FRANÇOIS BERNIER

APIA : ARCHITECTURE PROPRIÉTÉ INTERACTION ACTEUR POUR DES MONDES VIRTUELS AGILES

Thèse présentée
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de doctorat en génie électrique
pour l'obtention du grade de Philosophiae Doctor (Ph.D.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

MAI 2008

Résumé

Une majorité de mondes virtuels (MVs) existants mettent l'accent sur la qualité graphique, les capacités réseaux ou sur la qualité de la modélisation. Toutefois, ces MVs manquent souvent d'*agilité*, tant au niveau de leur conception que de leur exécution. Il en résulte des MVs monolithiques avec peu de réutilisabilité. Cette thèse propose une architecture appelée "Architecture Propriété Interaction Acteur" (**APIA**) qui a pour objectif d'améliorer l'*agilité* des MVs en se concentrant sur les caractéristiques sous-jacentes : la composabilité, la réutilisabilité, la modifiabilité, l'extensibilité et une grande liberté d'action.

La solution proposée consiste en une architecture générique basée sur un méta-modèle conceptuel comprenant cinq éléments de base, supportés par un processus et des algorithmes de gestion. Comparativement aux autres approches qui sont centrées sur l'entité, le méta-modèle conceptuel propose un paradigme centré sur l'interaction entre les entités. Ce paradigme permet de regrouper les protocoles entre les entités et les aspects de l'interaction dans un élément du MV nommé interaction. Plusieurs exemples, dont des applications de cryochirurgie et d'inspection de barrages, viennent démontrer que les entités et les interactions sont moins interdépendantes avec cette approche. De plus, des exemples montrent que l'émergence de comportements est facilitée due à la meilleure composabilité de cette approche. Ensuite, le processus développé avec **APIA** définit trois groupes d'actions qui influencent à différents niveaux l'*agilité* future du MV. Finalement, **APIA** définit des algorithmes de gestion de ces actions afin de maintenir le MV cohérent tout au long des modifications, des extensions ou des compositions en cours d'exécution. Les gestionnaires qui appliquent ces algorithmes maintiennent automatiquement la cohérent du MV lors d'opérations de modification, d'extension et de composition. Ces caractéristiques sont désormais possibles en cours d'exécution sans l'intervention humaine.

Abstract

Most existing Virtual Worlds (VWs) emphasize on graphic quality, networking capacities or modeling quality. However, these VWs often lack of *agility*, both from a design and an execution point of view. It then results in monolithic VWs with little reusability. This thesis proposes an architecture called Actor Property Interaction Architecture (**APIA**) which aims at improving the *agility* of MVs by tackling its inherent characteristics: composability, reusability, modifiability, extensibility and a high degree of freedom.

Firstly, the proposed solution consists of a generic architecture based on a conceptual meta-model defining five basic elements, supported by a process and management algorithms. While other approaches are centered on the entity, the conceptual meta-model proposes an interaction-centric approach. This paradigm gathers the inter entities protocols and the interaction aspects of a VW in an element called interaction. Many examples, including cryosurgery and dam inspection applications, show that entities and interactions are less interdependent with this approach. These examples also show emergent behaviours due to a high-level of composition. Moreover, **APIA** defines a process based on three groups of actions influencing at various levels the future *agility* of a VW. Finally, **APIA** defines management algorithms for these actions to maintain the consistency of the VW throughout run-time modifications, extensions or compositions. By using these algorithms, a manager ensures VW consistency during modification, extension and composition. Since human intervention is no longer required, these characteristics are becoming available at run-time.

Avant-propos

Les travaux de recherche présentés dans cette thèse ont été réalisés au Laboratoire de vision et systèmes numériques (LVSN) du Département de génie électrique et de génie informatique de l'Université Laval de septembre 1999 à avril 2007. Les recherches ont été effectuées en collaboration avec l'Institut de recherche d'Hydro-Québec (IREQ) et l'Université de Victoria.

Je tiens d'abord à remercier Denis Poussart qui, par sa passion et sa vision, m'a fait prendre conscience du potentiel et m'a transmis la passion des environnements virtuels. Beaucoup d'efforts ont été nécessaires pour produire **APIA** ainsi que tous les modules et projets qui s'y sont connectés et continuent de le faire. Ceci n'aurait pu être réalisé sans de nombreuses personnes avec qui j'ai eu la chance de travailler : Éric, Martin & Martin, Hugues, les trois François, Denis, Brad, Frédéric, Alexandre, Richard, Dominique, Charles, Marie-Ève, Jean-François & Christophe. Je veux tous les remercier de leur persévérance et de leur patience pour avoir travaillé avec une architecture **APIA** en constante évolution.

Je tiens également à remercier les professionnels du LVSN, Sylvain Comtois et Denis Ouellet, qui m'ont apportés leur soutien. J'aimerais aussi mentionner les encouragements et le support de Denis Laurendeau dans les publications. Merci également à Nathalie Harrison pour son apport au projet et son soutien moral.

Enfin, je dois souligner le soutien financier du Conseil de recherche en sciences naturelles et en génie (CRSNG) et du Fonds québécois de la recherche sur la nature (FCAR) pour le financement de mes travaux de doctorat ainsi que celui de l'Institut pour la robotique et l'intelligence des systèmes (IRIS) pour le financement du projet VERTEX et de la Fondation canadienne pour l'innovation (FCI) pour le financement des infrastructures de réalité virtuelle.

Table des matières

Liste des tableaux

Liste des figures

Chapitre 1

Introduction

La simulation de la réalité fut longtemps un sujet philosophique décrit dans des oeuvres telles que *Simulacres et simulation* (Baudrillard, 1981) ou réservé à la science fiction avec *Burning Chrome* (Gibson, 1982), *Neuromancer* (Gibson, 1984) et *Ender's Game* (Card, 1985). L'arrivée des ordinateurs puis celle des technologies de visualisation tridimensionnelle (3D), d'une puissance de calcul adéquate et d'une meilleure connectivité a permis l'émergence des simulations numériques et de la réalité virtuelle. Les composantes de base devenaient désormais disponibles afin de recréer des mondes artificiels ou, plus spécifiquement, des mondes virtuels (MVs). Ces MVs contribuent aujourd'hui de différentes façons à de nombreux domaines d'application. Par exemple, ils peuvent contribuer au succès des interventions de nature critique telles que les inspections de barrage hydro-électriques par un sous-marin téléopéré ou le traitement du cancer par la cryochirurgie. Étant donné que ces interventions impliquent des manipulations complexes et délicates, les MVs constituent des environnements idéaux pour effectuer la planification, l'entraînement et l'assistance à l'exécution de ces manipulations. Les MVs s'emploient également dans le domaine du divertissement qui recrée des représentations du monde réel ou de mondes fantaisistes.

1.1 Problématique

La conception et l'évolution des MVs posent actuellement de nombreux défis. Dès le début de leur développement, on a cherché à construire le plus rapidement possible et à moindre coût des MVs performants et réalistes. Les solutions apportées s'inspirent des récents progrès du domaine logiciel : développer des bases génériques, modulaires et flexibles ;

réutiliser les composantes ; fournir des outils spécialisés ; créer des logiciels d'intégration ; etc. Elles visent à améliorer la composabilité, l'extensibilité, la réutilisabilité et la modifiabilité. De façon similaire, ces caractéristiques devraient se retrouver dans les MVs. La conception et l'évolution s'effectuent en plus dans un contexte où le changement accéléré de la société actuelle et sa complexification (Poussart, 2006) rendent les prévisions et la planification difficiles pour les entreprises. Les MVs, développés par des entreprises et répondant à des besoins du monde réel, doivent alors être constamment et rapidement modifiés à des coûts parfois importants.

