MAGS-COA : Une extension de MAGS pour représenter, simuler et critiquer des suites d'actions

Rapport de contrat

Auteurs

Jimmy Perron, Jimmy Hogan, Hedi Haddad

NSim Technology

Laboratoire d'informatique cognitive
Université Laval

Direction du projet

Micheline Bélanger
RDDC-Valcartier

Bernard Moulin
Laboratoire d'informatique cognitive
Université Laval

[W7701-4-3825]

[DRDC Valcartier CR 2006-049]

Février 2006

Résumé

Le présent contrat vise à explorer, définir et implémenter un environnement capable de supporter le développement d'un système de critique de suites d'actions militaires. L'approche de critique retenue consiste à simuler la suite d'actions dans un environnement géo-spatial et à analyser le résultat de la simulation pour générer des critiques. Le travail consiste alors à développer les différents modules qui supportent cette approche, à savoir un module de spécification de scénarios, un module de simulation dans un environnement spatial, un module d'enregistrement des résultats de la simulation et un module d'analyse de ces résultats pour générer des critiques. Excepté le module d'analyse de résultats qui est en cours d'implémentation, des premières versions de tous les autres modules sont déjà implémentées et testées sur un cas simple. Les premières explorations du prototype montrent que l'architecture et les principaux modules sont stables. Toutefois, le module de critique reste à implémenter, et les autres modules nécessiteront encore quelques améliorations dans les travaux futurs.

Abstract

This contract aims to explore, define and implement a first prototype of an environment to support the development of a military COA critiquing system. The critiquing approach consists in simulating the COA in a geospatial environment and to analyse the results in order to generate critics. In order to implement this approach, several modules have been developed and tested, such as the scenarios specification module, the geosimulation module, the data collector module, and the critiquing module. Except the critiquing module that is under development, a first version of each of all the other modules have been developed and tested using a simple scenario. The first tests and experimentations with these modules showed that the architecture of the prototype is stable. However, the critiquing module must be finished, and the other modules must be improved.

Sommaire

Le projet MAGS-COA vise à explorer et développer un prototype qui illustre le concept de systèmes de critique de suites d'actions militaires. Le prototype se base sur une approche de géo-simulation multi-agent qui comporte trois étapes principales. Dans la première étape, l'utilisateur doit spécifier son plan (suite d'actions) appelé *scénario* : il définit les ressources, leurs positions initiales, les tâches à accomplir, les contraintes temporelles et la chronologie d'exécution des tâches. La deuxième étape consiste à simuler le scénario dans un environnement spatial virtuel géo-référencé. Les ressources du plan sont alors des agents plongés dans l'environnement virtuel et qui tentent de réaliser les tâches qui leur sont assignées. Durant la simulation, les évènements qui sont jugés pertinents pour la dimension à critiquer sont enregistrés dans un registre de données (trace de simulation). Finalement, la troisième étape consiste à analyser la trace de la simulation en utilisant différentes sources de connaissances pour générer des critiques.

Le prototype est développé en utilisant comme base la plate-forme MAGS, une plate-forme de géosimulation multi-agent développée par le laboratoire d'informatique cognitive de l'Université Laval. Plusieurs améliorations ont été faites à cette plate-forme pour l'adapter au projet MAGS-COA, en particulier l'ajout d'un module de spécification de scénarios, d'un module d'enregistrement de données de simulation et d'un module d'analyse et critique de résultats de simulation.

Le but de ce document est de présenter les améliorations qui ont été apportées à la plate forme MAGS et les différents modules qui ont été implémentés pour développer le prototype MAGS-COA. Les résultats et les travaux futurs sont également discutés.

Jimmy Pierron, Jimmy Hogan et Hedi Haddad, 2005. Le projet MAGS-COA. Université Laval.

Executive summary

The COA-MAGS project aims to explore and develop a prototype illustrating the concept of military COA critiquing system. The prototype is based on multi-agent geosimulation approach composed of three steps. In the first step, we allow the user to specify a course of action (COA) that we call *scenario*. The user defines the resources, their positions, their tasks and the temporal constraints that define the chronology of these tasks. The second step consists in simulating the scenario within a virtual geospatial environment. The resources are represented by agents immersed in the virtual environment and trying to execute their assigned tasks (objectives). During the simulation, all events that are considered relevant for the critic system are saved in a data repository. Finally, the third step consists in analysing the simulation results using different knowledge sources in order to generate critics.

The prototype is developed using the MAGS platform, a multi-agent geosimulation platform developed by the laboratory "Laboratoire d'informatique cognitive" at Laval University. Several modifications have been introduced in the MAGS platform in order to enable it to support the MAGS-COA project, especially the development of a scenario specification module, a data repository module and a critiquing module.

The aim of this document is to present the different modules that have been introduced in the MAGS platform in order to develop the MAGS-COA prototype. The results and the future works are also discussed.

Jimmy Pierron, Jimmy Hogan et Hedi Haddad, 2005. Le projet MAGS-COA. Université Laval.

Table des matières

1 Introduction _1

2 Concepts de base _1

2.1 Suite d'actions 1

2.2 Actions composées et actions élémentaires 2

2.3 Contraintes 2

3 Le simulateur MAGS-COA _3

3.1 Le simulateur 4

3.2 Le scénario 5

3.2.1 Les événements (VSEvent) 5
3.2.2 Les ordres (VSOrder) 6
3.2.3 Les conditions (VSCondition) 10
3.2.4 Interface utilisateur pour la spécification du scénario 12
3.2.5 Conclusion sur le scénario 13

3.3 Le répertoire d'actions (VSActionRepository) 13

3.4 La simulation 14

3.4.1 AgentContainer 15
3.4.2 OpContainer 17
3.4.3 Updater 20
3.4.4 TimeStep 20

3.5 L'environnement 20

3.5.1 Conclusion sur l'environnement 23

3.6 Le registre de données 24

3.7 L'observateur 26

3.8 Le critiqueur 26

3.8.1 Les traces de simulation 27
3.8.2 Les connaissances du domaine 28
3.8.2.1 Connaissances sur les ressources et les actions 28
3.8.2.2 Connaissances sur l'environnement spatial de la simulation 29
3.8.2.3 Connaissances sur les critiques 29

4 Exemple d'application _29

4.1 Exemple 1 29

4.1.1 Création des événements 29
4.1.2 Sélection des événements 31
4.1.3 Conception de la COA 31
4.1.4 Contraintes temporelles 35

4.2 Exemple 2 37

4.2.1 Le scénario 37
4.2.2 La simulation 41
4.2.2.1 Les Agents de la simulation 42
4.2.2.2 Les observateurs de la simulation 45
4.2.2.3 Les données recueillies lors de la simulation 46
4.2.2.4 La structure des observateurs 46

5 Discussion _______ *48*

5.1 Le processus de critique _______ 48

5.2 La création de nouveaux éléments _______ 51

5.3 La spécification du scénario _______ 51

5.3.1 Les avantages _______ 51
5.3.2 Les limitations _______ 52
5.3.3 Travaux futurs _______ 52

5.4 La simulation _______ 53

5.4.1 Les avantages _______ 53
5.4.2 Les travaux futurs _______ 53

5.5 Les observateurs _______ 54

5.5.1 Les avantages _______ 54
5.5.2 Les limitations _______ 54
5.5.3 Les travaux futurs _______ 56

5.6 Les données recueillies _______ 57

5.7 La critique _______ 57

5.8 L'environnement _______ 58

6 Conclusion _______ *61*

7 Annexe A _______ *62*

7.1 L'ajout des nouvelles actions au VSAction Repository _______ 62

7.2 Description des comportements développés _______ 66

7.3 Enregistrements de l'exemple d'application _______ 69

8 Annexe B _______ *76*

Liste de figures

Figure 1 **:** Les composants principaux de MAGS-COA 3
Figure 2 **:** Les gestionnaires du simulateur 5
Figure 3 : Attributs d'un ordre (c++) 7
Figure 4 : Un événement dans le scénario..8
Figure 5 : Un ordre assigné à un événement 8
Figure 6 : Affectation des objectifs à un agent par un ordre 10
Figure 7 : Un exemple de scénario avec 3 nœuds de condition 11
Figure 8 : Barre d'outils de MAGS-COA..12
Figure 9 : Structure d'une simulation COA 15
Figure 10 **:** Un exemple d'opérateur de navigation..19
Figure 11 : La relation d'inclusion..21
Figure 12 **:** La combinaison des relations d'inclusion et des réseaux routiers..23
Figure 13 : Création des ressources..30
Figure 14 : Attributs du LocationState..30
Figure 15 : Sélection des éléments du scénario..31
Figure 16 Description et affichage des ordres..31
Figure 17 : Processus de création d'un comportement 33
Figure 18 : Comportement généré à partir des ordres 34
Figure 19 : Attributs de l'action élémentaire GotoAction 35
Figure 20 : Règle d'échec concernant une contrainte temporelle..36
Figure 21 : Représentation du scénario de l'exemple 2 37
Figure 22 : Structure du scénario de l'exemple 2 dans MAGS-COA..38
Figure 23 : Structure du scénario 39
Figure 24 : Attributs d'un DrawState..40
Figure 25 : AttributeState de l'order "Strike" 41
Figure 26 : Scénario → VSAction → Simulation..42
Figure 27 : Un agent CF18 de la simulation 42
Figure 28 : Attributs d'un CF18 de la simlation..43
Figure 29 : Création d'un CF18 et assignation de deux ordres 44
Figure 30 : Règle d'échec concernant une contrainte temporelle..45
Figure 31 : Les enregistrements des données de simulation 46
Figure 32 : Structure d'un observateur..47

1 Introduction

Le projet MAGS-COA vise à explorer et développer un prototype qui illustre le concept de systèmes de critique de suites d'actions militaires. Le prototype se base sur une approche de géo-simulation multi-agent qui fera l'objet de ce document.

Le prototype est développé en utilisant comme base la plate-forme MAGS, une plate-forme de géosimulation multi-agent développée par le laboratoire d'informatique cognitive de l'Université Laval. La base technologique de MAGS permet, d'une façon collaborative, de spécifier des scénarios et de les simuler dans un environnement géographique représenté en 2D ou en 3D.

Le but de ce document est, dans la section 1, de présenter les concepts de base nécessaires pour atteindre l'objectif qui est de développer un système permettant de critiquer les suites d'actions militaires. La section 2 aborde l'architecture et le développement du prototype permettant la spécification des scénarios (suites d'actions), la simulation et l'analyse de ces scénarios. L'objectif du prototype est de définir un cadre suffisant pour développer des méthodes pour la critique de ces scénarios. La section 3 présente deux exemples d'application du prototype et la section 4 discute en profondeur des résultats et des travaux futurs suite à ce projet.

2 Concepts de base

Dans cette section nous présentons les concepts de base utilisés dans MAGS-COA.

2.1 Suite d'actions

Une suite d'actions (Course of Action, COA) est un ensemble d'actions qui décrit les étapes proposées par l'utilisateur pour réaliser une mission donnée. Ces actions sont généralement spécifiées avec un haut niveau d'abstraction et ne sont pas détaillées, et donc ne peuvent pas être directement simulées par MAGS-COA. Pour cette raison, nous distinguons les notions *d'actions composées* et *d'actions élémentaires*.

2.2 Actions composées et actions élémentaires

Une action composée est une action qui peut être raffinée en une ou plusieurs actions composées ou élémentaires. Une action élémentaire est une action qui ne peut pas être raffinée en d'autres actions. Ainsi, la hiérarchie des actions pourrait être représentée sous forme d'arbre dont la racine et les nœuds intermédiaires sont des actions composées et les feuilles représentent des actions élémentaires.

2.3 Contraintes

En spécifiant son COA, l'utilisateur peut spécifier plusieurs contraintes que nous résumons principalement en deux types :

- **Contraintes inter actions**

Ce sont des contraintes temporelles qui permettent de spécifier l'ordre chronologique de l'exécution des actions. Par exemple, une action peut s'exécuter avant, après ou en même temps qu'une autre action.

- **Contraintes sur les actions**

Une action pourrait être soumise à plusieurs types de contraintes. On distingue principalement :

- Les contraintes spatiales et temporelles

  Une action pourrait devoir s'exécuter durant une période donnée de temps. Elle pourrait devoir s'exécuter également suite, pendant ou juste avant la production d'un *événement* particulier dans l'environnement de simulation. Un événement correspond à la création d'un composant dans la simulation à un instant et une position donnés.

  De la même façon, une action pourrait devoir s'exécuter dans un espace particulier, par exemple un avion ne doit pas voler au-dessus des zones spécifiées comme risquées.

- Les contraintes de ressources

Une action consomme des ressources matérielles (avions, etc.) et humaines pour qu'elle puisse s'exécuter. Dans MAGS-COA, nous modélisons ces ressources par des agents. Ainsi, une ressource dans MAGS-COA a les mêmes caractéristiques qu'un agent (attributs statiques et les comportements). Nous présenterons les caractéristiques d'un agent MAGS-COA plus tard dans ce document.

3 Le simulateur MAGS-COA

Les éléments de base utilisés pour construire la structure du simulateur sont appelés des *composants*. Ainsi, la simulation et l'architecture sont constituées par plusieurs composants qui interagissent ensemble, chacun ayant sa fonctionnalité, et qui peuvent être, à leur tour, constitués par d'autres composants. La figure 1 illustre les différents composants de base du simulateur et leurs interactions. Dans ce qui suit nous présentons successivement les structures de ces composants et leurs interactions.

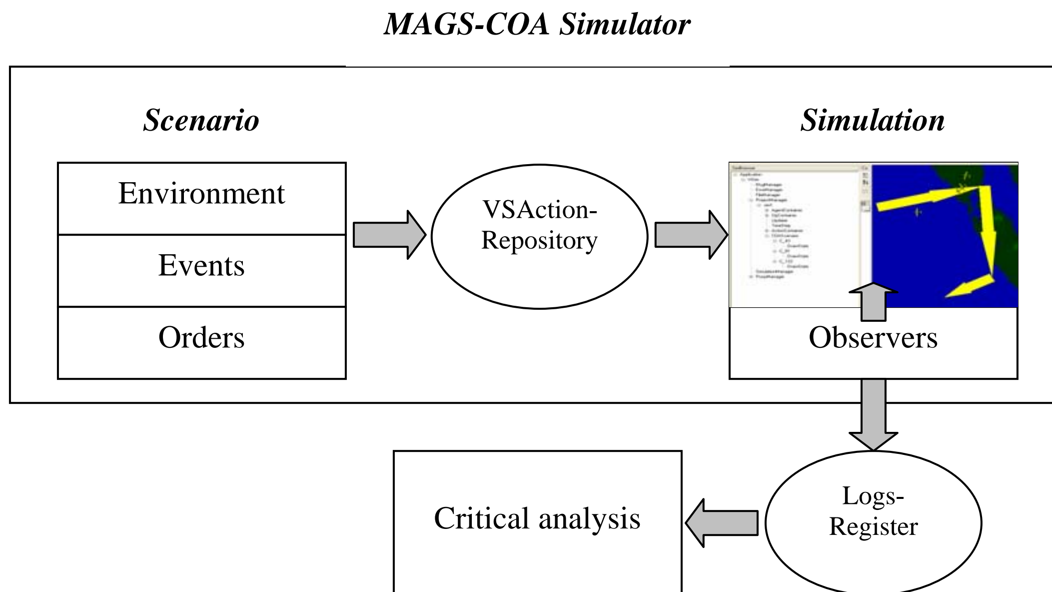


Figure 1 **:** Les composants principaux de MAGS-COA

3.1 Le simulateur

Le simulateur MAGS-COA est basé sur la plateforme de développement SdkSim de NSim Technologies qui est le successeur de MAGS [MAGS, 2003]. Cette plateforme permet de développer une architecture de simulation organisée en deux grands volets : le(s) client(s) et le simulateur. Toute l'architecture du système est organisée sous forme d'un arbre auquel des composants (plug-in) sont rajoutées. Les composants de base de SdkSim sont les suivants :

- *MessageManager*

  Le MessageManager s'occupe de la gestion des messages dans le simulateur. Il permet de créer et de récupérer facilement des messages qui s'échangent au niveau de l'architecture. Il ne s'agit pas de messages entre agents mais de messages généraux entre composants du simulateur. Un agent étant un composant, les agents pourront s'échanger des messages à l'aide de ce mécanisme.

- *ProxyManager*

  Le ProxyManager permet de gérer toute la communication et la synchronisation entre les clients et le serveur qui contient le simulateur. C'est donc par ce Manager que sont envoyés et reçus tous les messages provenant des clients (par exemple, actualiser une simulation, etc.).

- *FileManager*

  Le FileManager permet la gestion et la standardisation des accès aux fichiers pour sauvegarder et recharger des données. À titre d'exemple, c'est ce Manager qui permet de sauvegarder les résultats des simulations.

- *ErrorManager*

  Ce Manager sert à stocker et gérer les erreurs provenant de la simulation.

- *ProjectManager*

  Le ProjectManager a comme objectif d'organiser et de gérer les projets de simulation. Il permet de créer, modifier et supprimer des projets. Un projet contient un scénario, une

simulation ainsi qu'une liste d'actions disponibles pour créer des éléments dans la simulation.

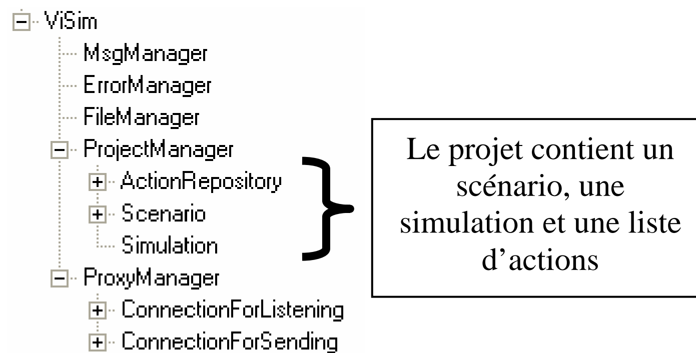


Figure 2 **:** Les gestionnaires du simulateur

Les détails sur le fonctionnement et les principes de base du framework sont présentés dans la documentation de SdkSim [SdkSim, 2005].

3.2 Le scénario

Le scénario est un composant qui sert à spécifier la COA telle que définie dans la section 1.1. Concrètement, un scénario permet à l'utilisateur de définir trois types d'éléments : des événements, des ordres et des conditions. Un scénario est représenté sous forme d'un arbre où les nœuds sont de type ***VSEvent***, ***VSOrdrer*** et ***VSCondition***.

3.2.1 Les événements (VSEvent)

Nous avons déjà défini les évènements externes en tant qu'évènements qui se produisent dans le simulateur et qui ne sont pas contrôlés par les actions du scénario. Dans le simulateur, ces évènements sont appelés des *VSEvents*. Un VSEvent est caractérisé par la structure suivante :

- **VSEvent**
  - **Name** : Nom de l'événement.

