Liste primitives nécessaires pour le mode Warbot classique

Introduction : **2**

If & co : **3**

Si et Sinon 3

Ou 3

Variables : **4**

Actions : **5**

Définir objectif 5

Supprimer objectif 5

Conditions : **6**

A un objectif 6

Niveau de vie 6

Niveau de ressource 6

Inférieur, supérieur, égal... 7

Primitives à supprimer ou à remplacer : **8**

Task 8

SelectUnite 8

TargetNearest 8

Near 8

Turn 9

HalfTurn et RandomTurn 9

Not full, not empty 9

Reloaded 9

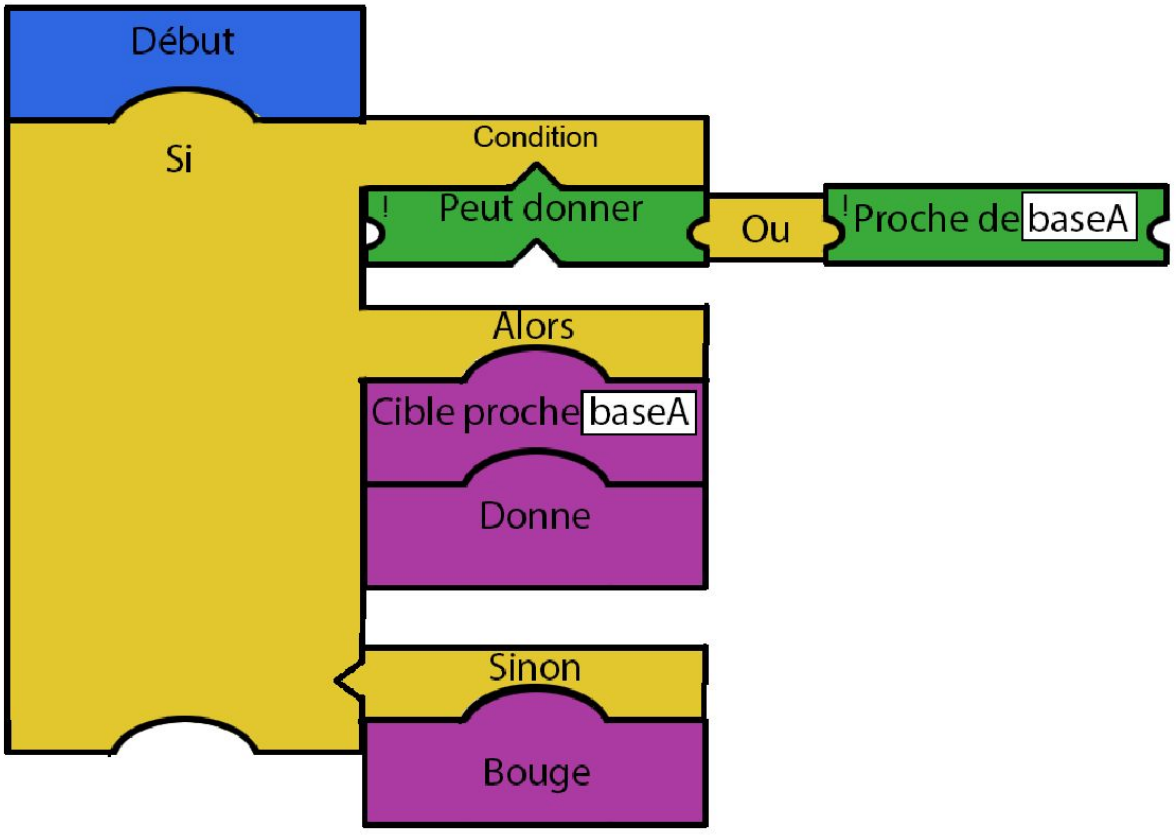
Idée de conception **10**

Bloc comportementale 10

Plusieurs type de variable : 10

Introduction :