Afin de s'adapter à cette nouvelle réalité, les MVs devraient posséder une capacité rapide d'adaptation aux changements afin de répondre à des besoins difficilement prévisibles. Ce concept se nomme *agilité* (Levine, 2005). Quoique certains MVs possèdent une certaine *agilité* avant leur exécution, elle est peu présente pendant leur exécution, ce qui constitue une lacune critique.

Par exemple, un MV peut être conçu dans le but d'entraîner des opérateurs de sous-marins téléopérés qui inspectent de conduites d'eau de barrages. Suite à un bris imprévisible dans une conduite du barrage, l'eau se brouille et la visibilité des caméras embarquées, utilisées pour guider le sous-marin, devient fort réduite, ce qui empêche l'intervention. Il devient donc critique de compenser cette limitation en convertissant rapidement le MV d'entraînement en un MV d'assistance à l'opérateur qui reçoit ainsi une prédiction du comportement du sous-marin lors de l'intervention. Un MV véritablement *agile* permettrait cette conversion rapide.

Un autre défi est lié à la nécessité de faire vivre une expérience virtuelle riche et réaliste aux usagers. L'interconnexion sensori-motrice de l'humain avec les MVs, supportée par la réalité virtuelle et ses technologies, permet l'immersion et la présence (Heeter, 1992) qui améliorent cette expérience. Cette interconnexion, quoique très importante, n'est pas la seule à influencer la qualité de l'expérience virtuelle. La façon dont le MV est construit peut également y contribuer. Comme illustré à la figure 1.1, un MV doit pouvoir supporter des changements de comportements qui, bien que simples en apparence, posent néanmoins d'importants défis au niveau de leur représentation formelle. Lors de l'implantation par des comportements physiques, le degré de fidélité, défini comme l'exactitude de la représentation lorsque comparée au monde réel (US DoD, 1995b), se doit d'être assez élevé pour obtenir un MV réaliste.

Toutefois, l'exactitude mathématique de la représentation du comportement de combustion n'est pas une condition suffisante pour obtenir une expérience virtuelle riche et réaliste. Pour poursuivre avec l'exemple de la figure 1.1, il serait intéressant que cette branche puisse être empilée avec d'autres branches afin de créer un feu de camp. Cette possibilité dépend également de l'aspect générique des comportements, de l'interaction possible entre entités, de la capacité de permettre une utilisation non prévue à l'origine ou de la capacité d'introduire dynamiquement des comportements.

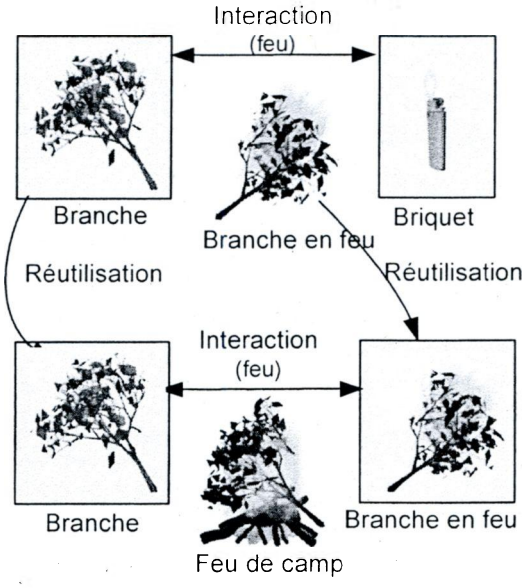

Figure 1.1 : Exemple d'un MV *agile* où une branche qui peut prendre feu est réutilisée afin de créer un feu de camp.

La figure 1.2 illustre une analogie d'*agilité* d'un MV avec un ensemble de pièces LEGO^{MD}. Ce MV peut déjà être constitué d'éléments de base et offrir la possibilité de composer ces éléments afin de créer de nouveaux éléments. Cette capacité de composer de nouveaux objets à partir de ceux existants ou de réutiliser ceux-ci repose également sur une grande *agilité*. Les architectures sur lesquelles les MVs conventionnels sont basés ne mettent pas suffisamment l'accent sur ce type d'*agilité*, surtout en cours d'exécution.

Un MV *agile* possède donc de nombreux avantages, tant pour faciliter son développement et son évolution que pour offrir une expérience virtuelle intéressante. Cette

agilité a toutefois une contrepartie. Lorsqu'un MV peut être modifié, étendu, composé ou réutilisé en cours d'exécution, il faut qu'il demeure *cohérent* (Vaghi *et al.*, 1999), c'est-à-dire qu'il se comporte comme un tout de façon adéquate et prévisible.

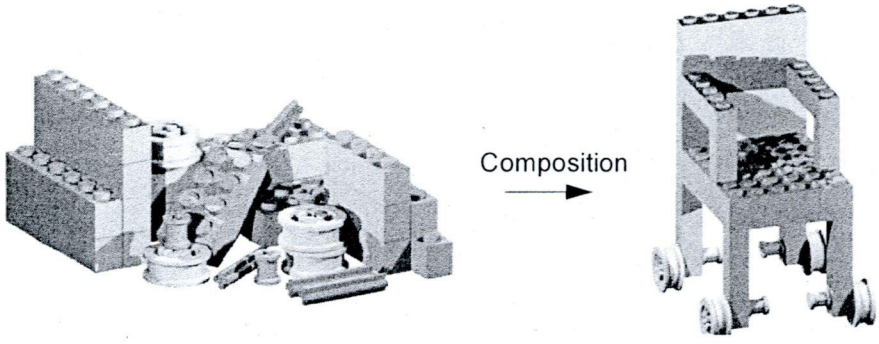

Figure 1.2 : Exemple de composabilité (par réassemblage d'éléments déjà existants) des LEGO^{MD} pouvant être transposé aux MVs.

1.2 Solution proposée

Cette thèse propose une architecture, dénommée Architecture Propriété Interaction Acteur (**APIA**) pour concevoir des MVs *agiles*, ayant un plus haut niveau de réutilisabilité, d'extensibilité, de modifiabilité et de composabilité ainsi qu'une plus grande liberté d'action que les approches actuelles. La spécification de ce que sont des MVs *agiles* constitue une contribution de cette thèse. Toutefois, l'innovation principale de cette architecture est d'être centrée sur l'*interaction* et de gérer les règles d'interaction en cours d'exécution. Il sera démontré que, lorsqu'utilisée avec les bons éléments et un processus adéquat, l'interaction accroît l'*agilité*. Étant donné que ces MVs se modifient en cours même d'exécution, **APIA** gère continuellement les conséquences de ces changements. **APIA** fournit un encadrement de conception et d'exécution de MVs basés sur les modèles conceptuels. **APIA** est constitué d'un méta-modèle conceptuel, d'un processus et de gestionnaires. Une implantation, décrite en annexe, a permis de valider l'architecture sur des applications de cryochirurgie et d'inspection de barrages. D'autres exemples démontrent l'*agilité* résultante de cette architecture.

1.3 Structure de la thèse

Cette thèse débute, dans le chapitre 2, par une présentation détaillée des caractéristiques requises pour obtenir des MVs *agiles*. On y formule ainsi les objectifs généraux de cette thèse. Le chapitre 3 analyse les architectures de MVs existants en regard des objectifs prédéfinis et identifie les composantes d'une nouvelle architecture nécessaires pour en améliorer l'*agilité*. Il présente les lacunes des MVs existants et propose des réflexions et suggestions pour y palier. Le chapitre 4 présente **APIA** et donne des exemples qui viennent en expliquer le fonctionnement. Le chapitre 5 démontre l'*agilité* d'**APIA** avec des exemples dont un de fine granularité représentant un câble en interaction avec son environnement et un autre pour la cryochirurgie. Le chapitre 6 revisite et analyse les principales caractéristiques d'**APIA**. Enfin, le chapitre 7 conclut avec les contributions spécifiques de ce travail.

