Selecting Robust Strategies in RTS Games via Concurrent Plan Augmentation

Abdelrahman Elogeel♣ Andrey Kolobov◇ Matthew Alden♣ Ankur Teredesai♣

♣Institute of Technology
University of Washington, Tacoma
{elogeel,mealden,ankurt}@uw.edu

◇Microsoft Research
Redmond, WA, USA
akolobov@microsoft.com

ABSTRACT

The multifaceted complexity of real-time strategy (RTS) games forces AI systems to break down policy computation into smaller subproblems – strategic planning, tactical planning, reactive control, and others. To further simplify planning at the strategic and tactical levels, state-of-the-art automatic techniques for this task, such as case-based planning, produce deterministic plans for what is inherently an uncertain environment, and fall back on replanning when the game situation disagrees with the constructed plan. A major weakness of this approach is its lack of robustness: repairing a failed plan is often impossible or infeasible due to real-time computational constraints, causing a game loss. This paper presents a technique that selects a robust RTS game strategy by using ideas from contingency planning and by exploiting action concurrency of these games. Specifically, starting with a strategy and a linear tactical plan that realizes it, our algorithm identifies the plan's failure modes using available game traces and adds concurrent branches to it so that these failure modes are mitigated. In this manner, our approach may train an army reserve *concurrently with* an attack on the enemy, as defense against a possible counterattack. After augmenting each strategy from an available library (e.g., learned from human demonstration) our approach picks one with the most robust augmented tactical plan. An extensive evaluation on the popular RTS games of StarCraft and Wargus, which shares its engine with several other games, shows that concurrent augmentation significantly improves win rate and lets the agent prevail in scenarios where baseline strategy selection consistently leads to a loss.

1. INTRODUCTION

Ten years after real-time strategy (RTS) games were proposed as a challenge for AI [2], computational state of the art in them is still well short of the human performance level [13]. Among the aspects that make RTS games so difficult for computers to play well are the complexity and partial observability of the worlds they depict, their adversarial nature, and the necessity to plan in real time with durative actions and a high degree of concurrency. In an attempt to make RTS game policy computation more tractable, researchers have broken this problem into subproblems varying by planning timescale (strategy, tactics, reactive control) and by their function (opponent modeling, economy/resource management, infrastructure construction, and others). Some of them, e.g., planning at micro-timescales and terrain analysis, have been tackled with relative success using reinforcement and machine learning techniques [13], which affected at least one application area that shares characteristics with RTS games, combat environments simulation [11]. Nonetheless, AI agents in modern commercial RTS games are still far from intelligent. A major reason for this is that, despite progress in other areas, strategic and high-level tactical planning in RTS games have remained largely open problems; in this paper, we focus on them.

Selecting game strategy and high-level tactics are tightly intertwined tasks. On one hand, we would like to choose a strategy, i.e., a sequence of subgoals to achieve to win the game, for which a tactical plan that attains these subgoals is easy to find. On the other hand, tactical plans with a good chance of winning may be available only for specific subgoal sequences, thereby implicitly constraining strategy selection. Case-based planning [6] and the On-line Case-Based Planning (OLCBP) architecture derived from it [12] are among the few automatic approaches that have attempted to solve these planning problems jointly in RTS games.[1] Taking a set of subgoal specifications and gameplay logs (e.g., generated by humans) as inputs, OLCBP builds a hierarchical task network (HTN) [15] from them. It uses this HTN to produce a deterministic strategic and tactical plan at the start of a game. It follows this plan until the game state deviates from the plan's expectations, in which case OLCBP replans. While a significant step towards automating strategy selection, OLCBP doesn't explicitly attempt to maximize the chance of winning the game. Moreover, its reliance on replanning causes a game loss when the existing plan fails and no new one can be found [9].

The technique we present in this paper, which we call *concurrent plan augmentation (CPA)*, mitigates the drawbacks of OLCBP by aiming to select the strategy with a set of tactical plans that will empirically least likely require replanning when executed. We implement CPA in an architecture called *ROLCBP (Robust OLCBP)*. Like OLCBP, ROLCBP identifies promising strategies and tactical realizations for them from game logs *without using a conventional planner*. Importantly, the number of such distinct strategies in RTS games is usually small, allowing ROLCBP to analyze each such strategy with its tactical plans and try to address the plan's weaknesses. For each strategy, ROLCBP first identifies the likely steps at which its tactical plan will fail. Then ROLCBP searches for plans that can be executed *concurrently with* parts of the strategy's main tactical plan and can decrease its empirical chance of failure. For example, if the original plan calls for immediately attacking the enemy and the attack often fails in simulation, ROLCBP might propose to train an army reserve simultaneously with the attack so that the player's base does not get overrun by the opponent's counterattack if the player's attack peters out. Augmenting the tactical plans for each strategy and assessing their failure probability identifies

[1] See Related Work for a discussion of a hard-coded approaches.

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May, 4–8, 2015, Istanbul, Turkey.*
Copyright © 2015, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

the most empirically robust strategy, while storing the augmented plans reduces the need to replan in critical situations. The ability to produce robust tactical behavior automatically in the presence of action concurrency and without reliance on conventional deterministic or probabilistic planners is what makes ROLCBP functionally novel.

We conduct an extensive empirical evaluation of ROLCBP on the popular games of *StarCraft* and *Wargus* (an open-source clone of *Warcraft II* that shares its game engine with several other real-time strategies). It demonstrates CPA's broad applicability and shows that, thanks to CPA, ROLCBP not only has a significantly higher win rate against built-in AI than OLCBP, but can also win on maps where OLCBP always loses. Moreover, whenever CPA does not affect the strategy choice, it still improves the win rate by making tactical plans for the selected strategy much less failure-prone. Last but not least, the experiments reveal that ROLCBP easily wins against opponents very dissimilar from those encountered during training.

To sum up, our paper makes the following contributions:

- We present concurrent plan augmentation (CPA), a technique that helps pick robust strategic and tactical plans.
- We implement CPA in an OLCBP-based architecture called ROLCBP and extensively test it on complex commercial RTS games. The results indicate that CPA provides quantitative as well as qualitative win rate gains by improving decision-making at both strategic and tactical levels.