- **Effect** : C’est l’algorithme qui permet de d’implémenter l’effet de l’événement sur l’environnement. À titre d’exemple, la création d’un agent, le déclenchement d’un phénomène tel que la pluie, une explosion ou un incendie, etc.
- **Position** : La position ou se produira l’événement.
- **Time** : Le temps de déclenchement de l’événement lors de la simulation.

Par exemple, lorsque le concepteur désire faire exploser une bombe, il devra créer un événement qui aura la structure suivante :

- Name : ExploseBomb
- Effect : L’algorithme qui simule l’explosion d’une bombe et son effet.
- Position : Pourrait être les coordonnées exactes ou le nom d’une zone, un pont par exemple.
- Time : Par exemple, 10 minutes après le début de la simulation.

Un événement représente donc tout ce qui peut se créer dans le temps et dans l’espace (création d’agents, d’explosions, de zones, etc.).

3.2.2 Les ordres (VSOrder)

Un ordre est un composant qui décrit une action telle que définie dans la section 1.2. En fait, quand l’utilisateur spécifie sa COA, il génère une suite d’ordres, chaque ordre correspond à une action de la COA. L’ordre est le moyen qui permet d’assigner les objectifs imposés par l’usager aux agents concernés par le scénario. Un ordre contient les informations suivantes :

- **ActorName** : Le nom de l’agent auquel cet ordre sera assigné ;
- **ActionType** : Le type d’action à appeler pour exécuter cet ordre ;
- **Object** : La position ou l’agent qui est l’objet de cet ordre. (par exemple, un ordre de mouvement nécessite de spécifier la destination) ;
- **Les contraintes temporelles** : il s’agit de spécifier les contraintes temporelles qui régissent la chronologie de l’exécution des ordres (Temps d’activation de l’ordre, temps maximum alloué pour l’exécution de cet ordre).

```
//-----------------------------------
// ACTION or OPERATION
//-----------------------------------
// Action executed by this order
std::string m_ActionType;

//-----------------------------------
// ACTOR NAME
//-----------------------------------
// Name of the Refered components to assign this order
std::string m_ActorName;

//-----------------------------------
// TEMPORAL CONSTRAINT
//-----------------------------------
// Time to active OR assign this order to the actor
float m_ActivationTime;

// Time to complete this order
float m_TimeToComplete;

//-----------------------------------
// OBJECT - Position OR Component
// with a LocationState
//-----------------------------------
// Position
float m_ObjectPosX;
float m_ObjectPosY;
float m_ObjectPosZ;

// Component with a locationState
std::string m_ObjectName;
```

Figure 3 : Attributs d'un ordre (c++)

Un ordre est impliqué dans les deux composants du simulateur : le scénario et la simulation. Lorsque l'utilisateur spécifie un ordre dans le scénario, il doit préalablement sélectionner les éléments auxquels il désire assigner cet ordre. Rappelons que le scénario ne contient que trois types d'éléments : les VsEvents, les VsConditions et les VsOrders. L'ordre ne peut donc pas être affecté directement à des agents du scénario car les agents ne sont pas encore crées à ce stade. Les agents seront créés seulement dans la simulation lorsque celle-ci sera lancée. Ce sont les événements du scénario qui servent à spécifier la création des agents dans la simulation. Donc lorsqu'un ordre devra être affecté, il le sera à un événement et non pas à un agent.

Voici un exemple simple permettant d'illustrer ceci (figure 4). Tout d'abord, le scénario est vide lorsque l'utilisateur place un événement pour créer un CF18. Cet événement est visualisé à l'aide d'un modèle 3D de CF18 dans l'environnement.

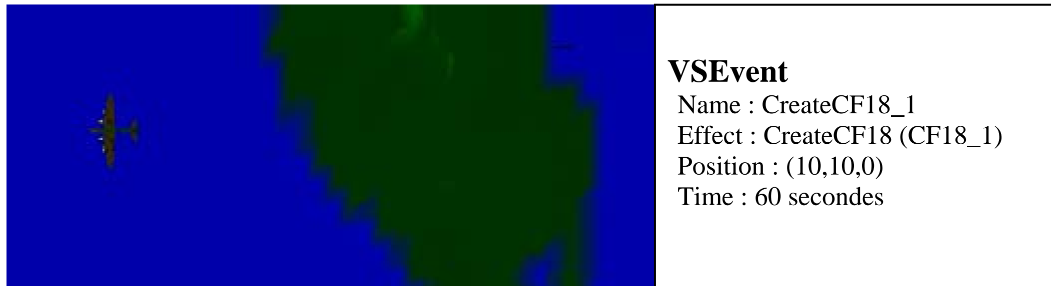


Figure 4 : Un événement dans le scénario

Ensuite l'utilisateur sélectionne l'événement car celui-ci possède une caractéristique spatiale (sa position). Lorsque l'événement est sélectionné, l'utilisateur lui affecte un ordre « goto » qu'il appelle Goto1.

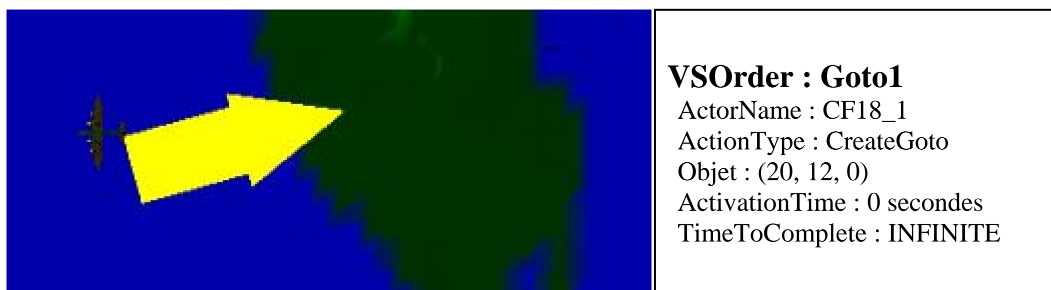


Figure 5 : Un ordre assigné à un événement

L'ordre est représenté dans l'interface du simulateur sous forme de flèche (voir figure 5). L'ordre est affecté à l'événement mais l'attribut ActorName de l'ordre lui indique la référence de l'agent qui sera créé dans la simulation. C'est ainsi que le lien est effectué entre l'événement, l'ordre et le futur agent.

Pour continuer la COA, l'utilisateur sélectionne l'ordre du futur agent CF18_1. La sélection de l'ordre est possible parce que celui-ci possède également une position spatiale qui se trouve à être l'objet de l'ordre (dans ce cas c'est la position (20, 12, 0)). L'utilisateur assigne ensuite un ordre Goto2 à Goto1. L'assignation d'un ordre à un autre ordre a pour effet de construire une

suite d'actions pour le même futur agent. Le fait de sélectionner l'ordre et non l'événement pour continuer la COA est un choix pour simplifier la spécification. Il est plus naturel de sélectionner le dernier ordre assigné pour continuer la suite. De plus, il sera plus simple de modifier un ordre en particulier dans COA.

Lorsque le scénario est spécifié, l'utilisateur peut démarrer la simulation. Le fait de démarrer la simulation permet d'instancier le scénario et de créer les composants réels dans la simulation à partir de ce qui est spécifié dans le scénario.

La procédure qui permet aux ordres d'assigner des objectifs aux agents est expliquée dans la figure 6. Initialement, quand l'utilisateur spécifie sa COA à l'aide de l'interface, un scénario composé d'une suite d'événements et d'ordres est généré. Normalement, c'est l'agent qui doit avoir la connaissance de comment interpréter l'ordre qui lui est assigné lors de la simulation. Cependant, pour ne pas dupliquer toute la connaissance pour tous les agents, ces connaissances ne sont pas stockées dans l'agent mais plutôt dans un autre module que nous avons appelé *VSActionRepository* qui contient toutes les connaissances comportementales. L'agent se fait donc assigner ces connaissances du VSActionRepository qui deviennent des comportements. Pour chaque agent, l'instanciation des comportements initialement spécifiés par le scénario va créer le *COABehaviour*. Une fois la simulation lancée, chaque agent, vu que l'environnement est dynamique, pourrait avoir besoin d'exécuter des comportements autres que ceux qui lui sont assignés par le scénario. Ces autres comportements sont définis dans le composant *MyBehaviour*. Ceci permet de distinguer les comportements introduits par le scénario des comportements propres à l'agent.

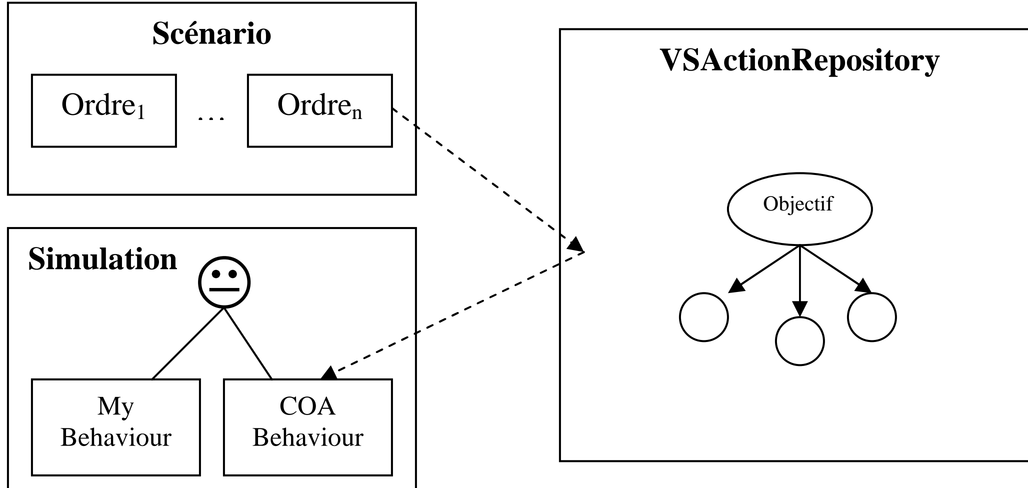


Figure 6 : Affectation des objectifs à un agent par un ordre

3.2.3 Les conditions (VSCondition)

Jusqu'ici, le scénario contient une arborescence d'événements et d'ordres qui devront être exécutés pour créer les bons éléments dans la simulation. La question est ***Quand*** seront-ils exécutés ou sous quelles conditions devront-ils être exécutés ?

La réponse se trouve dans le dernier type d'éléments d'un scénario : la ***VSCondition***. Le scénario est représenté comme un arbre de composants de type événement ou ordre. Il s'agit donc d'ajouter un type de nœud dans l'arbre de scénario qui servira à déterminer si le sous-arbre de ce nœud est valide ou non. Le nœud de type condition permet de définir des règles qui seront testées pour retourner l'état Vrai ou Faux. Si le nœud de type VScondition est vrai, tout le sous-arbre du scénario de ce nœud est exécuté.

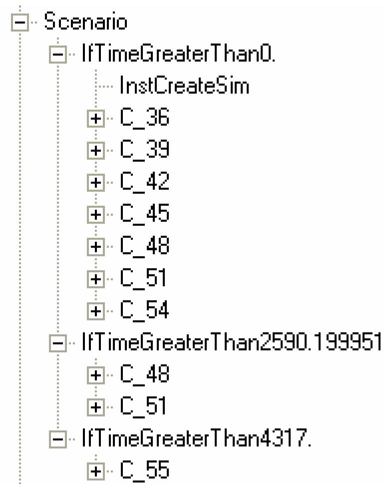


Figure 7 : Un exemple de scénario avec 3 nœuds de condition

La figure précédente présente un exemple d'arbre de scénario composé de trois nœuds de type VSCondition. Pour le moment, le nœud de type VScondition ne supporte qu'une condition temporelle qui détermine si le temps de simulation est supérieur à la condition. Donc le premier nœud de l'arbre « IfGreaterThan0 » test le fait que le temps courant de la simulation est supérieur ou égal à 0 secondes. Si tel est le cas, tout le sous-arbre de ce nœud est exécuté.

Le type de condition temporelle permet actuellement de créer des événements à un instant donné ou d'assigner des ordres à un instant donné. Il est donc possible de définir, pour un sous-arbre de scénario, qu'il soit exécuté *avant*, en *même temps* ou *après* un autre sous-arbre simplement en ajustant le temps des nœuds de conditions.

Ensuite, il sera possible d'implanter des règles plus complexes pour représenter des branches conditionnelles dans le scénario. Par exemple, une condition pourrait vérifier si un événement E1 s'est produit dans la simulation et ainsi exécuter un sous arbre du scénario lié à cet événement. Cependant, il faut se questionner sur le type de condition à implanter dans le scénario pour ne pas déplacer les comportements des agents vers des conditions du scénario.

3.2.4 Interface utilisateur pour la spécification du scénario

Le client du système SdkSim est implanté en C# à l'aide d'une architecture « pure plug-in », ce qui lui permet d'intégrer des éléments supplémentaires pour l'interface utilisateur. MAGS-COA définit des barres d'outils pour supporter les fonctionnalités du simulateur. Actuellement, on utilise quatre barres d'outils pour définir :

- une palette de ressources amies (présentement limitées à des CF18 et des radars),
- une palette de ressources ennemies (qui peuvent être différentes des ressources amies),
- une palette de commandes de spécification des scénarios,
- et une palette de commande de manipulation des simulations.

La figure suivante présente les différents boutons jusqu'à présent développés pour spécifier, modifier et simuler des scénarios et interagir avec le système de serveur ViSim.

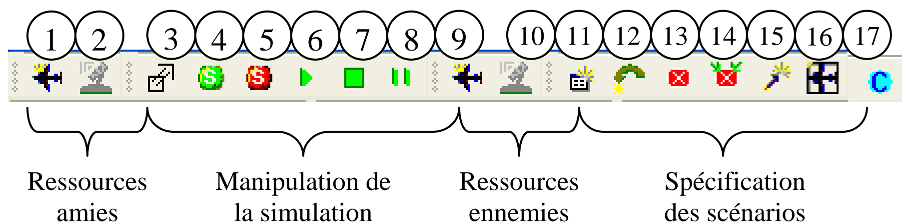


Figure 8 : Barre d'outils de MAGS-COA

1-Bouton pour ajouter un événement dans le scénario pour créer un CF18 ami;
2-Bouton pour ajouter un événement dans le scénario créant une zone radar amie;
3-Bouton pour manipuler (Déplacement et agrandissement) la carte de l'environnement de la simulation;
4-Ce bouton permet d'afficher le scénario (événements et ordres);
5-Ce bouton permet de cacher le scénario (événements et ordres);
6-Bouton pour démarrer la simulation à partir du scénario;
7-Bouton pour arrêter la simulation;
8-Bouton pour pauser la simulation;
9-Bouton pour ajouter un événement dans le scénario pour créer un CF18 ennemi;
10- Bouton pour ajouter un événement dans le scénario créant une zone radar ennemie;
11-Ce bouton permet d'ajouter un événement pour créer la structure d'un simulateur COA au début de la simulation;
12-Ce bouton sert à assigner un ordre « Goto » aux événements et aux ordres sélectionnés ;
13-Bouton pour assigner un ordre « Wait » (attente pendant une durée donnée de temps) à un agent du scénario;

14-Bouton pour assigner un ordre « Appointment » (attente d'un ou plusieurs agents) à un agent du scénario;
15-Bouton pour assigner un ordre « Strike » à plusieurs agents sélectionnés dans le scénario
16-Bouton pour sélectionner les agents (événements) et les ordres du scénario ;
17-Bouton permettant d'appeler le CriticManager pour démarrer les algorithmes de critiques.

3.2.5 Conclusion sur le scénario

Un scénario est donc un arbre servant à représenter une COA spécifiée par l'utilisateur. Lorsque l'usager interagit avec le système pour sélectionner des éléments, assigner des ordres ou créer des agents, il ne fait qu'ajouter ou modifier des nœuds du scénario. En aucun cas, l'utilisateur ne peut interagir directement avec la simulation qui ne sert qu'à « jouer » le scénario.

À ce stade-ci, la structure du scénario semble assez expressive et permet de spécifier une suite d'action (COA) efficacement.

3.3 Le répertoire d'actions (VSActionRepository)

L'un des objectifs principaux du système est de séparer le scénario et sa spécification (COA) de la simulation proprement dit. Ainsi, un scénario peut être simulé de différentes façons selon le type d'algorithme utilisé ou le type de comportement implanté. Le scénario n'est qu'une spécification de conditions, d'événements et d'ordres ayant certains attributs. Il ne contient donc aucune connaissance sur les éléments à exécuter lors de la simulation.

Lorsque l'arbre de scénario est spécifié et que l'utilisateur désire exécuter ce scénario, le système doit créer la simulation correspondant au scénario. Toute la connaissance reliée aux événements et aux ordres se trouve dans ***VSActionRepository*** qui contient toutes les ***VSActions*** qui seront exécutées pour créer les événements et assigner les ordres. Ce sont ces VSactions qui connaissent la façon de créer les agents et les comportements dans la simulation. La connaissance se trouve à l'extérieur du scénario et de la simulation dans l'objectif de garder nos agents « portables » et indépendants du contexte de simulation. Les agents peuvent ainsi se faire assigner le même comportement dans deux contextes de simulation différents sans modifier leurs structures.

Actuellement, voici la liste des actions qui sont implantées :

- **CreateCOASimulation** : Cette vsaction sert à créer une simulation de type COA.
- **CreateCF18** : Cette vsaction permet de créer un CF18 à un endroit donné dans la simulation.
- **CreateRadarZone** : Cette vsaction permet de créer une zone radar ayant un état de perception à un endroit précis dans la simulation.
- **CreateBaseGoto** : Cette vsaction sert à assigner un comportement de déplacement « Goto » à l'agent référencé par l'ordre.
- **CreateStrike** : Cette VSAction permet d'assigner un comportement d'attaque et de support aux agents référencés par cet ordre. La VSAction supporte deux types de rôle : « Leader » et « Supporter ». Les leaders se verront assigner un comportement leur permettant d'attaquer directement la cible désignée. Les « Supporter », quant à eux, auront un comportement visant à protéger le(s) leader(s) de toute attaque ne venant pas directement de la cible. Cette VSAction vise donc à simuler un comportement simple de groupe ayant le même objectif commun.

Dans l'annexe du présent rapport, nous présentons les étapes principales qui permettent de programmer des nouvelles actions et donc d'alimenter le répertoire d'actions. En fait, le système MAGS-COA ne permet à l'usager de spécifier des nouvelles actions (des nouveaux comportements), et donc elles doivent être programmées directement dans l'environnement du projet de développement du système.

