# Extending Case-Based Planning with Behavior Trees *

**Ricardo Palma†, Pedro A. González-Calero, Marco A. Gómez-Martín and Pedro P. Gómez-Martín**
rjpalma@estumail.ucm.es, {pedro,marcoa,pedrop}@fdi.ucm.es
Dpto. Ingeniería del Software e Inteligencia Artificial
Facultad de Informática, Universidad Complutense de Madrid
C/ Prof. José García Santesmases, s/n
28040-Madrid (Spain)

### Abstract

The combination of learning by demonstration and planning has proved an effective solution for real-time strategy games. Nevertheless, learning hierarchical plans from expert traces also has its limitations regarding the number of training traces required, and the absence of mechanisms for rapidly reacting to high priority goals. We propose to bring the game designer back into the loop, by allowing him to explicitly inject decision making knowledge, in the form of behavior trees, to complement the knowledge obtained from the traces. By providing a natural mechanism for designers to inject knowledge into the plan library, we intend to integrate the best of both worlds: learning from traces and hard-coded rules.

## Introduction

Real-time strategy (RTS) games are very demanding in terms of AI complexity. They require fast pathfinding algorithms for moving large numbers of units through extensive levels, which need to be manually or procedurally annotated with tactical information. Regarding decision making, RTS games require a multi-tiered AI approach, with decisions made at a low level for individual characters, at an intermediate level for a formation of characters, and at the high level for reasoning about an entire team in the game. Usually simple techniques, such as state machines, are used for low level decision making, while some form of rule-based system is the most common approach for decision making at higher levels (Millington and Funge 2009).

Building a rule-based system for decision making at the tactical and strategic level of an RTS game is a complex task for game designers. In order to alleviate this authoring effort, there is an open line of research on the automatic acquisition of decision making knowledge for RTS games from recorded traces of human experts playing the game. Such approaches, through the application of machine learning techniques, seek to make possible a form of *programming by demonstration*, where the human author shows the game AI how to play the game. One way to accomplish this is to automatically extract sequences of expert actions as plans for later reuse. A case-based planning (CBP) system maintains a library of plans, a plan base, where a plan used for a situation in the past can be retrieved to be re-applied in a similar situation, possibly after some adaptation.

On-line case-based planning (OLCBP) (Ontañón et al. 2010) has been proposed as an extension to CBP for those domains, such as RT games, where plan generation and execution need to be interleaved in order to react to a partially observable changing environment. Plans need to be monitored in order to be discarded whenever the plan execution engine detects that a given plan can not possibly success and a new one must be devised. In such a situation, OLCBP expands the CBP cycle to include plan execution monitoring, while postponing plan expansion and adaptation.

The combination of learning and hierarchical case-based planning has proved an effective solution for RTS games (Ontañón et al. 2009). Nevertheless, learning hierarchical plans from expert traces also has its limitations. A game AI generated by an expert playing the game may only be as good as the recorded traces, which may contain "holes" in those areas where not enough training has been provided. Identifying and selectively providing training examples to fill such holes may prove a difficult task. Moreover, in some situations, exemplified in this paper, OLCP may fail to react adequately to the changing situations in a RTS games, stubbornly sticking to a plan when a higher priority goal should be taking its attention, or hastily discarding a plan when a concrete non-essential action fails.

In order to solve these problems, this paper proposes to bring the game designer back into the loop, by allowing him to explicitly inject decision making knowledge to complement the knowledge obtained from the traces. Behavior trees are the technology of choice for representing decision making knowledge in commercial videogames. They can be built both by programmers and designers, and provide the capability of reacting to more urgent goals that hierarchical planning lacks of. By providing a natural mechanism for designers to inject knowledge into the plan library, we intend to integrate the best of both worlds: learning from traces and hard-coded rules.

The rest of the paper runs as follows. First two Sections briefly describe the two technologies we propose to

*Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03).

†Currently under grant from Obra Social Fundación "la Caixa".

integrate: hierarchical case-based planning and BTs. Next Section describes the main issues we have identified in CBP systems for games. The next two Sections describe the proposed solution, integrating CBP and BTs, along with detailed examples of their use and the architecture required to support it. The final Section presents related work and concludes the paper.