2. BACKGROUND

Real-time strategy games. RTS games are a genre that typically involves managing resources, building infrastructure, training an army, and ultimately defeating other players by destroying their units and infrastructure. *StarCraft*, an RTS developed by Blizzard Entertainment™, and *Wargus*, an open-source clone of *Warcraft II*, are popular games that we use as experimental testbeds in this paper. In *StarCraft*, every player controls one of three warrior races: Zerg, Protoss, or Terran. In *Wargus*, there are two of them: humans and orcs. In both games, warriors of each race differ in their combat characteristics and in the resources required to produce them. Accordingly, good choices of strategy and tactics vary among races, and also depend on the world configuration, called a *map*, where the game is played.

Strategic planning in RTS games. The objective of RTS game AI is to come up with a *policy* — a mechanism for deciding which actions to initiate at any point in time, given the agent's current belief about the state of the world and actions (both of the agent and of the opponents) that are already being executed. RTS games have several distinctive attributes that put them beyond the capabilities of most standard game tree search and reinforcement learning techniques: the size, complexity, and non-determinism of the game environment, durative and concurrent action execution, partial observability, and others. In fact, even simply finding action outcome trajectories that lead to victory under simplifying assumptions is difficult for modern planners due to actions having durations and being concurrent. For this reason, finding a policy in RTS games is broken down into several layers of abstraction, which effectively decomposes the problem into smaller, more manageable planning tasks. Strategic planning is the highest level in this abstraction hierarchy. It chooses a set of subgoals that need to be achieved to win the game, and the order in which they need to be achieved. While some of the lower levels of the abstraction hierarchy, e.g., unit control, can be tackled with automatic techniques, good strategic behavior in real-time games is very difficult to come by with existing planning technology and, as a result, is either scripted in game AI or relies on adapting plans gleaned from human gameplay. On-Line case-based planning is an example of the latter approach.

On-line case-based planning. OLCBP [12] is a planning architecture derived from hierarchical task networks (HTN) [15] and case-based reasoning. Its inputs, along with a game description, are a set of subgoal specifications and gameplay logs (e.g., generated by humans). OLCBP operates in a cycle with two main phases, behavior acquisition and online expansion-execution (Figure 1). During behavior acquisition, it heuristically matches up parts of the logged game traces with the subgoals, thereby identifying both the strategies used in the game and the tactical plans for achieving the subgoals. After some additional processing, OLCBP stores the extracted plans, called *cases*, in its *case base*.

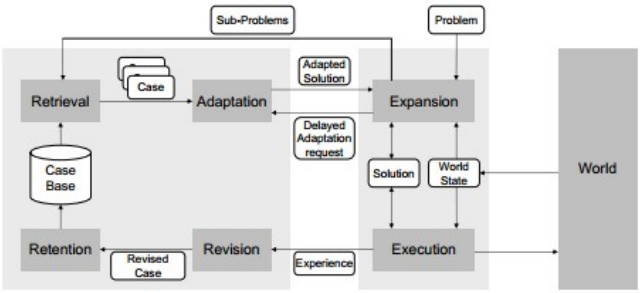


Figure 1: OLCBP cycle. Image taken from [12]

OLCBP assumes its opponents to be relatively static and views strategy and tactics selection as a classical planning problem with a possibility of replanning. OLCBP's expansion-execution step is responsible for dividing this planning problem into subproblems using an HTN constructed from the provided subgoals with the help of the game logs. The chosen strategy is passed to the retrieval submodule, which selects and adapts tactical plans for it from the case base. OLCBP executes these plans until the game situation deviates from the predicted one. In this case, it looks for a different tactical plan, strategy or both, given the player's current situation. Reliance on replanning is OLCBP's significant weakness. Replanning takes time, but even a small delay in acting may mean losing the game. In addition, OLCBP's case base my not contain a back-up plan for unforeseen circumstances, also leading to a game loss.

Contingency planning. In contingency planning [14], an agent situated in a partially observable non-deterministic environment needs to devise a policy for reaching the goal from any state in which the agent may end up. Unlike in (partially observable) Markov Decision Processes (MDPs), the agent does not know the probabilities of its actions' outcomes, only the sets of possible outcomes themselves. In this paper we show that, despite having been considered for non-adversarial settings so far, contingency planning can be potent in RTS game AI as well.

3. CONCURRENT PLAN AUGMENTATION

To address the drawbacks of existing strategy selection approaches such as OLCBP, we propose a technique called *concurrent plan augmentation* (CPA). CPA's high-level idea is to start with a tactical plan for each available strategy, analyze the ways in which the game situation can deviate from it, and come up with *contingent* plan branches that prevent the deviations from becoming

catastrophic, to be executed *in parallel* with the main plan. Repeating these steps iteratively builds a partial *policy* for every strategy. This policy's ability to avoid failure is taken as a measure of the corresponding strategy's robustness; based on it, a strategy selection framework, e.g., OLCBP, can choose a strategy for playing the game. At first glance, CPA is reminiscent of contingency planning, but has several important differences:

- Most contingency planning algorithms assume that catastrophic failures cannot happen, i.e., that it is possible to reach the goal (win the game) starting from any state. Even the few exceptions such as [10] assume that failures can be fully avoided from any state. In contrast, failures impossible to avoid with certainty are common in RTS games, e.g., due to opponents' actions. For this reason, contingency planners can usually rely on replanning, whereas CPA attempts to preempt catastrophic events.
- Contingency planning mostly considers settings with no action concurrency, where actions must be executed one after another. CPA relies on concurrency for its operation.
- Unlike in contingency planning, which has no widely accepted measure of policy quality, characteristics of RTS games suggest gauging policy quality with its ability to avoid failures. CPA uses one such measure, a policy's failure probability. Although in a game this measure is only heuristic, since it depends on the training opponent's play, our experiments show that it works well in practice. In games with no draws, as is the case for most RTS games, failure probability minimization can be alternatively viewed as maximizing the win rate.