3.4 La simulation

Lorsque le scénario est spécifié, l'utilisateur pourra démarrer une simulation afin de « jouer » son scénario et de recueillir des données. Pour démarrer une simulation, il suffit d'appuyer sur le bouton « Start » et le scénario commence à s'exécuter. La structure d'une simulation COA est la suivante :

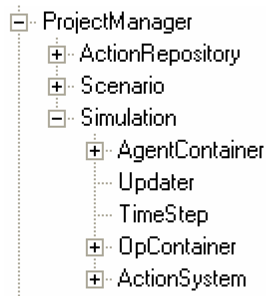


Figure 9 : Structure d'une simulation COA

3.4.1 AgentContainer

Il s'agit simplement du composant qui contient tous les agents de la simulation. Les agents sont des composants qui représentent soit les ressources instanciées faisant partie du scénario simulé soit les éléments de l'espace géographique. Ces agents sont dotés de capacités spatiales et cognitives dont nous citons la perception, la navigation et le comportement.

La perception

Les agents sont dotés d'une capacité de perception visuelle et non visuelle [Moulin et al., 2003]. On distingue par exemple :

- La perception des caractéristiques du terrain de l'environnement géographique de simulation, telles que l'élévation, les pentes, etc.
- La perception, via un champ visuel, des objets statiques (montagnes, bâtiments, etc.) et mobiles (autres agents logiciels) qui entourent l'agent.
- La perception des chemins et des routes.
- La perception des aires dynamiques ayant des propriétés spécifiques, telles que les zones d'odeur et les aires de fumée.
- La perception des messages communiqués par d'autres agents [Moulin et al., 2003].

La navigation

Les agents se déplacent dans leur environnement de façon autonome en se basant sur leurs systèmes de perception. On distingue deux modes de navigation : le mode *following-a-path*

(suivre un chemin particulier) et le mode *obstacle-avoidance* (éviter les collisions avec les obstacles et les autres agents).

Le comportement

Le comportement d'un agent est un comportement à base *d'objectifs*. En effet, durant la simulation, l'agent essaye de réaliser des objectifs qui lui sont assignés ou qu'il choisi lui même. Un objectif peut être *composé* ou *élémentaire*. Un objectif composé est un objectif qui pourrait être décomposé par d'autres objectifs composés ou élémentaires. Un objectif élémentaire ne peut pas être décomposé en d'autres objectifs. Ainsi, on parle d'arbres d'objectifs constitués par des objectifs composés et des objectifs élémentaires.

Formellement, un objectif se définit selon la grammaire BNF suivante :

Objectif ::= Objectif_Composé | Objectif_élémentaire
Objectif_Composé ::= Objectifs, Règles_D'Activation, Règles_De_Complétude
Objectif_Élémentaire ::= Action, Règles_D'Activation, Règles_De_Complétude
Objectifs ::= {Objectif_1, …, Objectif_n}

Un objectif a des pré-conditions qui doivent être satisfaites afin qu'il puisse s'exécuter et a des effets qui décrivent l'état de l'environnement suite à son exécution. Les pré-conditions sont représentées par les *règles d'activation.* Les effets sont mesurés par des *règles de complétude* qui permettent de déterminer sous quelles conditions un objectif est réussi ou échoué.

Un objectif a un cycle de vie. Initialement *inactif*, il devient *actif* dès que ses règles d'activation sont satisfaites. Le fait qu'un objectif soit *actif* veut dire qu'il devient candidat pour la sélection à être exécuté. Une fois sélectionné, l'objectif devient *en cours* et ses instructions sont exécutées. Les instructions constituent les algorithmes qui permettent de modifier les états d'un agent ou de l'environnement et qui permettent d'implémenter l'objectif. Par exemple, si un agent veut se déplacer, sa vitesse doit augmenter, sa position doit changer, etc. Il est à noter qu'un objectif composé peut être *en cours* seulement si au moins un de ses enfants est sélectionné, car les instructions se trouvent au niveau des objectifs élémentaires. Le passage à *en cours* d'un objectif élémentaire entraîne le passage à *en cours* de ses objectifs parents jusqu'à la racine de

l'arbre. L'objectif sélectionné est l'objectif qui est le plus prioritaire au moment de la sélection. L'étape de sélection constitue la décision que prend l'agent face à tous les objectifs candidats. Un objectif *en cours* peut être arrêté ou complété. Si l'objectif est arrêté et que c'est un objectif qui peut être repris, il devient *suspendu*. L'objectif *suspendu* redevient *en cours* lorsqu'il est le plus prioritaire pour l'agent. Si l'objectif est arrêté avant sa complétude et qu'il ne peut pas être *suspendu*, il devient *non-atteint*. Lorsqu'une règle de complétude s'applique alors qu'un objectif est *en cours* ou *suspendu,* il devient *atteint* ou *non-atteint*, selon la règle appliquée.

Les objectifs sont à la base des comportements des agents de la simulation. Ces agents sont des objets 'intelligents' concrets ou abstraits ou encore des groupes d'agents. La notion de groupe d'agents est primordiale dans la simulation de plusieurs scénarios. En fait, dans le domaine militaire, les actions du scénario sont souvent exécutées par des groupes d'agents qui jouent des *rôles* différents. L'exemple typique est l'opération d'attaque dans laquelle il y a des avions qui ont la mission d'attaque proprement dite et des avions qui les supportent. Bien qu'ils réalisent la même action du scénario, ces deux types d'avions ont des comportements (des objectifs) différents du moment où ils jouent des rôles différents. Un exemple plus simple consiste aux rôles de leader et du suiveur du même groupe. La notion du rôle est implémentée, elle aussi, en utilisant des objectifs. De ce fait, un agent appartenant à un groupe exécute les objectifs correspondant au rôle qu'il joue dans ce groupe.

L'affichage

Chaque agent possède un état d'affichage (DrawState) lui permettant de se lier à un modèle 3D et de fournir certaines informations relatives à l'affichage (couleur, modèle 3D, animation).

3.4.2 OpContainer

Ce composant est un conteneur d'opérateurs. Un opérateur représente, dans le simulateur, un composant qui possède un algorithme qui doit être exécuté systématiquement à chaque cycle de simulation. Un exemple est l'algorithme de navigation des agents qui doit être, pour chaque agent, exécuté à chaque cycle de simulation. Il s'agit d'une méthode pour séparer les algorithmes d'exécution de la structure de l'agent lui-même. Ainsi, il est possible de changer dynamiquement

l'algorithme utilisé par un agent pour la perception ou la navigation par exemple. MAGS implantait les algorithmes directement dans les états des agents, ce qui, en plus de dupliquer le code source, rendait les algorithmes difficilement interchangeables. De plus, en séparant les algorithmes des agents, les agents sont indépendants du contexte d'exécution et ils deviennent donc « portables ».

Par exemple, considérons le cas où un agent doit être transporté dans une autre simulation où l'échelle est différente. Il est important de conserver l'état de l'agent (la valeur de ses attributs) intacte mais les algorithmes agissant sur ces attributs changent. L'algorithme de navigation change d'une échelle à l'autre mais pas les attributs relatifs à l'agent (vitesse, position destination). C'est donc ainsi que les agents sont séparés de leurs opérateurs.

```
int ViOpNavRand::Update(std::string token, void* values)
{
   // ---------------------------------------------------
   // Update all agent's location states
   // ---------------------------------------------------
   // Get agent number
   int NbAgent = this->m_pParent->GetParent()->GetComponent("AgentContainer")
                 ->GetNbComponent();

   // browse all agents
   for (int i = 0; i < NbAgent; i++)
   {
      // Get Agent
      ViBaseAgent* agent = (ViBaseAgent*)this->m_pParent->GetParent()
      ->GetComponent("AgentContainer")->GetComponent(i);

      // get location state of the agent
      ViLocationState* loc = ((ViLocationState*)agent
      ->GetComponent("LocationState"));

      float Dist = 0.001f;

      if (loc != NULL)
      {
         if ((loc->m_XPosTrg) != 0 && (loc->m_YPosTrg != 0))
         {

            // Compute next position from current position to target position
            float XDep = loc->m_XPosTrg - loc->m_XPos;
            float YDep = loc->m_YPosTrg - loc->m_YPos;

            float Angle = atan2(YDep,XDep);

            XDep = fabs(XDep) * cosf(Angle) * Dist;
            YDep = fabs(YDep) * sinf(Angle) * Dist;

            loc->m_XPos += XDep;
```

```
            loc->m_YPos += YDep;
            loc->m_ZAngle = Angle;
         }
      }
   }
   return 1;
}
```

Figure 10 **:** Un exemple d'opérateur de navigation

Tous les opérateurs qui doivent être mis-à-jour à chaque cycle doivent se retrouver dans le OpContainer. Voici la description sommaire de chacun des opérateurs de MAGS-COA.

- NavOp : c'est une composante qui implante un algorithme simple de navigation pour les agents avions de base. Il calcule une trajectoire entre la position actuelle d'un agent et sa destination. Il prend la vitesse de l'agent en considération. Les informations relatives à la navigation se trouvent dans le *LocationState* de chaque agent (vitesse, position, destination).
- OpPerception : Il s'agit de l'opérateur qui implante l'algorithme de perception circulaire de base. La perception s'effectue à l'aide du *PerceptionState* d'un agent avec le rayon spécifié et place les agents perçus dans le *PerceptionState* de l'agent;
- OpBehavior

  L'opérateur OpBehavior contient l'algorithme qui actualise, lors de la simulation, tous les objectifs qui se trouvent dans le composant *My Behavior* de l'agent ;
- OpDraw

  Cet opérateur parcourt tous les DrawState des agents et envoie l'information nécessaire pour l'affichage 3D au(x) client(s) ;
- COAOpBehavior

  Cet opérateur contient l'algorithme qui actualise et exécute le comportement des agents se rapportant à la suite d'actions (le composant *COA Behavior*) ;
- COADraw

  Cet opérateur parcourt les DrawState des ordres et envoie l'information nécessaire pour l'affichage des ordres. Il faut savoir ici que ce sont les ordres au niveau du

scénario qui sont affichés et non les actions des agents. L'avantage est d'avoir un seul affichage pour un ordre même s'il contient des dizaines d'agents.

3.4.3 Updater

Le composant ***Updater*** est un composant spécial qui sert à appeler, à chaque cycle, tous les opérateurs. Par exemple, si les opérateurs devaient s'exécuter une seule fois par deux cycles, il serait possible de modifier ***Updater*** pour implanter ce comportement.

3.4.4 TimeStep

Ce composant s'occupe de gérer le temps de la simulation. Il permet de calculer le temps écoulé et de régler la vitesse de la simulation.

3.5 L'environnement

On distingue deux types d'environnements : l'environnement physique de la simulation, dans lequel les agents naviguent, et l'environnement 'conceptuel' qui décrit comment l'environnement physique est conceptuellement et logiquement structuré. Dans cette section nous décrivons le modèle de l'environnement conceptuel. L'implémentation du modèle physique et son lien avec le modèle conceptuel n'est pas implanté et seront donc traités dans la section 4 (conclusion et travaux futurs).

Conceptuellement, l'espace de la simulation est découpé en des zones qui sont liées par différents types de relations. En particulier, nous distinguons trois types de relations : la relation d'inclusion, les relations de positionnement et les réseaux routiers.

La relation d'appartenance

Une zone de l'espace peut contenir plusieurs autres zones. Tel qu'illustré par la figure 11, les zones *z1*, *z2*, *z3*, *z4* et *z5* sont incluses dans la zone *Z* et la zone *Zb* contient les zones *z6*, *z7* et *z8*.

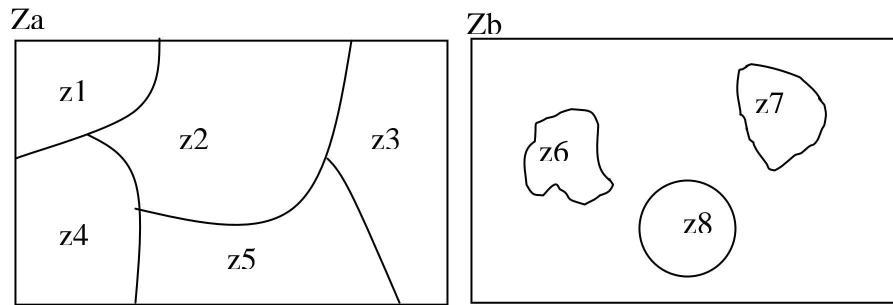
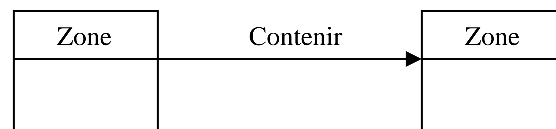


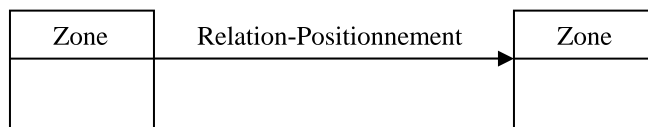
Figure 11 : La relation d'inclusion

La relation d'inclusion peut être modélisée par une relation orientée entre deux zones, une zone 'conteneur' et une zone 'contenue'.



Les relations de positionnement

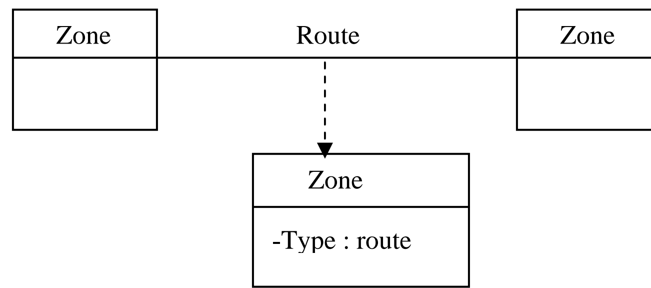
Les relations de positionnement permettent de décrire les différentes positions des zones de l'espace les unes par rapport aux autres. Ces relations peuvent décrire la distance quantitative (en mètres, kilomètres, etc.), la distance quantitative (près, loin, adjacent, etc.) ou l'orientation avec ses différents niveaux de précision (nord, sud, nord-sur, etc.). De la même façon, ces relations peuvent être modélisées par des relations orientées qui relient deux zones de l'espace.



Les réseaux routiers

Les routes qui relient deux zones de l'espace se distinguent des relations d'inclusion et de positionnement déjà mentionnées par leur aspect physique : en plus d'être une relation

conceptuelle entre deux zones, une route a sa propre existence comme une zone physique qui peut même, selon le cas modélisé, contenir d'autres zones. Un cas typique pourrait être le fait d'avoir une route contenant deux voies qui sont séparées par un trottoir. Ces relations peuvent être modélisées par des zones qui sont reliées par une relation conceptuelle *route* qui pointe vers une zone physique de type *route*.



Il est à noter que les réseaux routiers sont généralement modélisés par des graphes orientés dont les nœuds représentent les jonctions routières et les arcs représentent les routes. Le sens de circulation de la route est indiqué par la direction des arcs orientés du graphe. Ces réseaux routiers peuvent être modélisés avec le même concept de relation entre zones que nous avons présenté. Dans la figure 12, nous avons pris la zone *Za* de la figure 11 à laquelle nous avons ajouté un réseau routier constitué de quatre nœuds connectés par des routes. Les nœuds du réseau routier peuvent être modélisés en tant que zones de l'espace qui sont reliées par des routes. Nous avons utilisé le concept de *jonction* pour modéliser l'intersection entre les routes et les frontières de l'espace modélisé. En effet, une jonction n'est rien d'autre qu'une zone particulière de l'espace qui permet de garder le lien entre l'extérieur de l'espace et le réseau routier, et qui constitue donc un point d'entrée / sortie à cet espace.

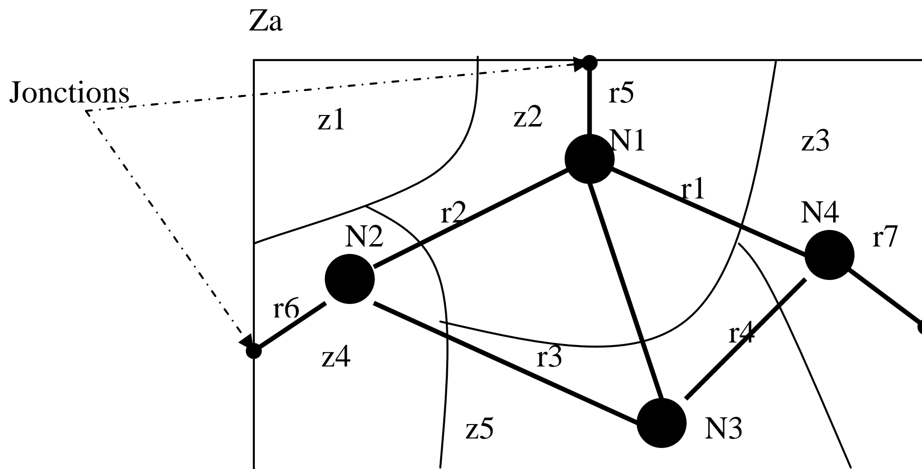


Figure 12 **:** La combinaison des relations d'inclusion et des réseaux routiers

De ce fait, il est possible d'utiliser le même concept de relations entre zones pour représenter les trois types de relations que nous avons citées. De plus, l'avantage de ce modèle est de permettre de modéliser l'espace avec une granularité à échelles multiples, et donc de pouvoir focaliser, selon le scénario simulé, sur les zones de l'échelle d'intérêt.

3.5.1 Conclusion sur l'environnement

L'environnement tel que décrit précédemment est le résultat d'une réflexion théorique sur la représentation sémantique de l'environnement. Il s'agit donc d'un essai qui, pour le moment, n'a pas été implanté dans le simulateur. L'objectif de la représentation sémantique de l'environnement est d'extraire des informations qualitatives pour alimenter les algorithmes de critiques. Ainsi, même si l'environnement ne provient pas directement du simulateur, il est possible de tester si cette représentation correspond aux exigences pour alimenter adéquatement les algorithmes de critiques. Pour ce faire, il suffit de décrire manuellement la structure formelle de l'environnement dans le registre de données présenté dans la section suivante.

3.6 Le registre de données (Logs Register)

Le registre de données est un composant qui permet de stocker toutes les données que l'on désire recueillir à partir de la simulation afin de les analyser par la suite pour faire des critiques. Présentement on enregistre quatre types de données au cours de la simulation qui sont :

1- le cycle de vie des objectifs : il s'agit d'enregistrer les étapes par lesquelles un objectif passe lors de la simulation, depuis son activation jusqu'à son achèvement avec ou sans succès.

2- les actions exécutées par un agent lors de la simulation

3- les changements des états (attributs) des agents

4- l'entrée à et la sortie d'une zone de l'espace

Afin que ces données puissent être exploitables par le module de critique, elles doivent être formatées sous un format compréhensible. Dans cette première version, nous avons opté pour le format suivant : *<Type, Acteur, Action, Objet, Localité, Temps>*. Ce format peut légèrement varier selon le type de données enregistrées. En effet, le champ *type* est utilisé pour indiquer si l'enregistrement en cours décrit l'un ou l'autre de ces types de données. De ce fait, les valeurs que peut prendre ce champ sont les éléments de l'ensemble {Objective-state, Action, Attribute-change, Zone-Event}. Ainsi, l'interprétation du reste des champs va dépendre de la valeur du champ *type*. Dans ce qui suit, nous illustrons le principe avec des exemples.

