Vers un modèle de stratégie basé sur la gestion des ressources pour la création des IA dans les RTS

Juliette LEMAITRE [1,2] Domitile LOURDEAUX [1] Caroline CHOPINAUD [2]

[1] Sorbonne universités, Université de technologie de Compiègne, CNRS, Heudiasyc - UMR CNRS 7253, CS 60 319, 60 203 Compiègne cedex, France
[2] MASA Group, 8 rue de la Michodière, 75002 Paris, France

juliette.lemaitre@hds.utc.fr

Résumé

Dans les jeux de stratégie en temps réel existants à ce jour, les comportements des opposants au joueur sont souvent critiqués. Les stratégies qu'ils suivent ne sont plus suffisamment intéressantes dès lors que l'on dépasse le stade du joueur débutant ou de l'entraînement, car trop simples et prédictibles. Le mode multijoueurs est alors largement préféré par une majorité de joueurs chevronnés. Pourtant de nombreuses recherches ont été menées sur le sujet de l'IA dans les jeux vidéo et en particulier pour les RTS, qui permettent aux entités d'exhiber des comportements de plus haut niveau et très efficaces pour remplir les objectifs du jeu. Mais les solutions proposées sont trop compliquées à comprendre et à utiliser pour intéresser le milieu industriel du jeu vidéo, soumis à de fortes contraintes de temps. Pour permettre le développement d'IA présentant un plus grand challenge, nous proposons dans ce papier un modèle de stratégie qui se veut intelligible et qui permet de créer facilement des comportements plus complexes que ceux actuellement proposés, capable de s'adapter à la dynamique du jeu.

Mots Clef

Stratégie, modélisation, game design.

Abstract

The artificial intelligence used for opponent non-player characters in commercial real-time strategy games is often criticized by players. It is used to discover the game but soon becomes too easy and too predictable. Yet, a lot of research has been done on the subject, and successful complex behaviors have been created, but the systems used are too complicated to be used by the video games industry, as they would need time for the game designer to learn how they function, which ultimately proves prohibitive. Moreover these systems often lack control for the game designer to be adapted to the desired behavior. To address the issue, we propose an accessible strategy model that can adapt itself to the player and can be easily created and modified by the game designer.

Keywords

Strategy, Behavior Modeling, Game Design.

1 Introduction

Les Intelligences Artificielles (IA) contrôlant les opposants dans les jeux vidéo commerciaux ne tirent pas pleinement avantage des résultats de la recherche académique sur le sujet. Certains systèmes comme la planification ou même les algorithmes d'apprentissage ont déjà été utilisés par des jeux[7, 4], mais cela reste des exceptions. La principale raison est l'existence de trop grands écarts entre leurs buts et leurs contraintes, rendant les solutions proposées par la recherche académique peu adaptées à l'industrie du jeu.

Au cours de la création d'un jeu vidéo, la conception des IA opposantes est rarement la priorité, on lui préfère souvent d'autres aspects comme les graphismes ou l'animation, rendant la contrainte de temps d'autant plus critique. Cette forte contrainte temporelle ne permet généralement pas au studio de développer une solution d'IA poussée et réutilisable d'un jeu à l'autre. Ces développements spécifiques obligent alors à systématiquement créer de nouvelles IA à partir de zéro. Nous cherchons donc à proposer une solution rapidement intégrable dans ce processus, qui facilite et accélère la création de stratégies tout en permettant leur réutilisabilité.

Les jeux de stratégie en temps réel (RTS) possèdent des environnements présentant de nombreux challenges pour la recherche en IA qui les utilise comme environnements de test. Pourtant, même si ces environnements nécessitent des IA complexes, les solutions utilisées par l'industrie pour créer les IA intégrées au jeu sont trop simples et mal-adaptées. En effet, des solutions de type machines à états (FSM), arbres de décisions (BT), ou dans le pire des cas, des scripts sont préférés à des solutions plus complexes car elles apparaissent comme simples d'utilisation et intelligibles de prime abord.

