A Primer on Writing Strategies for the AI of Battlefield1942™

Author: Tobias Karlsson

Abstract: This document will try to give a brief introduction to how to write strategies for the AI in Battlefield1942.

This document is part of a series of documents on the AI of Battlefield1942.

Writing Strategies

This document will try to explain how to write a strategy for the SAI of Battlefield 1942. The first part of this document will describe the different parts of a strategy, and the second part will give some general guidelines for how to write an effective strategy. It is beyond the scope of this document to explain all the details of the AI Battlefield1942, and some prior knowledge is assumed.

Strategy? – What, When, How?

The strategies' purpose is to help guiding the SAIs in their decisions on which strategic areas to attack and defend. The SAI ranks the importance of the strategic areas by their temperature. The strategy is used to skew these temperatures so that the SAIs will prioritise strategic areas depending on strategy.

The strategies are a very vague and subtle tool, but also a very flexible one. Unfortunately, it is impossible to learn how to write strategies only by reading this document, so the reader is urged to look at existing strategies and experiment with them on different maps.

Building Blocks of a Strategy

Strategies are built using three different files. Each file contains a specific part that comprises a strategy. Dividing the strategies' parts into files like this, is not mandatory, it is merely an organizational decision. The files are as follows:

- Strategies.con
- prerequisites.con
- conditions.con

Each of these files will be described in detail below.

Strategies.con

Strategies.con holds the definitions of the strategies. In this file, all information except the part that decides when the strategy is possible to use, is defined.

createStrategy

- aiStrategy.createStrategy *name(string)*

This is command creates a strategy and gives it the name *name*.

Aggression

- aiStrategy.Aggression *percent (float)*

This command sets the aggression level of the strategy. The value *percent* designates the percentage of the side's resources that will be devoted to offensive moves. *Percent* is a number in the range between 0.0 and 1.0.

NumberOfAttacks

- aiStrategy.NumberOfAttacks *number (int)*

This command sets the maximum *number* of attacks that the SAI may perform at any given time when the strategy is active.

NumberOfDefences

- aiStrategy.NumberOfDefences *number (int)*

This command sets the minimum number of defences that the SAI must sustain at any one time when the strategy is active. Note the difference between NumberOfDefences and NumberOfAttacks.

TimeLimit

- aiStrategy.TimeLimit *seconds (int)*

This command is optional. Setting it means that when the strategy has been active for the period of time indicted by *seconds*, its importance as a viable strategy slowly diminishes. This is a good way to force the SAI to change its strategy every once in a while.

setPrerequisite

- aiStrategy.setPrerequisite *name(string)*

This command sets the prerequisite that will decide how interesting a strategy is, and if it is even possible to use at the moment.

setStrategicObjectsModifier

- aiStrategy.setStrategicObjectsModifier *flagType(enum) modifier(float) status(enum)*

This command creates a modifier that affects the temperatures of the strategic areas of the specified type. *FlagType* is a flag that specifies the strategic areas (each strategic area can have one or more of these flags set by its creator). The last five flags are set dynamically and will be explained below. *FlagType* can have the following values:

- Flank
- Centre
- Base
- Close
- Remote,
- AirSpawner
- LandSpawner
- NavalSpawner
- SoldierSpawner
- StrongPoint
- ChokePoint
- Bridge
- AirField
- SupplyPoint
- Route
- ControlPoint
- North
- West
- South

- East
- Front
- Safe
- Enemy
- Neutral
- UnReachable

Front means that the strategic area has at least one neighbour that is not controlled by the SAI.

Safe means that all neighbouring areas are controlled by the SAI.

Enemy means that the enemy controls the strategic area.

Neutral means that the strategic area is not controlled by anyone.

UnReachable means that the strategic area has no neighbours controlled by the SAI.

Modifier is a float that specifies how much the temperature of the strategic area is to be changed. *Modifier* is a multiplier.

