In P. Gonzáles-Calero & M. Gomez-Martín (ed.), AI for Games: State of the Practice (2011)

Case-Based Reasoning and User-Generated AI for Real-Time Strategy Games

Santiago Ontañón and Ashwin Ram

Abstract Creating AI for complex computer games requires a great deal of technical knowledge as well as engineering effort on the part of game developers. This paper focuses on techniques that enable end-users to create AI for games without requiring technical knowledge by using case-based reasoning techniques. AI creation for computer games typically involves two steps: a) generating a first version of the AI, b) debugging and adapting it via experimentation. We will use the domain of real-time strategy games to illustrate how case-based reasoning can address both steps.

1 Introduction

Over the last thirty years computer games have become much more complex, offering incredibly realistic simulations of the real world. As the realism of the virtual worlds that these games simulate improves, players also expect the characters inhabiting these worlds to behave in a more realistic way. Thus, game developers are increasingly focusing on developing the intelligence of these characters. However, creating (AI) for modern computer games is both a theoretical and engineering challenge. For this reason, it is hard for end-users to customize the AI of games in the same way they currently customize graphics, sound, maps or avatars.

This chapter focuses on techniques to achieve , i.e. on techniques which would enable end-users to author AI for games. This is a complex task, since modern computer games are very complex. For example, () games (which will be the focus of

Santiago Ontañón
Artificial Intelligence Research Institute (IIIA-CSIC), Campus UAB, 08193 Bellaterra (Spain), e-mail: santi@iiia.csic.es

Ashwin Ram
CCL, Cognitive Computing Lab, Georgia Institute of Technology, Atlanta, GA 30332 (USA), e-mail: ashwin@cc.gatech.edu

this chapter) require complex strategic reasoning which includes resource handling, terrain analysis or long-term planning under severe real-time constraints and without having complete information. Because of all of these reasons, programming AI for RTS games is a hard problem. Thus, we would like to allow end-users to create AI without programming.

When a user wants to create an AI, the most natural way to describe the desired behavior is by demonstration. Just let the user play a game demonstrating the desired behavior of the AI. Therefore, a promising solution to this problem are (LfD) techniques. However, LfD techniques have their own limitations, and, given the complexity of RTS games and the lack of strong domain theories, it is not possible to generate an AI by of a few human demonstrations.

The first key idea presented in this chapter is to use (CBR) [1, 9] approaches for learning from demonstration. While it is hard to completely generalize an AI from a set of traces, it is possible to break demonstrations into smaller pieces, which contain specific instances of how the user wants the AI to behave in different situations. For instance, from a demonstration, the sequence of actions the user has used in a specific scenario to destroy an enemy tower can be extracted. These pieces correspond to what in CBR are called *cases*, i.e. concrete problem solving episodes. Each case contains the actions the user wants the AI to perform in a concrete specific situation. Moreover, it is also possible to adapt cases to similar situations. Using a CBR approach to learning from demonstration, we do not need to completely generalize a demonstration. It is enough with being able to adapt pieces of it to similar situations. Moreover, as we will see, classic CBR frameworks need to be extended in order to deal with this problem. In order to illustrate these ideas, we will introduce a system called *Darmok 2*, which is capable of learning how to play RTS games through learning from demonstration.

The second key idea presented in this chapter is that when creating AIs, either using learning from demonstration or directly coding them, it is very hard to achieve the desired result in the first attempt. Thus, by using *self-adaptation* techniques, given a particular AI, it can be automatically adapted fixing some issues it might contain, or making it ready for an unforeseen situation. Again, self-adaptation is a hard problem because of two main reasons: first, how to detect that something needs to be fixed, and second, once an issue has been identified, how to fix it. We will see how this problem can again be addressed by using CBR ideas, and specifically we will present a approach inspired in CBR that addresses this problem. The main idea is to define a collection of *failure-patterns* (which could be seen as cases in a CBR system), that capture which failures to look for and how to fix them. In order to illustrate this idea, we will introduce the *Meta-Darmok* system, which uses meta-reasoning in order to improve its performance at playing RTS games.

In summary, the main idea of this chapter is the following. Authoring AI typically requires two processes: a) creating an initial version of the AI, and b) debugging it. Learning from demonstration is a natural way to help end-users with a), and self-adaptation techniques can help users with b). Moreover, both learning from demonstration and self-adaptation are challenging problems with many open questions. CBR can be used to address many of these open questions and thus, make both

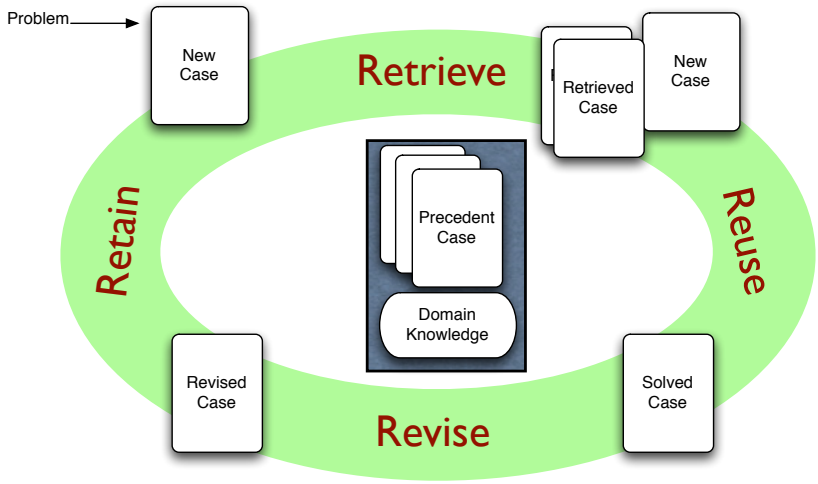

Fig. 1 The case-based reasoning cycle.

learning from demonstration and self-adaptation feasible in the domain of complex computer games such as RTS games.

The remainder of this chapter is organized as follows. Section 2 very briefly introduces CBR. Sections 3 and 4 contain the main technical content of the chapter. Section 3 focuses on CBR techniques for learning from demonstration in RTS games, and Section 4 focuses on CBR-inspired meta-reasoning techniques for self-adaptation. Section 5 concludes the paper and outlines open problems to achieve user-generated AI.

2 Case-Based Reasoning

[1, 9] is a problem solving methodology based on reutilizing specific knowledge of previously experienced and concrete problem situations (*cases*). Given a new problem to solve, instead of trying to solve the problem from scratch, a CBR system will look for similar and relevant cases in its *case base*, and then adapt the solutions in these cases to the problem at hand. A typical in a CBR system consists of a triple: problem, solution and outcome. Where the outcome represents the result of applying a particular solution to a particular problem.

The activity of a case-based reasoning system can be summarized in the CBR cycle, shown in Figure 1, which consists of four stages: *Retrieve*, *Reuse*, *Revise* and *Retain*. In the Retrieve stage, the system selects a subset of cases from the case base that are relevant to the current problem. The Reuse stage adapts the solution of the cases selected in the retrieve stage to the current problem. In the Revise stage, the obtained solution is examined by an oracle, which gives the correct solution (as

in supervised learning). Finally, in the Retain stage, the system decides whether to incorporate the new solved case into the case base or not.