États des objectifs

Supposant qu'on affecte à un avion CF18 l'objectif d'attaquer un radar ennemi. Lors de la simulation, quand l'avion va commencer à exécuter son objectif, un enregistrement de la forme suivante sera ajouté au registre des données :

<Objective-State, CF18:A1, Objective:Attack1, Nil, En-cours, Time:10:15>

La valeur 'Objective-State' du champ *type* indique que l'enregistrement courant sert à décrire l'état d'un objectif. La valeur de l'acteur est l'identificateur de ***l'instance*** de l'agent et qui est formé en concaténant le type de l'instance (CF18 dans ce cas) et son identifiant (A1).

L’utilisation des instances typées facilite par la suite l’analyse des données. La valeur ‘Objective:Attack1’ est l’identificateur de l’instance de l’objectif d’attaque. La valeur du champ *Objet* est *Nil* étant donné qu’on n’a pas d’objet pour ce type d’enregistrements. L’état de l’instance de l’objectif est ‘En-cours’, et la valeur ‘Time:10:15’ indique le moment de la simulation où l’instance de l’objectif a été activée.
Ainsi, l’enregistrement <Objective-State, CF18:A1, Objective :Attack1, Nil, En-cours, Time:10:15> dans le registre de données signifie que l’agent ‘CF18:A1’ a commencé l’exécution de l’objectif ‘Attack1’ à 10 :15 par rapport au temps de la simulation.

Actions

Supposons que, pour réaliser l’objectif d’attaque, l’avion doit se déplacer à la position du radar et par la suite le bombarder. Dans le registre de données, on aura deux enregistrements qui décrivent ces actions sous la forme suivante :

<Action, CF18:A1, Action:goto1, Nil, Zone:z1, Time:10:15>
<Action, CF18:A1, Action:fire2, Agent:radar1, Zone:z1, Time:10:45>

Le premier enregistrement signifie que l’agent ‘CF18:A1’ a commencé une action ‘goto1’ vers la zone ‘Zone:z1’ à 10 :15. Le deuxième enregistrement signifie que le même agent a commencé, à 10 :45, une action ‘fire2’ à la zone ‘Zone:z1’ contre le radar ‘Radar1’.

Changement des valeurs des attributs

Il est également possible d’enregistrer les changements des états des agents lors de la simulation en sauvegardant les changements des valeurs de leurs attributs. L’enregistrement <Attribute-Change, CF18:A1, Attribute:Fuel, -10%, Zone:z1, Time:10:20> signifie que la valeur de l’attribut ‘Fuel’ de l’agent ‘CF18:A1’ a changé de ‘-10%’ à l’instant 10:20 et dans la zone ‘Zone:z1’.

Les entrées et les sorties aux zones de l’espace

Les entrées et les sorties des agents aux zones de l'espace sont aussi sauvegardées par des enregistrements de la forme <Zone-Event, CF18:A1, Event :Enter, Nil, Zone:z2, Time:10:25> (l'avion 'A1' a entré dans la zone 'z2' à 10 :25) ou <Zone-Event, CF18:A1, Event :Exit, Nil, Zone:z2, Time:11:15> (l'avion 'A1' a quitté la zone 'z2' à 11 :15). Le même principe pourrait être appliqué aussi pour gérer les membres d'un groupe d'agents de façon dynamique, et ce en remplaçant la zone par le groupe dans les deux exemples précédents.

3.7 L'observateur

L'observateur est le composant qui fait le lien entre la simulation et le registre de données. En fait, il est formé par un ou plusieurs agents particuliers qui sont plongés dans la simulation juste pour observer les aspects qu'on leur demande d'observer dans la simulation et les enregistrer dans le registre de données. Par exemple, pour enregistrer les agents qui entrent ou quittent une zone de l'espace, il faut placer un observateur dans cette zone.
L'intérêt d'utiliser le concept d'agents observateurs est d'éviter le fait d'enregistrer tout ce qui se passe dans la simulation, ce qui peut générer une quantité énorme de données. L'idée est plutôt d'enregistrer seulement les données dont on a besoin. De plus, le fait que les observateurs soient des agents, donc ayant des comportements, ouvre la porte pour avoir plus d'intelligence lors du recueil de données de la simulation. Un exemple d'observateur est présenté dans la section 4.2.2.4.

3.8 Le critiqueur

Le module de critique implémente les algorithmes qui utilisent les différentes connaissances nécessaires pour générer des critiques. Dans cette section nous nous contentons d'énumérer les principales connaissances utilisées par ce module, les algorithmes sont en cours d'implémentation. Il est à noter que la structure du critiqueur est en place et elle est implantée à l'aide du module *ViStdCriticManager* qui se trouve dans le projet sous *ProjectManager* de

l'arbre d'application. L'utilisation du module est simple, il suffit d'appuyer sur le bouton de l'interface pour activer les algorithmes de critique.

Le module de critique utilise principalement deux types de connaissances : les traces de la simulation et les connaissances du domaine. Nous avons déjà présenté les traces de la simulation et leurs structures dans la section 2.6 (registre de données). Les connaissances du domaine sont toutes les connaissances du domaine d'application (dans ce cas, les COA) qui sont nécessaires pour analyser les traces de la simulation et générer des critiques. Nous allons présenter chacun de ces types de connaissances dans les sections qui suivent. Pour implémenter le critiqueur, il serait très pertinent de représenter les traces de simulation et les connaissances du domaine avec le même formalisme de représentation de connaissances. Il serait plus intéressant encore d'utiliser ce même formalisme pour faire du raisonnement et générer des critiques. Pour ce faire, nous avons choisi d'explorer l'utilisation de Prolog+CG, une version de Prolog qui supporte les graphes conceptuels (Sowa 1984). Le langage permet donc de représenter les connaissances sous forme de graphes conceptuels et offre des mécanismes pour raisonner sur ces graphes. En particulier, nous sommes en train d'explorer l'utilisation de la plate forme Amine[1] qui intègre des outils de manipulation des connaissances (ontologies, etc.) avec la dernière version de Prolog+CG. Dans ce qui suit, nous présentons brièvement les pistes que nous sommes en train d'explorer pour les deux types de connaissances.

3.8.1 Les traces de simulation

Pour que les traces de simulation soient exploitables par le critiqueur, nous proposons de leur appliquer deux transformations. La première transformation consiste à exprimer les faits qui constituent ces traces sous forme de graphes conceptuels. Cette transformation est possible car les valeurs de ces faits sont déjà typées (voir section 2.6). La deuxième transformation consiste à construire, à partir de ces faits, une description [en langage naturel] de l'enchaînement d'évènements de la simulation. Nous explorons les travaux de Anne Vilnat [Vilnat, 2005] sur le

[1] Plus d'informations sons disponibles sur le lien http://amine-platform.sourceforge.net

dialogue et analyse de phrases (définitions de processus, de verbe d'action et de verbe d'état) comme élément de départ à cette réflexion.

3.8.2 Les connaissances du domaine

Il est à noter que les connaissances du domaine qui sont utilisées par les algorithmes de critique peuvent avoir des formats différents (graphes conceptuels, etc.). Dépendamment des aspects à critiquer, un critiqueur peut avoir besoin de certaines ou de toutes les connaissances suivantes :

3.8.2.1 Connaissances sur les ressources et les actions

Il s'agit de l'ontologie qui décrit les différentes ressources du domaine (caractéristiques, etc.) et les actions qu'elles peuvent entreprendre. On distingue présentement deux types de connaissances sur ces actions: la hiérarchie des actions et les contraintes sur les actions.

a. Hiérarchie d'actions

La décomposition des actions en des sous-actions et actions élémentaires est stockée dans le répertoire d'actions (VSActionRepository). Cette connaissance doit être accessible au critiqueur.

b. Contraintes sur les actions

Les contraintes sur les actions permettent de définir les conditions nécessaires pour le bon accomplissement des actions. Les contraintes peuvent porter sur l'agent de l'action, sur l'espace ou le temps. Par exemple, si on considère une action de déplacement, les contraintes pourraient être que l'agent doit avoir suffisamment de Fuel et que le chemin ne doit pas être bloqué.

3.8.2.2 Connaissances sur l'environnement spatial de la simulation

Le critiqueur doit avoir des connaissances sur l'espace de simulation en termes de différents objets spatiaux (zones dangereuses, etc.), leurs caractéristiques, leurs positions réelles et les relations topologiques, etc.

3.8.2.3 Connaissances sur les critiques

Ces connaissances définissent, pour chaque dimension de critique, les différentes règles et les différents aspects à étudier pour générer une critique. Pour l'instant, certaines de ces connaissances pourraient être représentées sous forme de graphes conceptuels, d'autres pourraient être programmées sous forme d'algorithmes.

4 Exemple d'application

4.1 Exemple 1

Cette section illustre le fonctionnement de l'architecture avec un exemple simple d'une COA utilisant trois agents. L'objectif est de mettre en évidence les interactions entre les composants et de tester l'architecture du simulateur. Il est à noter cette exemple est réduit à sa plus simple expression, donc tous les éléments du simulateur ne sont pas présentés. Cet exemple présente un cas simple où trois CF-18 sont positionner, géographiquement et temporellement, dans le scénario. Ensuite, trois actions de déplacement sont affectées à chacune des ressources, ce qui constitue la COA.

4.1.1 Création des événements

La première étape de la spécification du scénario consiste à créer les événements et les placer dans l'environnement de la simulation. Pour le moment, les événements pour créer trois avions sont placés dans la partie océanique gauche de l'environnement. Les composantes C_31,

C_34 et C_37 représentent les événements sous le composant *Scénario*. La figure suivante montre que chaque agent possède, à cette étape, deux composantes : le *LocationState* et le *DrawState*.

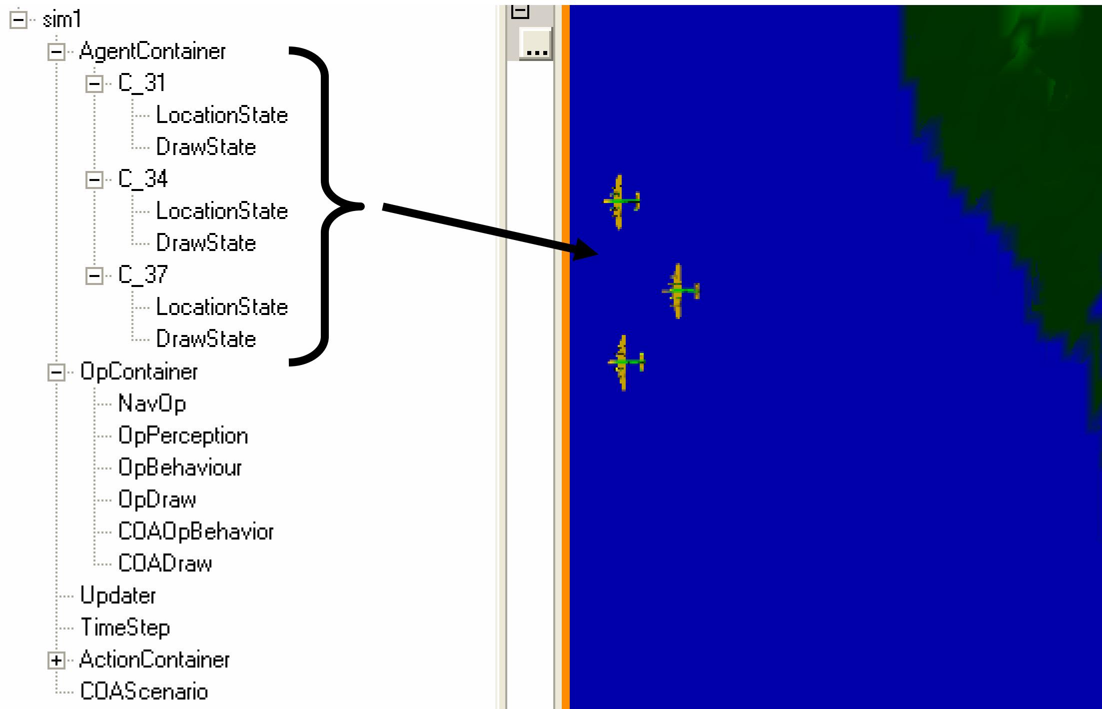


Figure 13 : Création des ressources

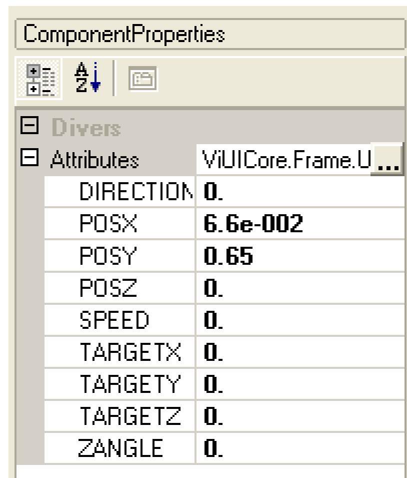


Figure 14 : Attributs du LocationState

Le LocationState définit les attributs de localisation de l'agent qui serviront à l'opérateur de navigation et de perception. Le DrawState, quant à lui, définit le modèle 3D à afficher pour chaque agent.

4.1.2 Sélection des événements

L'étape suivante consiste à sélectionner les événements sur lesquels on désire assigner un ordre. Lorsque les événements sont sélectionnés, ils deviennent des attributs de l'action *SelectOrder*.

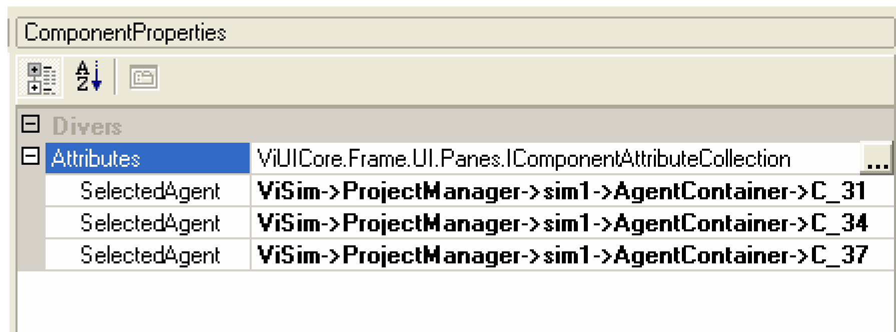


Figure 15 : Sélection des éléments du scénario

4.1.3 Conception de la COA

Lorsque les éléments désirés sont sélectionnés, il faut leur attribuer des ordres. Pour le moment, seul l'ordre *Goto* sera attribué. La suite d'ordres *Goto* génère une suite d'objectifs qui a pour effet de spécifier une trajectoire pour les ressources.

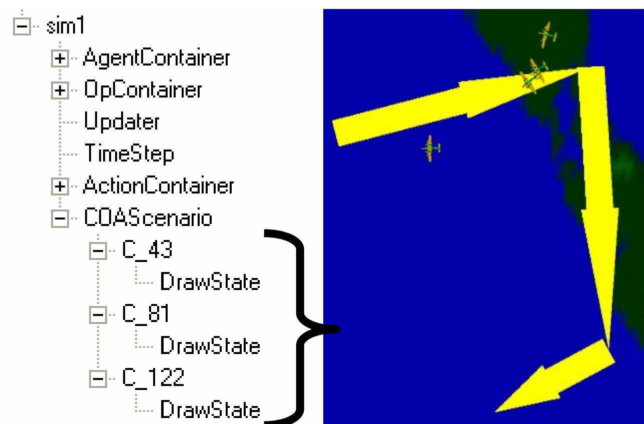
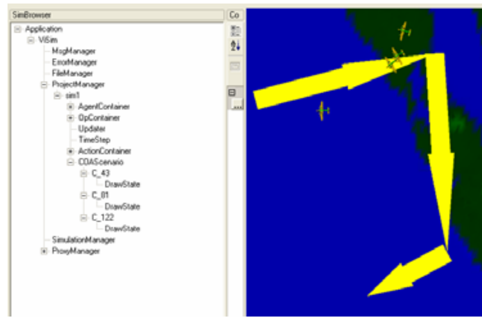
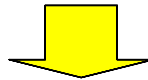


Figure 16 Description et affichage des ordres

La Figure 16 montre les trois ordres qui ont été spécifiés par l'utilisateur sur les trois agents sélectionnés. Chaque ordre possède un DrawState, ce qui permet l'affichage des ordres. Ensuite, chaque ordre est envoyé à chaque agent pour que celui-ci le prenne en charge. La Figure 17 présente le processus d'assignation d'un ordre spécifié par l'usager et la génération du comportement correspondant.

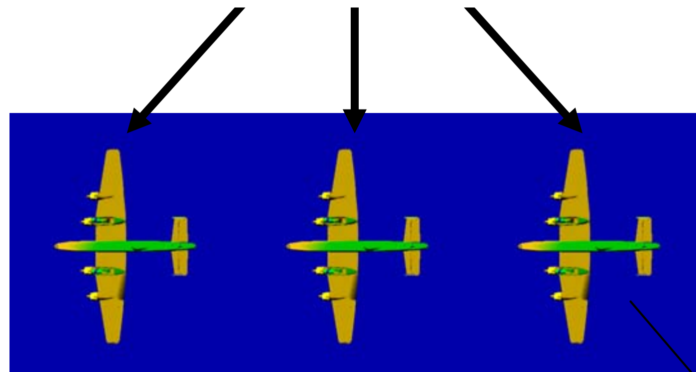


Spécification des ordres par l'utilisateur



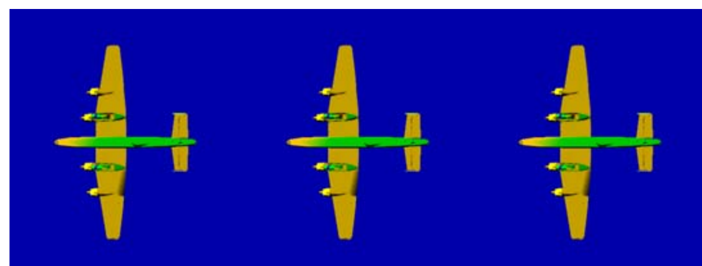
Liste d'ordres

Envoi des ordres à chaque agent impliqué



Chaque agent va chercher la connaissance relative à l'ordre qu'il doit instancier

ActionFactory
Connaissance des comportements à instancier



COABehavior	COABehavior	COABehavior

Figure 17 : Processus de création d'un comportement

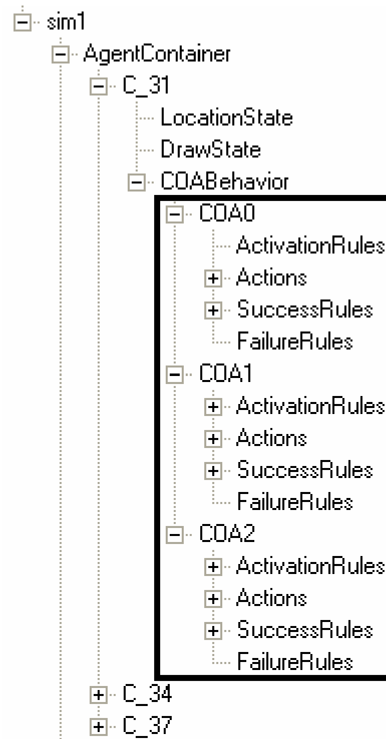


Figure 18 : Comportement généré à partir des ordres

La figure précédente présente les comportements générés à partir de trois ordres *goto*. les composants COA0, COA1, COA2 sont les objectifs permettant d'atteindre les trois objectifs *goto*. Chaque objectif contient ses règles d'activation qui sont généralement liées à la complétude de l'objectif précédent. Ainsi la règle d'activation de COA1 est que COA0 soit *complétéAvecSuccès*. COA0 ne possède pas de règles d'activation puisqu'il est le premier objectif à s'exécuter.