Chapitre 2

Caractéristiques de l'agilité des Mondes Virtuels (MVs)

La conception et l'évolution des MVs ainsi que l'expérience virtuelle vécue lors de leur utilisation peuvent être améliorées en rendant les MVs plus *agiles*. Le chapitre précédent a brièvement présenté ce concept. Une définition plus complète est requise pour expliquer l'*agilité* afin de proposer une architecture visant son amélioration. Dans le domaine du génie logiciel, Levine (2005) définit l'*agilité* sur la base de trois qualités. La première est la rapidité de mise en oeuvre. La seconde est la capacité d'improviser et d'utiliser des patrons pour concevoir dynamiquement et avec créativité de nouvelles solutions et d'être ainsi flexible. La dernière est l'adaptabilité, soit la capacité de répondre de façon dynamique et interactive à un changement. Ces qualités donnent quelques pistes de solutions telles que le dynamisme, l'interactivité, la rapidité, la flexibilité et l'adaptabilité. La littérature (Saleh *et al.*, 2001) utilise le terme flexibilité pour définir un concept similaire à l'*agilité*. Toutefois, le terme *agilité* a été préféré car il contient, selon Fricke *et al.* (2000), une dimension rapidité qui s'avère importante pour les MVs. Le terme *dynamique*, souvent rencontré dans la littérature, n'est pas suffisamment explicite pour qualifier les MVs. L'expression *en cours d'exécution* sera plutôt employé. Une définition de l'agilité d'un MV peut être maintenant posée.

> L'*agilité* d'un MV se définit comme sa capacité d'adaptation rapide à des besoins nouveaux et difficilement prévisibles de ses créateurs et de ses usagers.

Cette thèse repose sur l'hypothèse que l'amélioration d'un ensemble de caractéristiques

accroît l'*agilité* des MVs. Les caractéristiques retenues sont la *composabilité*, la *réutilisabilité*, la *modifiabilité* et l'*extensibilité* des MVs ainsi que la *liberté d'action* des usagers et scénaristes. Ce choix est motivé par leur effet direct sur l'*agilité* comparativement à d'autres caractéristiques, non retenues, qui y contribuent indirectement.

Ces caractéristiques ont été définies par plusieurs auteurs. Afin de lever toute ambiguïté sur ces termes, les prochaines sections les décrivent dans une nouvelle classification supportée par des définitions déjà établies. Cette classification servira à analyser les approches existantes et à évaluer la solution proposée. Bien que ces caractéristiques soient présentées séparément, elles dépendent les unes des autres et s'influencent mutuellement. Ces dépendances seront soulignées dans les prochaines sections. D'autres caractéristiques tels que l'aspect générique d'un MV, l'interopérabilité (Capps *et al.*, 1996; Soto & Allongue, 2002) et l'adaptabilité se retrouveront à l'intérieur des caractéristiques retenues. Finalement, dans les prochaines sections et chapitres, les caractéristiques de l'*agilité* seront présentées d'un point de vue de "novice" dans l'implémentation et la modélisation des MVs.

Ce chapitre présente donc des exemples de situations qui peuvent à première vue sembler simplistes. Celles-ci sont effectivement "simples" mais néanmoins assez riches de sens pour pouvoir illustrer les concepts fondamentaux qui sont exploités ici. Des observations, importantes pour la suite, accompagneront ces exemples.

2.1 Définitions

Certains termes doivent être définis avant de poursuivre avec une présentation détaillée des caractéristiques et des objectifs généraux de cette thèse.

2.1.1 Description, déroulement et modèles

Le *déroulement* se définit comme le résultat de l'exécution d'un MV sur une période de temps donné. Ce déroulement, qui équivaut à l'histoire dans les structures narratives, est contrôlé par une certaine *description*. Celle-ci correspond à une représentation approximative du monde réel ou à un monde fantaisiste. Les scénarios et les modèles jouent ce rôle. Pour des raisons liées à la solution, le terme modèle a été retenu dans cette thèse pour désigner tout ce qui influence le déroulement. Un modèle est défini comme une représentation physique,

mathématique ou logique d'un système, d'une entité, d'un phénomène ou d'un processus (US DoD, 1995a). Les modèles sont valides dans un contexte et pour certaines conditions. La figure 2.1 illustre que, conceptuellement, le déroulement d'un MV est contrôlé par sa description alors que, d'un point de vue implantation, les modèles en contrôlent l'exécution.

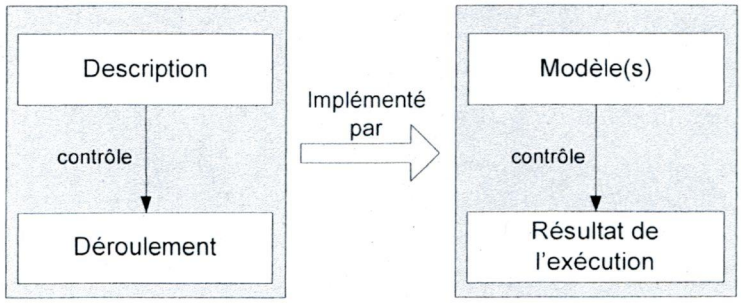

Figure 2.1 : Représentation de la description d'un MV qui en contrôle le déroulement et correspondance avec les modèles qui contrôlent le résultat de l'exécution

2.1.2 Usager et scénariste

Deux types d'intervenants peuvent influencer le déroulement : l'*usager* et le *scénariste*. Le premier, l'*usager*, pose des actions ou réagit dans les limites définies du déroulement du MV. Il peut être réel ou virtuel. La personne humaine est un usager qui peut intervenir par l'intermédiaire d'avatars. Dans certains cas comme un MV reproduisant une cryochirurgie, l'usager est un chirurgien qui manipule une sonde cryogénique. Dans d'autres cas, il est un robot intelligent, basé sur des agents logiciels, qui évolue dans le MV.

Le second intervenant, le scénariste, peut agir au niveau de tous les éléments constituants le MV. Il peut être réel ou virtuel. De plus, il est moins contraint par le MV. Alors qu'un usager tend la corde d'un arc pour en augmenter la portée, un scénariste peut changer directement l'attribut de tension de la corde. Le terme scénariste englobe tous les rôles liés à la conception des MVs : concepteur de modèles, concepteur d'entités, concepteur de la scène, etc. Il est à noter qu'une même personne, désignée usager-scénariste, peut jouer les deux rôles simultanément.

2.1.3 Éléments du MV

Les MVs reposent généralement sur les mêmes éléments fondamentaux. Ces éléments seront employés tout au long de cette thèse et sont définis au tableau 2.1.

Élément	Définition
Entité	Le terme entité est synonyme d'objet ou de tout ce qui peut être circonscrit dans l'espace cartésien ou dans tout autre espace.
Attribut	Le terme attribut est emprunté de l'orienté objet (OO) et est employé comme un quantitatif ou un qualitatif de l'entité.
Comportement	Le terme comportement représente tout ce qui modifie les attributs et anime les MVs. Il provient de la traduction du mot anglais *behavior*. Le terme comportement est une généralisation des termes méthode, fonction et partie fonctionnelle d'un modèle.

Tableau 2.1 : Définition des éléments intrinsèques des MVs.

2.2 Composabilité

Tel que le mentionnent Capps, Watsen et Zyda (1999), la *composabilité* est un objectif important à atteindre dans les MVs. La définition suivante provient de Pratt, Ragusa et von der Lippe (1999).

> La *composabilité* se définit comme la possibilité de concevoir de nouveaux éléments à partir de ceux déjà existants.