Status is an optional parameter and if set, it limits the modifier to only work on strategic areas that has the specified status. The possible values of *status* are:

- Hostile
- Neutral
- Owned

The status of a strategic area is perceived by the SAI, which means that it might not always be an agreement of the status of a strategic area between the two sides.

addRequiredPrecedingStrategy

- addRequiredPrecedingStrategy *name(string)*

This command is optional and if set, the SAI will not consider activating the strategy unless one of its required preceding strategies are active at the moment. It is possible to specify several required strategies. In that case, it is enough that one of these strategies is active. *Name* is the name of the required strategy.

addProhibitedPrecedingStrategy

- addProhibitedPrecedingStrategy *name(string)*

This command is optional and if set, the SAI will not consider activating the strategy if one of its prohibited preceding strategies are active. It is possible to specify several prohibited strategies. In that case, it is enough that one of these strategies is active. *Name* is the name of the required strategy.

addSpecificObjectModifier

- addSpecificObjectModifier *name(string) modifier(float)*

This command is optional. This command allows for the strategy to have a modifier for a specific strategic area specified with *name*. *Modifier* is a multiplier that specifies how the strategic area's temperature is going to be modified.

completeStrategies

- completeStrategies

This command must be run if the "addSpecificObjectModifier"-command has been used. If not, the game will most likely crash sooner or later. The command cannot be successfully issued before the strategic areas have been loaded. The purpose of the command is to initiate the "addSpecificObjectModifier"-commands, when run the game checks that the specified areas really do exists.

Prerequisites.con

The purpose of the prerequisites is to decide whether the strategy is a valid choice at the moment, as well as to decide how interesting the strategy is, if it is valid. Whenever the SAI is interested in checking a strategy, the prerequisite of that strategy is called, and is expected to return a decimal value. When the value is zero or below, the strategy is not valid. If it is higher than zero, the strategy is valid to that degree. The scale of this value, what counts as a high and a low value, is only dependent on the other strategies available to the SAI at the moment. That is, the writer of the strategy arbitrarily chooses the scale. The higher relative value a prerequisite returns, the more likely it is that strategy is chosen.

createPrerequisite

- createPrerequisite *name(string)*

This command creates a prerequisite with the name *name*.

addCondition

- addCondition *name(string) weight(float)*

The main component of the prerequisites is its conditions. The prerequisite combines the results of its conditions and uses it to return its own result. This command is used to add a condition of name *name* to the prerequisite. The modifier *weight* is optional, and is a multiplier that the result of the condition is multiplied with. If not set, it is defaulted to 1.0.

Conditions.con

A condition is a unary or binary comparison. The point of the conditions is to check a particular part of the state of the world. The conditions return a float value that indicates how true they are. This value is dependant on which kind of condition one has chosen to create.

To produce its result, each condition compares two values (one might be a constant) against a *targetValue*. How the comparison is made, depends on the parameters given to the condition. The default *targetValue* is 0.0, which may not always be a suitable value.

Creating conditions

There are several commands for creating conditions for the strategies:

- createConstantCondition
- createHomogenousCondition
- createHeterogeneousCondition

The first two commands are just specialisations of the last, createHeterogeneousCondition. The commands take the same parameters, except that the first two only need a subset of the full parameter list of the last one. The commands with parameter lists are as follows:

- createConstantCondition *name(string) compareType(enum) compare(enum) conditionSide(enum) conditional(enum) constant(float)*
- createHomogenousCondition *name(string) compareType(enum) compare(enum) conditionSide(enum) conditional(enum)*

- createHeterogeneousCondition, *name(string) compareType(enum) compare(enum) conditionSide(enum) conditional1(enum) conditional2(enum)*

The meaning of the parameters will be explained in turn.

Name is the name of the condition.

CompareType specifies which ranges of values that the conditions will return its result in, and how the function that defines how the values evolve with stronger/weaker truthfulness. The possible variants are as follows:

- Crisp
- Fuzzy
- FuzzySqr

If the compareType is *Crisp*, the result from a condition is either 1.0 (true), 0.0 (undecided), or –1.0 (false). This is the easiest compareType to use, but also the least flexible.

If the compareType is *Fuzzy*, the condition will return a continuous value between $-\infty$ and $+\infty$. The higher the value, the more true the condition is. Correspondingly, the lower the value, the more false the condition is. If the value is 0.0, the result is undecided.

