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; the world may change between observations. (Binding of resource variables is discussed in more detail in section 5.4)

In the last section, we defined some goals for our Pacman agent. Here is a program to achieve the goal `GetAwayFromGhosts`:

```
[EscapeFromGhosts
[Rule0 NOT(?Ghost1 Infinite [[Type ghost]
[Dist < 10]] [])
OR NOT(?Ghost1 Infinite [[Type ghost]
[Blue false]] [])
⇒
```

```

[null]]

[Rule1 (?Ghost1 Infinite [[Type ghost]
      [Blue false] [Dist < 10]] [])
      AND (?PowerPill1 Infinite [[Type powerpill]]
      [[Dist < 10]])
  =>
  [move toward PowerPill1]]

[Rule2 (?Ghost1 Infinite [[Type ghost]
      [Blue false] [Dist < 10]] [])
  =>
  [move away from Ghost1]]
]

```

Programs can be hierarchical. That is, one program can call another program as an action. This allows the agent designer to write generic programs for common sub-tasks which can then be used in multiple places. For example, an agent may have a generic GOTO program, which moves the agent to a location.

```

[GOTO [?target]
  [Rule0 (?AgentState [[TYPE state] [Location *]] [])
    AND property(Location, ?AgentState) =
      property(Location, ?target)
  =>
  [null]]

  [Rule1 ...]

  ...

  [RuleN ...]
]

```

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. We assume for the purposes of this example that the world sends a message to the agent listing information about the agent's current state (this is true of games such as Unreal Tournament) and that this information is stored in the database as a resource by the world interface. The first rule therefore 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.

A call to GOTO might look like:

```

[Rule4 (?Enemy1 Infinite [[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.

5.3 Tasks

Tasks are created from goals by the program selector. Since the goal contains a condition to be achieved, the program selector must simply find a program with the corresponding success condition. The program replaces the condition in the goal structure, and the resulting data structure is a task.

Runnable Tasks are those tasks that have had enough resource variables bound to make one or more rules runnable. Suspended Tasks are tasks which are not runnable due to lack of resources.

The tasks corresponding to the above goals are listed below.

```
[GetAwayFromGhosts 90 Non-maintenance
  [EscapeFromGhosts
    [Rule0 NOT(?Ghost1 Infinite [[Type ghost]
      [Dist < 10]] [])
      OR NOT(?Ghost1 Infinite [[Type ghost]
        [Blue false]] [])
    ]
  =>
  [null]]

[Rule1 (?Ghost1 Infinite [[Type ghost]
  [Blue false] [Dist < 10]] [])
  AND (?PowerPill1 Infinite [[Type powerpill]
    [[Dist < 10]])
  ]
=>
[move toward PowerPill1]]