As described, CPA leaves many details unspecified. How to find contingent plan branches? When should a contingent branch's execution begin? Should the policy tree be computed eagerly before the start of the game or lazily as the game progresses? Their answers depend on the RTS game and strategy selection framework to which CPA is applied. This paper combines CPA with the OLCBP framework to produce the *Robust OLCBP* algorithm, which we describe next.

4. ROBUST OLCBP

The operation of CPA with OLCBP in an architecture we call Robust OLCBP (ROLCBP) is broken down into an offline stage (Algorithm 1) and an online stage (Algorithm 2). The offline stage is devoted to training, while the online stage exploits the learned knowledge for strategy and tactics selection. Throughout our explanations, we will be referring to ROLCBP's pseudocode, denoting line l of Algorithm a as "line $l : a$". In the following section, we illustrate the pseudocode's operation with an example from *StarCraft*.

Offline stage. At this stage, ROLCBP relies on the OLCBP's machinery to learn a set of strategies $\mathcal{S}$ and a case base of tactical plans $\mathcal{P}$ based on a human-specified set of subgoal descriptions $\mathcal{G}$, action descriptions $\mathcal{A}$, and a set of gameplay logs (line 10:1). The details of this step [12] are beyond the scope of our paper but, importantly for the rest of ROLCBP, each plan in the resulting case base $\mathcal{P}$ achieves a subgoal from $\mathcal{G}$ assuming some other subset of $\mathcal{G}$'s goals has already been achieved. Generally, $\mathcal{P}$ contains several plans that achieve a given subgoal.

After tactical plan extraction, ROLCBP gathers statistics for policy augmentation. Using a simulator for the RTS game to which it is applied, ROLCBP reproduces the initial conditions of each plan $P \in \mathcal{P}$ (i.e., generates a game state in which all subgoals that P assumes to be achieved have indeed been achieved). Then, with the help of game logs, it simulates the execution of P by an OLCBP-based bot against different opponents, recording the number of times each action in P was executed and failed.

```
1  Input: M — a game map, G = {g_i}_{i=1}^n — a set of game
   subgoals, A = {a_i}_{i=1}^m — a set of game actions,
2  L — a set of gameplay logs,
3  AugDegree — augmentation degree
4
5  Output: StarCraft strategy & high-level tactical plan
6
7  // Learn S = {S_i = (g_{j_1}, ..., g_{j_{n_i}})} — a set of strategies
8  // (subgoal sequences) & P = {P_i^g = (a_{j_1}, ..., a_{j_{m_i}})} —
9  // a set of high-level tactical plans, each for some subgoal g
10 S, P ← OLCBP-Learn(G, A, L)
11
12 foreach P_i^g ∈ P, a_j ∈ P_i^g do
13     p_fail(a_j, P_i^g) ← ProbOfCausingFailure(a_j, P_i^g)
```

Algorithm 1: ROLCBP: **the offline stage**

For the purposes of CPA, we define an action a's failure probability $p_{\text{fail}}(a, P^g)$ as the fraction of times its execution caused a *deviation* from the corresponding tactical plan P^g and forced OLCBP to resort to replanning (lines 12:1-13:1). In reality, OLCBP's replanning is sometimes successful at attaining P^g's goal g. Therefore, $p_{\text{fail}}(a, P^g)$ is generally a pessimistic measure of a's quality in P^g. Nonetheless, the overestimation of a's probability of causing an unrecoverable failure actually helps CPA produce more robust policies.

Online stage. At runtime, ROLCBP follows the MainGameLoop method (lines 3:2-10:2). It begins by selecting a game strategy with the most *robust* tactical plan (line 4:2) as explained below, using the SelectStrategy&Tactics procedure. In addition to returning the tactical plan for the chosen strategy, SelectStrategy&Tactics augments it with concurrent contingent branches. In effect, this method constructs a tactical *policy tree*, although this tree may be only partial. ROLCBP plays according to the chosen tactical plan, launching its contingent branches prescribed by the policy tree, until OLCBP's mechanisms detect a deviation from it (line 6:2).

SelectStrategy&Tactics (lines 12:2 - 25:2) operates by iterating over all strategies S in library $\mathcal{S}$, letting OLCBP concoct a tactical plan P_S for each strategy, augmenting this tactical plan with contingent branches using CPA, and evaluating the *failure probability* of the policy tree based on this tactical plan. The failure probability of a strategy's augmented tactical plan serves as a proxy of the strategy's robustness; SelectStrategy&Tactics chooses the strategy with with lowest such probability.