While inductive techniques learn from sets of examples by constructing a global model (a decision tree, a linear discrimination function, etc.) and then forgetting the examples, CBR systems do not attempt to generalize the cases they learn. CBR aligns with the ideas of [2] in machine learning, where all kind of is performed at problem solving time (during the Reuse stage). Thus, CBR systems only need to perform the minimum amount of generalization required to solve the problem at hand. As we will see, this is an important feature, since, for complex tasks like RTS games, attempting to learn a complete model of how to play the game by generalizing from a set of examples might be unfeasible.

3 Generating AI by Demonstration

A promising technology to achieve is *learning from demonstration* [20]. The goal of LfD is to learn how to perform a task by observing an expert. In this section we will first introduce the main ideas of LfD, with a special emphasis on case-based approaches. Then we will explain how can they be applied to achieve user-generated AI by explaining how this is solved in the Darmok 2 system, which has been used to power a social gaming website, Make ME Play ME, based around the idea of user-generated AI.

3.1 Background

Learning from demonstration (also known as *programming by demonstration* or *programming by example*) has been widely studied in artificial intelligence since early times [4] and specially in robotics [11] where lots of robotics-specific algorithms for learning movements from human demonstrations have been devised [14]. The main motivation behind LfD approaches is that learning a task from scratch, without any prior knowledge is a very hard problem. When humans learn new tasks they extract initial biases from instructors or by observing other humans. LfD techniques aim at imitating this process. However, LfD also poses many theoretical challenges.

LfD techniques typically attempt at learning a policy for a dynamic environment. This task cannot be addressed directly with inductive techniques because of several reasons: first, the performance metric might not be defined at the action level (i.e. we cannot create examples to learn using supervised learning); and second, we have the temporal blame assignment problem (it's hard to know which actions to blame or reward in case of failure or success). Without background knowledge, as evidenced by research in reinforcement learning, there is a prohibitively large space to explore.

In the same way as for supervised learning, we can divide the approaches to learning from demonstration in two large groups: *eager approaches* and *lazy approaches*, although work on LfD has focused on eager approaches [10, 4, 15, 20] except for a handful of exceptions like [8]. Eager methods aim at synthesizing a strategy, policy or program, where as lazy approaches simply store the demonstrations (maybe with some pre-processing), and only attempt to generalize when facing a new problem. Let us present some representative work of LfD.

Tinker [10] is a programming by demonstration system, which could write arbitrary Lisp programs (containing even conditionals and recursion). The user provides examples as input/output pairs, where the output is a sequence of actions, and Tinker generalizes those examples to construct generic programs. Tinker allows the user to build incrementally, providing first simple examples and then move on to more complex examples. When Tinker needs to distinguish in between two situations, it prompts the user to provide a predicate that would distinguish them. Tinker is a classic example of an eager approach to LfD, where the system is trying to completely synthesize a program from the examples. Other eager approaches to LfD have been developed both in abstract AI domains [4], as well as in robotics domains [15].

In Tinker, we can already see one of the recurring elements in LfD systems: *traces*. A is the computer representation of a demonstration. It usually contains the sequence of actions that the user executed to solve a given problem. Thus, a pair problem/trace constitutes a demonstration, which is equivalent to a training example in supervised learning.

Schaal [20] studied the benefits of LfD in the context of . He showed that under certain circumstances, the Q-value matrix can be primed using the data from the demonstration and achieved better results than a standard approach. This priming of the value matrix is a way to use the knowledge in the demonstrations to bias subsequent learning, and thus avoid blind search of the search space of policies. However, not all reinforcement learning approaches benefited from using the knowledge in the demonstrations. Notice, moreover, that reinforcement learning also falls into the eager LfD approaches category, since it tries to obtain a complete policy.

Schaal's work evidences another of the important aspects in learning from demonstration: not all machine learning techniques easily benefit from the knowledge contained in the demonstrations.

In this chapter, however, we will focus on lazy approaches to LfD, based on (CBR). Which are characterized for not attempting to learn a general algorithm or strategy from demonstration, but at storing the demonstrations in some minimally generalized form to then adapt them in order to solve new problems. Other researchers have pursued similar ideas, like the work of Floyd et al. [8], which focuses on learning to imitate RoboCup players. Lazy approaches to LfD are interesting, since they can potentially avoid the expensive exploration of the large search space of programs or strategies. While the central problem of eager LfD approaches is how to *generalize* a demonstration to form a program, the central problem of lazy LfD approaches becomes how to *adapt* a demonstration to a new problem.

In order to apply learning from demonstration to a given task, several problems have to be addressed: how to generate demonstrations, how to represent each

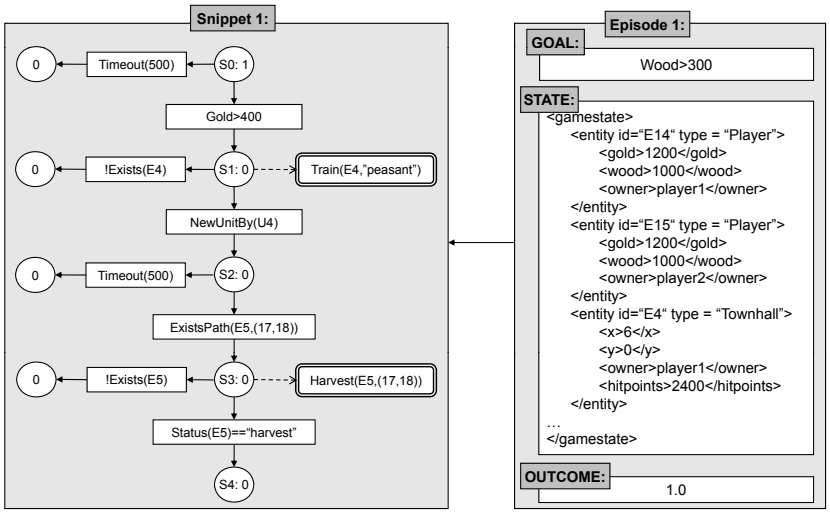

Fig. 2 A case in D2 consisting of a snippet and an episode. The snippet contains two actions, and the episode says that this snippet succeeded in achieving the goal $Wood > 300$ in the specified game state. The game state representation is not fully included due to space limitations.

demonstration (), how to segment demonstrations (which parts demonstrate which tasks and subtasks), which information to extract from the demonstrations, and how this information will be used by the learning algorithm. The remainder of this section will focus on a lazy LfD approach to learn AI in the context of computer games, and on how to address the issues mentioned above.