Chaque objectif contient la même action élémentaire ***GotoAction.*** Cette action possède les attributs permettant de modifier le LocationState avec les valeurs XTarget et YTarget de l'agent référencé.

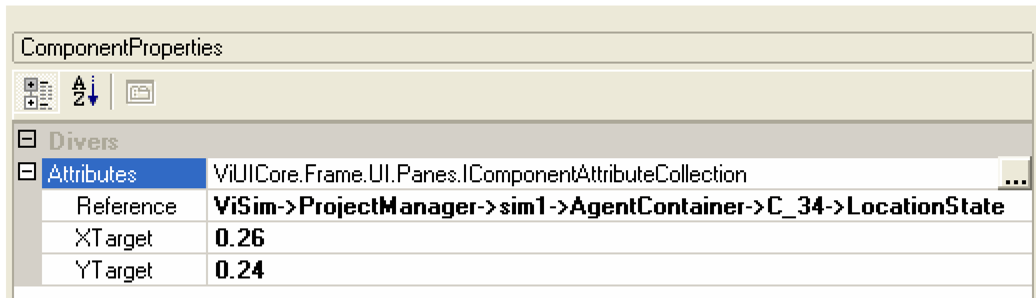


Figure 19 : Attributs de l'action élémentaire GotoAction

La règle de succès qui a été générée vérifie si l'agent a atteint sa destination. Aucune règle d'échec n'a été implantée.

Cet exemple ne montre pas l'intégration des contraintes temporelles. De telles contraintes viendront simplement ajouter des règles d'activation et de complétion en rapport avec le temps écoulé depuis l'activation de l'objectif.

4.1.4 Contraintes temporelles

L'exemple précédent doit être modifié afin d'y ajouter des contraintes temporelles sur les comportements plus spécifiquement sur les actions élémentaires. Un élément important est la spécification de temps maximal alloué pour atteindre un objectif. Si ce temps n'est pas respecté, l'objectif doit échouer.

Pour implanter cette fonctionnalité, il est nécessaire d'ajouter une ***règle d'échec*** (FailureRules) pour l'objectif. De plus, l'objectif doit être en mesure de calculer le temps écoulé depuis son activation. Ensuite, il suffit d'interoger l'attribut *ElapsedActivationTime* de l'objectif et de le comparer avec le temps maximum qui a été aloué. Si la règle s'avère vrai, l'objectif est *completedwithfailure*.

Pour compléter l'exemple précédent, la régle suivant à été ajouté à chaque objectif décrivant un comportement à effectuer :

```
If Obj(ElapsedActivationTime) > MaxTime then NIL
```

La seule précondition de la règle vérifie si le temps écoulé de l'objectif depuis son activation est plus grand que le temps alloué. Il n'y a pas d'actions dans cette règle car lorsqu'une règle de d'échec est vrai, l'objectif devient automatiquement CompletedWithFailure.

C_30
LocationState
DrawState
COABehavior
COA0
ActivationRules
Actions
SuccessRules
FailureRules
IfTimeIsPass
ElapsedTime

ComponentProperties

Divers

Attributes	ViUICore.Frame.UI.Panes.IComponentAttributeCollection
ComponentType	NSimComponents.Precondition
ComponentName	ElapsedTime
LeftMember_ReferenceFloat	ViSim->ProjectManager->sim1->AgentContainer->C_
LeftMember_ReferenceString	ViSim->ProjectManager->sim1->AgentContainer->C_
LeftMember_String	ViSim->ProjectManager->sim1->AgentContainer->C_
LeftMember_Type	ReferenceFloat
Operator	>
RightMember_ReferenceFloat	
RightMember_ReferenceString	
RightMember_String	
RightMember_Type	Float
LeftMember_Float	0.
RightMember_Float	40

Figure 20 : Règle d'échec concernant une contrainte temporelle

4.2 Exemple 2

L'exemple 2 est plus complexe et vise à présenter tous le processus pour effectuer des critiques sur un scénario COA. Il présente donc la spécification du scénario, l'instanciation de la simulation à partir du scénario, le recueil des données de la simulation et le stockage de ces données pour les algorithmes de critiques puissent les exploiter.

4.2.1 Le scénario

Le scénario que nous avons implémenté est illustré par la figure 21. Les avions amis A1, A2 et A3 ont un rendez-vous dans la zone *Z1* pour former ensuite un groupe qui a pour mission d'attaquer le radar ennemi R2. Dans cette opération, A1 a le rôle d'attaquer la cible principale (R1), alors qu'A2 et A3 doivent se préoccuper de tous les autres éléments ennemis qui peuvent entrer dans la scène lors de l'attaque. Dans cet exemple, ils doivent faire face aux avions ennemis E1, E2, E3 et E4 qui font le va et vient pour protéger le radar. Dans ce scénario, le fait qu'A2 et A3 arrivent à temps pour supporter A1 est un aspect critique pour la réussite de la mission.

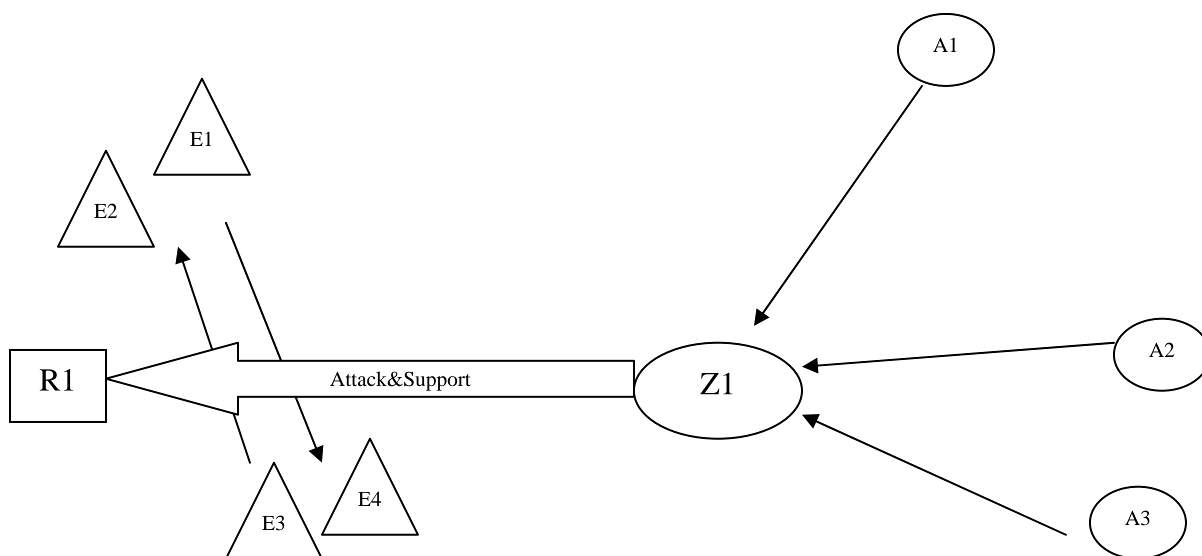


Figure 21 : Représentation du scénario de l'exemple 2

En utilisant une démarche semblable à celle que nous avons utilisée pour le scénario de l'exemple 1, la spécification du deuxième exemple avec l'outil MAGS-COA donne la structure du scénario présenté dans la Figure 22.

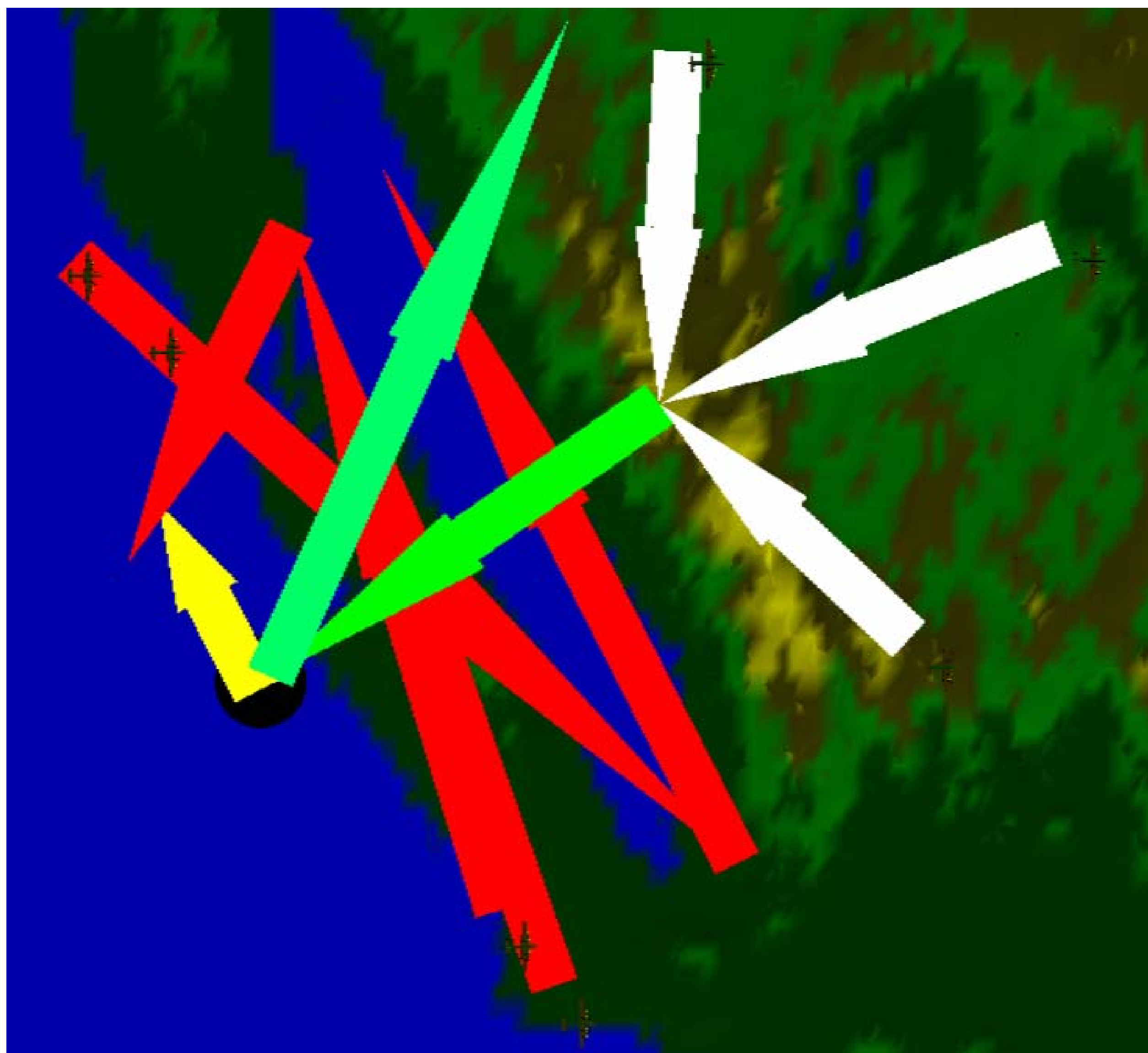


Figure 22 : Structure du scénario de l'exemple 2 dans MAGS-COA

Les flèches blanches représentent l'ordre « Goto&Wait » pour les trois avions qui devront exécuter l'attaque. Lorsque cette objectif « Goto&Wait » sera atteint, les 3 CF18 se verront assigner l'ordre d'attaque (flèche verte). Cette ordre assignera un comportement « Leader » qui attaquera directement la cible (zone noir) et deux comportements « Supporter » visant à soutenir

l'attaque du Leader en le protégeant des ennemies qui patrouillent (flèches rouges). Lorsque la cible sera détruite, le groupe devra rejoindre la base (flèche verte pâle).

Voici tous les composants VSCondition, VSEvent et VSOrder nécessaires pour ce scénario :

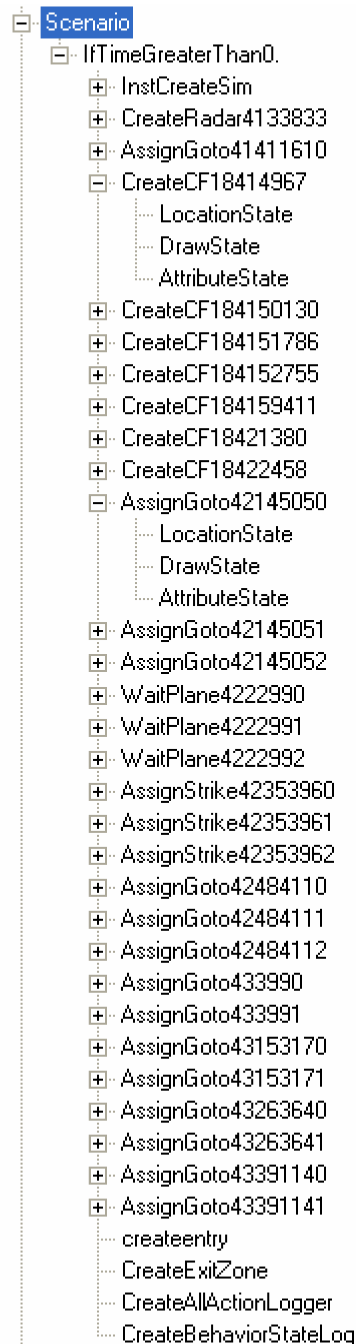


Figure 23 : Structure du scénario

Dans ce scénario, il n'y a qu'un noeud VSCondition nommé « IfTimeGreaterThan0 », ce qui signifie que tout le scénario s'exécute lorsque la simulation démarre (Temps de simulation > 0).

Ensuite, il y tous les événements et les ordres spécifiés par l'utilisateurs pour créer le scénario. Toutes les structures du scénario contiennent trois composantes servant à décrire certaines caractéristiques d'un scénario :

- **LocationState** : Tous les éléments du scénario doivent être représentés dans l'espace afin que l'utilisateur soit en mesure de les positionner, les sélectionner et leur assigner des comportements spatiaux. Ce composant « LocationState » vise donc à décrire la composante spatial d'un élément du scénario VSEvent ou VsOrder.
- **DrawState** : Le scénario doit aussi être représenté visuellement. Pour ce faire, le composant DrawState associé à un VSEvent ou un VSOrder permet de décrire les caractéristiques visuelles de l'élément :
    - Le type de modèle à afficher à afficher (avion, flèche, zone circulaire)
    - La couleur de l'élément
    - La dimension de l'élément

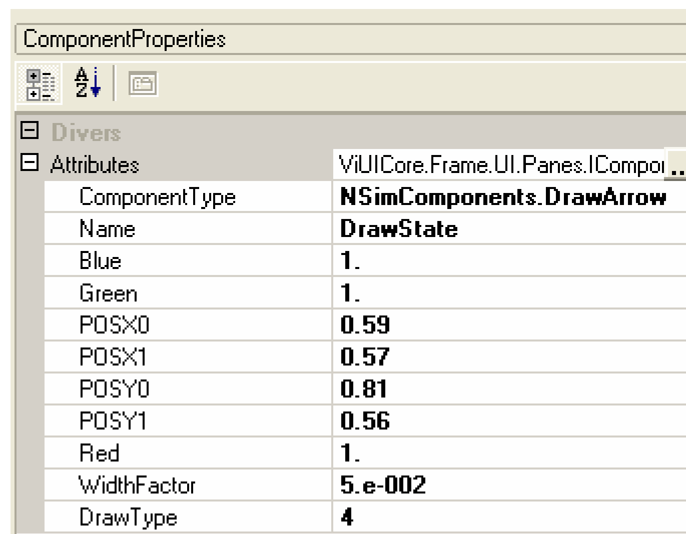


Figure 24 : Attributs d'un DrawState

- **AttributeState** : Le dernier composant d'un élément de scénario est le « AttributeState » servant à décrire des caractéristiques spécifiques pour la création d'un élément dans la simulation. Par exemple, la spécification d'un ordre « Strike » doit indiquer le groupe d'appartenance et le rôle de l'agent à la VSAction qui créera le comportement d'attaque.

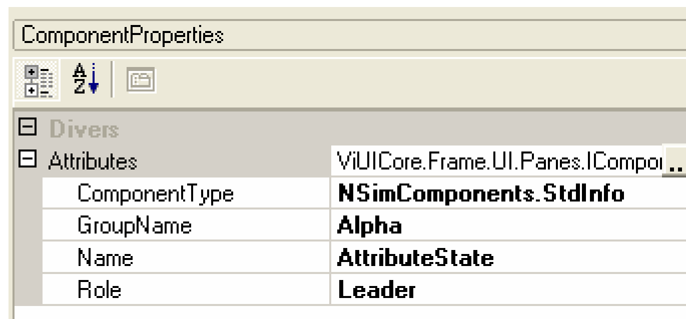


Figure 25 : AttributeState de l'order "Strike"

Les quatre derniers événements du scénario servent à créer les observateurs dans la simulation. Ces 4 observateurs recueilleront différentes données de simulation pour les stocker dans la composantes « Log » de la simulation. Tous les « logs » seront ensuite formatés et enregistrés dans un fichier.

4.2.2 La simulation

Lorsque le scénario est complètement spécifié, la simulation peut être lancée pour tester l'une des évolutions possibles de ce scénario.

Lorsque l'utilisateur appuie sur le bouton « Start », la boucle principale du scénario s'active. La première vérification consiste à demander les éléments actifs du scénario. Dans notre cas, tous les éléments sont retournés puisqu'ils sont tous valides au temps 0 de la simulation. La simulation exécute donc toutes les VSAction associées aux événements et aux ordres du scénario et crée ainsi tous les agents et les comportements dans la simulation.