The CPA logic at the heart of SelectStrategy&Tactics is captured in eponymous method (lines 27:2-41:2). CPA analyzes a tactical plan P_S by breaking it down into constituent plans P^g from case base $\mathcal{P}$. Recall that at the offline stage, ROLCBP estimated failure probabilities for each action in each such plan (lines 12:1-13:1). With the help of these estimates, CPA identifies actions with nonzero failure probabilities in P_S (line 29:2). For each of these actions a, CPA models the hypothetical situation immediately after a's failure by determining the "latest" subgoal g_j of strategy S that will remain achieved in that situation (line 31:2). In order to be considered a valid contingency branch for a's failure, a plan P_S^+ needs to

```
1  // See Algorithm 1 for the description of M, G, S, P, and AugDegree
2
3  MainGameLoop(M, G, S, P, AugDegree) begin
4      S, P_S ←
       SelectStrategy&Tactics(M, G, S, P, AugDegree)
5      while game has not ended do
6          a_fail, P^g ← PlayUntilActionFailure(P_S)
7          P_S ← ContingentBranch[a_fail, P^g, P_S]
8          if AugDegree == 1 then CPA(M, S, P_S, P)
9          else if AugDegree == 0 then OLCBP(M, S, g, P)
10         else AugDegree ← AugDegree − 1
11
12 SelectStrategy&Tactics(M, G, S, P, AugDegree) begin
13     foreach S = (g_j, ..., g_k) ∈ S do
14         // Compose a tactical plan for strategy S
15         // by concatenating several plans from P
16         P_S = (P_{i_j}^{g_j}, ..., P_{i_k}^{g_k}) ← OLCBP(M, S, ∅, P)
17         LeafBranches ← {P_S}
18         foreach d from 1 to AugDegree do
19             NewBranches ← ∅
20             foreach plan P ∈ LeafBranches do
21                 NewBranches ← NewBranches ∪
22                                   ∪ CPA(M, S, P, P)
23             LeafBranches ← NewBranches
24         p_fail(P_S) ← EvalFailProb(P_S)
25     return S, P_S s.t. p_fail(P_S) is the lowest
26
27 CPA(M, S, P_S, P) begin
28     LeafBranches ← ∅
29     foreach a ∈ P^g ∈ P_S s.t. p_fail(a, P^g) > 0 do
30         (g_1, ..., g_j) ← achieved prefix of S if a fails
31                           during P^g's execution
32         // Compose a plan for S's remaining subgoals
33         P_S^+ ← OLCBP(M, S, (g_1, ... g_j), P)
34         // Schedule P_S^+ to start executing concurrently
35         // with P_S ASAP after g_j is achieved but
36         // before a starts executing, if possible
37         if ScheduleConcurrentExec(P_S, P_S^+, g_j) then
38             ContingentBranch[a, P^g, P_S] ← P_S^+
39             LeafBranches ← {P_S^+}
40         else ContingentBranch[a, P^g, P_S] ← ∅
41     return LeafBranches
42
43 EvalFailProb(P_S) begin
44     if P_S is ∅ then return 1
45     p_fail(P_S) ← 0
46     foreach a_i ∈ P_S, i ranging from |P_S| to 1 do
47         p_fail(P_S) ← p_fail(a_i, P_S)·
48           ·EvalFailProb(ContingentBranch[a_i, P^g, P_S])+
49           +(1 − p_fail(a_i, P_S)) · p_fail(P_S)
50     return p_fail(P_S)
```

Algorithm 2: ROLCBP: **the online stage**

achieve all subgoals of S after g_j. CPA delegates it to OLCBP to find such a plan (line 33:2).

When/if P_S^+ is found, it needs to be scheduled for execution. Intuitively, we would like to start P_S^+ in parallel with P_S as early as possible after g_j is achieved. (By P_S^+'s construction, we cannot start it before achieving all of S's subgoals up to g_j.) We would also like to schedule it before a, because otherwise in case of a's failure, while OLCBP is looking for another plan, for some time the player's units will not know what to do (and hence will be extremely vulnerable). Whether P_S^+'s execution can start in this time window depends on game resource constraints. For example, it may be impossible to train several kinds of units concurrently without enough minerals. Scheduling P_S^+ is again delegated to OLCBP (line 37:2). If it turns out to be possible, P_S^+ is designated as P_S's contingency for a's failure (line 38:2). Otherwise, a's failure remains unaccounted for (line 40:2). Ultimately, CPA's inability to add contingency branches to a tactical plan may point to the plan's, and hence the strategy's, inherent weaknesses.

A single CPA invocation provides back-ups for the failure of a given tactical plan, but not for failures of the back-ups themselves. The latter would require applying CPA recursively to the contingent branches. SelectStrategy&Tactics provides the option of doing that via the *augmentation degree* (*AugDegree*) parameter. The first-degree augmentation applies CPA to a strategy's main tactical plan only, the second-degree one — to that plan and its contingent branches, and so on (lines 17:2-23:2). Intuitively, higher-degree augmentations reveal complete contingency structure and provide a more thorough assessment of a strategy's robustness. Independently of augmentation degree, evaluation of a strategy's failure probability can be done in a single pass. This is due to the fact that an augmented tactical plan is always a tree. Computing its failure probability is a matter of recursively propagating information from the tree's leaves to its root (lines 43:2 -50:2).

If augmentation degree is at least 1, deviating from the main tactical plan of the chosen strategy in ROLCBP's MainGameLoop likely does *not* cause replanning, as it would in OLCBP. This is because a contingent branch for this failure has already been launched, if this was possible given the game resource constraints. This branch becomes the main execution plan (line 7:2). However, this plan may be vulnerable: recall that, depending on augmentation degree, SelectStrategy&Tactics may not have scheduled any contingencies for it. Therefore, once ROLCBP switches to this plan, if necessary it augments it with contingent branches (line 8:2), to avoid hasty replanning in case of another failure.

Practical considerations. In its computations, ROLCBP iterates over all strategies in its library $\mathcal{S}$, and for each of them builds a policy tree whose size is influenced by the augmentation degree parameter. Thus, the size of $\mathcal{S}$ and augmentation degree are two major factors affecting ROLCBP's performance. Fortunately, the following observations imply that sensible values of these quantities are small in practice:

- *RTS games typically have few viable strategies, and OLCBP's learning from human demonstration naturally identifies them.* Indeed, although in theory there are many possible goal orderings for any scenario, human players tend to use only those belonging to a small set of successful ones. Since ROLCBP's strategy library $\mathcal{S}$ is populated by strategies extracted from human gameplay logs, it ends up containing only a few entries. This makes it possible to analyze each one of them in a short time at the beginning of the game.

- *High-degree augmentation does not pay off.* While n high-degree augmentation gives a better estimate of a strategy's robustness, and hence enables better-informed strategy choice, its computational cost grows exponentially in augmentation degree. ROLCBP can afford some deliberation time at the beginning of the game, but excessively long delay before committing to a strategy heightens the risk of the opponent's attack against the player's unprepared infrastructure. Moreover, most of the contingent branches in a highly augmented policy tree will end up not getting followed, wasting computational effort invested into them. Last but not least, high-degree contingencies are increasingly difficult to schedule because of the multiple resource conflict resolutions OLCBP needs to perform. In all our tests, *first-degree* augmentation, which applies CPA only to the main tactical plan of a policy, has provided the best balance of computational cost and strategy evaluation quality. Therefore, we set *AugDegree = 1* in all our experiments.