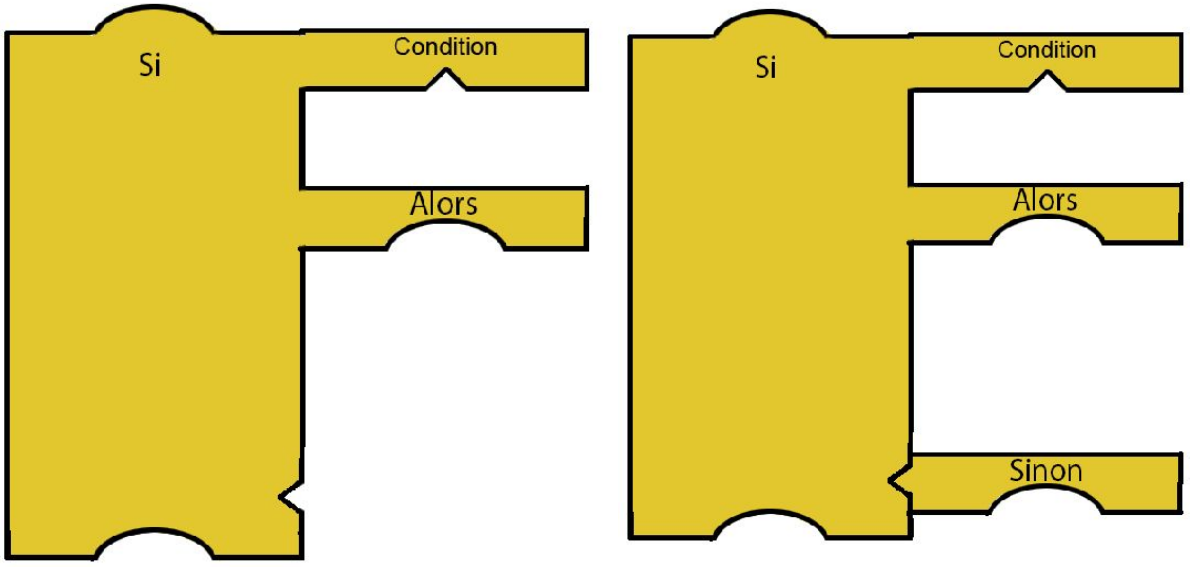
Actuellement, Warbot ne comprend que 3 catégories de primitives (Actions, Condition et Control). Au vu de nos attentes pour le logiciel, il nous semble obligatoire de créer une quatrième catégorie : Variable. Cette dernière contiendra les getters de l'unité.
Le nom de la catégorie "Control" (qui comprend les if et task) ne me plaît pas et il faudra lui trouver un nouveau nom. Pour le moment, elle a été rebaptisé "If & co" parce que c'est déjà plus logique. Le principal problème est que le nom le plus approprié pour cette catégorie serait "Condition" et elle est déjà utilisée par une autre catégorie qui, elle, est composé des primitives qui retourneront des booléens. Ce nom me semble plus approprié à ce groupe qu'à celui des "If & co".
Au vu du public visé qui s'est étendu pour viser des personnes plus jeunes qui ne maîtrise ni l'informatique ni l'anglais, il a été proposé de changer la langue de Warbot pour le français. Les noms des primitives listés dans ce document seront donc en français. **<u>En revanche, nous ne proposons pas de traduction du nom des primitives déjà existantes dans ce document.</u>**
Exemple de code après la refonte du langage :



If & co :

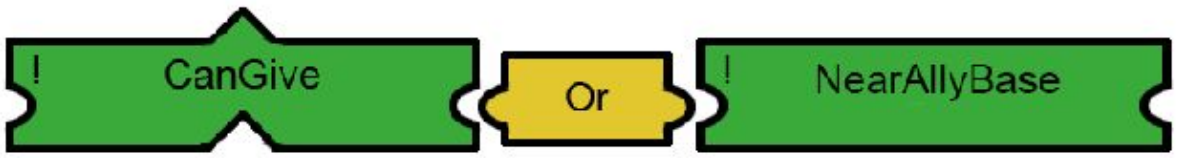
Si et Sinon

Le if classique. Par défaut, la primitive Si est composé des blocs “condition” (dans laquelle on place notre condition) et “alors” (dans laquelle on place les actions qu’on souhaite exécuter si la condition est vraie). Si aucune primitive n’est placé dans la partie condition, on estime que le code présent dans le bloc “alors” doit être exécuté. Il est possible de rajouter une primitive “Sinon” à la primitive “Si” pour exécuter des actions seulement si la condition est fausse.



Ou

L’opérateur or classique. Comme l’opérateur and est exprimé par le fait de placer des primitives de conditions les une en-dessous des autres, nous avons décidé d’exprimer les ou en posant une primitive entre deux primitives alignées. Les primitives à droite de la primitive “ou” perdent leurs “pics”, pour éviter que les utilisateurs puissent exprimer le même and en plaçant une primitive à plusieurs endroits différents.