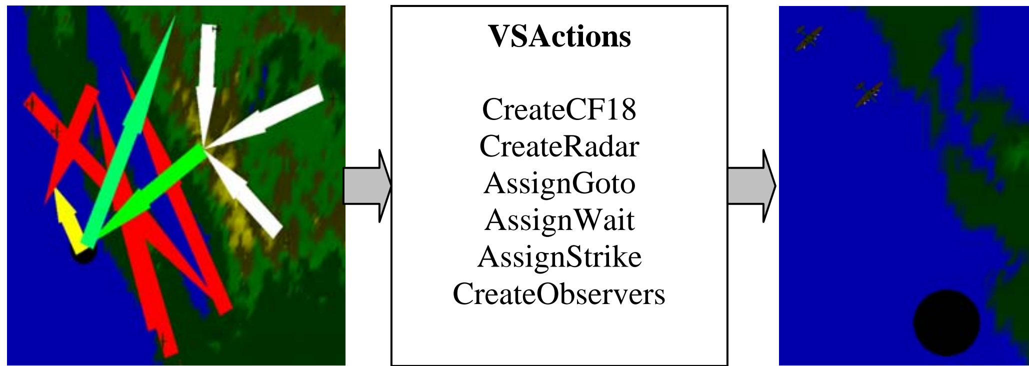


Figure 26 : Scénario → VSAction → Simulation

Lorsque la simulation est démarrée et que les éléments du scénario ont été créés, il est pertinent de présenter les composantes de la simulation qui sont générés.

4.2.2.1 Les Agents de la simulation

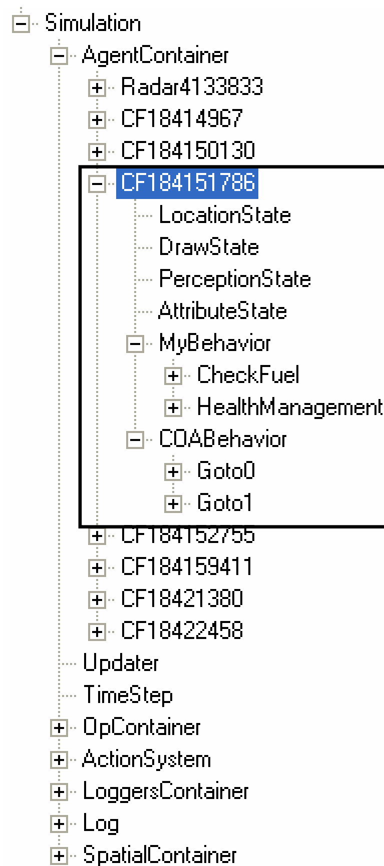


Figure 27 : Un agent CF18 de la simulation

Les agents de la simulation sont tous contenus dans le composant ***AgentContainer***. C'est la VSAction CreateCF18 qui a créé la structure de l'agent de la
Figure 27. Cette VSAction a créé les composantes suivantes de l'agent :

- **LocationState** : Composant décrivant la composante spatiale de l'agent CF18 dans l'environnement de simulation
- **DrawState** : Composant décrivant les caractéristiques visuelles de l'agent dans la simulation ;
- **PerceptionState** : Composant décrivant, en tout temps, l'état de la perception de cet agent (le rayon, la liste des agents perçus)
- **AttributeState** : Il s'agit d'un composant permettant de décrire tous les attributs de l'agent. Par exemple, un CF18 possède les attributs suivants qui sont définis dans le AttributeState de l'événement du scénario :

ComponentProperties

Divers	
Attributes	ViUICore.Frame.UI.Panes.IComponen
ComponentType	**NSimComponents.StdInfo**
Force	**Canada**
HostilityLevel	**High**
Name	**AttributeState**
FuelLevel	**1.3e+004**
HealthLevel	**1.e+002**

Figure 28 : Attributs d'un CF18 de la simlation

- **MyBehavior** : Il s'agit du comportement de base du CF18 qui est généré par la VSAction « CreateCF18 ». L'exemple courant décrit deux comportements de base :
  - CheckFuel : Ce comportement propre au CF18 permet de vérifier en tout temps de carburant de l'agent et de réagir en conséquence. Si le carburant est à 0, l'avion s'écrase (HealthLevel = 0). Le comportement pourrait définir facilement des actions plus complexe comme le ravitaillement ou la retraite.

- HealthManagement : Cet objectif permet au CF18 de vérifier sont niveau de dommage. Si le niveau de dommage est à 0, l'avion s'écrase. De la même façon, il serait possible de spécifier des comportements plus complexes comme la retraite ou l'atterrissage d'urgence.

Il est à noter que le COABehavior du présent CF18 n'est pas généré par la VSAction CreateCF18 mais par l'appel de la VSAction « CreateGoto » servant à assigner deux ordres « Goto » du scénario.

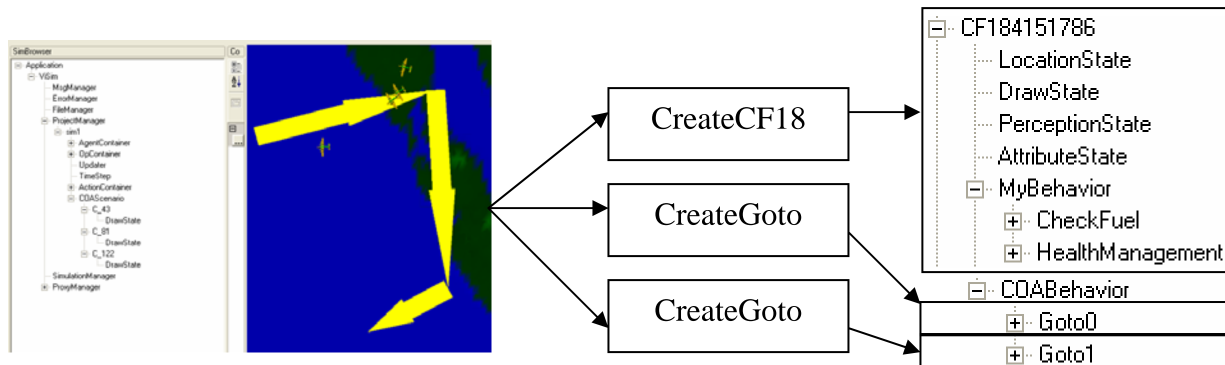


Figure 29 : Création d'un CF18 et assignation de deux ordres

Contraintes temporelles

Un élément important est la spécification de temps maximal alloué pour atteindre un objectif. Si ce temps n'est pas respecté, l'objectif doit échouer.

Pour implanter cette fonctionnalité, il est nécessaire d'ajouter une ***règle d'échec*** (FailureRules) pour l'objectif. De plus, l'objectif doit être en mesure de calculer le temps écoulé depuis son activation. Ensuite, il suffit d'interoger l'attribut *ElapsedActivationTime* de l'objectif et de le comparer avec le temps maximum qui a été alloué. Si la règle s'avère vraie, l'objectif est *completedwithfailure*.

Pour compléter l'exemple précédent, la règle suivant à été ajouté à chaque objectif décrivant un comportement à effectuer :

```
If Obj(ElapsedActivationTime) > MaxTime then NIL
```

La seule précondition de la règle vérifie si le temps écoulé de l'objectif depuis son activation est plus grand que le temps alloué. Il n'y a pas d'actions dans cette règle car lorsqu'une règle de d'échec est vraie, l'objectif devient automatiquement CompletedWithFailure.

Figure 30 : Règle d'échec concernant une contrainte temporelle

4.2.2.2 Les observateurs de la simulation

Le ***LoggersContainer*** contient tous les observateurs de la simulation. L'exemple contient les quatre observateurs décrits précédemment dans la section 1.1. Ces observateurs sont exécutés à chaque cycle de la simulation et ils enregistrent l'information pertinente dans le composant « LogManager » ainsi que dans le fichier texte « simulationlog.txt ».

4.2.2.3 Les données recueillies lors de la simulation

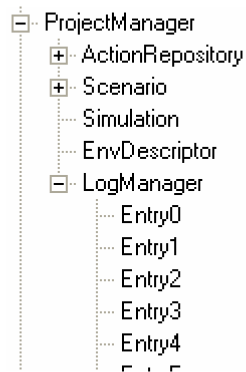


Figure 31 : Les enregistrements des données de simulation

Toutes les entrées (Entry component) sous le composant LogManager représente un fait enregistré dans la simulation. Tous les faits enregistrés pour cet exemple sont présentés en Annexe.

La simulation permet ainsi de simuler un scénario en particulier et d'en recueillir des informations pour que celles-ci soient analysées par le critiqueur.

4.2.2.4 La structure des observateurs

Les observateurs sont structurés de façon hiérarchique. Les noeuds contenus dans cette hiérarchie représentent des opérations sur des données. Chaque noeud de l'arbre utilise les noeuds qui sont directement sous celui-ci pour obtenir les données dont il a besoin pour effectuer son opération. Une fois celle-ci complétée, il retourne la valeur calculée au noeud qui le contient.

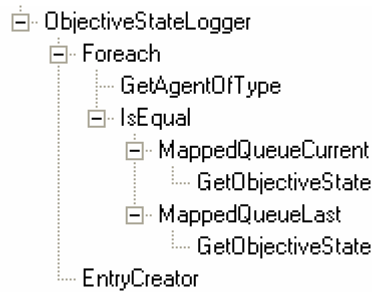


Figure 32 : Structure d'un observateur

La Figure 32 montre l'agent observateur *ObjectiveStateLogger*. Cet agent observe les objectifs contenus dans les comportements des agents. Lorsque l'état d'un objectif change d'un cycle d'exécution à l'autre, ce changement est enregistré dans le composant *Log* sous la forme d'un *LogEntry*.

Le noeud *Foreach,* ainsi que tout les sous noeuds qu'il contient, sont des composants dérivés du patron de classe *FixedExpOperator*. Ce patron de classe accepte comme paramètre le nombre d'opérandes qu'il doit contenir pour être considéré comme valide et exécutable. Dans le cas de *Foreach,* l'héritage est déclaré comme suit :

```
class ForeachConstructList : public FixedExpOperator<2>
{
public:
	virtual IExpElement* Evaluate(const std::vector<IExpElement*>& Param);

};
```

Dans la figure 32, on constate que *Foreach* contient effectivement 2 noeuds, soit *GetAgentOfType* ainsi que *IsEqual*. *Foreach* est donc valide. Il est donc possible pour *ObjectiveStateLogger* d'appeler la fonction *Evaluate* de *Foreach.*

Lors de l'appel de la fonction *Evaluate* de *Foreach,* ce dernier utilise ses deux noeuds pour obtenir les informations dont il a besoin en appelant à leur tour les fonctions *Evaluate* de *GetAgentOfType* et *IsEqual*. *GetAgentOfType* retourne la liste de tous les agents d'un type particulier. *IsEqual* prend en paramètre une référence sur un agent et vérifie si les données retournées par ses deux noeuds opérande sont égales.

Les noeuds *MappedQueueCurrent* et *MappedQueueLast* acceptent en paramètre des références sur les agents desquels l'appelant souhaite tester l'attribut. *MappedQueueCurrent* retourne l'attribut courant tandis que *MappedQueueLast* retourne l'attribut du cycle précédent.

Finalement, *GetObjectiveState* retourne l'état de l'objectif pour un agent dont la référence est passée en paramètre par le noeud appelant.

Si les résultats retournés par la hiérarchie de noeuds sont pertinents *ObjectiveStateLogger* utilise l'objet *EntryCreator* pour créer l'enregistrement dans le conteneur *Log*.

5 Discussion

Ce projet visait à explorer, définir et développer un environnement pour la critique de suites d'actions militaires. Voici donc le résultat des expérimentations qui ont été menées au cours du projet MAGS-COA. Les sections suivantes présentent les conclusions pour chacun des modules développés en y résumant les tests effectués, les choix de conception ainsi que les améliorations possibles à partir de ce qui a été développé.

5.1 Le processus de critique

Avant même de débuter la conception et les expérimentations, il fallait déterminer tout le processus de critique. Ce processus consiste à définir les éléments nécessaires à la critique d'une suite d'actions et la façon de développer ces éléments.

La critique d'une suite d'actions doit nécessairement débuter par la spécification même de la COA (Course of Action). Cette COA est spécifiée à l'aide d'un scénario servant à positionner les ressources et assigner des actions à ces mêmes ressources. Pour ce faire, il faut, préalablement à la spécification du scénario, concevoir et définir les ressources disponibles, leurs comportements propres ainsi que toutes les actions que l'utilisateur pourra assigner aux ressources. Ces deux étapes sont décrites dans les sections 5.2 et 5.3.

Ensuite, lorsque les nouveaux éléments sont spécifiés et que le scénario est créé, il est nécessaire de se poser la question sur la possibilité d'effectuer des critiques directement sur le scénario. Le scénario est la description statique d'une situation donc il peut, en théorie, être critiqué. Le problème est que la critique ne portera que sur l'évolution spécifiée par l'utilisateur. Le scénario spécifie les objectifs à atteindre mais ne détermine pas ou très peu la façon de les atteindre car l'espace d'états pour passer de l'état initial du scénario jusqu'à l'état où tous les objectifs sont complétés est énorme. Deux types de solutions peuvent être développés pour déterminer l'évolution possible du scénario initial : la planification dans l'espace des états possibles ou la simulation du scénario.

La planification

La planification est une technique intéressante qui, une fois tous les paramètres spécifiés (états, actions de transition entre les états), trouvera les actions qui devront être exécutées pour atteindre les objectifs. Ainsi, la critique sera basée sur une planification infructueuse qui serait analysé. Cette solution n'a pas été abordée dans ce projet étant donné les l'espace d'états très grand (scénario spatio-temporel) et la difficulté à déterminer la distribution de probabilité des actions exécutées pour chaque état.

Cette avenue de recherche reste néanmoins possible à plus long terme et mérite que l'on y accorde quelques recherches théoriques de faisabilité.

La simulation

La méthode choisie pour déterminer l'évolution possible du scénario COA est la simulation. La simulation parcourt l'espace d'états à l'aide d'algorithmes sophistiqués (navigation, perception, comportements) pour atteindre ou non l'état final. Cette technique est plus simple que la planification mais possède quand même certaines difficultés :

- La conception des algorithmes de simulation

  C'est la qualité et la cohérence des algorithmes utilisés pour parcourir les états qui détermineront la validité de la simulation. Durant ce projet, des algorithmes utilisés pour parcourir l'espace d'états (i.e. pour simuler le scénario), sont la navigation, la

perception, le comportement à base d'objectifs et les actions élémentaires (Strike, Wait). Ce sont ces algorithmes qui font évoluer le scénario vers sont état final. Il faut donc s'assurer, à l'aide d'expert, que les algorithmes développés sont représentatifs de la réalité.

- L'observation des états et de la transition entre les états

Étant donné que la transition entre les différents états se fait à travers les algorithmes, il est très difficile de déterminer la suite exacte de transitions qui ont conduit à l'état observé. Pour ce faire, il faut définir des « Observateurs » (voir section 3.7) spécialisés qui devront enregistrer explicitement la succession de certains états de la simulation. Évidemment, les observateurs ne pourront enregistrer toutes les transitions d'états étant donné la quantité énorme de données que cela représente.

La technique de la simulation permet donc de réduire le nombre d'états à spécifier en intégrant des algorithmes de simulation. Cependant, le fait de calculer de façon algorithmique les transitions entre les états fait perdre des informations qui pourraient s'avérer importantes pour la critique du scénario. Par exemple, l'algorithme de navigation d'un CF18 ne permet pas de retrouver explicitement toutes les transitions de positions et de temps (i.e. sa trajectoire dans l'espace). Pour ce faire, il faudrait définir un observateur dédié qui aurait pour tâche d'enregistrer tous les changements d'états sur un LocationState d'un CF18.

Finalement, suivant le choix algorithmique de la simulation pour calculer l'évolution du scénario, voici le processus à mettre en place pour produire des critiques :

- Spécification des ressources et des actions possibles à affecter ;
- Spécification d'un scénario à l'aide des ressources et des actions précédemment définies ;
- Simulation du scénario à l'aide des algorithmes développés ;
- Conception des observateurs nécessaires pour recueillir l'information sur les états de la simulation ;
- Formatage et stockage des informations ;
- Conception des algorithmes de critique sur les informations recueillies.

5.2 La création de nouveaux éléments

Avant même de pouvoir spécifier des scénarios, il faut savoir quels sont les types d'éléments disponibles pour l'utilisateur. Le prototype MAGS-COA possède, à ce stade-ci, 2 types de ressources sont disponibles (CF18 et zone radar) et 4 types d'ordres à assigner (Goto, Strike, Wait, WaitPlane).

Pour être en mesure de créer **facilement** de nouvelles ressources et de nouvelles actions, il faut un niveau de généralité qui n'est pas encore atteint. Pour le moment, les nouveaux éléments du scénario doivent être définis directement dans le code source (voir Annexe à ce sujet) et il n'existe aucun interface utilisateur pour la spécification des ordres.

5.3 La spécification du scénario

La spécification d'un scénario consiste à définir, dans l'espace et dans le temps, les éléments du scénario (événements et ordres). L'objectif est de définir l'état initial du système (les événements) et d'en spécifier les objectifs (les ordres). Ainsi, le simulateur pourra, à l'aide des algorithmes de simulation, faire évoluer le scénario initial vers les objectifs fixés.

Pour permettre au scénario de s'instancier une simulation, l'introduction d'une nouvelle structure à été nécessaire pour faire le lien entre la description des ordres et les algorithmes de simulation. Cette structure (VSActionRepository) consiste à créer les algorithmes de comportements de la simulation qui exécuteront les ordres du scénario. Le comportement à base d'objectifs est l'un des algorithmes de simulation important permettant de faire évoluer adéquatement l'état de la simulation. Il est donc nécessaire de bien définir le VSActionRepository pour que les ordres soient exécutés correctement dans la simulation.

5.3.1 Les avantages

Le fait de séparer la spécification du scénario, la simulation et le lien entre les deux a plusieurs avantages intéressants. Premièrement, le scénario devient indépendant de son implantation dans la simulation et est ainsi facilement portable d'un système à l'autre (évite les contraintes

techniques). De plus, il est possible de modifier la spécification des algorithmes comportementaux (VSActionRepository) sans affecter la description du scénario. Un utilisateur peut donc définir un comportement de plusieurs façons selon le niveau de réaliste désiré, la configuration matérielle disponible ou selon tout autre considérations susceptibles d'affecter la simulation.

5.3.2 Les limitations

La limitation principale est que les algorithmes qui définissent comment évolue la simulation ne peuvent pas tous se situer au niveau du VSActionRepository. L'idéal serait que le VSActionRepository contienne la spécification de tous les algorithmes de contrôle de la simulation. Ainsi, le VSActionRepository ne ferait qu'instancier une simulation directement à partir de la description du scénario. Cependant, certains algorithmes doivent être préalablement développés directement dans le simulation puisqu'il est, techniquement, trop complexe de définir une simulation temps réel performante à partir d'algorithmes complètement génériques. Ce projet a mené à la conception de plusieurs algorithmes qui doivent être directement implantés dans le simulateur. Ces algorithmes sont la ***navigation***, la ***perception*** ainsi que les ***actions élémentaires*** des objectifs **Strike** et **WaitPlane**. Les algorithmes comportementaux qui définissent les objectifs des agents dans la simulation sont par contre définis d'une façon générale à l'aide du VSActionRepository.