5. EXAMPLE

Suppose that after offline training (Algorithm 1), ROLCBP playing the game of *StarCraft* starts by analyzing a strategy whose subgoal ordering is depicted as a sequence of dark-gray rectangles on the *left* side of Figure 2. Suppose further that OLCBP, which ROLCBP uses to find a tactic for this strategy, composes a plan given by the white rectangles inside the dark-gray rectangles. Namely, the plan is *[GatherPrimaryResource, BuildBarracks, TrainMarine, AttackUnit]*, where several *TrainMarine* and several *AttackUnit* actions are executed in parallel. As Figure 2 shows, this tactical plan consists of primitive plans for achieving each subgoal given the previously achieved ones. For the simplicity of the example, each such subplan of the main tactical plan has only one step.

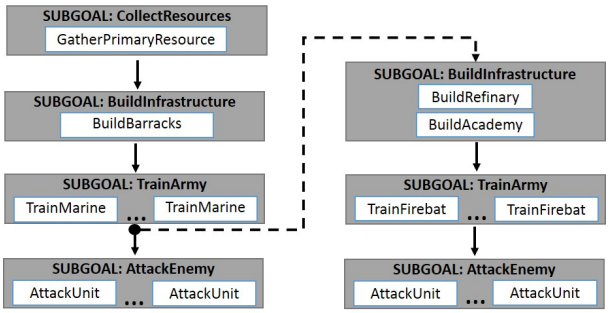


Figure 2: A tactical plan (left) with a contingent branch (right).

The analysis of this plan involves computing the failure probabilities of the steps that compose it. Imagine that during the offline stage (Algorithm 1), ROLCBP established that the first three subgoals, up to and including *TrainArmy*, are always achieved successfully, e.g., because at the start of the game the opponent has too little time to reach the player's base and disrupt the plan. However, ROLCBP may have discovered that the last subgoal of the plan, *AttackEnemy* is difficult to reach: the marine units, though quick to train, are fairly weak, and their attack often fails. When it does, the *TrainArmy* subgoal also gets "unachieved", because the army is destroyed in the attack.

Having performed this analysis, ROLCBP decides to inject a concurrent contingent plan before going for the *AttackEnemy* subgoal. This contingent branch needs to achieve all subgoals that remain unachieved if the main plan fails, i.e., *TrainArmy* and *AttackEnemy*. One such plan is shown on the right side of Figure 2. It trains an army of units more powerful than marines, which requires it to build additional infrastructure. This infrastructure takes a lot of resources, which are unavailable while the main army of marines is being trained, so it schedules this branch for execution right after the marines are trained and ready to attack. If the marines fail, a more potent army of firebats will be available to guard the base and attack the enemy soon afterwards.

After augmenting the main tactical plan, ROLCBP evaluates its failure probability. If more strategies are available, ROLCBP analyzes them too and picks the least likely to fail.

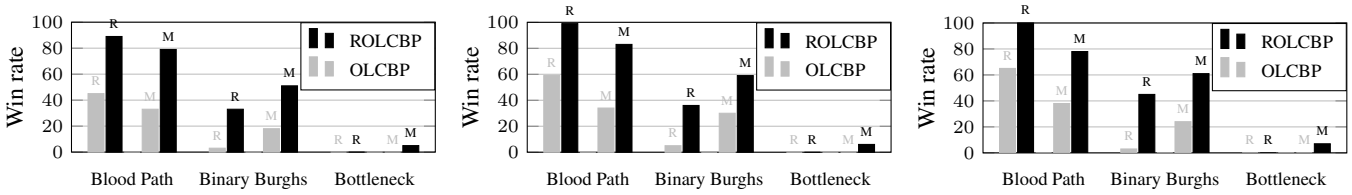
6. EVALUATION

We evaluated CPA by implementing it as part of ROLCBP in software bots for *StarCraft* and *Wargus* (an open-source clone of *Warcraft II*). Most of our experiments focus on *StarCraft*, since it is the more complicated of the two and has a better-developed publicly available experimentation infrastructure that greatly speeds up running tests on it. At the same time, *Wargus* shares its game engine *Stratagus*, the mechanism responsible for its planning capabilities, with a range of other RTS games such as *Aleona's Tales*, so our successful experiments on *Wargus* imply CPA's effectiveness on all these games as well and attest to the universality of our approach. Our experiments address the following questions: (1) How does concurrent policy augmentation affect the bot's performance if used purely at the tactical level when a strategy is fixed, by increasing the tactical plan's robustness? (2) How does CPA affect the bot's performance if used in both at the tactical and strategic levels in ROLCBP, by guiding the strategy selection process? (3) How successfully does CPA deal with opponents who are significantly different from those it plays during the offline training stage (Algorithm 1)? Note that demonstrating the overall superiority of our bot over state-of-the-art bots for *StarCraft* or *Wargus* is *not* an objective of our experiments. Building such a system would require extensive research into components other than strategy and tactics selection modules, which is beyond the scope of this work. Instead, our evaluation uses simple algorithms in bot modules such as those responsible for low-level control, and concentrates on the influence of CPA on strategic and high-level tactical performance.

Experimental setup. We gauge the performance of ROLCBP and the baselines (built-in AI, OLCBP, or both, depending on the experiments) by running many games against an opponent and measuring the *win rate* — the percentage of won games. Win rate subsumes all other gameplay characteristics of potential interest, such as speed or memory usage: if the player's AI is unacceptably slow or memory usage too high, this is reflected in a low win rate. We identify statistically significant performance differences in win rate with two-tailed z-test for equality of proportions at the 95% confidence level.