La modularité désigne un concept similaire. La figure 1.2, présentée précédemment, illustre une analogie avec un exemple de Lego™ permettant la composition d'une entité à partir de pièces élémentaires. Les avantages de la conception par pièces sont nombreux. Premièrement, la réutilisation de l'entité composée est supérieure car l'entité peut être désassemblée et les pièces peuvent servir à d'autres fins. Ensuite, lorsque les composantes interagissent déjà avec leur environnement, l'entité composée possède certaines capacités

d'interaction de ses constituants. L'exemple qui suit montre un avantage de la composition.

> Un MV contient des pièces possèdant la capacité d'entrer en collision avec l'environnement. Une composition efficace permettra au véhicule conçu avec celles-ci d'hériter de cette capacité.

Finalement, des comportements ou propriétés émergents peuvent résulter de la composition de blocs élémentaires. Ces comportements émergents peuvent contribuer à l'*agilité* car ils étendent les MVs avec de nouveaux comportements qui se manifestent en cours d'exécution. L'exemple suivant illustre ce concept.

> La composition d'une séquence de petits cylindres attachés les uns aux autres peut résulter en un fouet qui possède la capacité de créer un son de claquement dû au dépassement de la vitesse du son (Bernstein *et al.*, 1958) et ce, pour une impulsion minimale, ce qui ne peut être atteint par aucun des cylindres le composant lorsqu'ils sont soumis individuellement à une force équivalente.

Dans l'exemple du Lego™, la création d'entité est rendue possible avec l'existence de pièces élémentaires (éléments) qui respectent des normes. Ces normes tiennent compte de la granularité et de la spécialisation. Selon Szyperski (2002), la granularité est un facteur prépondérant dans la réutilisation et l'interopérabilité des composantes logicielles. Il semble en être de même pour les MVs et la simulation (Davis & Anderson, 2004). Un autre critère important consiste à permettre l'interaction des pièces. Avec les blocs Lego™, cette interaction est garantie par un standard qui définit le multiple de la dimension des pièces ainsi que la taille et l'espacement des connecteurs. Similairement, l'utilisation de standards, qui viennent contrôler la conception de MVs, peut également améliorer la composabilité. Ce contrôle de la conception sera traité au chapitre 3 avec les méta-niveaux.

Aucun sous type n'est proposé pour la composabilité. Quoique Petty & Weisel (2003) en proposent un lexique, il ne permet pas d'établir des types utiles à cette thèse.

2.3 Réutilisabilité

Un MV peut être conçu ou étendu sur la base de ceux existants. Il s'agit du concept de

réutilisabilité dont la définition qui suit s'inspire du domaine logiciel (Reese & Wyatt, 1987).

> La *réutilisabilité* se définit comme la capacité d'isoler, de maintenir et d'utiliser les éléments existants dans le développement de nouveaux éléments.

La réutilisation s'applique tant au niveau de l'usager qui réutilise les éléments qu'au niveau du scénariste qui copie des parties du scénario. Des éléments peuvent être réutilisés pour créer, modifier ou étendre un MV, en cours d'exécution ou non. Toutefois, comme les exemples de cette section le montreront, la réutilisation en cours d'exécution offre de nombreux avantages dont une plus grande liberté d'action à l'usager. Puisque la possibilité de réutiliser un ou plusieurs éléments d'un MV pendant l'exécution implique une réutilisation avant l'exécution, la réutilisation en cours d'exécution sera principalement considérée dans cette thèse. Il en sera de même pour les autres caractéristiques.

La réutilisation peut prendre ici plusieurs formes (Pidd, 2002; Paul & Taylor, 2002) dont certaines seront reprises dans les prochaines sections qui en décrivent différentes catégories.

2.3.1 Réutilisation par contexte

La réutilisation peut s'effectuer dans un même contexte ou dans un contexte différent que celui prévu. Le premier cas, illustré dans l'exemple qui suit, survient lorsqu'une entité est réutilisée pour une même tâche mais avec des entités similaires.

> Un MV contient un arc pouvant lancer une flèche en interagissant avec elle, tel qu'illustré à la figure 2.2 a). Toutefois, est-il possible de lancer, avec un arc, un bout de branche taillée comme illustré à la figure 2.2 b) ?
>
> La réponse à cette question dépend de la capacité de la branche taillée à se substituer à la flèche ou à l'arc d'accepter des projectiles similaires à des flèches. D'une façon ou d'une autre, il s'agit d'un problème d'interaction. Le terme interopérabilité est parfois utilisé pour un concept similaire. Une généralisation de cet exemple permet d'affirmer que la réutilisabilité dépend de la généricité des interactions entre les entités ou de la versatilité des entités.

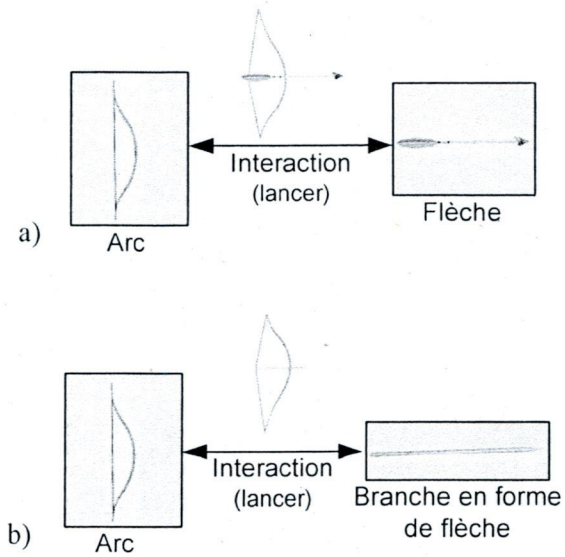

Figure 2.2 : Réutilisation d'un arc dans un même contexte : a) avec une flèche tel qu'il a été conçu b) avec une branche en forme de flèche.

La généricité des interactions est importante pour que les entités puissent interagir. L'interaction peut être conçue comme prenant place entre un arc Robin modèle #1242 et une flèche Stilkon modèle #2342; elle peut l'être aussi comme se produisant entre un corps élastique et n'importe quel corps possédant une masse et un coefficient d'aérodynamisme. Ce second type s'avère nettement préférable car il offre davantage de possibilités de réutilisation.

De plus, la versatilité des entités permet aussi d'augmenter la réutilisation en augmentant la capacité d'interaction. D'un côté, un bout de bois peut se substituer à une flèche sans que l'arc ne voit de différence. De l'autre, l'arc peut s'adapter pour lancer n'importe quel projectile.

La réutilisation est également possible dans un contexte différent. Ce cas, illustré par l'exemple suivant, permet à une entité et ses attributs d'être réutilisés dans une fonction totalement différente.

La figure 2.3 illustre un MV avec un briquet dans lequel l'arc précédent est importé. Si l'arc est constitué de bois, il possède les caractéristiques requises pour brûler. Cette réutilisation implique une grande versatilité de la part des entités ou une grande généricité de l'interaction qui met le feu.

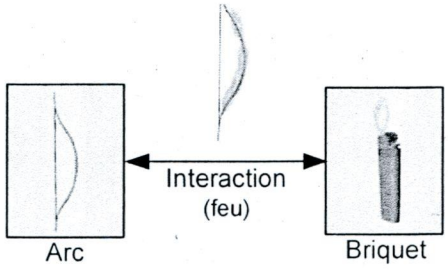

Figure 2.3 : Réutilisation d'un arc dans un contexte différent où un briquet peut y mettre le feu.