3.2 Learning from Demonstration in Darmok 2

Darmok 2 (D2) [16] is a real-time [21] system designed to play RTS games. D2 implements the *on-line case-based planning* cycle (OLCBP) as introduced in [17]. The OLCBP cycle attempts to provide a high-level framework to develop case-based planning systems that operate on-line, i.e. that interleave planning and execution in real-time domains. The OLCBP cycle extends the traditional CBR cycle by adding two additional processes, namely *plan expansion* and *plan execution*. The main focus of D2 is to explore learning from unannotated human demonstrations, and the use of adversarial planning techniques. In this section we will focus on the former.

3.2.1 Representing Demonstrations, Plans and Cases

A in D2 is represented as a list of triples $[\langle t_1, G_1, A_1 \rangle, ..., \langle t_1, G_n, A_n \rangle]$, where each triple contains a time stamp t_i game state G_i and a set of actions A_i (that can be empty). The set of triples represent the evolution of the game and the actions executed by each of the players at different time intervals. The set of actions A_i represent actions that were issued at t_i by any of the players in the game. The game state is stored using an object oriented representation that captures all the information in the state: map, players and other entities (entities include all the units a player controls in an RTS game: e.g. tanks).

Unlike in traditional STRIPS [7], actions in RTS games may not always succeed, they may have non-deterministic effects, and they might not have an immediate effect, but be durative. Moreover, in a system like D2 it is necessary to be able to monitor executing actions for progress and check whether they are succeeding or failing. Thus, a typical representation of preconditions and postconditions is not enough. An action a is defined in D2 as a tuple containing 7 elements including success conditions and failure conditions [16]. However, for the purposes of learning from demonstration, precondition and postcondition suffice.

Plans in D2 are represented as *hierarchical* . Petri nets [13] offer an expressive formalism for representing plans that include conditionals, loops or parallel sequences of actions. In short, a petri net is a graph consisting of two types of nodes: *transitions* and *states*. Transitions contain conditions, and link states to each other. Each state might contain *tokens*, which are required to fire transitions. The flow of tokens in a petri net represents it's status. In D2, the plans that will be learned by observing demonstrations consist of hierarchical petri nets, where some states will be associated with sub plans, which can be primitive actions or sub-goals. The left hand side of Figure 2 shows an example of a petri net representing a plan consisting of two actions to be executed in sequence: *Train(E4,"peasant")* and *Harvest(E5,(17,18))*. Notice that the handling of preconditions, postconditions, etc. is handled by the petri net, making the execution module of D2 is a simple petri net simulation component.

When D2 learns plans from demonstrations, each plans is stored as a . Cases in D2 are represented like cases in the Darmok system [17], consisting of a collection of plan *snippets* with *episodes* associated to them. As shown in Figure 2, a snippet is a petri-net, and an episode is a structure storing the outcome obtained when a particular snippet was executed in a particular game state intending to achieve a particular goal. The outcome is a real number in the interval $[0,1]$ representing how well the goal was achieved: 0 represents total failure, and 1 total success.

3.2.2 Learning Plans and Cases from Demonstration

D2's case base is populated by learning both snippets and episodes from human demonstrations. The input to the learning algorithm is one demonstration D (of length n), a player p (D2 will learn only from the actions of player p in the demonstration D), and a set of goals G for which to look for plans. The output is a collection

Demonstration	g_1	g_2	g_3	g_4	g_5
$\langle t_1, G_1, A_1 \rangle$					
$\langle t_2, G_2, A_2 \rangle$					
$\langle t_3, G_3, A_3 \rangle$					
$\langle t_4, G_4, A_4 \rangle$					
$\langle t_5, G_5, A_5 \rangle$					
$\langle t_6, G_6, A_6 \rangle$				✓	
$\langle t_7, G_7, A_7 \rangle$			✓	✓	
$\langle t_8, G_8, A_8 \rangle$		✓	✓	✓	
$\langle t_9, G_9, A_9 \rangle$		✓	✓	✓	✓
$\langle t_{10}, G_{10}, A_{10} \rangle$		✓	✓	✓	✓
$\langle t_{11}, G_{11}, A_{11} \rangle$		✓	✓	✓	✓
$\langle t_{12}, G_{12}, A_{12} \rangle$	✓	✓	✓	✓	✓

Table 1 Goal matrix for a set of five goals $\{g_1, g_2, g_3, g_4, g_5\}$ and for a small trace consisting of only 12 entries (corresponding to the actions shown in Figure 3, $A_{12} = \emptyset$).

of snippets and episodes. The set of goals G can be fixed beforehand for every particular domain, and is equivalent to the list of *tasks* in the framework (thus, the inputs are the same as for the HTN-Maker algorithm). The learning process of D2 can be divided in four main stages: *goal matrix generation*, *generation*, and *hierarchical composition*.

The first step is to generate the *goal matrix*. The goal matrix M is a boolean matrix, where each row represents a triplet in the demonstration D, and each column represents one of the goals in G. $M_{i,j}$ is true if the goal g_j is satisfied at time t_i in the demonstration. An example goal matrix can be seen in Table 1.

Once the goal matrix is constructed, a set of *raw* plans P are extracted from it in the following way:

1. For each goal $g_j \in G$ do
   a. For each $0 < i \leq n$ such that $M_{i,j} \wedge \neg M_{i-1,j}$ do
      i. Find the largest $0 < l < i$ such that $\neg M_{l,j} \wedge (l = 1 \vee M_{l-1,j})$
      ii. Generate a raw plan from the actions executed by player p in the set $A_l \cup A_{l+1} \cup ... \cup A_{i-1}$ and add it to P

For example, five plans could be generated from the goal matrix in Table 1. One for g_1 with actions $A_l \cup ... \cup A_{12}$, one for g_2 with actions $A_l \cup ... \cup A_8$, one for g_3 with actions $A_l \cup ... \cup A_7$, one for g_4 with actions $A_l \cup ... \cup A_6$, and one for g_5 with actions $A_l \cup ... \cup A_9$. Notice that the intuition behind this process is just to look at sequences of actions that happened before a particular goal was satisfied, since those actions are a plan to reach that goal. Many more plans could be generated by selecting subsets of those plans, but since D2 works under tight real-time constraints, currently it learns only a small subset of plans from each demonstration.

Notice that this process is enough to learn a set of raw plans for the goals in G. The snippets will be constructed from the aforementioned sets of actions, and the episode will be generated by taking the game state in which the earliest action in a particular plan was executed. Notice that all plans extracted using this method are

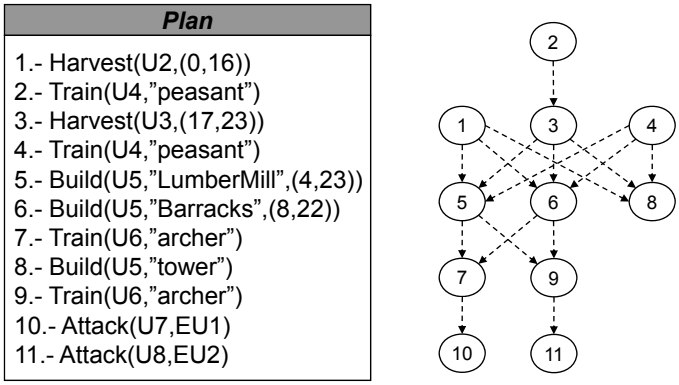

Fig. 3 An example dependency graph constructed from a plan consisting of 11 actions in an RTS game.

plans that succeeded, thus all episodes have outcome equal to 1. However, these raw plans might contain unnecessary actions and would be monolithic, i.e. they will not be decomposable hierarchically into subgoals. Dependency graph generation and hierarchical composition are used to solve both problems.

