Authoring Behaviours for Game Characters Reusing Automatically Generated Abstract Cases [*]

Antonio A. Sánchez-Ruiz, David Llansó,
Marco Antonio Gómez-Martín and Pedro A. González-Calero

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
email: {antonio.sanchez,llanso,marcoa,pedro}@fdi.ucm.es

Abstract. Authoring the AI for non-player characters (NPCs) in modern video games is an increasingly complex task. Designers and programmers must collaborate to resolve a tension between believable agents with emergent behaviours and scripted story lines. Behaviour trees (BTs) have been proposed as an expressive mechanism that let designers author complex behaviours along the lines of the story they want to tell.
On the other hand, BTs appear as a too complex mechanism for non programmers. In this paper, we propose the use of *abstract cases* automatically generated through planning to assist designers when building BTs. In order to make this approach feasible within state-of-the-art video game technology, we generate the planning domain through an extension of the component-based approach, a widely used technique for representing entities in commercial video games.

1 Introduction

According to the number of papers dedicated to the subject in the editions 3 and 4 of the AI Game Programming Wisdom [8, 9], Behaviour Trees (BTs) are the technology of choice for designing the AI of NPCs in the game industry. BTs are proposed as an evolution for hierarchical finite state machines (HFSM) intended to solve its scalability problems by emphasizing behaviour reuse.

Behaviour trees have been proposed as an expressive mechanism that let designers author complex behaviours along the lines of the story they want to tell, but at the same time, BTs appear as a too complex mechanism for non programmers [2, 3]. Commercial game development teams usually build some support tools in the form of tree editors, where the designer can choose from a set of predefined composite nodes, conditions to be checked, and basic actions that can be included in the tree. Nevertheless, in practice, there is a tension between the freedom that the designers require to include their narrative in the game and the effort required from programmers to debug faulty AI authored by non-programmers.

In this paper we propose the use of ontologies and planning techniques to assist game designers when authoring the AI for non-player characters. The design of BTs

[*] Supported by the Spanish Ministry of Science and Education (TIN2006-15140-C03-02 and TIN2006-15202-C03-03)

is usually an interactive process in which the designer incrementally adds new tree branches to make the NPC react to new situations. We propose an automatic way to compute abstract cases or solutions to those new situations, that designers may adapt before incorporating them to the current BT.

A drawback for using declarative knowledge-intensive AI techniques in games is the additional effort required to model the domain. In this case we require having a model of the actions that NPCs can do in the game world. In order to close this gap between academic and industrial game AI, we propose generating the planning domain through an extension of the component-based approach for representing entities, which is widely used in commercial video games.

The rest of the paper runs as follows. Next Section introduces the two techniques from commercial video games incorporated into our approach: component-based game entities, and behaviour trees. Next Section, describes the main ideas of our proposal for authoring BTs from automatically generated abstract cases. Next two Sections provide the details and exemplify the approach, first describing the generation of the planning domain and then the use of abstract cases to author behaviour trees. Last Section reviews related work and concludes the paper.

2 Background

2.1 Components

In the development of a virtual environment, the layer responsible of the management of the entities is usually created using an object-oriented programming language such as C++. Over the years this object-management system has been based on an inheritance hierarchy, where all different kinds of entities derive from the same base class often called `CEntity`.

Some of the consequences of this extensive use of class inheritance are, among others, an increase in the compilation time [5], a code base difficult to understand and big base classes. To mention just two examples, the base class of Half-Life 1 had 87 methods and 20 public attributes while Sims 1 ended up with more than 100 methods.

Due to all these problems developers tend to use a different approach, the so called component-based systems [13, 10]. Instead of having entities of a concrete class which define their exact behaviour, now each entity is just a component container where every functionality, skill or ability that the entity has, is implemented by a component. From the developer point of view, every component inherits from the `IComponent` class or interface, while an entity becomes just a list of `IComponents`.

As entities are now just a list of components, the concrete components (or abilities) that constitute them may be specified in an external file that is processed in execution time. This approach eases the creation of new kind of entities, because it does not require any development task but just the selection of the different skills we want our new entity to have from a set of components. In order to allow fine-grained adjustment of the behaviour (or skills) of different entities, their definition may also set the values of different attributes that components use as parameters of their behaviours.