## On-line Case-based Planning

*On-line case-based planning* (OLCBP) systems (Ontañón et al. 2010) suitable for RTS games are based on a three step process:

- *Case acquisition*: a human expert *plays* the game multiple times, and *traces* are generated containing detailed descriptions of all the events produced throughout each session.
- *Plan extraction*: traces are analyzed to extract plans. Depending on the capacities of the system, this stage may require the supervision of an expert.
- *Plan execution*: once traces have been processed and stored as plans, they are used to play the game using the on-line case-based planning cycle.

Traces store, among other data, sequences of the domain dependent *basic actions* that were carried out during the game. They are manually or automatically analysed afterwards in order to create a *hierarchy of goals* that will be used in the next step. The analysis provides groups of actions that, when executed (in sequence or in parallel), pursue a common *goal*. This process is generalised in order to find higher-level goals composed of subgoals that will be decomposed into basic actions, creating the previously mentioned goal hierarchy. The most general goal will be something like *WinGameGoal*, that would be decomposed, for example, into *DefeatEnemy*, *HaveGoliath*, etc. They will be afterwards broken down into more subgoals and, eventually, into basic actions such as *Move* or *AttackEnemy*.

The goal hierarchy is afterwards *split* to store each node and its *direct successors* (actions or subgoals) as independent *plans*. When comparing with Hierarchical Task Networks (HTN) (Erol, Hendler, and Nau: 1994), goals play the role of *tasks* and plans are similar to *methods*.

Plans are stored in a *case base* indexed both by the goal they pursue and the game state when the expert put them into practice. Plan execution consists of iteratively retrieving plans to fill *open goals* (those that should be executed but have not been expanded yet) until specific actions are reached (leaf nodes). These actions will be returned to the game to be executed by the AI player.

In RTS games, primitive actions are not atomic, so they could take some time to finish or even could fail. In order for the CBP system to track the actions execution, they may be enriched with conditions inspired by ABL (Mateas and Stern 2002): *alive conditions* of an action are those that must be satisfied throughout the execution of the action (otherwise, the action should be stopped), and *success conditions* will be used to determine when an action can be considered to have *correctly* finished.

## Behaviour Trees

The low level decision making located in the units of a strategy game has traditionally used some variant of finite state machines (FSMs). Other approaches, such as scripting languages, that allow a finer grained control of the behaviors have also been applied, and, in the last few years, a new technique known as behavior trees (BTs) has gained momentum. BTs can be seen as an evolution of hierarchical finite state machines (HFSMs) that promotes behavior reuse by replacing the explicit state transitions with predefined procedural mechanisms that allow computing the next state.

Though BTs were initially proposed as a tool for programmers, they are also used by professional *game designers* to create the behaviors of the entities from scratch (Isla 2008; Krajewski 2009). One of the key points is that, as FSMs, BTs open up the possibility of developing tools for creating and editing behaviors with a graphical user interface. A BT is a hierarchical structure where every node can be seen as a behavior. While an inner node is a composite behavior, leaves in the tree represent concrete actions to be performed in the virtual environment. In both cases, the execution of the behavior represented by the subtree (or leaf) may *succeed* or *fail*. At that moment, the parent node of the finished behavior takes the control of the execution and reacts accordingly.

BTs designers have a set of inner nodes that can be used to create complex behaviors. The simplest one is known as *sequential node* (depicted as an arrow in Figure 3), which is composed of a set of children that are executed sequentially; if *all* of them can be successfully accomplished, the complete behavior succeeds, failing otherwise. A different strategy is taken by the *selector node* (depicted as a question mark in Figure 3), that tries to execute sequentially its children and it succeds when *one* of them is executed successfully and terminates with failure when none of them can be accomplished.

In order to promote reusability, behaviors do not include conditions that lead to transitions, but they can be labelled with a guard or condition that must be true for the behavior to be activated. The inner node known as the *static priority list* allows the designer to label every child with a condition. Prior to the execution of the children, the node evaluates the guard and it launches the child only if the guard is true. Otherwise, it tries with the next child. If no guard is satisfied, the complete behavior fails. Otherwise, the result of the node is the result of the first child whose guard is true. A variant of this inner node is the *dynamic priority list* where even when a child behavior has already been selected and its execution has started, the guard of the remaining siblings are continually checked against the world state. Whenever one of them becomes true, the current child behavior is aborted and the higher priority child starts executing.