If the compareType is set to *FuzzySqr*, the condition will return the squared value as if it used the *Fuzzy* compareType with one exception, the value keeps its sign.

A condition basically compares two values. *Compare* is a value that decides how this comparison is made. The different possible values area as follows:

- Equal
- EqualGreater
- EqualSmaller
- Quotient
- QuotientGreater
- QuotientSmaller
- Difference
- DifferenceGreater
- DifferenceSmaller

If the comparison is *Equal* and the type is *Crisp* the condition will return 1.0 if the difference between the first and the last value equals *targetValue*, and –1.0 if the values are not equal. If the type is *Fuzzy* it will return negative of the absolute value of the difference between the first and the second value minus the *targetValue*.

If the comparison is *EqualGreater* and the type is *Crisp* the condition will return 1.0 if the difference between the first and the second value is equal or greater than the *targetValue*. It will return –1.0 otherwise. If the type is *Fuzzy* it will return the difference between the two values minus the *targetValue*.

If the comparison is *EqualSmaller* and the type is *Crisp* the condition will return 1.0 if the difference between the first and the second value is equal or smaller than the *targetValue*. It will return –1.0 otherwise. If the type is *Fuzzy* it will return the negative of the difference between the two values minus the *targetValue*.

If the comparison is *Quotient* and the type is *Crisp* the condition will return 1.0 if the first value divided with the second is equal to the *targetValue* and –1.0 otherwise. If the type is *Fuzzy* it will return negative of the absolute value of the difference between the quotient of the first and the second value and the *targetValue*. It will return –1000.0 if the second value is equal to 0.

If the comparison is *QuotienGreater* and the type is *Crisp* the condition will return 1.0 if the first value divided with the second is equal or greater than the *targetValue* and –1.0 otherwise. It will return 1.0 if the second value is 0.0. If the type is *Fuzzy* it will return the difference between the quotient of the first and the second value and the *targetValue*. It will return 1000.0 if the second value is equal to 0.

If the comparison is *QuotienSmaller* and the type is *Crisp* the condition will return 1.0 if the first value divided with the second is equal or smaller than the *targetValue* and –1.0 otherwise. It will return -1.0 if the second value is 0.0 If the type is *Fuzzy* it will return the negative difference between the quotient of the first and the second value and the *targetValue*. It will return –1000.0 if the second value is equal to 0.

The comparisons of *Difference*, *DifferenceSmaller*, and *DifferenceGreater* are aliases for the corresponding *Equal*-types of comparisons. Note that there is a bug with *Difference* that has been fixed for version 1.3 of Battlefield1942.

The *ConditionSide* parameter decides which information database (each SAI has a database for each side) should be used when the parameters are extracted. The possible values are as follows:

- Friendly
- Enemy
- Both

If the side is set to *Friendly*, both parameters will be extracted from the database describing the own side.

If the side is set to *Enemy*, both parameters will be extracted from the database describing the enemy side.

If the side is set to *Both*, the first parameters will be extracted from the database describing the own side, the second parameter is extracted from the database describing the enemy side.

The *Conditional(1&2)* specifies which values that should be extracted from the selected databases. The possible values are:

- Carrier
- HeavyNaval
- LCVP
- Naval
- Bomber
- Fighter
- AntiAir

- Air
- Infantery
- Artillery
- AntiTank
- HeavyAttack
- FastAttack strength
- Security
- AverageStrategicStrength
- AttackStrategicStrength
- DefenceStrategicStrength
- Ticket
- ControlPoint
- Time
- StartTime
- Attacks
- Defences
- NumberOfFriendlyAreas
- NumberOfNeutralAreas
- NumberOfHostileAreas
- Flank
- Base
- Close
- Centre
- Remote
- Route
- Bridge
- North
- West
- South

- East
- FrontFlank
- FrontBase
- FrontClose
- FrontCentre
- FrontRemote
- FrontNeutral
- Front
- Safe
- EnemyObject
- UnReachable
- UnitConstant

Carrier counts the number of objects of the types *Naval*, *AirField* and *Mobile*.