Cette simplicité n'est pas la seule raison pour le choix de ces systèmes d'IA, en effet les BT et FSM offrent non seulement une simplicité de compréhension et d'utilisabi-

lité mais aussi de la fiabilité et du contrôle. Au moins un de ces critères manquent à la plupart des solutions proposées par la recherche académique : certains systèmes ne trouvent pas toujours de solution et les entités contrôlées peuvent se retrouver inanimées ; il est également souvent difficile voire impossible pour les systèmes proposés de personnaliser la prise de décision pour obtenir le comportement voulu. En effet, l'objectif orientant la plupart des recherches académiques est l'optimisation des résultats (plus rapides ou plus efficaces) alors qu'un jeu vidéo a pour objectif de divertir le joueur. Si ces deux buts ne sont pas incompatibles, ils sont bien souvent résolus par des solutions différentes.

Pour résumer, l'industrie du jeu vidéo est soumise à des contraintes de temps qui diffèrent de celles de la recherche académique, et qui nécessitent d'utiliser des systèmes simples de compréhension et d'utilisation, d'autant plus pour la conception d'IA qui est rarement une priorité. De plus l'objectif d'un jeu vidéo étant de produire une expérience divertissante pour le joueur, les IA créées doivent être faciles à contrôler et personnaliser pour qu'elles contribuent à l'expérience souhaitée par le game designer. Dans le but de répondre à l'ensemble de ces attentes, nous proposons une solution pour concevoir et tester de manière efficace des comportements d'opposants dans les RTS, qui prend la forme d'un modèle de "stratégies" adaptatives et réutilisables. A partir d'une présentation des limites des approches actuellement existantes, nous détaillerons notre modèle de conception de stratégie et nous discuterons de ses avantages et de sa place dans une approche plus globale d'aide à la conception de stratégies.

2 Etat de l'art

Un aperçu des travaux réalisés au sujet des IA dans les RTS a été publié par Ontanon [6], il y cite à la fois le travail effectué dans le cadre des compétitions d'IA pour le jeu Starcraft organisées lors des conférences CIG et AAAI, et également les autres travaux de recherche qui ont été conduits sur le sujet.

Plusieurs raisons font des compétitions mentionnées ci-dessus de mauvais candidats pour fournir un système utilisable par l'industrie. Premièrement, la victoire lors des duels est le seul aspect servant à la comparaison de deux candidats. Le principe général est de mettre le système d'IA de chaque participant face aux systèmes de ses adversaires et de comparer les pourcentages de jeux gagnés, perdus ou nuls pour effectuer un classement. Deuxièmement, les architectures créées pour ces événements sont spécifiques au jeu Starcraft, elles sont découpées à la fois de façon parallèle et hiérarchique en sous-modules[6] correspondant aux spécificités du jeu. La spécialisation des sous-modules les rendent difficiles à réutiliser dans un environnement différent, cependant cela montre la complexité de la tâche de décision et l'importance de la décomposer. Avec notre modèle nous cherchons à fournir une structure de stratégie qui lui permette d'être réutilisée dans des environnements diverses.

Les environnements de RTS sont très utilisés dans la recherche académique comme environnement de test pour les nombreux challenges qu'ils présentent [1]. Plusieurs techniques d'IA y ont été appliquées pour tester leur efficacité mais elles manquent souvent de l'utilisabilité nécessaire pour être utilisées par l'industrie. Par exemple [3] utilise des ensembles de logs de jeux sur Starcraft pour créer des modèles de Markov cachés. Les comportements obtenus s'appuient intégralement sur des probabilités le rendant imprédictible et diminuant grandement le contrôle du game designer sur les comportements en résultant.

La planification à base de cas appliquée dans [5], et étudié plus largement dans [8] manque également de contrôle sur les comportements obtenus, car elle se base exclusivement sur des bibliothèques d'exemples obtenus à partir de jeux joués et commentés par des experts. Le fonctionnement de la planification à base de cas rend également difficile la compréhension des raisons de la génération de comportements non souhaités.

De plus, ces deux systèmes utilisent une méthode d'apprentissage qui peine à s'introduire dans le processus de création de jeux vidéo. En effet l'apprentissage est un processus long et qui nécessite une quantité importante de données en entrée. Si l'apprentissage peut faire économiser du temps lors de l'étape de création des comportements, les données nécessaires au processus peuvent prendre du temps à être générées. De plus, pour que des modifications puissent être apportées aux comportements produits par apprentissage, il est indispensable que les résultats obtenus soient intelligibles par le game designer, ce qui rend le processus d'autant plus complexe.