Given a plan consisting of a partially ordered collection of actions, a [24] is a directed graph where each node represents one action in the plan, and edges represent dependencies among actions. Such a graph is used by D2 to remove unnecessary actions from the learned plans.

Such a graph is easily constructed by checking each pair of actions a_i and a_j in the plan, and checking first of all, if there is any order restriction between a_i and a_j. Only those pairs for which a_i can happen before a_j will be considered. Next, if one of the postconditions of a_i matches any precondition of a_j, and there is no action a_k that has to happen after a_i that also matches with that precondition, then an edge is drawn from a_i to a_j in the dependency graph, annotating it with which is the pair of postcondition/precondition that matched. Figure 3 shows an example (where the labels in the edges have been omitted for clarity). The plan shown in the figure shows how each action is dependent on each other, and it is useful to determine which actions contribute to the achievement of particular goals.

D2 constructs a dependency graph of the plan resulting from using the complete set of actions that a player p executed in a demonstration D. This dependency graph will be used to remove unnecessary actions from the smaller raw plans learned from the goal matrix in the following way:

1. For each plan $p \in P$ do
   a. Extract the subgraph of the dependency graph containing only the actions in p.
   b. Detect which is the subset of actions A from the actions in p such that their postconditions match with the goal of plan p.

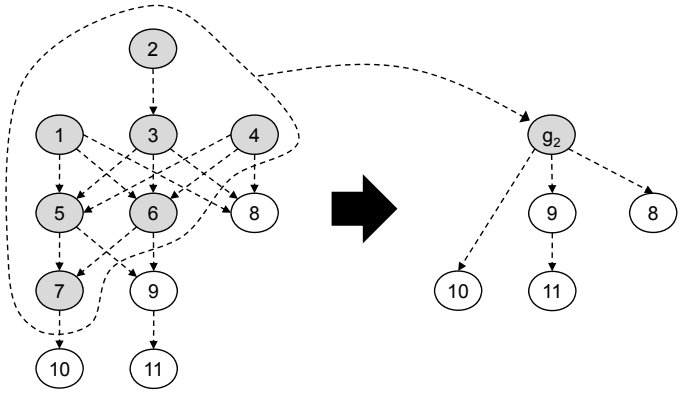

Fig. 4 The nodes greyed out in the left dependency graph correspond to the actions in the plan learned from a goal g_2, after substituting those actions by a single subgoal, the resulting plan graph looks like the one on the right.

c. Remove from p all actions that, according to the subgraph do not contribute directly or indirectly to any of the actions in A.

Moreover, the plan graph provides additional internal structure to the plan, indicating which actions can be executed in parallel, and which ones have to be executed in a sequence. All this information is exploited when generating the petri net corresponding to the plan.

Finally, D2 analyzes the set of plans P resulting from the previous step using the dependency graph to see if any of those plans are a sub-plan of another plan. Given two plans $p_i, p_j \in P$, if the set of actions in p_i is a subset of the set of actions in p_j, D2 assumes that p_i is a sub-plan of p_j, and all the actions in p_i also contained in p_j are substituted by a single sub-goal in p_j. Converting flat plans into hierarchical ones is important in D2, since it allows D2 to combine plans learned from one demonstration with plans learned from another at run time, increasing its flexibility.

Figure 4 shows an example of this process taking the plan graph of the plan learned for goal g_1 in Table 1, and substituting some of its actions by a single subgoal g_2. The actions marked in grey in the left hand side of Figure 4 correspond to the actions in the plan learned for g_2.

Notice that the order in which we attempt to substitute actions by subgoals in plans will result in different final plans. Currently, D2 uses the heuristic of attempting first to substitute larger plans first. However, this issue is a subject of our ongoing research effort. Let us explain how can D2 be used for achieving user-generated AI.

Finally, it is worth to remark that D2's goal is not to learn how to play the game in an optimal way, but to learn the player's strategy. In this sense, it differs from other LfD strategies. For example, the techniques presented by Schaal [20], used LfD only to bias the learning process, which would proceed then to optimize the strategy using standard reinforcement learning.

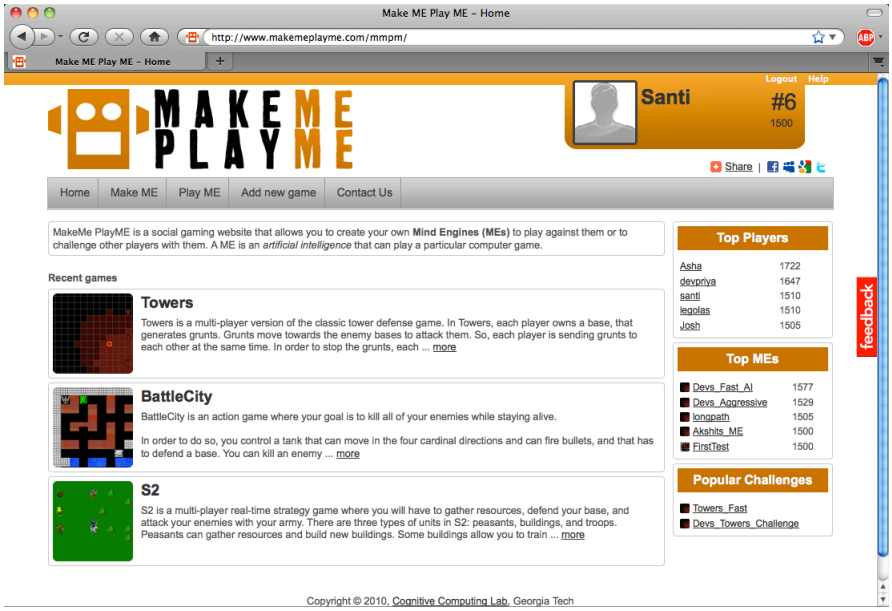

Fig. 5 The game selection page of Make ME Play ME.

3.3 Using Darmok 2 for User-Generated AI: Make ME Play ME

Make ME Play ME (MMPM)[1] is a project to build a social gaming website (see Figure 5) based on the idea of and powered by D2. In MMPM, users do not just play games, they create their own AIs, called *MEs* (Mind Engines). Users train their own MEs, which can play the different games available in the website, and compete against the MEs created by other players. MMPM is not the first web or game where users can create their own AIs and make them compete with others, but it is the first one where users can create their own AIs by demonstration: users do not require programming knowledge, they just have to play a series of games demonstrating the strategy they want their ME to use.