HeavyNaval counts the strength against ships of the objects of types *Naval*, with ship armour.

LCVP counts the number of objects that are of the types *Naval*, *Transport*, and *Mobile.*

Naval holds the sum of all strength against ship armour.

Bomber holds the sum of the strength against heavy armour of all objects of type *Air*.

Fighter holds the sum of the strength against air armour of all objects of type *Air*.

AntiAir holds the sum of all strength against air armour.

Air holds total strength of objects of type *Air*.

Infantery (yes, it is misspelled), holds sum of all the strength of objects of type *Infantry*.

Artillery holds sum of all the strength of objects of type *Artillery*.

AntiTank holds the sum of all strength against heavy armour of units of type *Air* or *Ground*.

FastAttack holds all the strength of objects of type *Mobile* that can move faster than certain speed (available to change if need be).

Security holds a value between 0.0 and 1.0 that measures the confidence the SAI has in its knowledge about the world. 0.0 means that the SAI does not really know anything, and 1.0 means that it belives it knows everything perfectly.

AvarageStrategicStrength holds the average of the sum of the strategic attack and defence strengths.

AttackStrategicStrength holds the sum of all strategic attack strengths.

DefenceStrategicStrength holds the sum of all strategic defence strengths.

Ticket holds the number of tickets that the side has.

ControlPoint holds the number of control points that the side controls.

Time holds the number of seconds since the last start of the map.

StartTime holds the number of seconds since the current active strategy was activated. This value is always set to 0.0 for the enemy.

Attacks holds the number of attacks that the side is currently performing.

Defences holds the number of defences that the side is currently performing.

NumberOfFriendlyAreas holds the number of friendly areas that the side is controlling.

NumberOfNeutralAreas holds the number of neutral areas.

NumberOfEnemyAreas holds the number of areas that the enemy side is controlling.

Flank holds the number of areas with the flag *Flank* that the side is controlling.

Base holds the number of areas with the flag *Base* that the side is controlling.

Close holds the number of areas with the flag *Close* that the side is controlling.

Centre holds the number of areas with the flag *Centre* that the side is controlling.

Remote holds the number of areas with the flag *Remote* that the side is controlling.

Route holds the number of areas with the flag *Route* that the side is controlling.

Bridge holds the number of areas with the flag *Bridge* that the side is controlling.

North holds the number of areas with the flag *North* that the side is controlling.

West holds the number of areas with the flag *West* that the side is controlling.

South holds the number of areas with the flag *South* that the side is controlling.

East holds the number of areas with the flag *East* that the side is controlling.

FrontFlank holds the number of areas the side is controlling that has the flag *Flank* and that have an enemy controlled neighbour.

FrontBase holds the number of areas the side is controlling that has the flag *Base* and that have an enemy controlled neighbour.

FrontClose holds the number of areas the side is controlling that has the flag *Close* and that have an enemy controlled neighbour.

FrontCentre holds the number of areas the side is controlling that has the flag *Centre* and that have an enemy controlled neighbour.

FrontRemote holds the number of areas the side is controlling that has the flag *Remote* and that have an enemy controlled neighbour.

FrontNeutral holds the number of areas that the side is controlling that have a neutral neighbour.

Front holds the number of areas that the side is controlling that have an enemy neighbour.

Safe holds the number of areas that the side is controlling that have no enemy neighbour.

EnemyObject holds the number of areas that the enemy side is controlling.

UnReachable holds the number of areas that the enemy side is controlling that has no neighbours that the side is controlling.

UnitConstant always returns the value 1.0.

Observe that the database that holds the values of the enemy side is mostly an estimate. These values may differ significantly from the ones that the enemy side has in its database concerning itself.

Constant is a constant that is used only when for *createConstantCondition*.

The different ways of creating a condition works in the following ways:

CreateConstantCondition creates a condition that only reads one value from the database and then compares it against a fixed value given by the *constant* parameter. If the *compareSide* is set to *Both*, the result is the same as setting it to *Friendly*.