La planification a également été testée lors de travaux de recherche, [2] l'utilise pour optimiser l'ordre des constructions dans Starcraft. Cette méthode peut donc être efficace sur une sous partie du raisonnement mais à cause du vaste espace de recherche, son utilisation pour le mécanisme entier de prise de décision est impossible car nécessitant un temps trop important de calcul non compatible avec des jeux en temps réel. De plus il peut être compliqué lors de l'observation d'un comportement non voulu, d'en comprendre les raisons.

3 PROPOSITION

L'objectif de notre travail est de fournir un modèle de stratégie accessible qui permet de simplifier la création de comportements complexes. Une stratégie est définie comme le processus de prise de décision pour l'allocation des ressources aux sous-tâches de façon à poursuivre un objectif global. Les comportements produits doivent être facilement modifiables et réutilisables. Cela permettra au système d'être facilement intégrable au processus de création de jeu vidéo durant lequel des modifications fréquentes des mécaniques de jeu nécessitent l'adaptation de tous les éléments, y compris de l'IA. Nous voulons également que le modèle soit appréhendable, autrement dit que la rai-

son de l'apparition d'un comportement non voulu peut être compris et résolu facilement de façon à permettre aux game designers de garder le contrôle sur les comportements obtenus.

3.1 Le modèle de stratégie

Notre modèle a pour objectif de faciliter la création de comportements collectifs dans les RTS, et plus généralement de comportements impliquant plusieurs agents ayant besoin de se coordonner. Pour construire une stratégie, nous utilisons une structure hiérarchique qui permet sa décomposition en sous-comportements plus simples, permettant ainsi au concepteur de se concentrer sur le niveau d'abstraction voulu et d'ajouter facilement un niveau supérieur utilisant les comportements précédemment créés. Un comportement est donc composé d'un ensemble de sous-comportements et la stratégie peut alors être représentée par un graphe orienté acyclique (DAG) avec un unique noeud racine et dans lequel les fils d'un noeud correspondent à ses sous-comportements. Le schéma 1 représente un exemple partiel de stratégie qui correspond à celle décrite dans le paragraphe 4. Tous les noeuds sous la forme d'un losange sont des tâches primitives, c'est-à-dire qu'elles peuvent être directement exécutées dans l'environnement virtuel, elles ne possèdent pas de sous-comportements. Les ronds et les carrés ne possédant pas d'arcs sortants sont incomplets et possèdent en réalité des sous-comportements qui ne sont pas représentés ici pour une question de place.

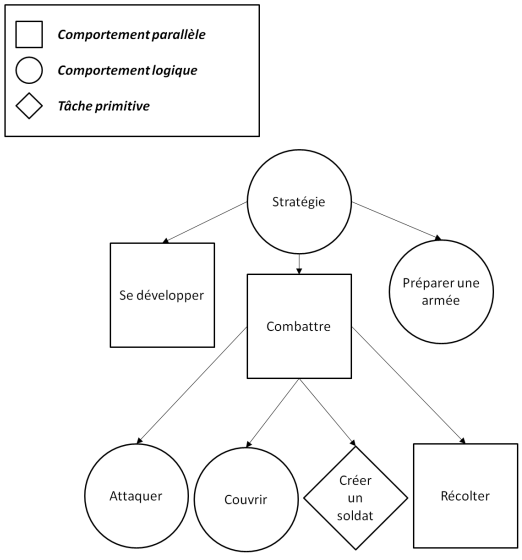

FIGURE 1 – Stratégie.

Un comportement est défini par un ensemble de sous-comportements, par son type, et par les ressources dont il a besoin. Le type d'un comportement peut être soit logique soit parallèle et détermine le mode d'utilisation des sous-comportements. Pour chaque comportement, l'ensemble des sous-comportements associé peut être composé d'un mélange de comportements parallèles, logiques ainsi que de tâches primitives. Le type parallèle aborde la problématique d'allocation des ressources car il va effectuer plusieurs de ses sous-comportements en même temps ; il est représenté dans nos graphes par un noeud carré. Le type logique permet à l'IA de choisir un seul de ses sous-comportements en fonction de l'état courant du jeu ; il est représenté dans nos graphes par un noeud rond. Chaque type a besoin d'informations supplémentaires relatives à leur type ; leur fonctionnement est davantage détaillé dans la suite de cette partie.