In order to make user-generated AI a reality, many user interaction problems need to be addressed in addition to the technical problems concerning learning from demonstration explained in the previous section. For instance, how to generate demonstrations, or how to visualize the result of learning. In our work on MMPM, we focused on the first of these problems. The latter is still subject of our future work.

The user flow works as follows:

1. Play demonstration games: The user selects a game, configures it (selecting number of players, opponents, map, etc.), and then simply plays. The user can repeat

[1] http://www.makemeplayme.com

this process as many times as desired. For each game played, a will be automatically saved by MMPM.
2. Create a ME: To create a ME, the user first selects which games does he wants to create a ME. Then MMPM lists the set of all available traces for that game (generated in the previous step). The user simply selects a subset of them (which will constitute the set of demonstrations) and the ME is created automatically, without further user intervention.
3. Play with the ME: at this point the user can already wither play against its own ME, or make the ME play with other users' MEs. MMPM lets users challenge other users' MEs. For each ME, a chess-like ELO score is computed, creating a leader-board of MEs. The users are thus motivated to create better MEs, which can climb up the leader boards.

Thanks to the technology developed in D2, the learning process is completely transparent to the user, who only needs to play games. There are no parameters that need to be set by the user. In order to achieve that, all the game-specific parameters of D2 are set before hand. When a new game is added to MMPM, the game creator is the responsible for defining the goal ontology, and for specifying any other parameter that D2 needs to know about the game (e.g. whether the game is turn-based or real-time). Currently, MMPM hosts three different games, but more are on preparation, and it even has the functionality to allow users to upload their own games.

3.4 Discussion

MMPM and D2 allow users to author AIs simply by demonstrations. For instance, in previous work, we showed how it is easy to author an AI for the game Wargus (a clone of WARCRAFT II) by demonstration which can defeat the built-in AI [17]. Moreover, the resulting AIs clearly use the strategies demonstrated by the users. The learning process of D2 is efficient and learning doesn't take any perceptible time. Moreover, the planning algorithms of D2 are also efficient enough to work on real time in the set of games available in MMPM.

However, MMPM and D2 still display a number of limitations, some of which clearly correspond to open problems in learning from demonstration.

- First of all, the approach of D2 is suitable for some kind of games (like RTS games), but breaks when the game becomes more reactive than deliberative. For example, one of the games in MMPM (BattleCity) is a purely reactive game, for which learning plans doesn't make much sense and where a more reactive approach like that in [8] should work much better.
- In addition to demonstrations, some learning from demonstration approaches also allow the user to provide feedback when the system performs the learned strategies in order to continue learning. In the context of D2 and computer games, it would be very valuable to allow such feedback, since it will enable the user to

fine tune the demonstrated strategies. However, this raises both technical and user-interface problems. The main technical problem is related to the delayed blame assignment problem: if the user provides a negative feedback, which of the previous decisions is to blame? Additionally, there would be user interface problems that need to be solved about how can the user provide feedback on the actions being executed by the AI. Specially in RTS games where a large number of actions is executed per second.

- Another issue, subject for our future research and common to all lazy learning approaches, is how to visualize the result of learning. Eager LfD techniques learn a policy or a program which can be displayed to the user in some form. But lazy LfD techniques do not. The only thing that could be displayed are the set of plans being learned. But that can be a potentially very large number of plans, and which does not include the procedure for selecting which plan to select in each situation (which is performed at run-time).
- Clearly, the biggest problem in LfD is how to generalize from a demonstration to a general strategy. Specifically, D2 is based on case-based planning and this problem is translated into how can plans be adapted to new situations. This is a well known problem in the case-based planning community [21], and has been widely studied. In D2 we used an approach with a collection of simplification assumptions which allow D2 to be able to adapt plans in real time [24]. However, those assumption have been designed with RTS games in mind. Finding general ways to adapt plans in an efficient way for other game genres is still an open research issue.

4 Self-Adaptive AI Through Meta-reasoning

Last section focused on techniques to easily generate AI for games. In this section we are going to turn our attention to the complementary problem of how can AI self-adapt to fix any flaws that might have occurred during the learning process, or to adapt the AI to novel situations. This is known as the *adaptive-AI* problem in game AI. This section will provide a brief overview of the problem, and then focus on a solution which combines with , specifically designed for the problem of achieving in games.

4.1 Background

The most widely used techniques for authoring AI in commercial games are scripts [5] and finite-state machines [19] (and recently, behavior trees [18]). These techniques share one feature: once they have been authored, the behavior of the AI will be static: i.e. it will always be the same game after game (ignoring the trivial differences which can be introduced adding randomness). Static behavior can lead to

suboptimal user experience, since, for instance, users might find a hole in the AI and exploit it continuously, or there might be an unpredicted situation or player strategy to which the AI does not know how to react. Trying to address this issue is known as achieving *adaptive game AI* [23].

Basically, adaptive game AI aims at developing techniques which allow for automatic self-modification of the game AI. A potential benefit is for fixing potential failures of the AI, but other uses have been explored, like using self-adaptation for automatically adjusting the difficulty level of games [22]. In this section we are interested in the former, and specifically, in developing techniques which ease user-generated AI. Algorithms which enable AI would enable the users to create AI in an easier way, since some errors in their AI could be automatically fixed by the adaptive AI. Before presenting how CBR can be used to address this issue, let us briefly introduce some brief background and existing work.

Spronck et al. [23] identified a collection of requirements for adaptive game AI. Four are computational requirements: speed, effectiveness, robustness and efficiency; and four are functional requirements: clarity, variety, consistency and scalability. Some of those eight properties, however, apply only to *on-line* techniques for self-adaptation. Our interest in self-adaptive AI concerns allowing user-generated AI, and thus, *off-line* adaptive AI techniques are also interesting. The most basic elements required to define adaptive AI are:

- Representation of the AI: a script, a collection of rules, a set of cases, etc.
- Performance criteria: if the AI has to be adapted, it is in order to improve in some measure. For instance we might want to make the AI better, or better exhibit a particular strategy, or better adjust to the skill level of the player.
- Allowed modifications: which adaptations are allowed? some times, adaptation simply means selecting among a set of given rule sets, some times, the rules or scripts can be actually adapted. This defines the space of possible adaptations.
- Adaptation strategy: which machine learning technique to use.

The most common approach to adaptive AI is letting the user define a collection of scripts or rules that define the behavior of the AI, and then learn which of those scripts, or which subset of rules work better for each particular game situation according to a predefined performance criteria. This approach has been attempted both using reinforcement learning [23] and case-based reasoning [3].

Let us now present a technique which can be combined to the learning from demonstration techniques presented in the previous section, in order to ease the job of a user who wants to create an AI.