CreateHomogenousCondition creates a condition that compares the friendly and enemy value that the *Conditional* specifies. Although it is possible to set the *CompareSide* to something other than *Both*, doing that does not make any sense.

CreateHeterogeneousCondition creates the most flexible condition of all the variants. It is possible to create all the conditions above using this method. The heterogeneous condition can compare any possible value with any other possible value.

createAreaCondition

- createAreaCondition *name(string) strategicArea(string) status(enum)*

The area condition is a special condition that does not work like the above described conditions. It is a condition that checks the status of a specific strategic area and returns 1.0 if the status equals *status*, and –1.0 otherwise.

Name is that name of the condition.

StrategicArea is the name of the strategic area that the condition is going to look at.

Status is the status of the strategic area that the condition is going to trigger on. The possible (self explanatory) values of *status* are:

- Owned
- Hostile
- Neutral

setConditionStrength

- setConditionStrength *ConditionStrength(enum)*

The *ConditionStrength* decides how important a condition will be for a strategy. The available strengths are as follows and their functions will be explained in greater detail below:

- Required
- RequiredPositive
- RequiredNegative

- Advisory
- AdvisoryPositive
- AdvisoryNegative

Required is the default strength. When this strength is set, it means that the condition must return a value greater than or equal than 0.0 for the prerequisite that uses it to be able to return any value other than 0.0.

RequiredPositive works the same way as *Required*. Its only reason for being is to be the opposite of *RequiredNegative* below.

RequiredNegative result in the prerequisite returning 0.0 if the condition returns a value greater than or equal 0.0. Otherwise, the condition works as normal.

Advisory strength means that the result of the condition is added to the prerequisite regardless of that result.

AdvisoryPositive strength means that the result will only be added to the prerequisite if it greater than or equal to 0.0.

AdvisoryPositive strength means that the result will only be added to the prerequisite if it smaller than or equal to 0.0.

setIsAbortCondition

- setIsAbortCondition *is(bool)*

This command is optional. If it is set, the condition will not be evaluated if the prerequisite is not the prerequisite of the active strategy. It is set to false by default.

TargetValue

- TargetValue *targetValue(float)*

This command sets the *targetValue* to *targetValue* (!). The *targetValue* is set to 0.0 by default.

CompleteConditions

- CompleteConditions

If a condition has been created using the *createAreaCondition* command, this command must be run once the strategic areas have been loaded. If this is not done, the game will probably crash.

Example

This part of the document will describe how to build a strategy. The first part will describe how the strategies for Kursk were created.

Kursk – An Example

First of all, Kursk is a mirrored map, which means that both sides can use the same strategies. On other maps (i.e. Omaha Beach, El Alamein), the setting requires different approaches for the different sides. This is nothing strange in itself, just some more strategies to write.

What Kind of Strategies are Required?

A game can normally be viewed as consisting of three distinct parts (this may vary from map to map, but I have found it true for most of the 31 maps I've written strategies for, with the possible exception of some assault maps). These three parts are:

- Map start
- Normal battle
- End game

Map Start is the very beginning of the map. This part of the game is about taking positions and rushing any neutral ground. This part of the game usually do not require more than one strategy, however, if it is desired that one side should be able to start differently from time to time, there might be a need for more strategies (the Americans on Market Garden is a good example of this).

Normal Battle is the normal part of the game. In this part, the battle will rage with no clear victor. There should be at least two strategies for this part, unless the map leaves absolutely no choice (i.e. British on Husky). The standard is three strategies, normally something on the lines of "attack west", "attack east", and "attack both".

End Game is the last part of the game when one side is obviously winning. This part of the game requires strategies both if the team is on the loosing side and if it is on the winning side.

Note that if a team loses possession of all its strategic areas, an automatic all out attack order is issued. It will make all bots attack the nearest control point.

Which Strategies are Required?

When creating a strategy, it is important to first know how the map is supposed to play. It is generally quite easy to get a rough idea what the designer had in mind when he/she created the map. Try zooming around the level with a free camera and get a first idea. Then, it is time to go and talk to the designer. Find out what kind of troop movement the designer envisioned when he/she created the map. Find out which control points/choke points/strong points that are important. With this information, it is easy to come up with a few strategies for the different parts of the game (described above).