BTs may be seen as goal structures that represent how high-level goals can be decomposed into lower level ones. In this sense, BTs resemble HTNs, although their purpose is totally different. While HTNs are used to *generate* plans, BTs are used to *store* hand-written plans. BTs can be seen as and-or trees that store a set of plans that a game entity can follow to obtain its goals.

## Issues of Case-based Planning in Games

We have identified three main problems in OLCBP systems: poor reactivity at the plan level, an excessive reactivity at the action level, and the difficulty in fine-tunning these learning by demonstration systems to solve the first two issues. We exemplify these problems in the StarCraft game controlled through Darmok (Ontañón et al. 2007), the reference implementation of an OLCBP system.

Regarding the first one, an OLCBP system presents a problem of reactivity. It builds a global plan by extracting subplans from its case base, finally reaching low level actions to be sent to the game engine. In long term scenarios (macro-management), these low level actions are good enough to make the plan evolve successfully. However, when these low level actions are to be changed or even aborted due to quick changes in the world, the fact that the system sticks to them deteriorates its performance: as long as the alive conditions of the actions remain true, the plan will keep running, even if some recent event suggests that the actions should be cancelled or modified. This is in part due to the way the OLCBP learns plans from the traces it processes. If it had learnt better structured plans, it could create more complex plans for every low level action. However, this would require not only much more traces, but also a vast effort from the expert. Unfortunately, the reactivity of such plans would still be compromised, since the execution model does not rethink plans unless they fail.

This problem is specially important when single units have to be controlled in a low level fashion. When a unit is given an order (low level action) from the plan, it may be the case that, while performing it, it should be somehow modified due to some events observed in the world. It is easy to find such situations in games like StarCraft. For instance, many StarCraft units have special abilities, which is the case of *high templars*. High templars are able to cast *psionic storms* on specific positions. A psionic storm is just a spell that occupies a small area of the map and causes great damage to the units under it.

When Darmok orders a high templar to cast a psionic storm on $\{X, Y\}$ (it issues the *CastPsionicStorm* action), the high templar will just go there and cast the storm. Since Darmok has carried out a previous adaptation step, $\{X, Y\}$ is likely to be a position similar to that of the original traces, which would usually be a region containing many enemy units and almost no ally units (in order for the storm not to kill ally units). However, as the high templar moves to $\{X, Y\}$, the enemy units could have moved around, making casting the storm totally useless, since no enemy unit would be damaged. What is more, if many ally units had moved to $\{X, Y\}$, they could get destroyed by the storm. In such cases it would be convenient for the templar to carefully think about the area on which he has been told to cast the storm. This problem could be solved by just adding a new alive condition to the action so that, if the reached position were not suitable for the storm any more, then the action would get cancelled.

Nevertheless, the idea behind OLCBP is to learn through examples, with as little domain knowledge as possible. In order for OLCBP to be very reactive, it should define countless alive conditions modelling all these scenarios, which would greatly increase the effort put when gathering the domain knowledge. If, for instance, the high templar had to run away from an eventual source of danger (in order not to get killed), new alive conditions should be defined accordingly for the *CastPsionicStorm* action.

On the other side of the spectrum, when thinking at the action level instead of the plan level, OLCBP systems encounters some situations in which they do not perform adequately due to an overly reactive behaviour. When Darmok learns plans from the traces it is fed, it may be the case that some of the learnt plans do not have a proper structure. This problem is in part related to the execution model of Darmok. Darmok associates plans to goals. When one of the individual actions or subplans the main plan is composed of fails, the main plan also fails and Darmok discards it, marking its goal as open again. As a result, Darmok will try to find a *different* plan for that goal, then run it.