5.3.3 Travaux futurs

Au niveau de la spécification des scénarios et de l'instanciation de la simulation par le VSActionRepository, plusieurs amélioration et travaux futurs sont possibles. Théoriquement la spécification d'un scénario est un processus assez simple où un utilisateur positionne, dans l'espace et dans le temps, les ressources utilisées et les ordres reliés à ces ressources. Cependant, la visualisation et la manipulation efficace du scénario reste un problème complexe. C'est évidemment une approche liée à la conception d'une interface utilisateur permettant de présenter et de manipuler de l'information spatio-temporel.

Une autre amélioration se situe au niveau de la description du VSActionRepository qui décrit les algorithmes utilisés dans la simulation. Cette structure devrait permettre la spécification

générique d'algorithmes complexes comme la navigation des agents et la perception. Ce n'est pas un problème simple, puisqu'il faut définir un langage déclaratif de spécification pour interpréter les comportements des agents dans la simulation. De plus, le résultat de la spécification doit permettre la création d'une simulation performante en temps réel, ce qui n'est pas simple avec un langage interprété..

5.4 La simulation

L'étape de simulation est une phase importante du processus de critiques puisqu'elle permet de créer dynamiquement des informations sur l'une des évolutions possibles du scénario spécifié.

5.4.1 Les avantages

Le système de simulation présenté dans la section 3.4 permet de décomposer les structures comme les agents des algorithmes qui les régissent. Le principe est exactement le même que dans le scénario où celui-ci était séparé de son implantation du VSActionRepository. Le simulateur sépare ainsi les agents et leur structure des algorithmes qui les font évoluer dans la simulation. L'avantage est de permettre à plusieurs algorithmes de s'exécuter sur un agent selon le contexte de la simulation. Par exemple un agent pourrait utiliser un algorithme de navigation très précis lorsque l'environnement est détaillé et un autre algorithme plus grossier lorsque l'environnement devient moins précis.

5.4.2 Les travaux futurs

Les travaux futurs sur le module de simulation sont essentiellement de développer des algorithmes de contrôle plus précis et plus complexes :

- Des algorithmes pour la coordination entre les membres d'un groupe pourrait être conçus ;
- Des algorithmes de navigation plus complexes ;
- Des algorithmes de perception plus sophistiqués ;
- Des algorithmes de modification temps réel de l'environnement virtuel ;
- Des algorithmes de physiques plus complexes (collision, gaz, feux, etc.) ;

Tout le travail effectué sur le module de simulation aura pour effet d'améliorer les résultats en fournissant des données plus cohérentes et plus précises.

5.5 Les observateurs

Vu la direction empruntée, soit celle d'utiliser la simulation pour faire évoluter le scénario, les observateurs sont des éléments essentiels et très importants. Ils permettent de recueillir l'information sur l'état courant de la simulation à un instant précis. Étant donné que le parcours des états est implicite et que l'enregistrement de toutes les transitions d'états génère trop de données, il est nécessaire d'utiliser des observateurs spécialisés. La section 1.1 présente les 4 types d'observateurs développés pour répondre aux besoins du projet et ainsi enregistrer les 4 type de données nécessaires.

5.5.1 Les avantages

Le fait d'utiliser des observateurs spécifiques pour recueillir l'information sur les changements d'états possède l'avantage de générer des quantités de données beaucoup moins importantes et d'être, ainsi, plus facilement manipulables. De plus, les données observées seront, si le choix de l'utilisateur est orienté vers un objectif précis, plus pertinentes pour les algorithmes de critiques qui seront probablement spécialisés. Un autre avantage, plus technique, est la performance du système de simulation qui est plus élevée.

5.5.2 Les limitations

La limitation la plus importante réside dans le fait que tous les états de la simulation ne sont pas enregistrés. Il y a donc une sélection des états qui seront conservés, ce qui réduit la probabilité de découvrir des liens inattendus. La critique sera basée sur les données disponibles provenant de la simulation, donc il y a un grand risque d'une perte d'expressivité de la critique.

La simulation sert à effectuer des changements d'états. Si certains états ne sont pas enregistrés sous prétexte qu'ils n'apportent peu ou pas d'informations supplémentaires à la critique, il devient donc obsolète de les simuler. Ce point est très important puisqu'il peut être une source de problèmes théoriques. Une simulation n'est, en fait, qu'une succession de transitions d'états

(changement de position d'un CF18 par l'algorithme de navigation, etc.). À partir de cette définition d'une simulation, il est pertinent de définir un « regroupement d'états » pour bien saisir l'impact de définir des observateurs.

Il se peut que, dans tout l'espace d'états, certains regroupements d'états soient isolés par rapport aux autres. Par exemple, si l'état courant de la simulation possède un attribut où un CF18 est abattu, la simulation n'a aucune chance de transiter vers un état où ce même CF18 est en plein vol. Il y a donc des regroupements d'états qui sont inaccessibles à partir d'autres regroupements d'états.

Tous ces regroupements possèdent donc des états qui sont possiblement accessibles dans tout le regroupement. Si ce n'est pas le cas, ils deviennent un regroupement en soi.

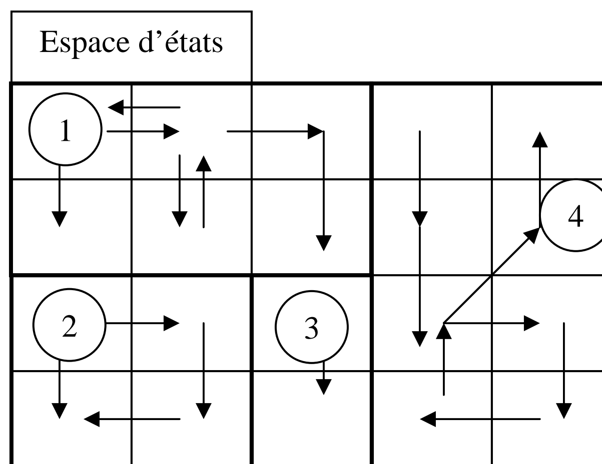


Figure 33 : 4 regroupements d'états

La figure précédente présente un espace d'états d'une simulation avec 4 regroupements d'états qui ne sont pas accessibles les uns avec les autres. Le fait de définir des observateurs peut limiter les données récupérées. Par exemple, un observateur 1 est placé sur un état et a pour mandat d'observer le changement de cet état. Il enregistrera l'information sur le changement vers la droite ou vers le bas lorsque celui-ce se produira. Dans un même regroupement, l'observateur ne pourra pas enregistrer l'intégralité des changements d'états, ce qui causera ainsi une perte d'information risquant d'affecter la qualité de la critique. Par exemple, l'algorithme de critique se

basant sur l'information fournit par l'observateur 4 ne pourra jamais déterminer la succession exacte de transitions qui a conduit à cet état observé. Un exemple très simple illustrant ce problème est que, actuellement, un observateur enregistre le fait qu'un CF18 entre ou sort d'une zone de visibilité. Cette transition d'états est enregistrée mais il n'y a aucun moyen de connaître la provenance de ce CF18, puisqu'il n'y a aucun observateur qui enregistre le changement d'états de la position des CF18. La critique ne pourra donc pas expliquer pourquoi ce CF18 se trouvait dans cette zone de visibilité.

Une autre considération importante est la possibilité de ne pas développer d'observateurs pour tous les regroupements d'états. Étant donné que les regroupements ne s'affectent pas entre eux, le fait de ne pas enregistrer d'informations sur les états d'un regroupement équivaut à ne pas simuler cet aspect. Il se peut donc qu'il y ait des aspects de la simulation qui soient inutiles étant donné le manque d'observateurs dans le regroupement d'états représentant ces aspects. Ce problème est difficile à cerner puisqu'une simulation peut contenir des milliards d'états et que le regroupement de ceux-ci n'est pas trivial.

5.5.3 Les travaux futurs

Les travaux futurs peuvent donc être théoriques ou techniques. Du point de vue technique, il serait intéressant de développer une méthode pour enregistrer efficacement tous les états de la simulation. Cette limitation technique amène des avenues de recherches alternatives pour catégoriser ou réduire le nombre d'états pertinents pour la critique.

Une solution intéressante serait de concevoir des méthodes pour évaluer et indexer les regroupements d'états d'une simulation. Ainsi, les algorithmes de critiques pourraient se spécialiser en ne critiquant que des aspects précis de la simulation. Il faudrait donc développer autant d'algorithmes de critiques qu'il y a de regroupements d'états. Ainsi, en simulant plusieurs fois (1 simulation par regroupement) et en combinant les critiques des différentes simulations, il pourrait y avoir des résultats intéressants et plus complets.

Une autre avenue de recherche est de réduire le nombre d'états d'une simulation. Une des solutions pour y arriver est de représenter qualitativement les attributs des états. Par exemple, un itinéraire d'un CF18 pourrait être exprimé qualitativement au lieu d'une représentation quantitative (suite de points). L'aspect spatial et temporel cause un grand nombre d'états dans la simulation puisqu'il s'agit de variables continues qui ont été discrétisées. La représentation qualitative de ces variables peut être une solution à explorer autant pour la représentation de la simulation que pour la critique.

5.6 Les données recueillies

Les observateurs recueillent des données sur l'état de la simulation et les stockent selon un format précis (voir section 1.1). Les avantages de stocker l'information dans cette forme est que l'enregistrement devient complètement général. Il peut être produit à partir de n'importe quelle source et peut être stocké dans une destination externe comme un fichier XML ou une BD relationnelle. De plus, ce format est pleinement extensible et ne dépend pas du type d'information qui est enregistré.

Un désavantage est qu'il s'agit d'un type d'enregistrement déclaratif qui produira beaucoup de données lorsqu'il y aura beaucoup de relations entre les enregistrements.

Pour le moment, les données sont stockées dans un fichier texte. Les travaux futurs se limitent donc au développement de modules permettant différents type de stockage (XML, BD, etc.) pour manipuler et accéder efficacement aux données.

5.7 La critique

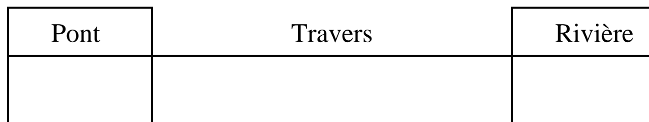
L'implémentation d'un première version du module de critique est en cours. L'objectif des prochaines semaines est d'implémenter un cas simple qui illustre comment le critiqueur utilise différentes connaissances pour analyser les traces d'une simulation et générer une critique. L'implémentation de ce cas simple devrait nous permettre d'organiser et de choisir les formalismes de représentation des différentes connaissances utilisées par le critiqueur.

5.8 L’environnement

Un travail de réflexion a été effectué à propos de la représentation de l’environnement. Le modèle préconisé se compose de deux parties, la portion réelle et la portion abstraite.

L’environnement réel est généré à partir de données géographiques précises provenant de source réelle ou imaginaire. Celui-ci est utilisé par les algorithmes s’exécutant directement dans l’agent. Par exemple, l’algorithme de navigation accède à une carte d’élévation afin de déterminer quels sont les obstacles à proximité afin de calculer l’itinéraire dans le cycle courant d’exécution. La portion réelle de l’environnement est donc constituée de données spatiales référencées qui forment ce que l’on pourrait nommer la réalité physique de l’environnement simulé.

La portion abstraite de l’environnement quant à elle représente les concepts de connaissance reliés à l’environnement réel. Il peut ainsi définir les lieux ainsi que les liens entre ces lieux. Au départ, le modèle caractérise ces liens de façon purement qualitative. Ainsi, la relation entre une rivière et un pont peut se représenter comme suit :



Une des caractéristiques voulues pour l’environnement réside dans l’indépendance des portions réelles et conceptuelles de celui-ci l’une par rapport à l’autre. C’est-à-dire que l’on souhaite pouvoir construire un modèle conceptuel sans nécessairement être lié a un modèle réel et vice-versa. Plusieurs raisons sont à l’origine de ce choix. D'abord, il serait intéressant de pouvoir effectuer des critiques sans environnement réel afin de limiter volontairement le traitement des informations nécessaires à la critique à des éléments et des relations abstraites.

Dans le cadre de la critique des suites d'action, la première utilisation considérée de l'environnement conceptuel consiste à déterminer quelles sont les causes et les conséquences d'un événement particulier. Dans l'exemple précédent, le pont devient infranchissable lorsque le niveau de l'eau est trop élevé. Un agent dont l'objectif est de traverser la rivière pourra rencontrer un échec donc la cause serait l'impossibilité de franchir le pont. La cause ultime devient ici le niveau trop élevé de la rivière. Ce premier exemple suscite à première vue quelques réflexions.

- Quelle est la relation entre l'objectif à atteindre et le concept de pont ? Cette relation existe-t-elle ? est-elle nécessaire ? Si elle existe, est-elle de même nature que la relation liant le pont et la rivière ?
- Comment fait-on pour remonter le fil des causes? Dans ce cas-ci, comment sait-on que la véritable cause du changement d'état du pont est le changement d'état de la rivière? Doit-on enregistrer tous ces changements d'état en prenant soin de déterminer sur le fait la cause.

De plus, pour que ceci soit possible, il faut que ces concepts possèdent des attributs dynamiques. Par exemple, le pont, au départ franchissable, devient infranchissable. La complexité provient du fait qu'en intégrant aux éléments de l'environnement conceptuel des attributs qui sont dynamiques, il faut en quelque sorte mettre à jour la chaîne des conséquences de ce changement, elle-même déterminée par les liens entre les différents éléments de l'environnement. D'où proviennent les changements ? Proviennent-t-ils de l'extérieur de l'environnement conceptuel ? Et si la modélisation de la rivière provient d'un modèle de simulation physique, donc non conceptuel, il faudra que le niveau de la rivière conceptuelle soit ajusté en temps réel, ce qui risque de provoquer une dépendance très grande entre la portion conceptuelle et la portion réelle de l'environnement.

Si la relation entre les éléments ponts et rivières semble intuitive, la relation entre l'augmentation du niveau de l'eau et la transformation d'un pont franchissable en un pont infranchissable est quant à elle plus complexe et dépasse le cadre de la définition d'un espace géographique conceptuelle. À ce point, la question est de savoir si la possibilité pour la rivière de transformer le

pont franchissable en pont infranchissable est une caractéristique de la relation entre ces deux éléments.

La philosophie même de séparation des environnements présente donc de nombreuses limitations que l'on peut résumer ainsi

- Un environnement conceptuel dynamique doit permettre sa modification de l'extérieur en même temps qu'il doit pouvoir agir sur l'environnement réel. Ce qui laisse de nombreuses interrogations quant à l'interaction mutuelle de ces deux parties.
- Puisque la représentation conceptuelle de l'environnement représente en quelque sorte une vision plus humaine de la connaissance, il serait sûrement souhaitable qu'il soit possible d'accéder à ces informations à partir même des comportements. Cette possibilité augmente encore plus la dépendance des deux représentations l'une par rapport à l'autre
- L'environnement conceptuel doit posséder des renseignements extrêmement précis à propos de ses éléments si l'on désire qu'il soit possible de spécifier d'éventuelle relation cause à effet. Sinon, ces relations devront être spécifiées dans la portion réelle de l'environnement, ce qui rendrait la portion conceptuelle plus ou moins utile.

6 Conclusion

Le projet MAGS-COA a permis de mettre en place un cadre et une méthode pour être en mesure de développer des critiques de suites d'actions militaires. Cette méthode peut se résumer en trois étapes. Dans la première étape, l'utilisateur doit spécifier son plan (suite d'actions) appelé *scénario* : il définit les ressources, leurs positions initiales, les tâches à accomplir, les contraintes temporelles et la chronologie d'exécution des tâches. La deuxième étape consiste à simuler le scénario dans un environnement spatial virtuel géo référencé. Les ressources du plan sont alors des agents plongés dans l'environnement virtuel et qui tentent de réaliser les tâches qui leur sont assignées. Durant la simulation, les évènements qui sont jugés pertinents pour la dimension à critiquer sont enregistrés dans un registre de données (trace de simulation). Finalement, la troisième étape consiste à analyser la trace de la simulation en utilisant différentes sources de connaissances pour générer des critiques.

Ces étapes ont été réalisées dans le prototype MAGS-COA et les résultats démontrent qu'il est possible d'utiliser ce processus et cette technologie pour critiquer les suites d'actions.

7 Annexe A

7.1 *L’ajout des nouvelles actions au VSAction Repository*

Dans cette annexe nous décrivons les étapes principales pour ajouter des nouvelles actions (des nouveaux comportements) au simulateur MAGS-COA. Pour ce faire, nous proposons de suivre un exemple simple qui consiste à implémenter un comportement permettant à un agent de ‘s’arrêter’ dans une zone spatiale pendant une période de temps donnée (attente).

1. Création de l’ordre

La première étape consiste à créer l’ordre correspondant à l’action d’attente dans le scénario. Dans le projet de développement de l’environnement VC++, les ordres sont placés dans le dossier COAClient –>Ordres –>AgentOrder. Tel qu’illustré par la figure 1, nous allons créer un fichier nommé ‘Order_CreateWait.cs’ dans ce dossier.

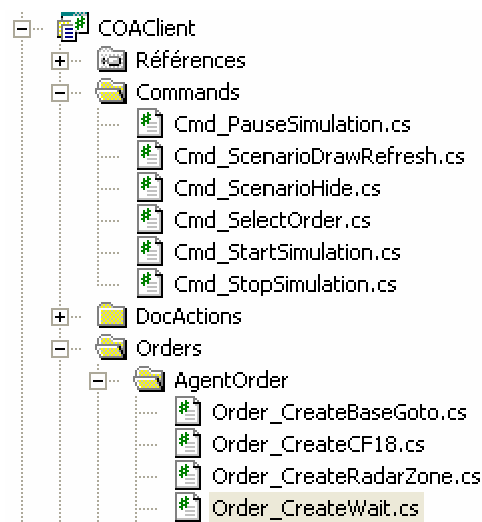


Figure 1 : L’emplacement du fichier de définition de l’ordre dans la structure du projet

Le fichier de code que nous venons d’ajouter permet de créer l’ordre d’attente dans l’arborescence du scénario et de faire le lien avec l’action qui l’implémente dans le

VSActionFactory. Les lignes qui permettent de créer l'ordre dans l'arborescence du scénario sont les suivantes :

```
Msg.SetType("CreateOrder");
Msg.SetDestination("remote->ViSim->ProjectManager->ActionRepository-
>CreateOrder");
Msg.SetSource("ViUI->MultiDocView");

Msg.AddElement("Wait" + date.Minute + date.Second + date.Millisecond); //Le
nom de l'ordre dans le scénario

Msg.AddElement("ViSim->ProjectManager->ActionRepository->CreateWait"); //La
référence à l'action de l'Actions Repository qui va créer le comportement

Msg.AddElement("CurrentSelection"); // Référence de l'agent qui est
sélectionné et qui est l'acteur de l'ordre

Msg.AddElement("NULL"); // Pas d'objet pour cet ordre, l'agent ne va pas se
déplacer ou faire une action sur un autre agent

Msg.AddElement("NSimComponents.DrawArrow"); // Le type de l'objet à afficher
pour l'ordre, dans ce cas c'est une flèche.

Msg.AddElement((float)-1);    // La position de l'ordre (la flèche)
Msg.AddElement((float)-1);
Msg.AddElement((float)-1);
```

Donc à ce niveau il s'agit tout simplement de créer l'instance de l'ordre et de définir ses paramètres (nom, objet, etc.). Il s'agit en fait à créer une instance du composant ordre comme ça été définie dans la section 2.2.2 du présent rapport. Notons bien la ligne de code '`Msg.AddElement("ViSim->ProjectManager->ActionRepository->CreateWait")`' qui permet de référencer l'action qui implémente l'ordre dans le VSAction Repository.