La réutilisation dans plusieurs contextes est très utile lorsqu'un usager exploite une entité dans un MV et veut l'amener dans un autre. Ce type de réutilisation peut être qualifié d'externe en comparaison avec la réutilisation interne qui exploite une partie d'un ou plusieurs éléments dans un même MV. La réutilisation externe s'avère difficile lorsque les éléments exportés et le MV hôte sont basés sur différents types de modélisations. Une modification de l'une des deux modélisations sera requise, opération d'autant plus difficile lorsqu'effectuée en cours d'exécution.

2.3.2 Réutilisation de différents types d'éléments

Il existe autant de types d'éléments que de modélisations possibles. Tel que l'illustre le prochain exemple, un plus grand choix de types d'éléments offre davantage d'opportunités de réutilisation.

> Un MV pour la simulation de combat militaire contient un ensemble configurable d'entités de combat, d'attributs pour le carburant et les munitions, de relations pour les formations et d'événements pour les engagements. Un autre MV ne contient que des entités de combat et des évènements pour les engagements et les formations. Le choix des éléments est différent entre ces MVs parce qu'ils sont basés sur des approches de modélisation distinctes. Cependant, la réutilisation serait probablement supérieure dans le premier cas à cause du plus grand nombre de type d'éléments.

2.3.3 Réutilisation de niveaux de granularité différents

Le type de modélisation employé peut limiter la granularité du MV et, par conséquent, la réutilisabilité. Une granularité trop fine complexifie la réutilisation alors qu'une granularité

grossière, tel qu'illustré dans l'exemple qui suit, empêche la réutilisation de sous éléments.

> Comme premier exemple, un MV représente des flèches sous forme d'attributs de l'entité *Arc*. L'encapsulation augmente ainsi la taille de la granularité. Il ne serait pas possible de prendre une flèche pour y mettre le feu indépendamment car le modèle qui met le feu n'agit, par exemple, que sur des entités.
>
> Dans un second exemple, illustré à la figure 2.4, une table à roulette provient de l'assemblage d'une table et de roulettes prélevées sur une chaise à roulettes. Cette réutilisation requiert la possibilité de prélever une partie d'une entité sur une autre entité. Ce désassemblage de la chaise à roulettes et la composition d'une table implique une granularité fine des entités. La réutilisation des roulettes dépend également de la capacité d'interaction qui permet de connecter les roulettes à la table.

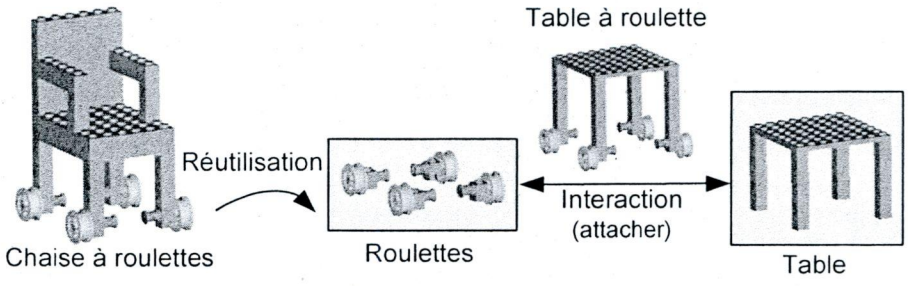

Figure 2.4 : Création d'une table à roulettes rendue possible grâce à la réutilisation d'une partie d'entité (les roulettes) et à l'interaction des roulettes et de la table.

2.3.4 Réutilisation par enchaînement

La réutilisation par enchaînement permet de créer des situations non prévues à l'origine, facilitant ainsi la conception de MVs et augmentant la qualité de l'expérience virtuelle. Cette succession de réutilisations, lorsqu'appliquée à grande échelle, peut augmenter considérablement les possibilités à l'intérieur d'un MV. De cette manière, l'usager façonne le MV à ses besoins au lieu de constamment dépendre d'extensions par les scénaristes. L'exemple qui suit en illustre un cas.

> La figure 2.5 montre une succession de réutilisations. Ici, un usager extrait initialement une branche d'un arbre pour y mettre le feu avec un briquet. Par la suite, l'usager utilise cette même branche pour mettre le feu à une autre branche dans le but de créer un feu de camp.

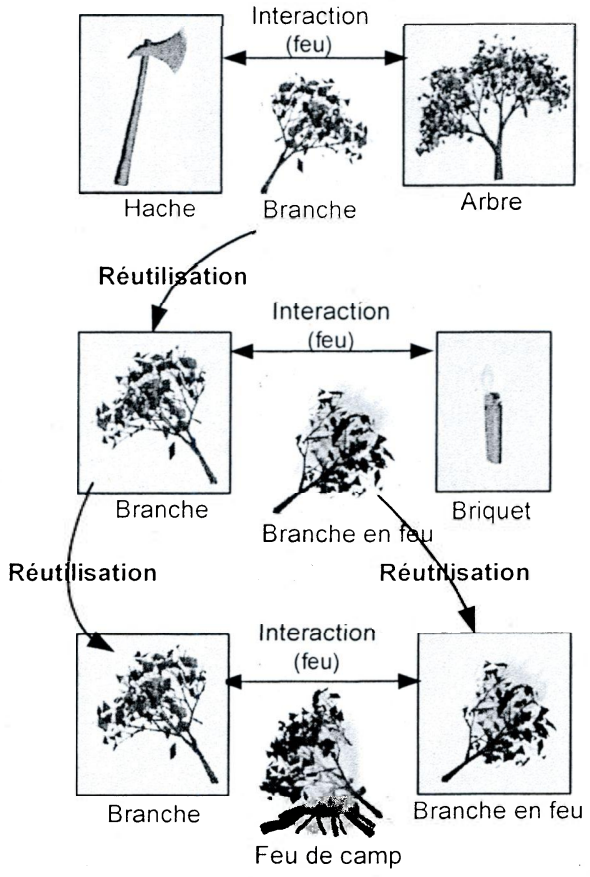

Figure 2.5 : Exemple d'un MV supportant la réutilisation par enchaînement.

2.3.5 Réutilisation en tant qu'usager, scénariste et usager-scénariste

La réutilisation peut concerner l'usager, ou encore le scénariste. Un scénariste-usager transcende le MV dans lequel il évolue, lui permettant d'effectuer des opérations comme prendre des propriétés d'une entité pour les appliquer sur une autre, ce qui demande une certaine connaissance du fonctionnement interne du MV. Un scénariste peut réutiliser tous les éléments en se situant à l'extérieur du MV, comme par exemple, changer les conditions de turbulence dans l'eau pour évaluer le niveau d'entraînement d'un opérateur de sous-marin.

La réutilisation usager est essentielle mais celles en tant que scénariste et scénariste-usager offrent des possibilités additionnelles. Le prochain exemple présente les avantages pour un usager-scénariste.

Un usager-scénariste évoluant dans un MV aperçoit deux entités. La première possède une masse mais non la seconde. Il peut donc se servir de la première pour tenir un poids mais cette utilisation s'avère impossible avec la seconde. Parce qu'il perçoit la géométrie des deux entités comme identique, il décide de copier la masse de la première sur la seconde selon une méthode prédéterminée. Quoique cette opération copier-coller semble relativement facile pour un attribut, elle s'avère beaucoup plus difficile lorsqu'il s'agit de comportements. Une telle opération n'est réalisable que par le scénariste ou des spécialistes du comportement en question. Il serait néanmoins possible de concevoir des MVs où les usagers eux mêmes peuvent réutiliser les comportements, surtout de façon implicite comme dans le cas de la figure 2.4.

2.4 Modifiabilité

La modifiabilité est à la base du déroulement et recoupe deux autres caractéristiques, la composabilité et l'extensibilité.

La *modifiabilité* se définit comme la capacité de changer les éléments du MV et leur arrangement.