There are scenarios in which this behaviour does not provide a good outcome. For example, if a goal *DefeatEnemy* is used to destroy all the enemy's units, plans for that goal will be composed of many actions or subplans, all of them with the purpose of destroying the enemy. In the middle of a battle, however, units are expected to fail their intended actions, since in a battle many situations arise that will prevent them from completing their tasks (e.g., they may get killed). Hence, in these cases, Darmok will always keep failing whatever plans it retrieves for the goal *DefeatEnemy*, due to the continuous failure of individual actions or subplans. In the end, after retrieving and failing several plans, Darmok ends up doing something that somehow makes sense, but of course the quality of the solution is not good at all. This is in fact the main reason why Darmok is not good at battles, usually needing to have twice as many units as the enemy does in order to defeat him (Ontañón et al. 2010).

Finally, it should be noted that OLCBP for strategy games may require a lot of effort to refine the case base. If an expert detects the lack of a reaction in the case base, he should provide a new plan to be learnt. Unfortunately, the only way of doing so is by playing a game emulating the very particular scenario, which may be very difficult due to the complexity and randomness inherent in strategy games.

## Extending Case-based Planning with BTs

Our contribution is to use BTs to overcome the problems described in the previous Section. Firstly, since they let the designer define behaviours at a very low level, complex scenarios can be easily modelled (designers are used to modelling complex behaviours). Secondly, they can be used to model low level actions. Finally, they can be used to model plans for goals in situations where it is convenient to have a better control.

We have extended the OLCBP architecture by means of a tactical layer based on behaviour trees, which is in charge of managing some low level actions and specific goals. Whenever the OLCBP issues a low level action or a goal, a decision is made about whether it is convenient for it to be managed by a behaviour tree; the main idea is to provide be-

haviour trees that, in some scenarios, are expected to be appropriately as substitutes for low level actions or goa

## Tactical Layer for Low Level Actions

In traditional OLCBP systems, low level actions fire them are directly performed by the game API. In ord have a better control over them, we propose to use B implement them. Initially, it may be thought that for e type of action, such as *AttackEnemy* or *Move*, there c be a BT implementing it. These would be action-orie BTs, that is, BTs that pursue a particular goal (the actic self), but also with the ability to change the overall beh if needed. This is just what we explained above: when a is given some order, it should stick to it, but it should als able to change its behavior in certain cases. However, not a realistic approach to expect a single BT to behave p erly when controlling one kind of action (for instance *AttackEnemy* or *Move* actions above) in every possible nario. Our approach proposes to have several BTs for kind of action. These BTs are stored in a BT library cre by designers. Every BT has the same result as the primitive action it is related to and it is labelled with a world description where this BT is expected to behave better. When the tactical layer receives the primitive action that OLCBP system want to execute, it use the BT library to check whether for that particular game state a suitable BT exists. In order to make this choice it uses CBR similarity measures described by (Flórez-Puga et al. 2009). Therefore, at run time, the tactical layer checks the similarity between the current game state and the states provided by all the trees for that kind of action, and the tree with the closest game state gets selected. In case no BT can be retrieved (for instance, if the closest BT is not close enough to a pre-established threshold), the resulting a BT is one with just the basic action on it.

Figure 1 shows how this architecture is actually implemented. For every action Darmok issues, our tactical layer processes it. The tactical layer requests an external *BT Base* to provide a BT to manage the action. Depending on the type of the action, the BT Base retrieves a set of potential BTs (those designed for that type of action); then, it compares the current game state to those of the BTs in the set, and the closest one is returned. This BT then is inserted into the *BT interpreter* that manages the collection of all the BTs currently in execution and is the layer that comunicates with the game. In case a BT could not be retrieved from the BT Base, the low level action is transformed in the simplest possible BT with just a leaf node with that action.

As reveals Figure 1, both the tactical layer and the BT intepreter have the game state available in order to have a richer control over what is going on in the game. Finally, we should mention that this extension to the OLCBP architecture does not need to modify the OLCBP system. By contrast, as we are about to describe, our next extension for managing goals will require some changes on them.