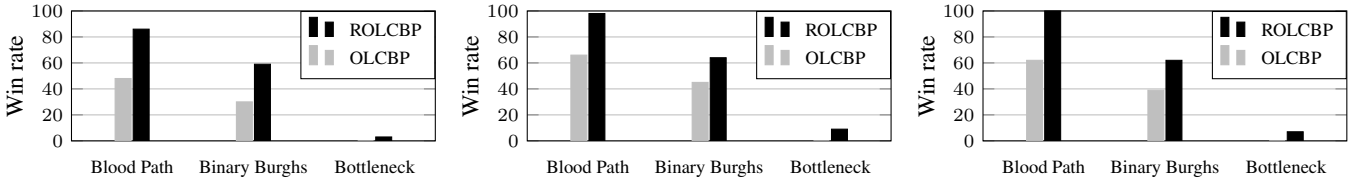
For *StarCraft*, we experiment on three maps frequently used in *StarCraft* tournaments: Blood Path, Binary Burghs, and Bottleneck. Each *StarCraft* map is characterized by the size of its grid, which is loosely correlated with the map's difficulty; Blood Path has size 64x64, Binary Burghs – 96x96, and Bottleneck – 128x128 cells. For *Wargus*, we use maps of similar sizes: Harrow (62x62), Hills of Glory (96x96), and Dust Storm (128x128). In all figures, map size increases along the x-axis. As mentioned previously, a *StarCraft* player can control one of three races and in *Wargus* — one of two, each with its own characteristics. Our bot always plays the *Terran* race in *StarCraft* and *orcs* in *Wargus*. For offline acquisition of strategies and tactical plans, our bot uses human gameplay logs. For *StarCraft*, they are available at http://www.teamliquid.net/replay/.

Our implementations of ROLCBP and OLCBP are in C++. The two differ only in strategy selection, and share all other components



(a) Training vs Terran/Zerg, playing vs Protoss (b) Training vs Terran/Protoss, playing vs Zerg (c) Training vs Zerg/Protoss, playing vs Terran

Figure 3: Effect of CPA on the tactical performance of the rush attack **(R)** and mid-game attack **(M)** strategies in *StarCraft*. *The advantage of ROLCBP is statistically significant in all experiments except for rush attack on the Bottleneck map.*



(a) Training vs Terran/Zerg, playing vs Protoss (b) Training vs Terran/Protoss, playing vs Zerg (c) Training vs Zerg/Protoss, playing vs Terran

Figure 4: Effect of CPA on overall performance in *StarCraft*. *The advantage of ROLCBP is statistically significant in all experiments.*

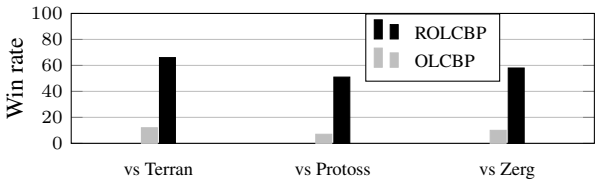


Figure 5: Effect of CPA on overall performance in *StarCraft* on the Bottleneck map with the Fabian strategy. In each case, ROLCBP trains on two races and plays versus the remaining one. *The advantage of ROLCBP is statistically significant in all experiments.*

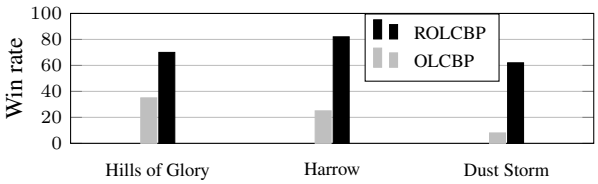


Figure 6: Effect of CPA on overall performance in *Wargus*. *The advantage of ROLCBP is statistically significant in all experiments.*

(modules for extracting strategies and tactical plans from game logs, low-level control, etc.) The augmentation degree parameter in ROLCBP was set to 1 (see the discussion at the end of the Robust OLCBP section). The experiments were run on an Intel Core i7 2.4GHz CPU with 12GB RAM under Windows 8.1.

Effects of Tactical-Level Augmentation. In this experiment, conducted on *StarCraft*, we measure how the win rate of individual strategies changes when first-degree CPA is applied to their main tactical plans. The results are shown in Figure 3. We focus on two strategies in this experiment. *StarCraft* has three different races, and we measure each strategy's win rate against each of the three races separately (Figures 3a, b, and c). Our bot always plays the Terran race.

To measure a strategy's win rate against a given race (e.g., Zerg) on a given map, we first run the offline training stage of ROLCBP and OLCBP on logs of games played against *the two other* races (e.g., Protoss and Terran) controlled by *StarCraft*'s built-in AI. Then we evaluate the strategy's performance against the target opponent race controlled by the built-in AI by comparing the win rates of OLCBP and ROLCBP when this strategy is the only one available in their strategy library. OLCBP and ROLCBP each ran every strategy 100 times against each race on each map.

The two strategies we use in this experiment are known among human *StarCraft* players as *rush attack* and *mid-game attack*. We chose these strategies because our bot judged them to be the most powerful ones it managed to extract from the game logs: in every game in the strategy selection experiments, whose results are presented later, our bot invariably choose one of these two strategies as the best one.

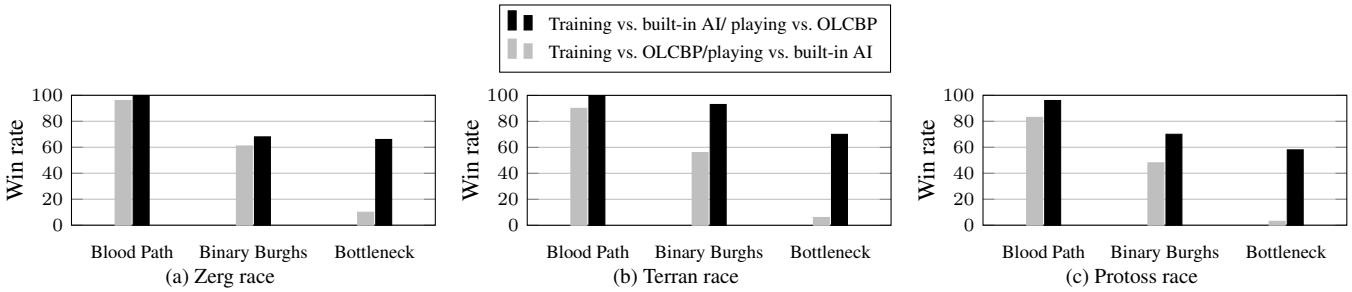
Figure 3, which summarizes the results on this experiment, reveals several patterns:

- *For most combinations of strategies, maps, and train-test splits in the experiment, ROLCBP's win rate is significantly higher than OLCBP's, i.e., CPA significantly improves strategies' robustness.* This happens because OLCBP resorts to replanning when a strategy's execution goes awry. Replanning is expensive, and if it does not yield a quick solution in a critical situation, the bot loses the game. CPA reduces the need for replanning in critical situations.