Both, the list of components of an entity and the initial values of their parameters, are usually stored in a separated file that can be seen as the file that describes the main

```
<blueprint>
<entity type="goblin" ontType="Goblin" parentOnt="Monster">
  <components list="Take, MoveTo, TakeCover, MeleeAttack,
                    LongRangeAttack, Charge-At, ..."/>
  <attributes>
    <attrib name = "strength" value = "weak"/>
    <attrib name = "weapon-tech"
            value = "rudimentary, elaborate"/>
    <attrib name = "height" value = "short"/>
     ...
  </attributes>
</entity>
...
</blueprints>
```

Fig. 1. Partial list of blueprints file

characteristics of every entity. That file, usually known as the *blueprint*, is parsed by the game engine at the beginning of its execution. When the game loads a new level from the map file, it iterates over the list of the level entities and uses the blueprint file for creating and launching them.

Figure 1 lists a portion of one of such files that describes the goblin entity. The entity description of the entity has two main parts, the list of components and the list of attributes. Goblin have components that allow them, among other things, go through the environment and pick up objects. All these skills are parameterized in the `attributes` section. For example, the `strength` attribute influences the `Take` component, while the `height` predefines the kind of objects the `TakeCover` component should consider as protections.

2.2 Behaviour Trees

BTs define an AI driven by goals, in which complex behaviours can be created combining simpler ones using a hierarchical approach. Nodes in a BT represent behaviours, where an inner node is a composite behaviour and a leaf in the tree represents an action. To promote reusability, behaviours do not include the conditions that lead to transitions. Those conditions are represented as guards in nodes of the tree so that the same behaviour can be used in different contexts with different guards. To further promote reusability, behaviours may be parameterized, so that in a particular context parameters are bound to actual values in the map. This way, a node in such a behaviour tree is represented through: a behaviour (be it composite or a primitive action); bindings for the parameters of that behaviour; and a guard condition that must be true at run time for that behaviour to be activated.

From the different models of execution for BTs, we choose one where a BT has an active branch, going from the root to a leaf, of behaviours being executed. Every tick of the game, some guards may get evaluated and some behaviours may finish, be it

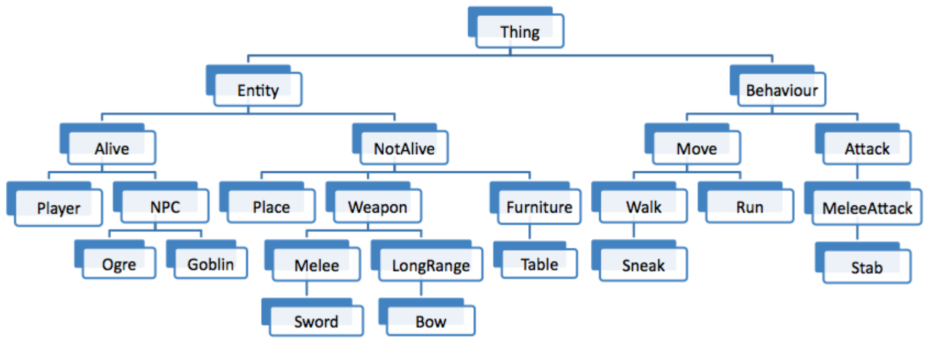

Fig. 2. Ontology that defines the domain vocabulary

successfully or with failure, eventually leading to the expansion of a new active branch in the tree.

Although more complex types of composites are described in the literature, for the goals of this paper we only require three types of composites: sequences, static priority list and dynamic priority list. A sequence composite behaviour executes its children in the order they are defined, succeeding when every children succeeds and terminating with failure whenever one of the children fails. Children behaviours of a sequence are not guarded by conditions. A static priority list is a composite node that evaluates its children guards in order and activates the first child whose guard is true. A dynamic priority list, in its turn, re-evaluates the guards of its children with higher priority (the first child being the one with highest priority) than the active one, and switches to a higher priority child whenever possible.