## Tactical Layer for High Level Goals

In those cases where Darmok has not been able to learn effective plans for some goals, BTs may be used as an alternative. The idea is similar to that of low level actions; BTs

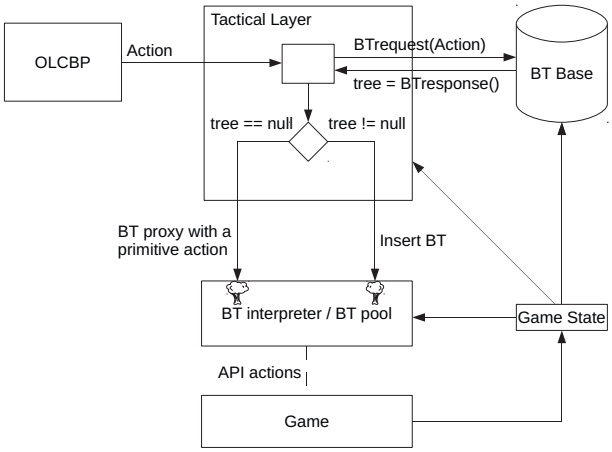


Figure 1: Low level tactical layer

are built by an expert. These BTs contain a description of the game state specifying when it is appropriate to use them. When a BT is retrieved for a particular goal, it will be the tree itself, along with the tactical layer, that will manage the original goal.

Figure 2 shows the architecture of the tactical layer at a high level. Initially, OLCBP proceeds as normal. However, when a goal is detected within a plan, the expansion module checks if that goal must be managed by a BT. In order to perform this check, Darmok asks the BT Base, just the same way as in the low level action scenario. If the BT Base can retrieve a BT for the current game state and goal, the current goal (Goal 3 in the figure) is replaced by a *BT Plan*, and the BT is marked to be sent out. A BT Plan is just a Darmok plan whose execution is not managed by Darmok, but by an external BT (as far as Darmok is concerned, it does not matter whether the plan is managed outside, as long as it provides the same interface as that of a standard Darmok plan). This way, at every game cycle, Darmok does not only issue low level actions (to be processed as described in the previous Section), but also BTs to be run by the tactical layer (they are inserted into the BT interpreter). In case a BT cannot be retrieved from the BT Base, Darmok proceeds as usual, that is, it expands the goal by retrieving from the case base a plan for it. Figure 2 corresponds to the situation in which a BT (called *tree* in the figure) can be retrieved for Goal 3.

## Global Architecture

The previous subsections described how BTs can be used to manage low level actions issued by Darmok as well as goals. The core of the architecture is the tactical layer mentioned above. The tactical layer manages a pool of BTs (BT Pool). This pool contains all the BTs currently in execution. At every game cycle, the tactical layer gives the BTs in the pool some time to run which in turn may provoke actual orders to be sent to the game API. As far as low level action BTs are concerned, they are terminated by the tactical layer whenever the success or failure condition (according to the semantics of Darmok) of the actions they represent are met, since it is at that time when Darmok expects them to stop

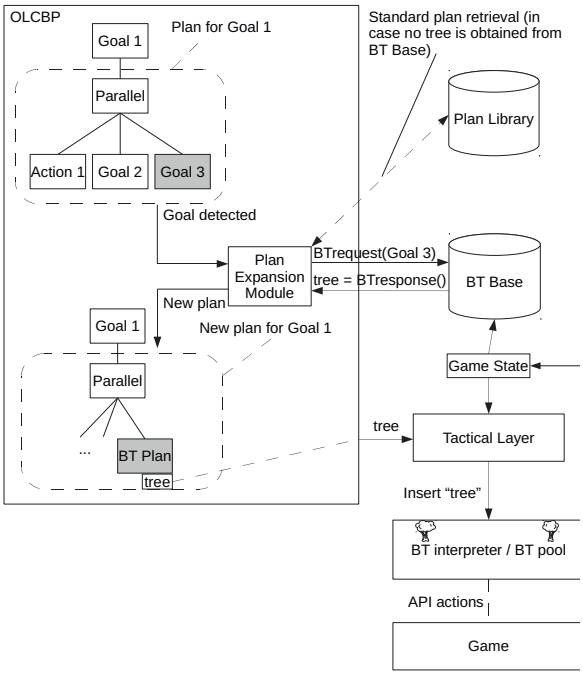


Figure 2: High level tactical layer

running so that it can make the global plan go on. V respect to high-level goals, their interaction with Darı is made through special BT Plans. As long as they vide a way of checking their failure and success conditi Darmok is able to interact with them as normal. Suc conditions for these plans are those of the goal they re sent. The key point here is failure conditions. In a noı situation, the plan would be marked as failed as soon a individual action or subplan of it failed. In this case, h ever, since the whole execution of the plan is managed BT, the failure condition is set by the BT itself, thus allov us to terminate it when we consider appropriate.