Comme pour la réutilisation, un MV ne peut être qualifié simplement de "modifiable" car il existe de nombreux types de modifications qui seront présentées dans les prochaines sous-sections. La composition, un cas particulier de modification, a déjà été traitée. L'extension, un autre cas particulier de modification, sera présentée à la section 2.5.

2.4.1 Nature des modifications

La forme la plus simple de modification consiste à changer la valeur des attributs. Cette forme est décrite dans le prochain exemple.

Un arc est déplacé dans une pièce froide. La tension de la corde augmente avec la diminution de température et comme la température y est inférieure, il en résulte une tension de l'arc plus élevée. Cette modification de la température permet à un archer de lancer des flèches plus loin.

Une modification de structure représente un cas nettement plus délicat. Diverses méthodes peuvent être invoquées à cette fin. La première, montrée dans l'exemple qui suit, est désignée comme une modification de la nature des éléments.

Un arc est modifié en canne à pêche, tel qu'illustré à la figure 2.6. Si cet arc est défini en tant que classe comme en OO, sa modification impliquera le changement de la classe *Arc* en classe *CanneaPeche*.

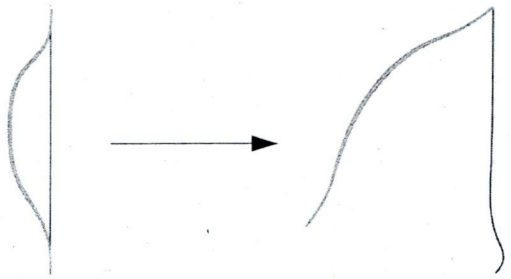

Figure 2.6 : Exemple de modification d'un arc en canne à pêche.

Ce type de modification peut s'avérer difficile car la transformation de la nature d'une entité implique une connaissance intrinsèque de celle-ci. Cette difficulté est à la base de l'OO. En effet, l'OO mise beaucoup sur le fait que la classe constitue une unité très cohérente qui réunit des attributs et des méthodes fortement couplés. La maintenance et la réutilisation en sont facilitées. Toutefois, sa modification peut être difficile et s'effectue généralement par l'héritage. Ce mécanisme d'extensibilité sera traité dans la section suivante.

Une autre méthode consiste en la modification fonctionnelle des entités. Lorsque les modèles qui agissent entre les entités et la granularité le permettent, ce type de modification s'avère plus facile, comme le montre l'exemple suivant.

L'arc de la figure 2.6 peut être perçu par l'usager mais peut ne pas exister en tant qu'entité unique. Il existerait sur la base de la combinaison d'une branche flexible et d'une corde extensible. Par conséquent, détacher la corde d'un côté de l'arc permettrait de modifier une entité arc en entité canne à pêche.

Une dernière forme de modification peut s'exercer par le changement des liens entre les éléments d'un MV.

Un usager d'un MV prend une corde et une branche pour créer un arc. Dans la forme précédente, l'entité nouvellement créée n'existait qu'en tant qu'agglomération de deux entités mais ici elle existe comme deux entités liées par une relation spécifique. Un MV possédant des comportements sensibles aux combinaisons d'entités (bois et corde) activerait alors celui pour lancer des entités (flèches).

2.4.2 Origine de la modification

Comme pour la réutilisation à la section 2.3.5, la modification peut être effectuée par un usager, un usager-scénariste ou un scénariste.

Un usager augmente la distance de tir d'un arc en le plaçant dans une pièce froide alors qu'un scénariste obtient le même résultat en modifiant directement l'attribut *Tension*.

2.5 Extensibilité

Il est possible de ne pas connaître tous les éléments à définir dans un MV avant le démarrage de celui-ci ou avant sa conception initiale. Un MV pourrait évoluer avec le temps et incorporer des modèles extérieurs à celui-ci. Avec l'ajout graduel d'éléments, le MV pourrait répondre à de nouveaux besoins de ses usagers.

L'*extensibilité* se définit donc comme la capacité de supporter l'ajout effectif, dans le MV, d'éléments de différents types et origines.

L'extension ne présente d'intérêt que si les éléments ajoutés peuvent interagir avec le MV déjà défini et produire l'effet désiré, d'où le terme effectif. Une forme commune se situe au niveau graphique avec l'interaction visuelle d'une caméra qui observe des entités. Cette interaction est généralement rendue possible par l'utilisation de standards graphiques comme le *Virtual Reality Modeling Language*, ou VRML (Carey *et al.*, 1997), et les graphes de scène (*scene graph*). Toutefois, les extensions de MVs devraient interagir à d'autres niveaux que la représentation graphique. Comme pour la composition, la modification et la réutilisation, il existe de nombreux types d'extensions. Celles-ci sont présentées dans les prochaines sections.

2.5.1 Extension en genre ou en nombre

La forme la plus simple d'extension, celle en nombre, consiste à ajouter au MV des éléments déjà connus. Dans une forme plus complexe, des éléments de nature inconnue peuvent y être ajoutés. Il s'agit dans ce cas d'une extension en genre. Des mécanismes sont requis pour que les éléments ajoutés interagissent avec les éléments déjà présents. L'exemple suivant montre les deux formes.

> Un MV conçu pour simuler des cryochirurgies pour le traitement du cancer contient, entre autres, des entités *Tumeur* et *Cryosonde*. Lorsqu'une tumeur d'un vrai patient est détectée par un système d'imagerie à résonance magnétique (IRM), de nouvelles entités *Tumeurs* sont ajoutées au MV tel qu'illustré à la figure 2.7 a). Les tumeurs ne diffèrent entre elles que par la valeur de leurs attributs. Ce MV est très spécialisé et dédié à cette application. Un MV plus générique et basé sur l'extension en genre permettrait de réduire des coûts de développement. Dans ce cas, l'entité *Tumeur* ne serait pas connue *a priori* mais créée après le démarrage du MV. Par exemple, la figure 2.7 b) illustre un MV contenant des types d'entités *CorpsCollisionable, Cryosonde* et *CorpsDétectéIRM*. La création d'une ou de plusieurs entités *Tumeur* requerrait l'héritage de *CorpsDétectésIRM*. Cette héritage indique que les tumeurs entrent en interaction avec les cryosondes et que deux tumeurs ne peuvent entrer en interaction l'une avec l'autre. La tumeur peut alors posséder des comportements et des attributs qui lui sont propres. Avec des mécanismes appropriés, l'extension en genre peut s'effectuer en cours d'exécution.

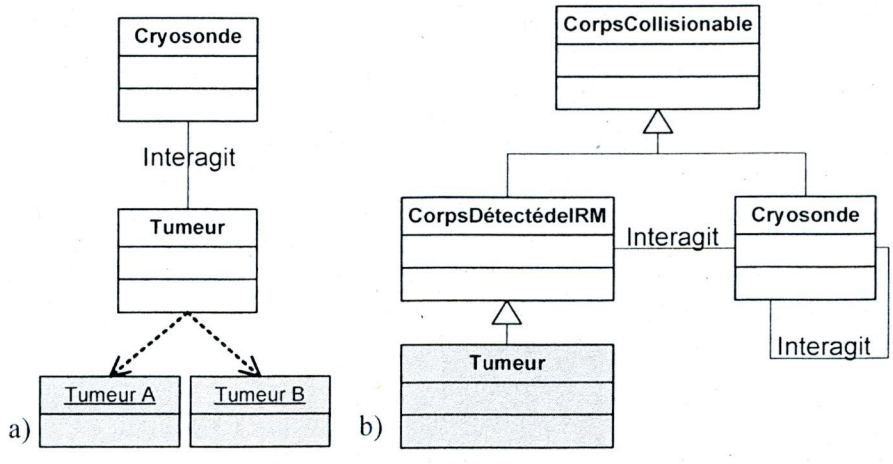

Figure 2.7 : Exemple d'extension en OO (Notation UML) : a) en nombre b) en genre.