BTs provide designers with an abstraction that allow them to treat a tree as a new complex behaviour that may be attached in other more general BTs. In that sense, during development time designers create a collection of basic behaviours in form of BTs, which are later attached to the branches of different BT for more than one entity. In order to improve the reusability, BTs may be parameterized. For example, designers may build a BT that an enemy using an available weapon attacks and after that picks an item up.

3 Generating Abstract Cases to Support BT Creation

The creation of BTs is a difficult task that designers usually perform by means of a try-and-fail process. The consequence is that the final quality of the BTs depends to a large extent upon the ability and experience of designers. Our proposal consists in helping them by means of abstract cases that are automatically computed using planning and ontologies. This way, we suggest sets of solutions that designers can adapt and add to the current BT.

In order to use planning we need to describe the domain and planning actions using a formal language.The description of a planning domain includes two main parts: (a) the description of the predicates that conform the domain vocabulary and how they are

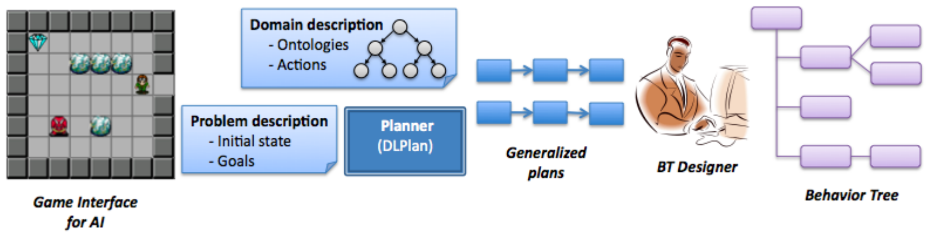

Fig. 3. Interactive process to create Behaviour Trees

related among them, and (b) a set of planning actions. We propose the use of *ontologies* to represent the first part, that is, the vocabulary and the domain constraints. Figure 2 shows an example of ontology that intuitively describes different components of a game. The set of planning actions, on the other hand, is strongly related to the available basic behaviours in the game. Planning actions are described in terms of preconditions and effects using the vocabulary available in the domain ontology.

Figure 3 summarizes our proposal to support the AI designer during the creation of BTs. By means of a graphical interface, that can be a simplified version of the game interface, the designer sets up a particular game scenario and some goals. Next, we automatically generate the equivalent symbolic description using the planning language, and by means of a planner, we compute all the possible plans that solve the problem. The planner that we use, DLPlan[1], is able to generalize the resulting plans using the domain ontology. This way, plans presented to the designer are not only specific plans for the current scenario but general strategies or abstract cases that can be reused in a broader set of situations (this will be described in detail later in section 5). Then the designer can use those cases to complete the BT that is currently building. Let's remember that BTs are useful in a broad set of scenarios and thus, this is actually an interactive process in which the designer proposes different scenarios to the system and incrementally completes the BT using the retrieved solutions.

Finally, the process by which abstract cases are integrated to the current BT is, nowadays, manual, the designer is the only responsible of changing the BT to add the new branches. It is important to remark that the planner works with a limited model of the game, while the designer can take into account much more factors (like story plot or special situations) in order to select behaviours, modify preconditions under certain circumstances and set the priority of each alternative. The planner output, therefore, only shows different solutions that the designer must modify, validate or reject. Anyway, preconditions of the planning operators and the generalized plans computed by DLPlan are usually a good source of inspiration to define the guards of new nodes in the tree.

4 Generating the Planning Domain

To be able to use planning techniques, we need a symbolic representation of the world as well as the actions that each type of entity can perform. The basic approach is to create

[1] Freely available at *http://sourceforge.net/projects/dlplan/*

this description from scratch. However, this information is, at least partially, already in the C++ classes that programmers have to implement to develop the game and in the configuration files that define the different types of entities.

Our proposal is to use this information in order to automatically generate all the information required by the planner. In order to do that we have to minimally extend the information contained in the components implementation and *blueprint* file.

The process starts with a base domain ontology that can be seen as the basic vocabulary of the game genre and it is independent of the concrete game being developed. This domain ontology is taken for granted and it includes the basic vocabulary for describing the new type of entities and actions that the game will incorporate. In other words, we start with an ontology similar to the one in figure 2 but without the leaves that correspond to the concrete types of entities in the game.