## Scenario

In this Section we describe two scenarios in which our posed architecture improves Darmok. One of them is a level action scenario, and the other is a goal plan scenar

### Low Level Action Scenario

In the low level scenario, let us have a look at the *CastPsionicStorm* action. Figure 3 shows a BT implementing the reactive model explained before: the high templar not only thinks about the target position where to cast the storm, but it also reacts to dangerous situations. As explained in previous sections, this BT must also define a game state specifying when it is convenient to use it. Since this is a very standard BT, it can be used for many scenarios, but it is probably best suited for scenarios in which both ally and enemy forces are balanced. In such cases the high templar should try not to damage ally units and run away in case of danger, in order not to lose advantage.

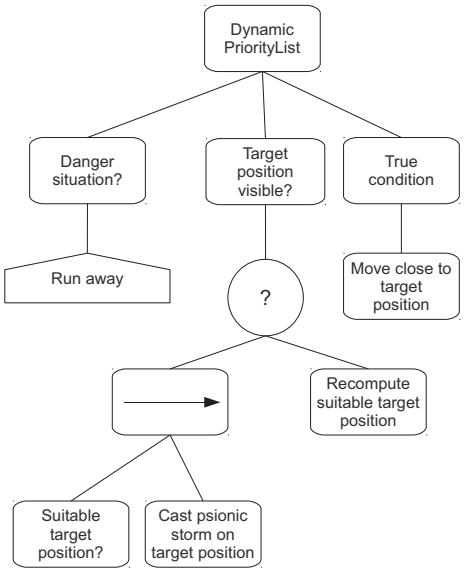


Figure 3: BT for the action *CastPsionicStorm*

In other scenarios, for instance when the ally armies outnumber the enemy, high templars could behave more aggressively or even boldly, so new behaviors could be implemented by another BTs with different associated game states.

When Darmok issues a *CastPsionicStorm* order, our tactical layer processes it. If in the current game state both ally and enemy forces are balanced, the BT of figure 3 could be retrieved. If so, it would be inserted into the BT Pool for subsequent use. If no BT could be found, the *CastPsionicStorm* action would be sent to the game API to be run right away.

### Goal Scenario

When dealing with some goals, Darmok is not expected to behave properly. Take as an example the *DefeatEnemy* goal mentioned in the previous section. When Darmok builds plans for it, they will take the form of many low level actions, maybe organized as sequence or parallel constructions. Even though these plans may look very complex, they would share the same structure, that is, being composed of many low level actions with almost no subplan.

Whenever an individual action or subplan fails, the whole plan is marked as failed, and a new plan has to be found for the goal *DefeatEnemy*. Since this will happen very often, Darmok will not behave properly in this situation.

We can define BTs to handle the *DefeatEnemy* goal in certain scenarios. For instance, we can build a BT to be used in scenarios where the ally armies outnumber the enemy. It is not our purpose to give a detailed description of what such a BT is like, but to show how our architecture overcomes the problem. An effective strategy in such situation consists in just ordering all the ally units to simultaneously attack all the enemy's buildings and units until the enemy is destroyed. As in the low level scenario, the BT contains a description of the game state for the situations in which it should be

used. When the *DefeatEnemy* goal is generated in the plan Darmok builds, Darmok's expansion module tries to find an appropriate BT from the BT Base, by using the current game state and the game state associated to the tree. If the ally armies outnumbered the enemy in the present game state, the described BT could be retrieved and subsequently sent out of Darmok along with the actions that it normally issues. From then on it would be managed by the tactical layer.