Tâche Primitive. Une tâche primitive correspond à la plus petite unité de décomposition d'un comportement. Elle ne peut plus être décomposée, mais elle est liée à une fonction qui lui permet d'être exécutée dans l'environnement virtuel. Elle est définie par un ensemble de ressources qui lui sont nécessaires pour être effectuée sous la forme de tuples *<R, Min, Max>* où :

```
R est le type de ressource
Min est la quantité minimum nécessaire
Max la quantité maximum possible
```

R est défini grâce au modèle de ressources présenté dans le paragraphe 3.2. Max peut être mis à -1 s'il n'y en a pas. Cette représentation permet une grande flexibilité et est adaptée pour des quantités variables de ressources, comme c'est le cas pour les jeux de type RTS. Par exemple, pour une tâche de construction, on peut lui associer les ressources suivantes :

```
<<ouvrier, 1, -1>, <terrain, 1, 1>>
```

De cette manière, plusieurs ouvriers peuvent être affectés à la même tâche de construction qui n'utilisera par contre qu'une unité de terrain. De façon à gérer le résultat d'une tâche, celle-ci peut retourner une valeur de retour qui sera envoyée au comportement parent, qui pourra alors la prendre en compte pour la suite du comportement.

Comportement Logique. Un comportement logique permet de représenter une logique de sélection de l'un des sous-comportements. Son exécution consiste à sélectionner le sous-comportement devant être exécuté, en tenant compte du sous-comportement précédemment sélectionné et de l'état courant du monde. Il est représenté par un tuple *<B, M, SB, CB>* où :

```
B est un ensemble d'états
M est un ensemble des transitions <OB, T,
    DB>
SB est le sous-comportement initial
CB est le sous-comportement couramment
    sélectionné
```

Une transition est composée d'un déclencheur T qui déclenche la sélection d'un état DB si OB est l'état courant. Un état peut être soit un sous-comportement, soit un état retour qui déclenche l'envoi d'un signal renfermant une valeur de retour au comportement parent.
Les déclencheurs peuvent être des signaux internes ou externes, ou des requêtes d'information : les signaux internes

correspondent aux retours du sous-comportement courant, les signaux externes viennent de modules dédiés au jeu, et les requêtes d'information sont représentées par des formules booléennes. Par exemple, un signal externe peut venir d'un module qui gère l'état du monde si la transition doit être activée par un changement dans l'état du monde, ou alors le signal peut venir d'un module de modélisation du joueur si la condition dépend de l'état du joueur. Dans le cas d'une requête d'information, la transition est activée si la formule est vraie. Les signaux sont utilisés pour des événements ponctuels alors que les requêtes d'information sont utilisées pour observer des états du monde plus statiques.

Considérons un exemple consistant de 3 sous-comportements : *Explorer*, *Combattre*, et *Récolter*. *Explorer* est défini comme étant le sous-comportement initial, puis, si des ennemis sont rencontrés ou de la nourriture est trouvée durant l'exploration du monde, on sélectionne respectivement le sous-comportement *Combattre* ou *Récolter*. Quand les sous-comportements *Combattre* ou *Récolter* renvoient un signal de succès, le comportement *Explorer* est de nouveau sélectionné. Dans le cas où des ennemis sont rencontrés alors que le sous-comportement *Récolter* est en cours de réalisation, le sous-comportement *Combattre* est préféré. Cet exemple peut être représenté comme suit, le comportement *Récolter* étant le sous-comportement actuellement sélectionné.

```
B = {Explorer, Combattre, Récolter}
M = {
      <Explorer, Signal(EnnemiRepéré),
          Combattre>,
      <Explorer, Signal(NourritureRepérée),
           Récolter>,
      <Combattre, Signal(Succès), Explorer
          >,
      <Récolter, Signal(Succès), Explorer>,
      <Récolter, Signal(EnnemiRepéré),
          Combattre> }
SB = Explorer
CB = Récolter
```

Un comportement logique peut ainsi être représenté par un graphe orienté qui peut être cyclique, les sous-comportements et les signaux de retour étant les noeuds et M énumérant les arcs avec OB le noeud origine et DB le noeud destination. Un graphe illustrant l'exemple décrit plus haut est représenté sur la figure 2.