4.2 Automatic Plan Adaptation in Meta-Darmok

Meta-Darmok [12] is a system based on the Darmok system [17], which is a predecessor to the D2 system described in the previous section. Meta-Darmok learns

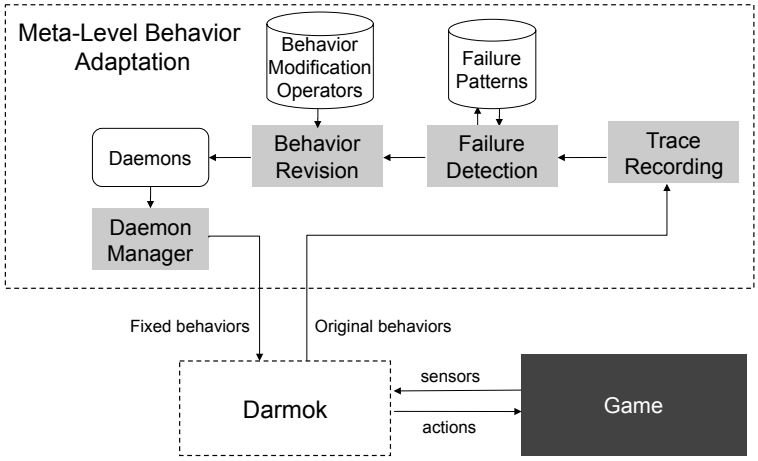

Fig. 6 Meta-reasoning flow of Meta-Darmok.

plans from expert demonstrations and then uses them to play games using case-based planning. Meta-Darmok is designed to play Wargus, and specially to automatically adapt Darmok's learned plans over time. The performance of Darmok, as well as D2, highly depends on the quality of the demonstrations provided by the user. If the demonstrations are poor, Darmok's behavior will be poor. If the expert that Darmok learnt from made a mistake in one of the plans, Darmok will repeat that mistake again and again each time Darmok retrieves that plan. The meta-reasoning approach presented in this section provides Darmok exactly with that capability, resulting in a system called Meta-Darmok, shown in Figure 6.

Meta-Darmok's adaptation approach is based on the following idea: instead of fixing the plans one by one a user can fix a collection of plans by defining a set of typical failures, and associating a fix with them. Meta-Darmok's meta-reasoning layer constantly monitors the plans being executed to see if any of the user-defined failures is happening. If failures occur, Meta-Darmok will execute the appropriate fixes. Moreover, Meta-Darmok's plan fixing happens off-line, after a game has been played. Notice that this approach is radically different from approaches like reinforcement-learning, where the behavior of is optimized by trial an error.

Specifically Meta-Darmok's approach consists of four parts: *Trace Recording*, *Failure Detection*, *Plan Modification*, and the *Daemon Manager*. During trace recording, a holding important events happening during the game is recorded. Failure detection involves analyzing the execution trace to find issues with the executing plans by using a set of [26]. These failure patterns capture the set of user-defined prototypical failures. Once a set of failures has been identified, the failed conditions can be resolved by appropriately revising the plans using a set of *plan modification routines*. These plan modification routines are created using a combination of basic modification operators (called *modops*, as explained later). Specifically, in Meta-Darmok, the modifications are inserted as *daemons*, which monitor for failure con-

ditions to happen during execution when Darmok retrieves some particular plans, but in general, they could be implemented in a different way. A daemon manager triggers the execution of such daemons when required.

4.2.1 Trace Recording

While Meta-Darmok is playing a game, the trace recording module records an *execution trace*, which contains information related to basic events including the name of the plan that was being executed, the corresponding game state when the event occurred, the time at which the plan started, failed or succeeded, and the delay from the moment the plan became ready for execution to the time when it actually started executing. The execution trace provides a considerable advantage in performing plan adaptation with respect to only analyzing the instant in which the failure occurred, since the trace can help localize past events that could possibly have been responsible for the failure.

Once a game finishes, an *abstracted trace* is created from the execution trace that Darmok generates. While the execution trace contains all the information concerning plan execution during the game, the abstracted trace contains only some key pieces of information: those needed to determine whether any failure pattern occurred during the game. The information included in the abstracted trace depends on which conditions are used in the failure patterns. For instance, for the set of patterns used in Meta-Darmok, information about hit points, location, actions being executed by the units, and in which cycles were units created or killed is included.

4.2.2 Failure Detection

Failure detection involves localizing the failures in the trace. Traces can be extremely large, especially in the case of complex RTS games on which the system may spend a lot of effort attempting to achieve a particular goal. In order to avoid the potentially very expensive search process of finding which actions are responsible for failures, the set of user-provided failure patterns can be used [6]. Failure patterns can be seen as a case-based approach to failure detection, and they gretly simplify the blame-assignment process into a search for instances of the particular problematic patterns.

Failure patterns are defined as finite state machines (FSMs) that look for generic patterns in the abstracted trace. An example of a represented as FSM is *Very Close Resource Gathering Location failure* (VCRGLfail) (shown in Figure 7) that detects whether a peasant is gathering resources at a location that is too close to the enemy. This could lead to an opening for enemy units to attack early. Other examples of failure patterns and their corresponding plan modification operators are given in Table 2. Each failure pattern is associated with modification routines. When a failure pattern generates a match in the abstracted trace, an instantiation of the failure pattern is created. Each instantiation contains which were the particular events in the

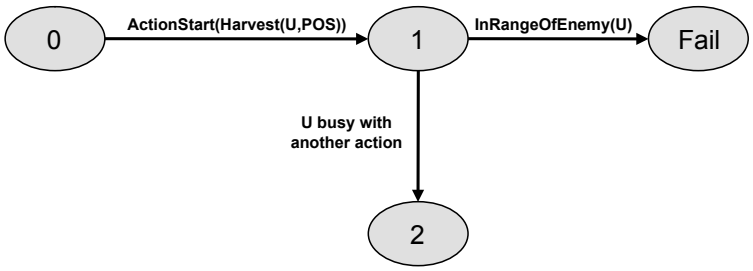

Fig. 7 FSM corresponding to the failure pattern VCRGLfail. This pattern detects a failure if the FSM ends in the *Fail* state. When a unit is ordered to start harvesting, the FSM moves to state 1, if the unit stops harvesting, it will move to state 2, and only when the unit gets in range of an enemy unit while harvesting, the FSM will end in the *Fail* state.

Failure Pattern	**Plan Modification Operator**
Resource Idle failure (e.g., resource like peasant, building, enemy units could be idle)	Utilize the resource in a more productive manner (for example, send peasant to gather more resources or use the peasant to create a building that could be needed later on)
Very Close Resource Gathering Location Failure	Change the location for resource gathering to a more appropriate one
Inappropriate Enemy Attacked failure	Direct the attack towards the more dangerous enemy unit
Inappropriate Attack Location failure	Change the attack location to a more appropriate one
Basic Operator failure	Adding a basic action that fixes the failed condition

Table 2 Some example failure patterns and their associated plan modification operators.

abstracted trace that matched with the pattern. This is used to instantiate particular plan modification routines that are targeted to the particular plans that were to blame for the failure.

4.2.3 Plan Modification

Once the cause of the failure is identified, it needs to be addressed through the appropriate modifications (modops). Modops can take the form of inserting or removing steps at the correct position in the failed plan, or changing some parameter of an executing plan. Each failure pattern has a sequence of modops associated with it. This sequence is called a *plan modification routine*.