Choosing Strategies on Kursk

Since Kursk is such a simple map (two control points in the middle of an oval map with each non-take able starting base approximately at equal distance from the middle), the map start and the normal battle turned out to be just the same. The choice is simple; either attack one or both the of the control points.

For the end game, the team that holds both the control points will leave a small force behind at each control point to prevent any commando raids of lone soldiers, while the majority of the units will move towards the enemy base and suppress/spawn camp the enemy.

The loosing side will concentrate all its forces on a single attack in order to try and take a control point back.

The Map Start and Normal Battle Strategies

These two strategies are simple, either attack one or two control points, they have the following definitions:

```
aiStrategy.createStrategy single
aiStrategy.Aggression 1.0
aiStrategy.NumberOfAttacks 1
aiStrategy.NumberOfDefences 0
aiStrategy.TimeLimit 200
aiStrategy.setPrerequisite attackPrereq

aiStrategy.createStrategy double
aiStrategy.Aggression 1.0
aiStrategy.NumberOfAttacks 2
aiStrategy.NumberOfDefences 0
aiStrategy.TimeLimit 200
aiStrategy.setPrerequisite attackPrereq
```

Both strategies have an aggression of 1.0, this is because there is nothing to defend in the home base, and an even stream of attacking units will come from the home base and a captured control point so there should be no need for any defensive forces. This is also why it is possible to use both of these strategies at startup.

There is a time limit of 200 seconds on both strategies. The purpose is that the SAI should change strategy every once in a while.

Both strategies uses the same prerequisite:

```
aiStrategy.createPrerequisite attackPrereq
aiStrategy.addCondition maxOneEnemyCP 10.0
```

The prerequisite is simple, its only purpose is to check if the game is in the end phase or not. For this, it only uses one condition, which has the following definition:

```
aiStrategy.createConstantCondition maxOneEnemyCP Crisp EqualSmaller
Enemy ControlPoint 1
aiStrategy.setConditionStrength Required
```

Yet again, the solution is simple; we are in the end of the game if one of the sides has control over all the control points. Correspondingly, in order to check that we are not in that phase of the game, we just have to check that the enemy has control of no more than one control point. Using a crisp comparison is also sufficient.

The End Game – The Winners

If the team is controlling all the strategic areas, its strategy is changed to keep the enemies away from the control points in order to drain the enemy's tickets. They do so by engaging the enemy in their own base and at the same time leaving a small force behind to guard the control points.

```
aiStrategy.createStrategy holdAndCamp
aiStrategy.Aggression 0.65
aiStrategy.NumberOfAttacks 1
aiStrategy.NumberOfDefences 2
aiStrategy.setPrerequisite holdAndCampPrereq
aiStrategy.setStrategicObjectsModifier Safe          0.2 Owned
aiStrategy.setStrategicObjectsModifier ControlPoint  2.0 Owned
```

The aggression is set to 0.65, meaning that only 35% of the force will be used in manning the two defenses. Since the enemy must pass through the attacking forces, it will be sufficient.

Any safe areas that the SAI's side control, will be unnecessary to defend, at the same time it is important that it is the control points that we are trying to defend. This is the purpose of the strategic object modifiers that are set on the two last lines of the strategy definition.

The prerequisite used has the following definition:

```
aiStrategy.createPrerequisite holdAndCampPrereq
aiStrategy.addCondition threeFriendlyCPCond       3.0
aiStrategy.addCondition minFiveFriendlyAreasCond  2.0
```

This prerequisite is a more complex prerequisite than the previous one. As we will see, this strategy is mutually exclusive to all other available strategies, and the both used conditions are required. This means that the chosen weights of the conditions in the prerequisite are completely arbitrary. However, if there had been more alternatives (like pushing towards the enemy camp with all available units), these weights would have played a crucial role in deciding which strategy would have been more likely to be chosen.