Comportement Parallèle. Un comportement parallèle répartit les ressources qui lui ont été allouées entre ses différents sous-comportements. Ceux-ci sont distribués entre plusieurs niveaux de priorité qui sont représentés par des couples <*M, C*>. M correspond au nombre de fois maximum où le niveau peut être exécuté en parallèle, mais il est facultatif : il est notamment inutile lors de la définition des tâches par défaut qui doivent donc être allouées à toutes les ressources restantes qui peuvent l'effectuer, quelque soit leur nombre. Par convention, nous utiliserons la valeur -1

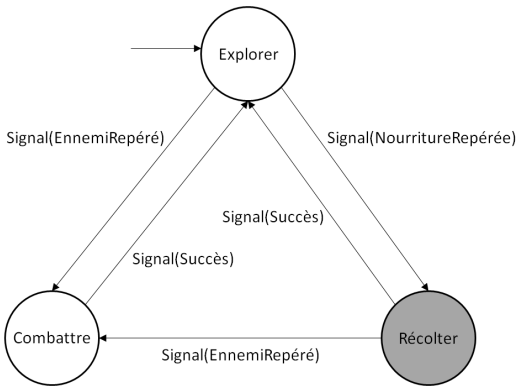

FIGURE 2 – Exemple simple de comportement logique.

pour indiquer qu'il n'y a pas de limite. C représente un ensemble de couples <*B, W*> où B est un sous-comportement et W une quantité, correspondant au nombre de fois où le sous-comportement est effectué à chaque exécution du niveau.

Les ressources disponibles sont d'abord allouées au niveau de plus haute priorité, et cela tant que c'est possible et dans la limite de M fois. Pour qu'une instance d'un niveau de priorité soit ajouté à la liste d'exécution, il faut que les ressources disponibles permettent d'effectuer l'ensemble de ses sous-comportements, autant de fois qu'indiqué par les quantités W. Un sous-comportement peut apparaître dans plusieurs niveaux de priorité, mais il ne peut apparaître qu'une seule fois à l'intérieur d'un niveau. Cette description permet une distribution des ressources qui s'adapte à la quantité des ressources disponibles.

TABLE 1 – Exemple simple de comportement parallèle.

Itérations maximum	Sous-comportement	Quantité
2	Combattre	1
	Couvrir	2
1	Alerter	
-1	Explorer	

Le tableau 1 décrit un exemple de comportement parallèle. Le premier niveau, pour être réalisé, a besoin de déclencher un sous-comportement *Combattre* et deux sous-comportements *Couvrir*, il peut être effectué au maximum 2 fois. Le deuxième niveau a besoin pour être réalisé de déclencher un sous-comportement *Alerter* et ne peut être effectué qu'une seule fois. Le dernier niveau représente le comportement par défaut pour toutes les ressources restantes, il a donc pour nombre maximum -1, et a besoin pour se réaliser de déclencher un sous-comportement *Explorer*.

En considérant que chacun des sous-comportements ne nécessite comme ressource qu'un soldat, et que nous avons à notre disposition 10 soldats, ils se répartiront alors comme suit :

- le premier niveau sera d'abord sélectionné, pour être

réalisé il a besoin de 3 soldats, 1 pour combattre, et 2 pour couvrir. Il nous reste donc 7 soldats.
- le premier niveau n'a été effectué qu'une fois, le nombre maximum d'itérations n'est donc pas encore atteint, il nous reste assez de soldats pour effectuer de nouveau le premier niveau, 3 soldats sont donc de nouveau alloués. Il nous reste 4 soldats.
- le nombre maximum d'itérations du premier niveau est atteint, nous passons donc au deuxième niveau qui a besoin d'un seul soldat, il nous en reste de disponible, un soldat est donc alloué pour le sous-comportement *Alerter*. Il nous reste 3 soldats.
- le nombre maximum d'itérations est atteint pour le deuxième niveau, nous passons donc au troisième qui a besoin lui aussi d'un seul soldat. Le niveau n'ayant pas de nombre maximum d'itérations, il sera effectué autant de fois qu'il reste de soldats, et le sous-comportement *Explorer* sera donc alloué à chacun des trois soldats restants.