Once the failure patterns are detected from the execution trace, the corresponding plan modification routines and the failed conditions are inserted as daemons for the plan in which these failed conditions are detected. The daemons act as a meta-level reactive plan that operates over the executing plan at runtime. The conditions for the

failure pattern become the preconditions of the daemon and the plan modification routine consisting of basic modops become the steps to execute when the daemon executes. The daemons operate over the executing plan, monitor their execution, detect whether a failure is about to happen and repair the plan according to the defined plan modification routines.

Notice that Meta-Darmok does not directly modify the plans in the case base of Darmok, but reactively modifies those plans when Darmok is executing them. In the current system, we have defined 20 failure patterns and plan modification routines for Wargus. The way Meta-Darmok improves over time is by accumulating the daemons that the meta-reasoner generates (which are associated to particular maps). Thus, over time, Meta-Darmok improves performance by learning which combination of daemons improves the performance of Darmok for each map. Using this approach we managed to multiply by two the win-ratio of Darmok against the built-in AI of Wargus [12].

The adaptation system can be easily extended by writing other patterns of failure (as described in [25]) that could be detected from the abstracted trace and the appropriate plan modifications to the corresponding plans that need to be carried out in order to correct the failed situation.

4.3 Using Meta-Darmok for User-Generated AI

In order to use Meta-Darmok for , we integrated Meta-Darmok into a behavior authoring environment, which we call an *intelligent* IDE (iIDE). Specifically, we integrated authoring by demonstration, visualization of the behavior execution, and self-adaptation through meta-reasoning. The iIDE allows the game developer to specify initial versions of the required AI by demonstration them instead of having to explicitly code the AI. The iIDE observes these demonstrations and automatically learns plans (that we will call *behaviors* in this section) from them. Then, at runtime, the system monitors the performance of these learned behaviors that are executed. The system allows the author to define new failure patterns on the executed behavior set, checks for pre-defined failure patterns and suggests appropriate revisions to correct failed behaviors. This approach to allow definition of possible failures with the behaviors, detecting them at run-time and proposing and allowing a fix selection for the failed conditions, enables the author to define potential failures within the learnt behaviors and revise them in response to things that went wrong during execution.

Here we will focus only on how meta-reasoning is integrated into the iIDE (for more details about the iIDE reported here, see [25]). In order to integrate Meta-Darmok into the iIDE, we added to functionalities:

- *Behavior Execution Visualization and Debugging*: The iIDE presents the results of the executing behaviors in a graphical format, where the author can view their progress and change them. The author can also pause and fast-forward the game to whichever point he chooses while running the behaviors, make a change in the behaviors if required and start it up again with the new behaviors to see the

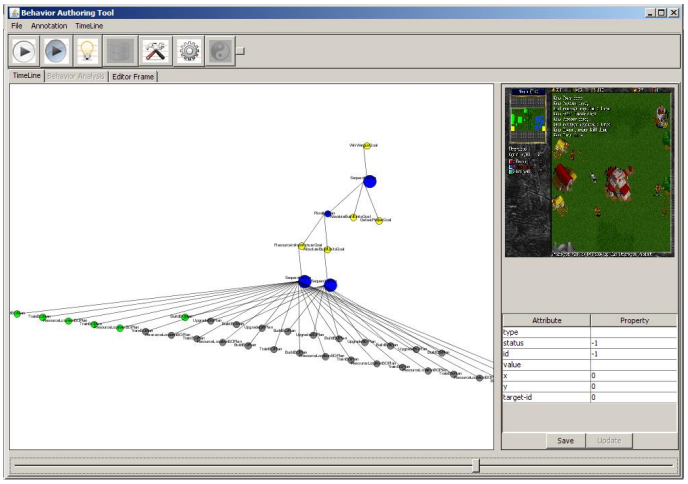

Fig. 8 A screenshot of the iIDE, showing the behavior execution visualization interface.

performance of the revised behaviors. The capability of the iIDE to fast forward and start from a particular point, further allows the author to easily replicate a possible bug late in the game execution and debug it. Figure 8 shows a screenshot of the execution visualization view in the iIDE, showing an executing behavior (including the current state of all the sub-goals and actions).

- *Failure Detection and Fixing*: The iIDE authoring tool allows the author to visualize relevant chronological events from a game execution trace. The data allows the author define new *failure patterns* by defining combinations of these basic events and pre-existing failure conditions. Each failure pattern is associated with a possible fix. A fix is basically a proposed modification for a behavior that fixes the error detected by the failure pattern. When a failure pattern is detected, the iIDE suggests a list of possible fixes, from which the author can select an appropriate one to correct the failed behavior. These techniques were also previously developed by us in the context of believable characters [26].

Figure 9 shows an overview of how all the components fit together to allow the author to edit a proper behavior set for the game. The iIDE controls Meta-Darmok by sending the behaviors that the author is creating. Meta-Darmok then, runs the behaviors in the game, and generates a trace of what happened during execution. This trace is sent back to the iIDE so that proper information can be shown to the author. Basically, the iIDE makes the functionality of Meta-Darmok (learning from demonstration and self-adaptation through meta-reasoning) accessible to the user to allow easy behavior authoring.

We evaluated this iIDE with a small set of users and the conclusions found that users felt authoring by demonstration was more convenient than writing our behaviors through coding. Notice that it took no more than 25 minutes to generate behaviors to play Wargus (that includes the time to generate the demonstration playing

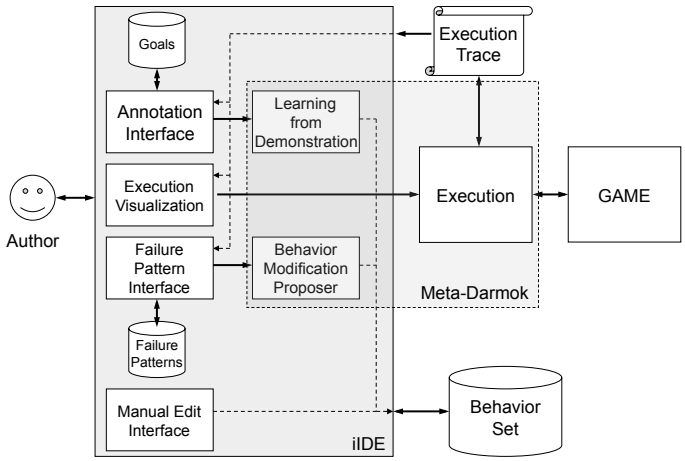

Fig. 9 Overview of how the iIDE interacts with the author and the game.

plus trace annotation). However, since Meta-Darmok is based on the old Darmok system, which required trace annotation, users felt that annotation was difficult, since it was difficult to remember the actions they had performed.