2. Création du COA-Behaviour correspondant à l'ordre

Nous avons déjà expliqué dans le rapport que l'agent a deux types de comportement : le comportement qu'on lui demande de faire dans le scénario (COA-Behaviour) et son comportement réel lors de la simulation (My Behaviour). L'étape suivante consiste à définir la structure du COA-Behaviour de l'agent sélectionné.

Les COA-Behaviours des agents sont définis dans le projet de développement dans le dossier COADLL->ActionFactory->AgentAction. À ce dossier, nous allons ajouter un fichier

nommé COACreateWait.cpp pour créer le comportement COA-Behaviour. Dans ce fichier on indique le nom de l'action du VSAction Repository qui doit être exécutée par l'agent, sa règle d'activation et ses conditions d'échec et de succès (donc les contraintes liées à l'action et qui sont spécifiées par l'usager).

Les lignes de code suivantes représentent l'implémentation de ces éléments pour l'ordre d'attente (Wait) :

Spécification de la règle d'activation

La règle d'activation de cet exemple consiste à vérifier si l'objectif précédent de l'agent (dans le cas ou il y en a) a été complété avec succès ou nom.

```
ViComponent* prec = COABehavior->GetComponent(COABehavior->GetNbComponent()-2);
  if (prec != NULL)
   {
     // Construct a activation rule (if precedent objective is completedWithSuccess)
     ViComponent* ActRule = m_pCore->CreateComponent("NSimComponents.Rule", "IfICanStart", obj->m_Parent + "->" + obj->m_Name + "->ActivationRules");

     ViComponent* prcSuccess = m_pCore->CreateComponent("NSimComponents.Precondition", "PrecSuccess", ActRule->m_Parent + "->" + ActRule->m_Name);
     prcSuccess->Init("LeftMember_Type", "ReferenceString");
     prcSuccess->Init("RightMember_Type", "String");

     prcSuccess->Init("LeftMember_ReferenceString", prec->m_Parent + "->" + prec->m_Name + "(LifeCycle)");
     prcSuccess->Init("RightMember_String", "CompletedWithSuccess");

     prcSuccess->Init("Operator", "==");
   }

  float time = *(float*)order->GetInfo("TimeToComplete");
```

Spécification de l'action du VSAction repository qui va être appelée pour implémenter le comportement d'attente

```
std::string ref = agent->m_Parent + "->" + agent->m_Name + "->LocationState";

ViComponent* c = m_pCore->CreateComponent("COADll.COAWait", "WaitAction", obj->m_Parent + "->" + obj->m_Name + "->Actions");
```

```
c->Init("Reference", ref);
```

Spécification des conditions de succès

Dans cet exemple, l'action réussit si l'agent a attendu la période de temps qu'on lui a demandée dans le scénario.

```
ViComponent* successRule = m_pCore->CreateComponent("NSimComponents.Rule", "IfTimeIsPass", obj->m_Parent + "->" + obj->m_Name + "->SuccessRules");

    ViComponent* ElapsedTime = m_pCore->CreateComponent("NSimComponents.Precondition", "ElapsedTime", successRule->m_Parent + "->" + successRule->m_Name);
    ElapsedTime->Init("LeftMember_Type", "ReferenceFloat");
    ElapsedTime->Init("RightMember_Type", "Float");

    ElapsedTime->Init("LeftMember_ReferenceFloat", obj->m_Parent + "->" + obj->m_Name + "(ElapsedActivationTime)");
    ElapsedTime->Init("RightMember_Float", time);

    ElapsedTime->Init("Operator", ">");
```

Spécification des conditions d'échec

Dans cet exemple nous n'avons des conditions d'échec

3. Programmation de l'action du VSAction Repository

L'étape suivante consiste à définir l'algorithme proprement dit qui permet d'implémenter le comportement de l'action d'attente. Les fonctions qui implémentent le comportement des agents sont définies dans le dossier COADLL->Agent->Actions. A ce dossier nous allons ajouter le fichier COAWaitAction.cpp qui permet de spécifier le comportement de l'agent associé à l'action d'attente. Dans le cadre de cet exemple, nous avons choisi un comportement simple : l'avion ne bouge pas, il reste dans sa position. Cependant, il aurait été possible de choisir n'importe quel autre comportement, par exemple de tourner dans un cercle ou dans une zone spatiale. Ces comportements doivent être implémentés au niveau de ce fichier.

Dans notre exemple, les lignes de code qui permettent à un agent de rester à sa place sont les suivantes :

```
float positioncouranteX = *(float*)CompReference->GetInfo("POSX");
float positioncouranteY = *(float*)CompReference->GetInfo("POSY");

CompReference->Init("TARGETX", positioncouranteX);
CompReference->Init("TARGETY", positioncouranteY);
```

4. Faire le lien entre les différents éléments ajoutés

Une fois qu'on a crée les fichiers de code qui permettent d'implémenter l'ordre, son COA-Behaviour correspondant et l'action qui décrit comment l'agent se comporte pour réaliser cet ordre, il faut faire le lien entre ces différents fichiers. Le lien est fait dans le fichier COADLL->DllInterface.h. Dans le cadre de notre exemple, ce lien est établi avec les lignes de code suivantes :

```
if (type == "COADll.Action.CreateWait")
  {
    c = new COACreateWait;
  }
if (type == "COADll.COAWait")
  {
    c = new COAWaitAction;
  }
```

Donc quand le fichier Order_CreateWait.cs fait appel a l'action *CreateWait*, une nouvelle classe de *COAWaitAction* sera activée. Quand cette classe fait appel à l'action *COAWait*, le comportement associé à l'agent sera exécuté par l'instanciation de la classe *COAWaitAction*. En suivant ces étapes, il est possible d'ajouter d'autres comportements d'agents de façon simpliste.

7.2 Description des comportements développés

Voici les différents comportements développés pour la réalisation de l'exemple d'application. Tous les comportements sont à base d'objectifs et sont générés par les VSAction correspondants.

CheckFuel

Cet objectif est assigné par la VSAction CreateCF18

Objective : CheckFuel

ActivationRule : If (Me→AttributeState(FuelLevel) <= 0)

Actions : Wait

SuccessRule : NIL

FailureRule : NIL

HealthManagement

Cet objectif est assigné par la VSAction CreateCF18

Objective : HealthManagement

ActivationRule : If (Me→AttributeState(HealthLevel) <= 0)

Actions : Wait

SuccessRule : NIL

FailureRule : NIL

Goto

Cet objectif est généré par la VSAction CreateBaseGoto. La règle d'activation vérifie si l'objectif précédent est complété avec succès avant d'activer l'objectif courant. C'est ainsi que la suite d'actions est produite dans le comportement de l'agent.

Objective : Goto

ActivationRule : If (ME→PrecedentObjective(LifeCycle) == "CompletedWithSuccess")

Actions : GotoAction

SuccessRule : If (ME→LocationState(Position) == TargetPosition)

FailureRule : If (ME→Goto(ElapsedTime) > TimeToComplete)

Wait

Cet objectif est généré par la VSAction CreateWait. La règle d'activation vérifie si l'objectif précédent est complété avec succès avant d'activer l'objectif courant. C'est ainsi que la suite d'actions est produite dans le comportement de l'agent.

Objective : WaitPlane

ActivationRule : If (ME→PrecedentObjective(LifeCycle) == “CompletedWithSuccess”)

Actions : WaitPlane

SuccessRule : If (ME→PerceptionState(PerceivedAgent) == PlaneToWait)

FailureRule : NIL

Strike

Cet objectif est généré par la VSAction CreateStrike. La règle d’activation vérifie si l’objectif précédent est complété avec succès avant d’activer l’objectif courant. C’est ainsi que la suite d’actions est produite dans le comportement de l’agent. De plus la VSAction peut générer deux types de comportement dépendant du rôle que doit jouer l’agent : Leader ou Supporter.

Objective : *Strike-Leader*

ActivationRule : If (ME→PrecedentObjective(LifeCycle) == “CompletedWithSuccess”)

Actions :

- Sub-Objective : Goto
- Sub-Objective : SearchTarget&Destroy

  ActivationRule : If (ME→PerceptionState(PerceivedAgent) == Target) THAN Strike

  Actions : NIL

  SuccessRule : NIL

  FailureRule : NIL

SuccessRule : If (Target→AttributeState(HealthLevel) <= 0)

FailureRule : NIL

Objective : *Strike-Supporter*

ActivationRule : If (ME→PrecedentObjective(LifeCycle) == “CompletedWithSuccess”)

Actions :

- Sub-Objective : Goto
- Sub-Objective : Search&Destroy

ActivationRule : If (ME→AttributeState(Force) != PerceivedAgent(Force)) AND (PerceivedAgent(HealthLevel > 0) THAN Strike

Actions : NIL

SuccessRule : NIL

FailureRule : NIL

SuccessRule : If (Target→AttributeState(HealthLevel) <= 0)

FailureRule : NIL

7.3 Enregistrements de l'exemple d'application

Voici un échantillon d'enregistrements de la simulation de l'exemple d'application. Tous les enregistrements se trouvent dans le fichier « simulationLog.txt »

Type:Agent, Object:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833, Time:0,

Type:Objective, Object:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0,

Type:Action, Object:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0->Actions->GotoAction,

Type:Contain, Container:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0, Element:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0->Actions->GotoAction,

Type:Contain, Container:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833, Element:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0,

Type:Agent, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Time:0,

Type:Objective, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->CheckFuel,

Type:Action, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->CheckFuel->Actions->Wait,

Type:Objective, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->HealthManagement,

Type:Action, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->HealthManagement->Actions->Wait,

Type:Contain, Container:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Element:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->CheckFuel,

Type:Contain, Container:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Element:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->HealthManagement,

Type:Contain, Container:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->CheckFuel, Element:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->CheckFuel->Actions->Wait,

Type:Contain, Container:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->HealthManagement, Element:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->MyBehavior->HealthManagement->Actions->Wait,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184150130->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184151786->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184152755->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0->Actions->GotoAction, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->Radar4133833->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184150130->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184151786->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184152755->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->Goto0, Objective-State:Ongoing, Time:0.03,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411, Time:188.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380, Time:188.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380, Time:191.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Time:191.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411, Time:203.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Time:203.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184151786, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184150130, Time:204.957,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184150130, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184151786, Time:204.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->Goto0, Objective-State:CompletedWithSuccess, Time:205.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->WaitObj1, Objective-State:Active, Time:205.957,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->WaitObj1->Actions->WaitPlaneAction, Time:206.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->WaitObj1, Objective-State:Ongoing, Time:206.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->WaitObj1, Objective-State:CompletedWithSuccess, Time:206.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->StrikeObj2, Objective-State:Active, Time:206.957,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->StrikeObj2->Actions->GotoObj->Actions, Time:207.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->StrikeObj2, Objective-State:Ongoing, Time:207.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411, Event:ExitVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Time:207.957,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380, Event:ExitVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Time:207.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->StrikeObj2->Actions->GotoObj, Objective-State:Active, Time:207.957,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->StrikeObj2->Actions->GotoObj->Actions->GotoAction, Time:208.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->StrikeObj2->Actions->GotoObj, Objective-State:Ongoing, Time:208.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184151786, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Time:237.957,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184151786, Time:237.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->Goto0, Objective-State:CompletedWithSuccess, Time:239.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->WaitObj1, Objective-State:Active, Time:239.457,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->WaitObj1->Actions->WaitPlaneAction, Time:239.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->WaitObj1, Objective-State:Ongoing, Time:239.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->WaitObj1, Objective-State:CompletedWithSuccess, Time:240.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->StrikeObj2, Objective-State:Active, Time:240.457,

Type:Event, Event:ActionExecuted, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->StrikeObj2->Actions->GotoObj->Actions, Time:240.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->StrikeObj2, Objective-State:Ongoing, Time:240.957,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF184152755, Event:EnterVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF184150130, Time:241.457,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Event:ExitVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Time:598.957,

Type:Event, Actor:ViSim->ProjectManager->Simulation->AgentContainer->CF18414967, Event:ExitVisibilityZone, Object:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458, Time:598.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184150130->COABehavior->Goto1, Objective-State:CompletedWithSuccess, Time:726.957,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18421380->COABehavior->Goto3, Objective-State:CompletedWithSuccess, Time:852.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF184159411->COABehavior->Goto3, Objective-State:CompletedWithSuccess, Time:862.457,

Type:Objective-State, Objective:ViSim->ProjectManager->Simulation->AgentContainer->CF18422458->COABehavior->Goto3, Objective-State:CompletedWithSuccess, Time:904.957,

8 Annexe B

Voici les étapes utilisées pour spécifier l'exemple 2 à l'aide des indications de la section 3.2.4 :

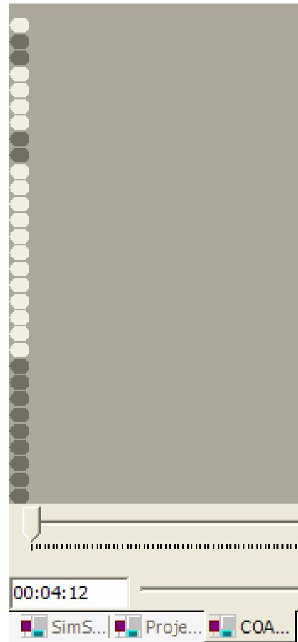
Placer une ressource

Pour placer une ressource dans l'environnement, il suffit d'appuyer sur le bouton correspondant à la ressource dans la barre d'outils. Ensuite, cliquer dans l'environnement à l'endroit approprié pour positionner la ressource.

Sélectionner une ou plusieurs ressources

Il y a deux méthodes pour sélectionner des ressources ou des ordres. La première consiste à appuyer sur le bouton de sélection dans la barre d'outils et ensuite de tracer, avec la souris, un rectangle de sélection dans l'environnement en appuyant sur le bouton de gauche. Tous les éléments dans ce rectangle seront sélectionnés mais attention, le rectangle n'apparaît pas à l'écran.

La deuxième technique consiste à utiliser la fenêtre temporelle à gauche nommée COATimeView. Cette fenêtre contient tous les éléments (ressources et ordres) qui sont dans l'environnement. Ces éléments sont représentés sous forme de gros points gris pour les ressources et blancs pour les ordres. Ils sont disposés selon leur temps de création dans la simulation. La figure suivante montre que tous les éléments présents dans le scénario seront créés au temps 0 dans la simulation (exemple 2). Pour sélectionner un élément, il suffit d'appuyer sur le point correspondant et il devient bleu. Lorsque que vous utiliserez la première méthode de sélection, le COATimeView se mettra automatiquement à jour. Cette fenêtre permet de sélectionner séparément les éléments se trouvant au même endroit géographique.



Affecter un ordre à une ressource

Pour affecter un ordre, il faut, premièrement, sélectionner le ou les éléments sur lesquels l'ordre devra être affecté. Ensuite, dans le cas d'un Goto ou d'un Striked cliquer dans l'environnement pour positionner l'ordre. Le clic de souris correspond aux coordonnées à atteindre. Dans le cas d'un Wait, le seul fait de choisir l'ordre à affecter dans la barre d'outils est suffisant. L'ordre s'appliquera sur l'élément courant : une ressource ou un ordre.

Pour construire une suite d'actions, il faut toujours sélectionner le dernier ordre assigné pour continuer la suite. C'est à ce moment que la fenêtre COATimeView peut être nécessaire car supposons que vous avez affecté un Goto à une ressource. Ensuite vous sélectionnez l'extrémité de la fléche pour sélectionner le goto et assignez un wait. Dans ce cas le Goto et le Wait seront aux mêmes emplacements géographiques car le wait s'appliquera à la fin du Goto. Donc si vous désirez poursuivre la suite d'action, vous devrez sélectionner seulement la dernière action (Wait). Pour ce faire, vous devrez utiliser la fenêtre COATimeView car si vous utilisez le bouton de sélection les deux ordres seront sélectionnés.

Modifier des paramètres à une ressource ou un ordre

Lors de la spécification du scénario, certains paramètres peuvent être modifiés concernant les ressources ou les ordres. Pour visualiser ou modifier ces paramètres, vous devez d'abord rafraîchir la liste des composants. Pour ce faire, double-cliquez sur le projet SimCOA dans la fenêtre *ProjectList*. Ensuite, allez dans la fenêtre *SimBrowser* pour avoir accès à tous les éléments d'architecture du simulateur. Le scénario se trouve sous ViSim→ProjectManager→Scenario. Pour visualiser les paramètres d'un élément, cliquez sur l'élément et allez dans la fenêtre *ComponentsProperties*. Cette fenêtre vous permet aussi de modifier des paramètres en changeant la valeur et en appuyant sur entrée.

Étapes pour construire l'exemple 2 :

1- Placer le radar ennemi et affecter lui un Goto.
2- Placer les avions ennemis et affecter leur une suite d'actions pour qu'ils se déplacent et protège le radar.
3- Placer les avions amis, sélectionner les tous et affecter un Goto.
4- Ensuite sélectionner les Goto et affecter un Wait.
5- Sélectionner les Wait à l'aide du COATimeView et affecter un Strike sur le radar ennemi
6- Modifier l'attribut Role de deux avions sur trois dans ViSim→ProjectManager→Scénario→IfTimeGreaterThan0→AssignStrike... →AttributeState. Par défaut le rôle, lors d'une attaque, est « Leader » mais vous pouvez mettre « Supporter » pour 2 des 3 avions.
7- Sélectionnez les ordres Strike et assignez un Goto pour le retour lorsque l'attaque sera terminée.

Références

[MAGS, 2003] Bernard Moulin, Walid Chaker, Jimmy Perron, Patrick Pelletier, Jimmy Hogan, Edouard Gbei: *MAGS Project: Multi-agent GeoSimulation and Crowd Simulation*. COSIT 2003: 151-168

[Vilnat, 2005] Anne Vilnat, « Dialogue et analyse de phrases », mémoire d'Habilitation à diriger des recherches, Université Paris-Sud 11, 2005.

[SdkSim, 2006]
NSim Technology, *Documentation du framework SdkSim*, 2006.