- *If a strategy is inherently unsuitable to a given map, applying CPA to it does not help.* For example, all tactical plans for the rush attack strategy on the Bottleneck map have fatal flaws that prevent them from winning even after CPA.

- *CPA can improve a strategy's tactical plan qualitatively, enabling it to win games that its non-augmented version would always lose.* For example, without CPA mid-game attack on the Bottleneck map always fails, while with CPA it is able to win a percentage (albeit small) of games.

Effects of CPA on Strategy Selection. Figures 4 and 6 show the main experimental results of this paper, which testify to the benefit of CPA for overall strategy selection in *StarCraft* and *Wargus*, respectively. For *StarCraft*, the experimental setup is the same as for the tactical-level augmentation experiments from the previous subsection, but now we allow ROLCBP and OLCBP to consider *all* strategies they extract from the game logs, instead of being restricted to any specific one. Thus, the quality of strategies' augmented tactical plans determines the strategy choice for the game. While ROLCBP makes this choice by minimizing strategies' empirical failure probability, OLCBP, our baseline, selects strategies heuristically based on the characteristics of the map and the races the players have picked, breaking ties randomly. In particular, for all *StarCraft* maps in our experiments both the rush and the mid-



(a) Zerg race (b) Terran race (c) Protoss race

Figure 7: Training against OLCBP and playing against static AI and vice versa.

game attack strategies turn out to match OLCBP's criteria, so it decides randomly between them.

As the graphs in Figure 4 demonstrate, CPA gives ROLCBP a statistically significant advantage in strategy selection. At the same time, similar to the tactical-level experiments, they show that if no strategy learned from the logs is very suitable for a map, then ROLCBP cannot fundamentally change AI's performance — this is the case in the *Bottleneck* scenario. Fortunately, the fix for this issue is as simple as giving ROLCBP a more diverse set of logs. To ascertain this, we re-ran Figure 4's experiments after adding logs with plays of a third strategy, known among *StarCraft* players as Fabian, in addition to rush and mid-game attack discussed earlier. This strategy is not a good option on Blood Bath and Binary Burghs, and ROLCBP indeed does not choose it there, so the results on these two maps remain as in Figure 4. The performance on Bottleneck, however, shown in Figure 5, improves by a lot across each of the three train-test splits and allows ROLCBP to beat built-in AI most of the time. The role of plan augmentation in the decision to choose the Fabian strategy for this map is vital. If failure probabilities of the main tactical plans for mid-game attack, rush attack, and Fabian are evaluated (lines 43-50 of Algorithm 2) against training opponents *before* plan augmentation, rush attack's failure probability appears the lowest, implying that it is a better strategy. (Figure 3 does not reflect this, because it shows win rates of strategies against *testing* opponents.) However, this map's dynamics make deviations from rush attack's tactical plan hard to recover from. Performing CPA and re-evaluating the augmented plans' failure probabilities reveals this and leads to the correct decision to play the Fabian strategy.

For *Wargus*, due to the limitations of the testing framework, we measure ROLCBP's and OLCBP's win rates against the built-in AI playing the orc race only. ROLCBP and OLCBP, which also controlled orcs, each played 20 games on each map. Results on *Wargus* (Figure 6) confirm our findings from *StarCraft*: CPA implemented in ROLCBP drastically improves strategy selection compared to OLCBP. On all *Wargus* maps we have tested, ROLCBP wins over half of the games against built-in AI, whereas OLCBP does not on any single map. Last but not least, since many other RTS games share the *Stratagus* game engine with *Wargus*, we can expect ROLCBP to enjoy similar advantage on them as well.

CPA's performance against unfamiliar opponents. Besides its ability to improve gameplay at strategic and tactical levels, the experiments above hint at CPA's robustness to previously unencountered opponents. As evidence of this, note that in the aforementioned setups ROLCBP's win rate was evaluated on opponent races against which it did not play during training. Nonetheless, both in training and in testing ROLCBP's opponents were controlled by the same (built-in) AI, which prompts a question: does ROLCBP perform as well when the enemy AIs in training and testing are different?

In our final set of experiments on *StarCraft*, we answer this question positively. We consider two opponent AIs: *StartCraft*'s built-in AI and OLCBP. For each *StarCraft* race, we train ROLCBP on game logs against that race controlled by the built-in AI, and measure its win rate against that race played by OLCBP. We also experiment with the case when built-in AI's and OLCBP's roles are switched. As before, we let ROLCBP play 100 games for each race, map, and train-test AI combination.

Figure 7 presents ROLCBP's win rates in this experiment. With the exception of training against OLCBP and playing against built-in AI on the Bottleneck map, OLCBP consistently and convincingly outperforms its opponents. Nonetheless, ROLCBP's win rate noticeably depends on the specific combination of train and test opponent AIs; playing OLCBP after training on the built-in AI usually gives better results than vice versa. This is not surprising. OLCBP's inferior win rates against the built-in AI in Figure 4c suggest that the OLCBP bot is weaker. ROLCBP's win rate suffers when it trains against this weaker AI and then faces the stronger built-in one. The converse also holds: note that the training provided by the built-in AI is so powerful that ROLCBP outmatches OLCBP on Bottleneck (Figure 7) despite the fact that, as already mentioned, all of the strategies in its library are fundamentally unsuitable for this map.