3.2 Le modèle de ressources

Une ressource peut être un objet, un agent, mais aussi un lieu, ou bien une compétence. Une hiérarchie de ressources doit être définie et seules les feuilles peuvent être utilisées pour définir le type d'une ressource lors de sa création dans l'environnement virtuel. En revanche, tous les types peuvent être utilisés pour définir les ressources nécessaires à une tâche primitive, autrement dit, quand une ressource est nécessaire pour la réalisation d'une tâche ou d'un comportement, son type peut être plus ou moins spécifique. Par exemple, la figure 3 représente une hiérarchie de types de ressources. Un *Agent* d'un jeu pour lequel cette hiérarchie est utilisée peut être soit un *Soldat* soit un *Ouvrier*, l'environnement inclut également des ressources de type *Epée* et *Pelle*. Cela signifie que si une tâche primitive a pour ressource nécessaire une ressource de type *Agent*, elle pourra être réalisée par un un *Soldat* ou bien par un *Ouvrier*. Des options plus avancées permettent au game designer d'indiquer que si une tâche a besoin de deux ressources de type *Agent*, ils ne doivent pas avoir le même sous-type, ou bien au contraire ils doivent avoir le même, ou bien encore cela n'a pas d'importance.
Une ressource est caractérisée par les propriétés suivantes : elle possède un type qui ne peut pas changer au cours de son existence ; une ressource peut être créée ou détruite par une tâche primitive ; la modification du type d'une ressource peut donc être simulée avec une destruction puis une création.

4 CAS D'USAGE

L'exemple suivant illustre un autre cas d'utilisation de notre modèle pour construire une stratégie dans un environnement utilisant quelques mécanismes des jeux RTS : la recherche et l'extraction de ressources minérales servant à la création de bâtiments et d'unités (ouvriers et soldats) ;

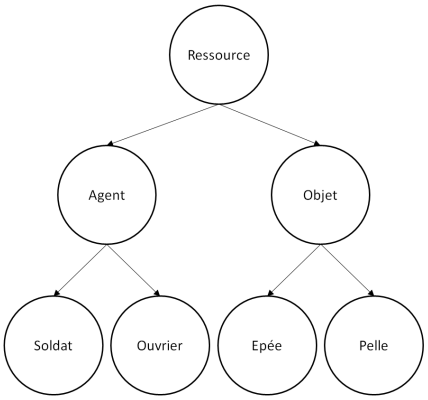

FIGURE 3 – Hiérarchie de ressources.

les bâtiments sont utiles pour augmenter la protection de la ville et l'efficacité des soldats produits. Pour simplifier la compréhension de la stratégie proposée, seules certaines parties seront détaillées.
La stratégie générale est présentée sur la figure 4, on y distingue trois phases du jeu : elle débute par une phase d'expansion durant laquelle des bâtiments sont créés jusqu'à ce qu'il soit considéré nécessaire de passer en phase de préparation au combat avec la construction d'une armée, pour finir inexorablement par une phase de combat avec l'ennemi. Plusieurs raisons peuvent expliquer la nécessité de créer une armée : cela peut être une durée de jeu, le nombre de bâtiments ayant déjà été créés, mais également si par espionnage il a été découvert que l'ennemi préparait une armée. Si l'ennemi attaque par surprise sans que la construction d'une armée est pu être débutée, le combat est directement engagé.

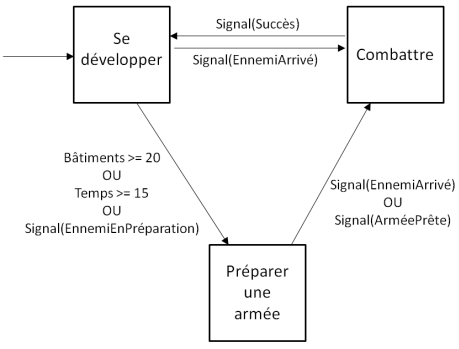

FIGURE 4 – Stratégie globale de l'exemple.

Le sous-comportement *Combattre* peut être décrit par un comportement parallèle, comme celui représenté dans le tableau 2. Les soldats disponibles vont alors commencer le combat pendant que les ouvriers continuent de collecter des ressources pour qu'elles puissent être utilisées pour créer davantage de soldats. Dans cet exemple, chaque soldat qui attaque un ennemi est couvert par un autre soldat.

Le sous-comportement *Récolter* est le comportement par défaut pour les ouvriers et est donc placé en dernier. Si les ressources nécessaires pour construire un soldat ne sont pas disponibles, le sous-comportement *Créer un soldat* ne sera pas effectué, et tous les ouvriers récolteront des ressources. Dès que les ressources nécessaires seront disponibles, un ouvrier sera assigné à cette tâche.

TABLE 2 – Comportement parallèle *Combattre*.