Variables :

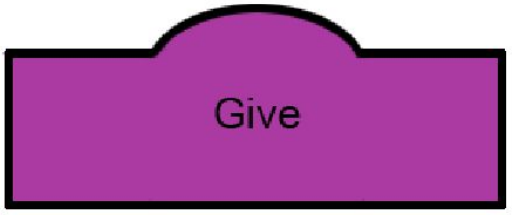
Les variables sont des éléments que l'on utilise comme paramètre pour les autres primitives (condition d'égalité, envoi de message, tourner d'un certain angle…).

- Vie actuelle : la vie restante de l'unité.
- Vie maximale : la vie qu'aurait l'unité si elle n'avait pas pris de dégâts
- Ressource actuelle : la quantité de ressource possédé par l'unité
- Taille inventaire : le nombre maximum de ressource que peut posséder l'unité
- Porté tir : la porté de l'arme de l'unité
- Distance objectif : la distance entre l'unité et son objectif
- Distance target : la distance entre l'unité et sa target
- Objectif : l'objectif sur le long terme de l'unité
- Cible : position de l'objectif à court terme de l'unité.
- Nombre membre : compte le nombre d'unité au sein d'un groupe dont le nom a été passé en paramètre via un textField.

- Lancé de dé : retourne un nombre random entre 1 et la borne maximum, passé en paramètre via un textField n'acceptant que les caractères numériques. Comment coder ce textField
- Constante : permet d'écrire un nombre ou du texte via un textField. Vaut null si rien n'est écrit dans la constante. **Les constantes ne permettent pas de faire de calcul.**

Actions :

Les actions sont les primitives qui agissent sur l'unité. Elles peuvent se placer un peu partout : en dessous de la primitive "Départ", d'une primitive "Si" ou d'une primitive "Sinon" ainsi que dans le bloc "Alors" des primitives "Si". Certaines actions terminent le tour de l'unité (Donner, Avancer, Tirer…) et ont donc une forme différente qui suggère le fait que l'unité ne pourra rien faire après cette action :



Définir objectif

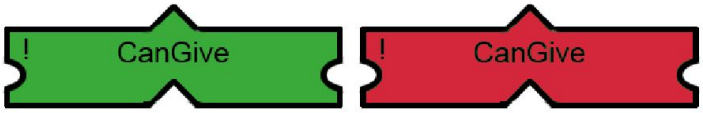
L'objectif est essentiel à la création de meilleures équipes car ce qui s'en rapproche le plus à l'heure actuelle est la cible mais elle se perd à chaque début de tour.
"Définir objectif" permet d'affecter la valeur présente dans la variable cible dans la variable objectif afin que l'unité ne perde pas son objectif de vue. Cette primitive ne termine pas le tour de l'unité.
Attention : l'objectif représente simplement un point dans l'espace. Il est impossible de placer une unité dans cette variable pour éviter que la primitive ne soit trop forte (sinon il serait tout à fait possible de suivre une unité mobile que l'on a plus dans son champ de vision).

Supprimer objectif

Une fois un objectif accompli, il doit être possible d'exprimer le fait que l'unité n'a plus d'objectif. Comme il n'existe pas de moyen de dire que l'on a plus de cible et qu'une telle primitive serait inutile vu que la cible se réinitialise à chaque nouveau tour, il vaut mieux modifier directement la valeur de l'objectif. CEtte primitive ne termine pas le tour de l'unité.

Conditions :

Les conditions se mettent dans le bloc “condition” des primitives “si”. Pour rendre claire quelles primitives se placent à quels endroits, la forme des primitives Conditions est différente de celle des primitives Actions.
Il est possible d’exprimer l’inverse de la condition en cliquant sur le “!” eu haut à gauche de la primitive. Après un clic, la couleur de la primitive change :



A un objectif

Renvoi true si l’unité a un objectif, false sinon.