[Rule2 (?Ghost1 Infinite [[Type ghost]
  [Blue false] [Dist < 10]] [])
  ]
=>
[move away from Ghost1]]
]
```

Notice that the task name, priority, and type come from the goal while the remainder of the structure is a TRP for achieving the condition that was listed in the goal.

5.4 Goal Arbitration

It is the job of the arbitrator to execute as many tasks as possible. To do so, it must allocate resources to the tasks, resolving conflicts where possible. This may involve suspending currently executing tasks and/or resuming suspended tasks as new tasks are proposed by the goal generators and the set of available resources change, e.g., due to changes in the game environment or the passage of time.

The arbitration process can be divided into four stages:

- the first stage resumes suspended tasks—this stage only runs if new resources have become available, or if resources have been freed during the last arbitration cycle;

- the second stage allocates resources to each task by binding resource variables to available resources;
- the third stage runs the first rule in each program that can run—in order to run, a rule must have all the necessary resources. The action is not actually executed, but rather added to a list of proposed actions.
- in the fourth stage, any conflicts between the proposed actions are resolved.

Steps 2, 3 and 4 are discussed in more detail below.

5.4.1 Binding the Resource Variables

Allocating resources to tasks consists of four main steps:

1. Sort the tasks according to priority.
2. Starting with the highest priority task, look for a rule which has a condition that evaluates to true. Resource variables are treated as conditions which evaluate to true when they are bound to resources.
3. If no rule in the program can run, then suspend the task.
4. Repeat steps 2 and 3 until all tasks have been processed, no more resources are available, or the maximum number of runnable tasks is reached. Suspend any tasks that are not runnable.

The main criteria used for binding resource variables is that a lower priority task may never take resources from a higher priority task. Therefore, we allow the highest priority task to bind its resources first. Each resource variable is matched against the resources stored in memory. A resource variable can only be bound to a resource which has all of the properties listed in the required properties list. In the case of a tie, the resource variable will bind to 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.

In cases using numerical quantities, if a resource variable binds to a resource which has more than the required quantity of a property, then the resource remains available to bind to additional resource variables. This is subject to the requirement that the total amount requested by all the bound resource variables is not more than the amount specified in the resource. Where a quantity is given a value of `Infinite`, then any number of resource variables can bind to the resource. `TIME` is handled like any other numerical property, except that if no time resources are present in memory, the amount of time available is assumed to be infinite.

When a resource variable is bound during one execution cycle and then used again during the next cycle, it remains bound to the same resource unless that resource is no longer available.

5.4.2 Running the Rules

Once all tasks have been made runnable or suspended, each runnable rule should propose an action. These actions are collected in a proposed actions list so they can be checked for conflicts before attempting to execute them in the environment. Ordinarily, any tasks whose top rules are runnable will be 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. If the top condition of a maintenance goal is not true and none of the rules can bind to the necessary resources, then the task is suspended.

5.4.3 Resolving Conflicts

Once the variables have been bound and each task has proposed an action, there may still be some conflicts. For example, two tasks might propose moving in opposite directions. The easiest way to resolve such conflicts is to simply discard the action proposed by the lower priority task. Alternatively, it may be possible to blend actions together. This will be dependant on the specific environment, so the programmer may optionally provide information specifying which actions can be blended. For example, in Unreal Tournament it is possible to shoot a gun while running, but only if the player is facing the target. The *run* action causes the player to turn toward the destination, but the *strafe* action allows the player to face somewhere else. If the arbitrator has this information available, it can change *run* actions to *strafe* actions when a *shoot* action has also been requested.

6 Related Work

Computer games and other interactive simulations have many attractions for AI researchers and there has been a considerable amount of recent work in this area.

Our approach extends that of Benson and Nilsson [11, 2]. 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 [2]. First, disjunctive conditions can be used to represent situations where there are several alternate ways of achieving the same goal. In section 3, we showed that there are potential problems when using stable nodes with disjunctive conditions.² Secondly, Benson and Nilsson's architecture has no way of using quantified resources. Using resources also 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 [2, 1] 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

²Disjunctive conditions are not discussed in [11], [2] or [1], however both [2] and [1] contain examples with disjunctive conditions. It is possible that the problems mentioned here were simply not encountered in the environments used in [2] and [1].

force the agent to work exclusively on a high priority goal. If a less important goal can be completed in a sufficiently short time, the arbitrator will always take the time to complete it. 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 [2] 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 [14]. 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 [15]. 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. However, MINDER1 can only execute one plan at a time. Combining the ability to suspend and resume goals based on both importance and availability of resources with the arbitrator's function of choosing the most appropriate action to work towards a set of active goals gives GRUE additional flexibility.

Bryson's work [3] 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, that means that tasks always have the same relative importance. We discussed problems with this approach in section 3. For comparison purposes, 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.

Much work in games and and real-time simulations has been done using the Soar architecture (eg. [10, 9, 13]). The Soar system is designed to reason about goals, but it is limited to a single goal hierarchy. Several approaches have been taken to the use of multiple goal hierarchies in Soar, however they require either representing some goals implicitly or forcing unrelated goals into a single hierarchy [8]. GRUE has been

designed to process parallel goals.

Horswill and Zubek [7] discuss the problem of game agents and argue that a hybrid approach is most appropriate. They discuss a technique using role passing [6] for fast inference in dynamic environments. DePristo and Zubek [4] describe a hybrid architecture used for an agent in a MUD (Multi-User Dungeon). This type of environment is essentially a role-playing game and typically involves tasks like killing monsters and buying equipment such as weapons and armor. 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 was capable of surviving in the MUD environment, but several problems were encountered. 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 handle the types of information and tasks that are commonly encountered in game environments.

7 Conclusions and Future Work

We have argued that game environments can be effectively handled by an agent architecture which uses an explicit representation of goals. We have proposed an approach 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 [2] and [1] has several limitations making it inappropriate for use in games. To overcome these limitations we have introduced the idea of *resources*. We have presented an architecture, GRUE, which extends teleo-reactive programs with resources and includes an algorithm for arbitrating between goals with conflicting resource requirements. GRUE facilitates code re-use and the development of flexible agents capable of balancing competing goals and responding appropriately to their environment.

Our future work will focus on building and testing more complete game agents using GRUE as described in this paper.

References

- [1] S. Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Stanford University, December 1996.
- [2] S. Benson and N. Nilsson. Reacting, planning and learning in an autonomous agent. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence*, volume 14. The Clarendon Press, 1995.
- [3] J. J. Bryson. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA, June 2001. AI Technical Report 2001-003.

- [4] M. DePristo and R. Zubek. being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 2001.
- [5] N. Hawes. 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, November 2000.
- [6] I. Horswill. Grounding mundane inference in perception. *Autonomous Robots*, 1998.
- [7] I. D. Horswill and R. Zubek. Robot architectures for believable game agents. In *Proceedings of the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games*, 1999.
- [8] R. Jones, J. Laird, M. Tambe, and P. Rosenbloom. Generating behavior in response to interacting goals. In *Proceedings of the 4th Conference on Computer Generated Forces and Behavioral Representation*, 1994.
- [9] J. Laird. 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*, 2000.
- [10] J. Laird and J. Duchi. Creating human-like sythetic characters with multiple skill levels: A case study using the soar quakebot. In *AAAI Fall Symposium Seris: Simulating Human Agents*, November 2000.
- [11] N. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, January 1994.
- [12] J. O'Brien. A flexible goal-based planning architecture. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 375–383. Charles River Media, 2002.
- [13] M. van Lent, J. Laird, J. Buckman, J. Hartford, S. Houchard, K. Steinkraus, and R. Tedrake. Intelligent agents in computer games. In *Proceedings of the National Conference on Artificial Intelligence*, pages 929–930, 1999.
- [14] I. Wright. *Emotional Agents*. PhD thesis, University of Birmingham, February 1997.
- [15] I. Wright, A. Sloman, and L. Beaudoin. Towards a design-based analysis of emotional episodes. *Philosophy Psychiatry and Psychology*, 3(2):101–137, 1996.