7. RELATED WORK

Automatic strategy and high-level tactics selection methods have featured in several AI systems for playing RTS games; many, if not most, are derived from case-based planning. Darmok [12], ITSrategizer [4], and EISBot [17] use this technique with various additions. An agent developed for the DEFCON game employs a related technique, case-based reasoning, combining it with simulated annealing and decision trees [1]. This makes case-based planning and its variants de-facto state of the art in *automatic* strategy selection for RTS games.

Hierarchical Task Networks have also been used for strategy selection. Besides OLCBP, which is partly based on HTNs, they have been used for this purpose in first-person shooters [7] and role-playing games [8].

Hard-coded techniques for strategy and tactics selection are still strongly competitive with automatic ones, especially in complex games such as *StarCraft* [13]. Finite state machines have been particularly effective for specifying fixed patterns for strategic decision-making and other aspects of RTS games [5].

Besides strategy and tactics selection, RTS game AI's behavior depends on many other components, which are discussed in a recent survey on this topic [13].

Several approaches related to CPA have been proposed in the area of sequential decision-making under uncertainty. Possibly the closest is *incremental contingency planning*, devised for planning Mars rover missions [3]. At a high level, it operates similarly to CPA, but considers non-adversarial environments, doesn't face the strategy selection problem, has different mechanisms and utility for deciding where to add contingent branches, and, crucially, assumes the availability of a planner to construct them.

In probabilistic planning modeled by MDPs, a related approach is *incremental plan aggregation* [16]. RFF, the planner that implements it, finds a linear plan to the goal, determines states where an agent may end up if action outcomes deviate from this plan, adds linear plans to the goal from these states, and iteratively repeats the process. Like ROLCBP, between iterations RFF estimates the "failure probability" of exiting the partially constructed policy tree. Unlike ROLCBP, RFF assumes no action concurrency, which makes choosing branching points much easier, and generates contingent plans on-demand with a dedicated planner, whereas ROLCBP does not need such a planner, relying on plans from its case base. Lastly, ROLCBP's failure probability estimates are only a heuristic for adding contingency branches, because it works in an adversarial, possibly non-stationary environment.

8. CONCLUSION

This paper presented concurrent policy augmentation (CPA), a technique for addressing shortcomings of existing methods for automatic strategy and tactics selection in real-time strategy games. These existing methods, such as OLCBP, treat strategy selection as a hierarchical deterministic planning problem with a possibility of replanning. Replanning in critical situations often fails, leading to a game loss. In contrast, CPA analyzes tactical plans for available strategies to identify their potential failure points. It attempts to mitigate the consequences of these failures by adding contingent plan branches to be executed concurrently with the main plan. We implemented CPA in an OLCBP-based architecture ROLCBP that selects strategies by evaluating the failure probabilities of their augmented tactical plans. Our experiments on popular and complex RTS games *StarCraft* and *Wargus* show that CPA gives ROLCBP significant advantages in win rate compared to OLCBP by improving both tactical and strategic decision-making. It also lets ROLCBP outperform opponents with different behavior than those ROLCBP encounters during training. In the future, we plan to improve our ROLCBP implementation and turn it into a bot competitive with entries in the *StarCraft AI* competition. We have discovered that despite superior strategic performance, our bot's low-level unit control often causes it to lose skirmishes (and hence entire games) that state-of-the-art bots would easily win. Low-level control issues account for most of the games ROLCBP lost in this paper's experiments. Thus, despite being out of this paper's scope, low-level control is a clear area of improvement. We believe that the usefulness of CPA extends beyond case-based planning techniques. The hypothetical advent of planners capable of efficiently generating winning trajectories for RTS games would remove CPA's dependence on plans stored in the case base and would greatly increase CPA's potential.

REFERENCES

[1] R. Baumgarten, S. Colton, and M. Morris. Combining AI methods for learning bots in a real-time strategy game. *Int. J. Computer Games Technology*, 2009, 2009.

[2] M. Buro. Call for AI research in RTS games. In *In Proceedings of the AAAI Workshop on AI in Games*, pages 139–141. AAAI Press, 2004.

[3] R. Dearden, N. Meuleau, S. Ramakrishnan., D. E. Smith, and R. Wasington. Incremental Contingency Planning. In *ICAPS'03*, 2003.

[4] I. Fathy, M. Aref, O. Enayet, and A. Al-Ogail. Intelligent online case-based planning agent model for real-time strategy games. In *ISDA*, pages 445–450, 2010.

[5] D. Fu and R. Houlette. The ultimate guide to FSMs in games. *AI Game Programming Wisdom 2*, 2003.

[6] K. Hammond and R. Head. Case-based planning: A framework for planning from experience. *Cognitive Science*, 14:385–443, 1990.

[7] H. Hoang, S. Lee-urban, and H. MuÃśoz-avila. Hierarchical plan representations for encoding strategic game ai. In *In Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05*. AAAI Press, 2005.

[8] J.-P. Kelly, A. Botea, and S. Koenig. Offline planning with hierarchical task networks in video games. In *AIIDE'08*, 2008.

[9] I. Little and S. Thiébaux. Probabilistic planning vs replanning. In *Workshop on International Planning Competition: Past, Present and Future (ICAPS)*, 2007.

[10] C. Muise, V. Belle, and S. McIlraith. Computing contingent plans via fully observable non-deterministic planning. 2014.

[11] J. Muñoz, G. Gutiérrez, and A. Sanchis. A human-like TORCS controller for the Simulated Car Racing Championship. In *Proceedings 2010 IEEE Conference on Computational Intelligence and Games*, pages 473–480, August 2010.

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

[13] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game AI research and competition in starcraft. *IEEE Trans. Comput. Intellig. and AI in Games*, 5(4):293–311, 2013.

[14] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[15] E. D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pages 206–214, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[16] F. Teichteil-Königsbuch, U. Kuter, and G. Infantes. Incremental plan aggregation for generating policies in MDPs. In *AAMAS'10*, pages 1231–1238, 2010.

[17] B. G. Weber, M. Mateas, and A. Jhala. Building human-level ai for real-time strategy games. In *AAAI Fall Symposium: Advances in Cognitive Systems*, volume FS-11-01 of *AAAI Technical Report*. AAAI, 2011.