Niveau de vie

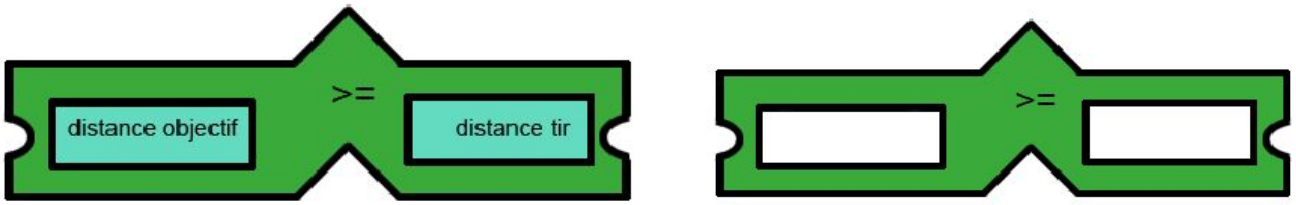
Prend en paramètre un chiffre entre 0 et 1. Retourne true si la vie de l’unité est supérieure ou égale à ce chiffre multiplié par la vie maximale de l’unité.

Niveau de ressource

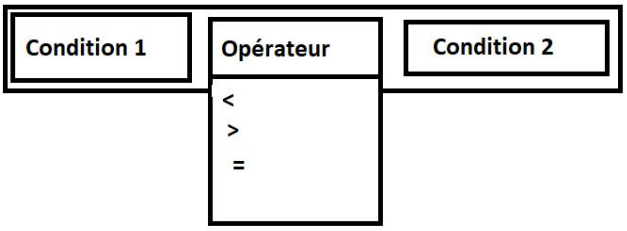
Prend en paramètre un chiffre entre 0 et 1. Retourne true si la quantité de ressource de l’unité est supérieure ou égale à ce chiffre multiplié par la capacité maximale de l’inventaire de l’unité.

Inférieur, supérieur, égal...

Prend deux variables en paramètre. Renvoi le résultat du test d'égalité/supériorité/infériorité.



L'opérateur se modifie via un menu déroulant sous cette forme :



Les opérateurs disponibles sont : égalité, non égalité, supérieur, supérieur ou égal, inférieur, inférieur ou égal. Vu que tous les opérateurs sont disponibles, il n'est pas nécessaire d'utiliser le "!" permettant d'inverser la condition.

Primitives à supprimer ou à remplacer :

Task

Task est une primitive qui a actuellement la même fonction que la primitive “si” mais ne dispose pas de bloc “sinon”. Au vu des modifications sur cette dernière, task est devenue inutile : il faut donc la supprimer.

Not full, not empty

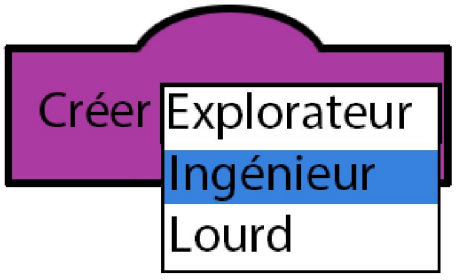
Vu qu’il sera possible d’exprimer la négation d’une condition, ces primitives sont devenus inutiles.

Reloaded

Plutôt que de tester si l’unité est rechargé, il est plus intéressant de savoir si l’unité peut tirer : en plus du test de rechargement, cette nouvelle primitive fera également un test pour savoir si la distance entre la cible et l’unité est inférieur à la porté maximale de l’arme. La primitive sera renommé “peut tirer”.

SelectUnite

Les primitives SelectWarEplorer, SelectWarHeavy, SelectWarEngineer… ne servent que pour les primitives “peut créer” et “créer unité”. Il est plus intéressant de modifier les primitives “peut créer” et “créer unité” pour qu’elles prennent en paramètre un type d’unité, rendant ainsi les primitives de sélection inutiles :



TargetNearest

Plutôt que d'avoir près de vingts primitives différentes qui ont en réalité le même objectif, il est plus intéressant de créer une simple primitive "Cible" qui prend en paramètre le type de la cible voulu (base alliée, message, explorateur ennemi…) sous la même forme que ce qui est prévu pour la création d'unité.
Il faudra également rajouter un champ pour pouvoir cibler son objectif car il doit être possible de tirer sur des unités n'apparaissant pas dans notre rayon de perception. Comme la primitive "peut tirer" calcul la distance entre l'unité et la cible, il doit être possible de cibler son objectif pour savoir si on peut tirer à la position de l'objectif.

Near

Pour les mêmes raisons que pour TargetNearest, il est plus judicieux d'ajouter un paramètre à une simple primitive "Proche de" plutôt que de conserver autant de primitives similaires.