## Related Work and Conclusions

There is no, to the best of our knowledge, any other work combining case-based planning and BTs. Nevertheless, this approach can be considered as an example of a more general AI trend of combining domain theory and empirical data: BTs encode a partial view of the expert's domain theory, and cases in the plan library are empirical data. From this point of view, multiple examples of integrations of a domain theory, usually in the form of a set of rules, and case-based reasoning (CBR) can be found in the research literature (Prentzas and Hatzilygeroudis 2007). Some systems take the output of a rule-based component as the input for a case-based one, such as the one described in (Lee 2008), a bank audit system that automatically detects abnormal, irregular, risky, and violated transactions from the standards at the first screening stage, and then applies CBR, which scrutinizes the detected transactions and provides the punishment levels at the second stage. While other systems take an approach, closer to the one presented here, where the output of a case-based module feeds a rule-based one, such as the system described in (Marling and Whitehouse 2001), a medical system for Alzheimer's Disease patients, where the case-based module is invoked to determine whether a neuroleptic drug should be prescribed to a patient and if this is so, the rule-based is invoked to select one of five drugs.

Considering BTs as a kind of planning artefact that stores hand-written plans, we can also find related work on the combination of case-based planning and other planning approaches. The SiN system (Muñoz-Avila et al. 2001) uses a case-based planning algorithm that combines conversational case retrieval with generative planning. SiN can generate plans given an incomplete domain theory by using cases to extend that domain theory, which is given in the form of a planning domain. SiN can also reason with imperfect world-state information by incorporating preferences into the cases. While the case-based module and the domain theory are independently developed in SiN, we propose a more efficient approach by purposely developing a domain theory to fill the holes in the empirical data.

This ongoing research still needs to be validated through empirical evaluation. A key aspect to be evaluated is to what extent the authoring effort of creating BTs to complement an existing plan library can improve the effectiveness of the result game AI.

Regarding future work, we intend to explore possible techniques for facilitating the task of identifying those areas in the plan library that require the expert intervention. At this point, a major drawback of the proposed approach is that the expert needs to analyse the plan library in order to identify those plans, sub-plans or basic actions that require improvement. We envision a computer-assisted identification process, where by generating traces of the AI controlled by the plan library the system can automatically pinpoint actions and goals that usually fail as places for improvement.

## References


Erol, K.; Hendler, J. A.; and Nau:, D. S. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In Hammond, K. J., ed., *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 249–254.

Flórez-Puga, G.; Gómez-Martín, M. A.; Gómez-Martín, P. P.; Díaz-Agudo, B.; and González-Calero, P. A. 2009. Query enabled behaviour trees. *IEEE Transactions On Computational Intelligence And AI In Games* 1(4):298–308.

Isla, D. 2008. Halo 3 - building a better battle. In *Game Developers Conference*.

Krajewski, J. 2009. Creating all humans: A data-driven AI framework for open game worlds. *Gamasutra*.

Lee, G. H. 2008. Rule-based and case-based reasoning approach for internal audit of bank. *Know.-Based Syst.* 21(2):140–147.

Marling, C. R., and Whitehouse, P. 2001. Case-based reasoning in the care of alzheimer's disease patients. In Aha, D. W., and Watson, I., eds., *4th International Conference on Case-Based Reasoning, ICCBR 2001, Proceedings*, 702–715.

Mateas, M., and Stern, A. 2002. A behavior language for story-based believable agents. *IEEE Intelligent Systems* 17(4):39–47.

Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition.

Muñoz-Avila, H.; Aha, D. W.; Nau, D. S.; Weber, R.; Breslow, L.; and Yamal, F. 2001. Sin: integrating case-based reasoning with task decomposition. In *IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence*, 999–1004.

Ontañón, S.; Mishra, K.; Sug, N.; and Ram, A. 2007. Case-based planning and execution for real-time strategy games. In *Proceedings of the 7th international conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*, 164–178. Springer-Verlag.

Ontañón, S.; Bonnette, K.; Mahindrakar, P.; Gómez-Martín, M. A.; Long, K.; Radhakrishnan, J.; Shah, R.; and Ram, A. 2009. Learning from human demonstrations for real-time case-based planning. In Kuter, U., and Muñoz-Avila, H., eds., *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge From Observations*.

Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2010. On-line case-based planning. *Computational Intelligence* 26(1):84–119.

Prentzas, J., and Hatzilygeroudis, I. 2007. Categorizing approaches combining rule-based and case-based reasoning. *Expert Systems* 24(2):97–122.