Concerning self-adapting behaviors using meta-reasoning, users felt it was a very useful feature. However, they had problems with our specific implementation because the requirement that a failure pattern should occur inside the game in order to be able to define it was a setback. Users could think of simple failure patterns which they would like to add without having to even run the game. However, despite of these problems, users were able to successfully define failure patterns. A more comprehensive explanation of the evaluation results can be found at [25].

4.4 Discussion

The techniques presented in this section successfully allow a system to detect problems in the behaviors being executed by the AI and fix them. However, we do so at the expense of letting the user be the one who specifies the sets of failures to look for, in the form of failure patterns.

Clearly, the problem of self-adapting AI contains two problems: detecting that something has to be changed, and change it. Both of them are, as of today, open problems. In our approach, we used a domain-knowledge intensive approach for detecting that something has to be changed, by letting the user specify domain dependent failure-patterns. Which, for the purposes of user-generated AI worked adequately, but at the expense of making the user having to manipulate concepts like conditions, actions, etc. when defining the failure patterns.

However, detecting that something has to be changed is a challenging problem. For example, in techniques such as dynamic scripting [23], we need to define a performance metric. In case the goal is just to adapt an AI to make it stronger or weaker, a performance metric is easy to define (percentage of wins, or some related measure should suffice). However, when the goal is to adapt an AI to better behave the way the user wants, this is harder, and interfaces to allow the user to provide feedback are required.

In general, for behavior creation, as we explained above, LfD is an intuitive way in which a user can provide domain knowledge. The iIDE presented in this section is an attempt to achieve the same thing for the problem of adapting an already created behavior. Other strategies which can be used are direct positive or negative reinforcement from the user when behaviors are being executed. Although this requires the user to constantly provide feedback, where as failure patterns can be given once and be reused multiple times.

5 Conclusions

This chapter has focused on techniques to achieve user-generated AI. We have presented two complementary techniques, learning from demonstration and self-adaptation, which combined can help the task of an end-user who wants to author AI without programming. In particular, the learning from demonstration technique presented in this chapter has been used to power the social gaming website *Make ME Play ME*, in which users compete to create good AIs by demonstration.

The work presented in this chapter indicates that enable user-generated AI, we need to address both technical and user-interface problems. D2 and Meta-Darmok are attempts at addressing the technical challenges, and Make ME Play ME and the iIDE are attempts at addressing the user-interface problems.

Moreover, although the techniques presented in this paper are useful for achieving user-generated AI, we have listed a number of open problems that need to be solved before they can be applied to a broad variety of games by end-users. In the case of learning from demonstration, the two main open problems of the approaches presented here are how to present the learned strategies to the user in a human-understandable way, and how to achieve generic and efficient plan adaptation (for adapting learned plans to new situations). Concerning self-adaptation, the main problems of a failure-pattern-based approach are enabling the easy definition of failure patterns for end-users in an intuitive way.

References

1. Aamodt, A., Plaza, E.: Case-based reasoning: Foundational issues, methodological variations, and system approaches. Artificial Intelligence Communications **7**(1), 39–59 (1994)
2. Aha, D. (ed.): Lazy Learning. Kluwer Academic Publishers, Norwell, MA, USA (1997)

3. Aha, D.W., Molineaux, M., Ponsen, M.J.V.: Learning to win: Case-based plan selection in a real-time strategy game. In: ICCBR, pp. 5–20 (2005)
4. Bauer, M.A.: Programming by examples. Artificial Intelligence **12**(1), 1–21 (1979)
5. Berger, L.: Scripting: Overview and code generation. In: S. Rabin (ed.) AI Game Programing Wisdom, pp. 505–510. Charles River Media (2002)
6. Cox, M.T., Ram, A.: Introspective multistrategy learning: on the construction of learning strategies. Artificial Intelligence **112**(1-2), 1–55 (1999)
7. Fikes, R., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intellicence **2**(3/4), 189–208 (1971)
8. Floyd, M.W., Esfandiari, B., Lam, K.: A case-based reasoning approach to imitating robocup players. In: Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society (FLAIRS), pp. 251–256 (2008)
9. Kolodner, J.: Case-based reasoning. Morgan Kaufmann (1993)
10. Lieberman, H.: Tinker: a programming by demonstration system for beginning programmers. In: Watch what I do: programming by demonstration, pp. 49–64. MIT Press, Cambridge, MA, USA (1993)
11. Lozano-Pérez, T.: Robot programming. In: Proceedings of IEEE, vol. 71, pp. 821–841 (1983)
12. Mehta, M., Ontañón, S., Ram, A.: Using meta-reasoning to improve the performance of case-based planning. In: Case-Based Reasoning, ICCBR-2009, no. 5650 in Lecture Notes in Artificial Intelligence, pp. 210–224. Springer-Verlag (2009)
13. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
14. Nakanish, J., Morimoto, J., Endo, G., Cheng, G., Schaal, S., Kawato, M.: Learning from demonstration and adaptation of biped locomotion with dynamical movement primitives. In: Workshop on Robot Learning by Demonstration, IEEE International Conference Intelligent Robots and Systems (2003)
15. Nicolescu, M.N.: A framework for learning from demonstration, generalization and practice in human-robot domains. Ph.D. thesis, University of Southern California, Los Angeles, CA, USA (2003). Adviser-Maja J. Mataric
16. Ontañón, S., Bonnette, K., Mahindrakar, P., Gómez-Martín, M., Long, K., Radhakrishnan, J., Shah, R., Ram, A.: Learning from human demonstrations for real-time case-based planning. IJCAI-09 Workshop on Learning Structural Knowledge From Observations (STRUCK-09) (2009)
17. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: On-line case-based planning. Computational Intelligence Journal **26**(1), 84–119 (2010)
18. Puga, G.F., Gómez-Martín, M.A., Díaz-Agudo, B., González-Calero, P.A.: Dynamic expansion of behaviour trees. In: AIIDE (2008)
19. Ryan Houlette, D.F.: The ultimate guide to fsms in games. In: S. Rabin (ed.) AI Game Programing Wisdom 2, pp. 283–302. Charles River Media (2003)
20. Schaal, S.: Learning from demonstration. In: Advances in neural information processing systems 9, pp. 1040–1046. MIT press (1997)
21. Spalazzi, L.: A survey on case-based planning. Artificial Intelligence Review **16**(1), 3–36 (2001)
22. Spronck, P., Kuyper, S.I., Postma, E.: Difficulty scaling of game AI. In: Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004), pp. 33–37 (2004)
23. Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., Postma, E.: Adaptive game AI2 with dynamic scripting. Machine Learning **63**(3), 217–248 (2006)
24. Sugandh, N., Ontañón, S., Ram, A.: On-line case-based plan adaptation for real-time strategy games. In: AAAI 2008, pp. 702–707 (2008)
25. Virmani, S., Kanetkar, Y., Mehta, M., Ontañón, S., Ram, A.: An intelligent IDE for behavior authoring in real-time strategy games. In: AIIDE (2008)
26. Zang, P., Mehta, M., Mateas, M., Ram, A.: Towards runtime behavior adaptation for embodied characters. In: IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence, pp. 1557–1562. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)