The conditions are defined in the following manner:

```
aiStrategy.createConstantCondition threeFriendlyCPCond Crisp Equal
Friendly ControlPoint 3
aiStrategy.setConditionStrength Required

aiStrategy.createConstantCondition minFiveFriendlyAreasCond Crisp
EqualGreater Friendly NumberOfFriendlyAreas 5
aiStrategy.setConditionStrength Required
```

The first condition checks if there are three (the untakeable control point at the base has to be counted too) control points in the SAI's possession. The second condition checks if there are at least five strategic areas the SAI controls (there are a few more strategic areas on the map than there are control points, basically, this condition checks if the SAI is in control of his part of the map).

The End Game – The Losers

The strategy for the losing side is generally to muster an attack at a single target with all available resources. The idea is simple, if the SAI cannot conquer a control point, it will soon loose. Its only option is to bet all on its final card and try to win a control point.

```
aiStrategy.createStrategy breakOut
aiStrategy.Aggression 1.0
aiStrategy.NumberOfAttacks 1
aiStrategy.NumberOfDefences 0
aiStrategy.setPrerequisite breakOutPrereq
aiStrategy.setStrategicObjectsModifier ControlPoint 4.0
```

Aggression is set to the maximum. Every unit is used for attacking. Only one attack is allowed, and Control Points are greatly prioritized.

The prerequisite consists of two conditions:

```
aiStrategy.createPrerequisite breakOutPrereq
aiStrategy.addCondition noFriendlyCPCond  10.0
aiStrategy.addCondition threeEnemyCPCond  10.0
```

Note that also this strategy is mutually exclusive to all other strategies. The following conditions ensures this:

```
aiStrategy.createConstantCondition noFriendlyCPCond Crisp Equal
Friendly ControlPoint 1
aiStrategy.setConditionStrength Required
```

```
aiStrategy.createConstantCondition threeEnemyCPCond Crisp Equal Enemy
ControlPoint 3
aiStrategy.setConditionStrength Required
```

The first condition checks that the SAI does not control any control points except for the base. The second condition checks if the enemy is controlling three control points.

Additional Notes and Pointers

This section has advice on various aspects on writing a strategy.

Keep it Simple

Try keeping your strategies simple. The more complex strategies can be quite difficult to debug.

Even though the AI engine supports a number of different, more or less, powerful functions, it may be a good idea to stay with the functions that give a more linear behaviour as long as it is possible. Also note that the strategies are rather general and that they do not allow for some more elaborate schemes that a designer might come up with.

Some Notes on Defending

The saying that attack is the best defence is indeed very applicable for Battlefield1942. If there is an attack emanating from a strategic area, chances are, that the own attacking units will meet any attack by enemy forces. This means that if you know which area an attack is going to be launched from, it is generally unnecessary to defend that area.

It is generally better to attack than defend. Battlefield 1942 is an action driven game. That means that it needs action. If a large portion of the bots is defending, chances are that there will be no, or little action.

Debugging

It is impossible to debug the strategies thoroughly. It is better to just try and play the map a few times and see if things work as they should. If that seems to be the case, then try to provoke some of the situations (like end games) that you think might not have been tested previously. Watch out for fragmented map situation, that is when fronts break down and both side's strategic areas cannot be easily separated with a (moderately) straight line. A common situation when this occurs, is when a player takes a fast vehicle and goes behind the front and capture a position far off in the enemy territory.

Bug Reports

Bugs reported in the AI are commonly bugs in the strategies (and unfortunately, the opposite is often true too). Make sure that you monitor all AI-related bugs and figure out which ones are tied to the strategies.

If the Mountain won't come to Mohamed…

Sometimes a map does not work with the AI for any of several reasons. Often, only small changes in terrain and/or object placement to clear up the pathfinding map will correct a problem. Do not be afraid of demanding adjustments of the map, it might be a lot less work, and give a better result than one might think.

The Difference Between NumberOfAttack and NumberOfDefences

Note that there is a significant difference in the meaning between NumberOfAttacks and NumberOfDefences. There can never be more attacks than the number given by NumberOfAttacks, but there may be fewer. There can never be fewer defences than the number given by NumberOfDefences, but there might be more.