2.5.2 Extension à partir d'éléments internes ou externes

Une extension peut s'effectuer en réutilisant des éléments déjà existants dans le MV, tel qu'illustré à la figure 2.7 a), ou à partir d'éléments externes au MV conçus spécifiquement pour y être ajoutés, tel qu'illustré à la figure 2.7 b). L'extension externe s'avère plus difficile car les éléments ajoutés doivent interagir avec les éléments déjà existants. Ce sujet sera traité dans la section 2.5.3. Les éléments externes peuvent provenir également d'un autre MV.

2.5.3 Extension de différents types d'éléments

Une extension implique l'ajout de différents types d'éléments. Les exemples suivants en présentent quelques cas.

La figure 2.8 illustre des exemples d'extension avec des éléments de différente nature. L'exemple a) illustre l'extension avec l'importation d'une entité. Dans ce MV hôte, la gravitation s'applique sur tous les corps qui possèdent une masse. Une branche est alors importée et lâchée à une certaine distance du sol. Pour que l'extension soit efficace, cette branche devrait subir des comportements déjà présents dans le MV hôte comme celle de la gravitation. Un tel ajout ne requiert que la présence de la masse, du centre de gravité et de la capacité d'ajouter l'attribut force à l'entité. Par conséquent, l'extensibilité dépend de la capacité de reconnaître des attributs d'entités importées et d'ajouter des attributs en cours d'exécution.

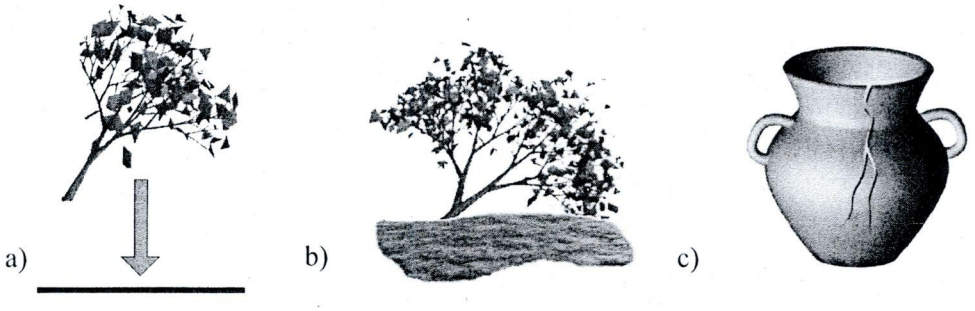

Figure 2.8 : Différentes formes d'extension de MVs : a) ajout de la capacité de faire tomber la branche, b) de faire flotter une branche ainsi que c) de casser un vase importé.

L'exemple b) illustre une extension en attribut et en application d'un comportement existant. Dans le MV hôte, les entités qui possèdent une densité peuvent flotter. Une branche qui ne possède pas l'attribut densité est alors plongée dans l'eau. Cette branche pourrait

flotter car elle possède une géométrie et une masse permettant de calculer la densité nécessaire pour interagir avec l'entité eau. Par conséquent, l'extensibilité dépend de la capacité à reconnaître des attributs, d'en ajouter en cours d'exécution et de les calculer à partir des autres présents.

L'exemple c) illustre un vase importé d'un autre MV qui possède la capacité de se briser sous l'action d'un choc. Pour que cette extension soit pleinement opérationnelle, le MV hôte doit être en mesure de s'entendre avec l'entité importée sur la définition d'un choc pour que celle-ci interagisse. L'extensibilité dépend donc de la capacité de l'entité à reconnaître une action prise sur elle ou de la capacité du MV hôte d'activer un comportement d'entité importée.

2.5.4 Extension de niveaux de granularité différents

Comme pour la réutilisation, la granularité des éléments influence l'extensibilité. Idéalement, un MV doit permettre l'extension d'éléments de granularité variable.

2.5.5 Extension en tant qu'usager, scénariste et usager-scénariste

L'extension en tant que scénariste est la plus courante. Celle en tant qu'usager s'apparente à une modification telle que décrite dans l'exemple de l'arc de la section 2.4.1.

2.6 Liberté d'action

La liberté d'action de l'usager et du scénariste est basée sur les autres caractéristiques.

> *La liberté d'action* se définit comme la capacité de définir, de modifier, d'étendre, de composer et de réutiliser les éléments en posant des actions.

L'*agilité* d'un MV dépend donc partiellement de la liberté d'action qui, elle-même, dépend en partie des autres caractéristiques combinées. Le seul fait de pouvoir effectuer une action n'est pas suffisant. Les conséquences des actions doivent être gérées afin de maintenir la cohérence du MV. De plus, ces actions ne doivent pas nuire à l'*agilité* future.

La figure 2.9 illustre quelques exemples de libertés souhaitables au sein des MVs. Dans l'exemple a), un usager trouve une flèche et un arc à différents endroits. Il décide de

lancer la flèche avec l'arc. Si l'arc peut interagir avec cette flèche (et *vice versa*), alors le déroulement correspond à la liberté que l'usager pense avoir. Dans cet exemple, une incapacité d'interaction entre l'arc et la flèche résulterait en un manque de liberté d'action.

Dans l'exemple b), un usager tente de mettre le feu à une chaise. Si le modèle physique de la combustion du bois ou de la chaise existe, l'usager a la liberté de mettre le feu à la chaise. Par conséquent, l'existence de modèles dans le MV est une autre cause améliorant la liberté d'action d'un usager.

L'exemple c) montre la corde de l'arc détachée afin de créer une canne à pêche. Cette liberté d'action est due à la composabilité de l'arc. La liberté d'action permet également l'opération inverse : il aurait été possible de créer un arc à partir d'une corde extensible et d'un bout de bois flexible. Si, suite à l'action de détacher l'arc, il était impossible de le rattacher, il s'agirait d'un exemple où une action limite l'*agilité* future.

Dans l'exemple d), un usager met le feu à une branche avec un briquet. Il dépose ensuite cette branche sur d'autres branches. Il s'attend à ce que les branches ajoutées prennent feu. Puisque la branche possède la capacité de mettre le feu, alors il peut créer un feu de camp. La liberté d'action est permise par la capacité de la branche à jouer le rôle d'instigateur. L'utilisation de modèles plus génériques augmente donc la liberté d'action.

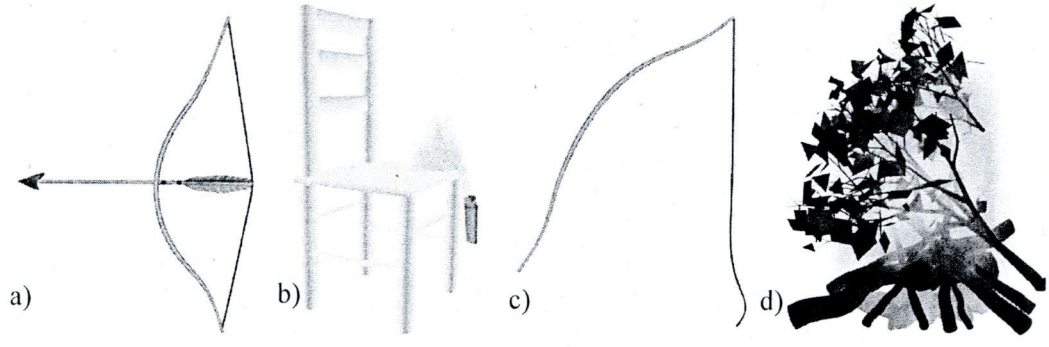

Figure 2.9 : Exemples de MVs qui démontrent une liberté d'action : a) lancement d'une flèche trouvée avec un arc b) mise à feu d'une chaise avec un briquet c) création d'une canne à pêche à partir d'un arc d) création d'un feu de camp à partir de plusieurs branches.