Itérations maximum	Sous-comportement	Quantité
-1	Attaquer	1
	Couvrir	1
-1	Créer un soldat	
-1	Récolter	

Les sous-comportements *Attaquer* et *Couvrir* sont respectivement détaillés sur les figures 5 et 6. Le comportement *Attaquer* est relativement simple : tirer sur l'ennemi à portée de tir le plus faible et si aucun ennemi n'est à portée de tir alors se déplacer vers l'ennemi le plus proche. Le comportement *Couvrir* est un peu plus complexe et nécessite des signaux externes et des requêtes d'information sur l'environnement. Il consiste à trouver une place pour se mettre à couvert, y déplacer le soldat, le mettant accroupi si nécessaire, puis tirer sur les ennemis visibles. Ensuite, si le soldat couvert s'est déplacé et que la place à couvert actuelle n'est alors plus adaptée, le soldat doit être déplacé.

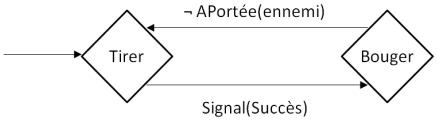

FIGURE 5 – Comportement logique *Attaquer*.

5 CONCLUSION

Dans ce papier nous avons présenté un modèle pour la définition de stratégies pour les jeux de type RTS dont la spécificité est de combiner la facilité d'utilisation avec la capacité à produire des comportements stratégiques performants. Ce modèle utilise les principes de hiérarchie et de parallélisme pour être facilement compréhensible et utilisable. Les agents sont manipulés à la manière de ressources, en utilisant des niveaux de priorité pour les répartir sur différentes tâches, rendant le modèle adaptable pour des ressources aux quantités variables. Une stratégie peut être facilement modifiée et ses sous-comportements peuvent facilement être extraits et réutilisés. Pour la suite il est nécessaire de prévoir une phase de test en faisant appel à des game designers afin d'évaluer la simplicité de compréhension et d'utilisation du modèle ainsi que son expressivité, de façon à étendre celle-ci en ajoutant des fonctionnalités répondant à leurs besoins tout en gardant l'utilisabilité. Ce modèle fait partie d'un projet plus vaste qui a pour ambition de fournir un générateur de stratégies, autrement dit un système qui proposerait au game designer une première stratégie à partir de laquelle travailler, et ce simplement en renseignant les mécanismes du jeu.

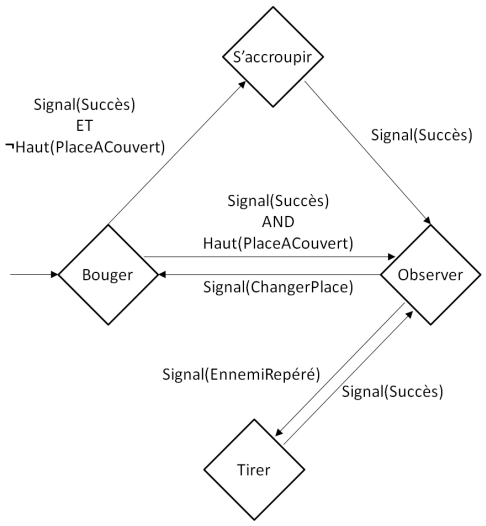

FIGURE 6 – Comportement logique *Couvrir*.

Références

[1] M. Buro. Real-time strategy games : A new AI research challenge. In *IJCAI*, pages 1534–1535, 2003.

[2] D. Churchill and M. Buro. Build order optimization in StarCraft. In *AIIDE*, 2011.

[3] E. W. Dereszynski, J. Hostetler, A. Fern, T. G. Dietterich, T.-T. Hoang, and M. Udarbe. Learning probabilistic behavior models in real-time strategy games. In *AIIDE*, 2011.

[4] R. Evans. Varieties of learning. *AI Game Programming Wisdom*, 2, 2002.

[5] S. Ontanon, K. Mishra, N. Sugandh, and A. Ram. Case-based planning and execution for real-time strategy games. In *Case-Based Reasoning Research and Development*, pages 164–178. Springer, 2007.

[6] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game AI research and competition in StarCraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(4) :293–311, 2013.

[7] J. Orkin. Three states and a plan : the ai of fear. In *Game Developers Conference*, volume 2006, page 4, 2006.

[8] R. Palma, P. A. Gonzalez-Calero, M. A. Gomez-Martin, and P. P. Gomez-Martin. Extending case-based planning with behavior trees. In *FLAIRS Conference*, 2011.