In order to populate the ontology with the new entities, we use the *blueprints* file. As we showed in section 2.1, this file has an entry per game entity, describing the set of components and attributes it has. The only new information we need to add to this file is two special fields in every entity, *ontType* and *parentOnt*, that set the corresponding symbolic name for this entity in the ontology and the branch or branches in which it must be added. This way, we can now add automatically new concepts in the ontology to represent these new types of entities.

Once the entity has been added to the ontology, we can also add information about their properties and the actions that they can carry out. This is done iterating over the list of components that the entity has, asking them which information has to be injected to the corresponding ontological entity. As an example, we would add the description of the `Goblin` entity that appears in figure 1 with `canWalk`, `canTake`, `hasStrength.weak` and other properties.

As regards the operators the planner uses, we can extract them from the components. Most of the components are in fact the responsible of the execution of one or more actions over the environment. They first check if the action can be carried out and then execute it. Our method of automatically generating the planning operators consists in extending the components with an extra task: their self-description. In that sense, every component that represents a behaviour must be able to provide, through its programming interface, the planning action that describes it.

Figure 4 shows the operator description that components `MoveTo`, `TakeCover` and `Take` generate. The preconditions are specify in terms of the entities' properties.

5 Plan Generation and BT Authoring

This section describes how to build BTs using our approach, i.e., taking advantage of planning techniques to support designers during the process. Next, we introduce a concrete example in which a BT must be created to control a greedy goblin that has entered in a room to discover a diamond in the opposite corner. We will assume the existence of a graphical interface (As complex games usually provide [6]) to define different initial states and goals without having to deal with logical predicates but just setting items and units in the map and defining theirs attributes.

```
WALK−TO(?who: alive, ?target: entity )
vars : ?r
pre : canWalk (?who), inRoom(?who, ?r), inRoom(?target , ?r ), aloneInRoom(?who)
post : nextTo(?who, ? target )

TAKE−COVER(?who: alive, ?c: cover)
vars : ?r , ?s1 , ?s2
pre : canTakeCover(?who), uncovered(?who), inRoom(?who, ?r), inRoom(?c, ?r ),
      cover(?c ), hasSize(?who, ?s1 ), hasSize(?c , ?s2 ), lessEqSize(?s1 , ?s2)
post : covered(?who)

TAKE(?who: alive, ?what: resource )
vars : ?w, ?s
pre : canTake(?who), nextTo(?who,?what), hasWeight(?what, ?w),
      hasStrength (?who, ?s ), enoughStrength(?s , ?w)
post : inInventary (?who, ?what)
```

Fig. 4. Planning operators corresponding to some basic behaviours.

Let's start with the simplest situation, where the goblin and the diamond are in the same room and there are no enemies near. This goblin is a warrior well armed with a short sword, a small knife, a short bow and a sling. The room, in turn, contains some furniture: a table, two chairs and a bookcase. Although the designer does not know it, behind the scene this information is been automatically translated to a symbolic representation for the planer using the vocabulary in the ontology. Now, when the designer defines the goal (the goblin gets the diamond), the planner shows the only possible plan: walk until the diamond location and take it:

1. `WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

Actually, using the abstraction capabilities of DLPlan we are able to point out that this plan is applicable in several more scenarios, because the plan only requires *goblin1* to be an entity that can walk and take things and that is alone in the room, and *diamon1* to be a small item. The generalization process followed by DLPlan to reach this conclusion is based on the ontological domain definition and it is described in [11].

Using this information, the designer builds the red branch, with dashed borders, of the BT shown in figure 5, that represents the only plan available in this scenario. It is important to mention that plans generated using the planner are sequences of actions that correspond to the leaves of the BT. The definition of internal nodes in the tree to group basic actions and to represent different alternatives is responsibility of designers.