Turn

Actuellement, cette primitive fais tourner l'unité dans la direction de sa cible. Cependant, nous avons maintenant deux variables vers lesquelles nous souhaitons que l'unité se tourne : la cible et l'objectif. Il faut donc rajouter un paramètre sous la forme d'un menu déroulant pour laisser l'utilisateur choisir s'il veut que son unité se tourne vers l'objectif ou la cible.

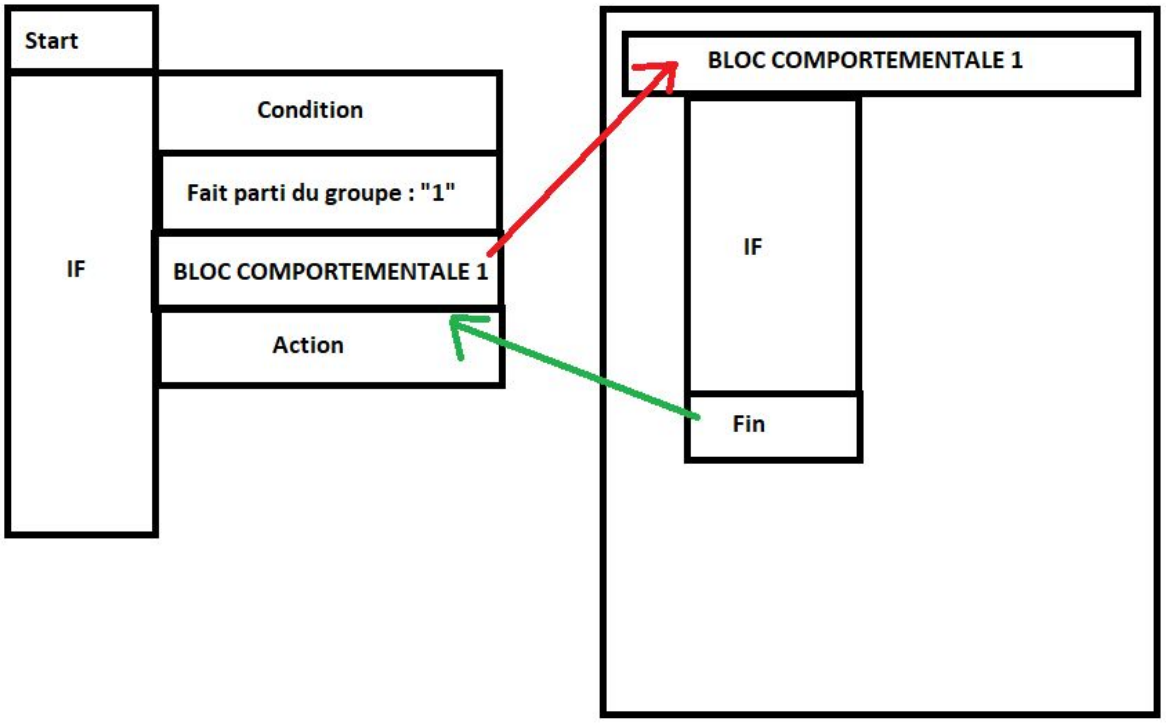
HalfTurn et RandomTurn

Il est important de pouvoir faire demi-tour. Il nous semble tout aussi important de pouvoir tourner d'un angle précis, sans forcément cibler quelque chose. Nous proposons d'ajouter un paramètre à cette primitive et de la renommer "tourner".
Il devient alors possible de placer la variable "lancé de dé" en paramètre, rendant la primitive RandomTurn inutile.

Idée de conception

Bloc comportementale

Ce bloc se diviserait en deux parties : un premier bloc s'intégrerai dans le comportement de base. Une fois ajoutée, une nouvelle zone permettant de définir un comportement s'afficherai. Fonctionnalité pouvant être utile dans le cas d'une unité ayant des comportements complexes différents en fonction du groupe auquel il appartient par exemple.



Plusieurs type de variable :

Il pourrait être intéressant de multiplier les formes des primitives de variables pour empêcher les utilisateurs de placer en paramètres de primitives des variables qui ne correspondent pas eu type demandé. Cependant, il nous semble difficile de rendre l'éditeur de comportement générique si nous choisissons d'implémenter ce genre de comportement. De plus, il faudrait apprendre quelle forme est associée à quelle primitive et on perdrait en simplicité de prise en main.