Next, the designer must complete this basic BT to make it useful in other scenarios as well, for example when there is an enemy in the same room that has already detected the goblin. This time the planner computes several more possible plans:

1. `ChargeAt(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1),`
   `Take(goblin1,diamon1)`

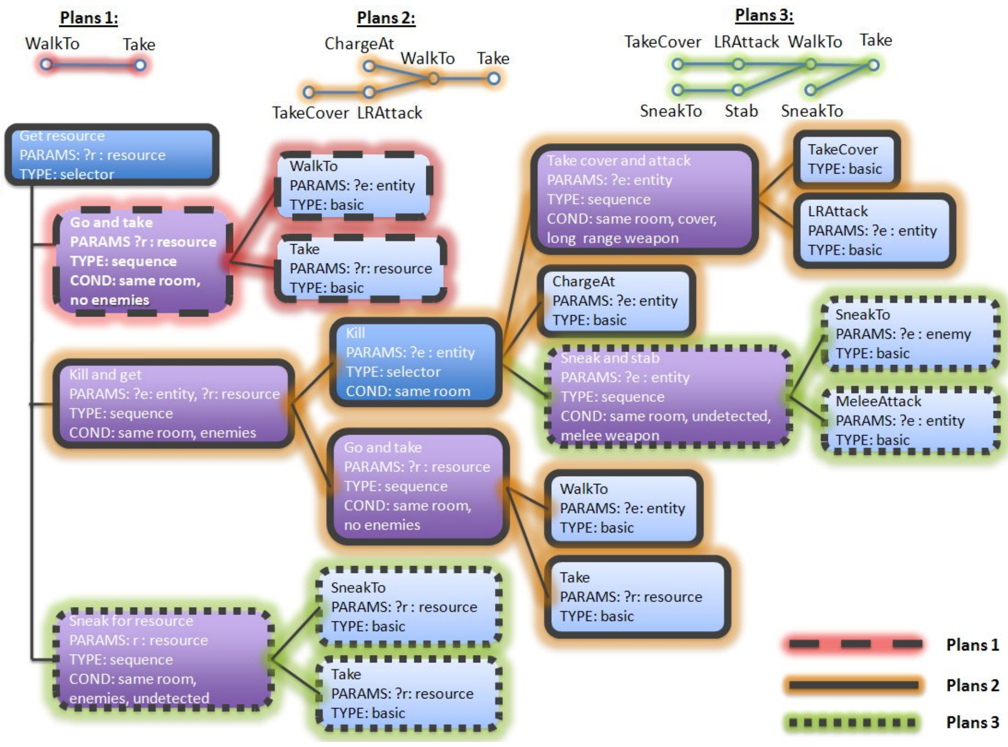

Fig. 5. Example of BT creation from different plans

2. `ChargeAt(goblin1,knife1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
3. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
4. `TakeCover(goblin1,table1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
5. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
6. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

It is important to mention that during the computation of these plans the planner has performed some interesting inferences using the domain knowledge. For example, the planner has used the table and the bookcase as possible covers and different weapons have been classified in melee or long range weapons. With those inferences, the six generated plans are in fact two different strategies parameterized with different values: charge against the enemy and then take the diamond; or look for a cover, attack the enemy from the distance and then take the diamond:

1. `ChargeAt(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

2. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`

Again, using the abstraction capabilities of DLPlan, those plans are applicable in more general scenarios than the current one, in this example *sword1* represents any melee weapon and *bow1* any long range weapon.

The computation of plans and the later generalization is performed behind the scene, and so, the designer only sees the generalized plans. Then, he has to complete the previous behaviour tree to incorporate the new possibilities. The resulting BT is built adding the orange branches, with continuous borders, shown in figure 5. Basically, the previous branch is only applicable if there are no enemies in the room, and in other case we have to kill the enemies first.

Finally, the designer wants to complete the BT with new branches that will be executed when there is an enemy in the room but he has not detected the goblin yet. This time, the planner computes several solutions that can be summarized in three strategies: (1) take a cover, attack from the distance, go until the diamond and take it; (2) sneak until the enemy, stab him, go until the diamond and take it; and (3) sneak until the diamond and take it (without killing the enemy).

Next, the designer adds these new alternatives to the BT, obtaining something similar to the figure 5 . In this case, the strategy of looking for a cover and attacking using a long range weapon was already in the previous BT so we only need to add the other two plans (green branches with dotted borders).

6 Related Work and Conclusions

Authoring the AI for non-player characters in modern video games is an increasingly complex task. Behaviour Trees, as successor of Hierarchical Finite State Machines, are a promising and emergent technology to represent the complex behaviours that the market demands. However, designers that have to build these BT usually do not have a programming background and do not find them intuitive enough. Consequently, we have to provide the designers with good tools to support the BT creation process. In particular, we propose the combination of planning and ontologies to propose abstract cases to the designer, that will incorporate them into the BT after the required adaptations.

We have described an interactive process in which the designer proposes different scenarios and goals and the planner computes all the possible solutions and generates abstractions for the initial state. Then the designer uses that information to build a more robust and versatile BT that represents different plans as different branches. The use of off-line planning techniques, let us to explore the space of design possibilities, and does not have the computational cost of planning during the game execution. The use of ontologies, on the other hand, provides an intuitive way to describe scenarios and interact with the planner. In order to make this approach feasible within state-of-the-art video game technology, we generate the planning domain through an extension of the component-based approach for representing entities which is widely used in commercial video games.

Related approaches have been described in [7, 4]. Pizzi et al. [7] use a planner to compute every combination of actions to solve each level, and show them to the human designer like a comic, to let him check if there are any gaps in the storyline or in the design of the level. Another related work is the one described in [4] where the authors propose the use of planning to coordinate the behaviours for NPCs that are not main characters in the storyline of role games. Our approach does not generate the full behaviour for an NPC as [4], but proposes particular plans for particular situations, and the human designer is responsible for incorporating those traces into the tree he is designing. On the other hand, we differ from [7] because we intend to assist the designer on building the behaviour for an NPC, instead of supporting a kind of validation of a game level by exploring the solutions that the player may try, so that the designer may choose to re-design the level in order to avoid certain solutions.

Regarding CBR in games, most of the works in literature focus on how to use cases during the execution of the game, either to improve the quality of the final AI or to improve the performance of the AI engine. In [1], cases consist of strategies applicable to specific situations in the real-time strategy game Stratagus. Using these strategies, they were able to beat opponents considered “very hard”. Another interesting work presents the multi-layered architecture CARL [12], that combines CBR and reinforcement learning to achieve a transfer-learning goal in another real time strategy game.

As future work we will study how to improve the interface between the planner and the BT authoring tool, and how to semi-automate the translation process between both representations. We are also interested in developing debugging tools to help the designer to understand the inferences that the planner does why the planner proposes some solutions and rejects others in particular scenarios.

References

1. D. W. Aha, M. Molineaux, and M. J. V. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR*, pages 5–20, 2005.
2. D. Isla. Handling complexity in the Halo 2 ai. In *Game Developers Conference*, 2005.
3. D. Isla. Halo 3 - building a better battle. In *Game Developers Conference*, 2008.
4. J. P. Kelly, A. Botea, and S. Koenig. Offline Planning with Hierarchical Task Networks in Video Games. In *AIIDE*, 2008.
5. J. Lakos. *Large Scale C++ Software Design*. Addison Wesley, 1996.
6. M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. Scriptease: Generating scripting code for computer role-playing games. In *ASE*, pages 386–387, 2004.
7. D. Pizzi, M. Cavazza, A. Whittaker, and J.-L. Lugrin. Automatic Generation of Game Level Solutions as Storyboards. In *AIIDE*, 2008.
8. S. Rabin, editor. *AI Game Programming Wisdom 3*. Charles River Media, 2006.
9. S. Rabin, editor. *AI Game Programming Wisdom 4*. Charles River Media, 2008.
10. B. Rene. *Game Programming Gems 5*, chapter Component Based Object Management. Charles River Media, 2005.
11. A. A. Sánchez-Ruiz, P. A. González-Calero, and B. Díaz-Agudo. Abstraction in Knowledge-Rich Models for Case-Based Planning. In *Proc. of International Conference on Case-Based Reasoning*, 2009.
12. M. Sharma, M. Holmes, J. Santamaria, , A. Irani, C. Isbell, and A. Ram. Transfer learning in real-time strategy games using hybrid cbr/rl. In *Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007.
13. M. West. Evolve your hiearchy. *Game Developer*, 13(3):51–54, Mar. 2006.