2.7 Conclusion sur l'agilité

La présentation détaillée des caractéristiques permet maintenant de définir l'objectif qui permettra d'accroître l'*agilité* des MVs. Cette thèse vise le développement d'une architecture augmentant plusieurs types de *composabilité*, de *réutilisabilité*, de *modifiabilité*, d'*extensibilité* ainsi qu'une plus grande *liberté d'action*. Tel qu'illustré à la figure 2.10, l'amélioration de ces caractéristiques contribuera à accroître l'*agilité*.

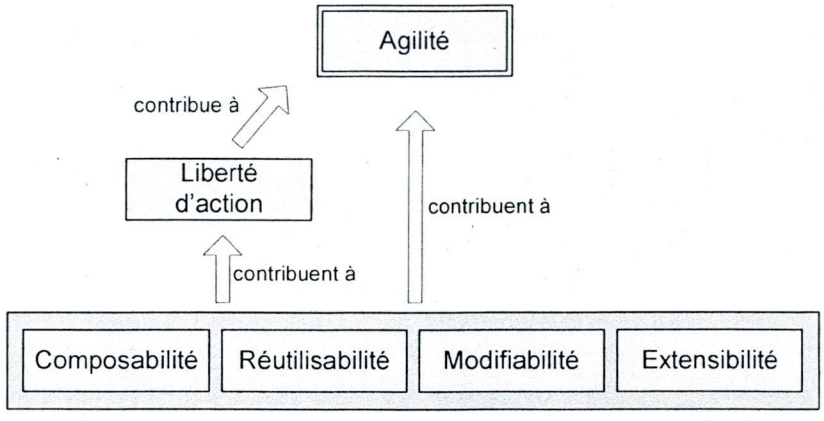

Figure 2.10 : Certaines caractéristiques contribuent à améliorer l'agilité.

Cette architecture comprendra un certain nombre de composantes qui seront identifiées dans l'état de l'art. Elle tiendra compte de plusieurs observations de ce chapitre. Premièrement, les MVs devront être décrits par partie et constitués par assemblage de nombreux éléments. Cette composition facilitera la mise en place de comportements émergents. La réutilisation dans un contexte différent sera ainsi favorisée. De plus, il faudra encourager une granularité fine et parfois variable. L'architecture devra permettre également l'amélioration de l'*agilité* tant du point de vue de l'usager que du scénariste. L'extension en genre et externe seront des problématiques également étudiées. De plus, il faudra faciliter l'ajout de comportements et de modèles, surtout de façon générique, afin d'accroître la liberté d'action. Finalement, il faudra éviter les spécialisations qui empêchent la réutilisation. L'annexe A en présente une implantation afin de montrer son employabilité et sa réalisabilité. Suite à la présentation détaillée des caractéristiques, l'état de l'art des architectures existantes de MVs au chapitre 3 permettra de proposer une architecture au chapitre 4.

Chapitre 3

État de l'art

Le chapitre 2 a présenté les objectifs visés dans cette thèse. Ce présent chapitre analyse les architectures à la base de MVs existants en regard des objectifs visés. Il se divise en quatre parties. La première situe les architectures dans une brève perspective historique. Les trois sections suivantes font une revue de littérature et analysent les méta-modèles conceptuels, le rôle des éléments et l'ordonnancement du MV. Quoique plusieurs architectures apportent des solutions partielles aux objectifs visés, la revue montrera qu'aucune ne donne une solution satisfaisante. Les lacunes des architectures existantes sont identifiées et permettent de formuler des suggestions pour la proposition d'une architecture plus *agile* atteignant les objectifs définis au Chapitre 2.

3.1 Revue générale

La revue de littérature des différentes architectures de MVs et de leur histoire a pour but de mieux situer les travaux de cette thèse par rapport au domaine et de comprendre les choix, les avantages et les inconvénients des travaux antérieurs. Ces architectures proviennent de trois domaines distincts : les simulations interactives réparties, les MVs répartis du domaine de la recherche et les MVs à la base des jeux 3D répartis sur l'internet. Toutes ces applications ont utilisé des MVs et contribué à leur avancement.

3.1.1 Origine des simulations interactives réparties

Le domaine de la simulation a pris son essor avec le développement des simulations à

événements discrets (Fujimoto, 2000). Originellement, les simulations exécutaient des modèles sans contrainte de temps et avec peu ou pas d'interaction avec un usager. Leur but était de vérifier des hypothèses et le temps d'exécution se devait d'être tout simplement "raisonnable". Seul l'ordre de grandeur du temps d'exécution avait de l'importance. Ce type de simulation est qualifié d'analytique. Par la suite, l'utilisation de plusieurs ordinateurs reliés en réseau a permis d'accélérer le temps d'exécution. Cette approche a donné naissance aux systèmes de simulation parallèle et répartie. En même temps que la répartition, les simulations ont commencé à inclure des éléments humains (*man-in-the-loop*) et matériels (*hardware-in-the-loop*) dans la boucle de simulation, à s'exécuter en conformité avec le temps réel et à utiliser la réalité virtuelle pour afficher les résultats et permettre aux utilisateurs d'interagir avec la simulation. Cette approche porte le nom de simulation interactive. Une importante part des travaux effectués dans le domaine des simulations interactives provient des applications militaires.

SIMNET (Miller *et al.*, 1989) est un pionnier dans le domaine des simulateurs interactifs répartis pour application militaire. Toutefois, il causait de nombreuses incohérences en cours de simulation. Par exemple, un observateur pouvait voir un char d'assaut se faire détruire avant l'impact du missile. De plus, le protocole d'échange demeurait propriétaire et non documenté.

Pour permettre l'ajout de stations hétérogènes, le protocole d'échange des paquets de SIMNET a été standardisé en 1993, sous le nom de Distributed Interactive Simulation, ou DIS (1278 IEEE standard, 1993). Il était toutefois impossible de simuler d'autres types de scénarios que ceux initialement prévus.

En 1993, l'armée américaine débuta la conception du *High Level Architecture*, ou HLA (2006), le successeur de DIS. Disponible depuis 1996, ce standard cherche à réduire les problèmes d'incohérences qui survenaient dans SIMNET et DIS et permet aux développeurs de définir eux-mêmes le contenu de l'information échangée entre les entités. HLA sera plus longuement discuté que les autres approches car cette thèse remet en question certains de ses éléments de base.

3.1.2 Origine des MVs répartis de recherche

Les MVs prennent leurs origines au début des années 80 dans le développement de la réalité virtuelle avec les avancées technologiques des engins de rendu graphique. Toutefois, avec le développement de la répartition émergent, à la fin des années 80, des MVs avec des comportements et offrant une plus grande interactivité à l'usager. Ces nouveaux MVs incluent désormais des comportements physiques complexes, l'interaction de nombreux intervenants humains entre eux et avec des entités virtuelles ainsi que l'interaction de systèmes physiques avec les entités virtuelles et des humains. Les humains ou le matériel qui interagissent avec le MV deviennent alors répartis géographiquement, permettant une plus grande extensibilité. Quoique les travaux aient porté principalement sur la gestion des ressources réseaux et la qualité graphique jusqu'au milieu des années 90, il demeurait difficile de développer des MVs ayant les qualités énumérées au chapitre 2. Au moment où l'*agilité* est devenue nécessaire, certaines solutions ont émergé dans des architectures dont le but initial n'était pas de résoudre ces problèmes. Plus récemment, les MVs collaboratifs ont fait leur apparition. Toutefois, ceux-ci ne répondent pas significativement à la problématique relative à l'*agilité*. La figure 3.1 présente ces principales architectures dans une perspective chronologique.

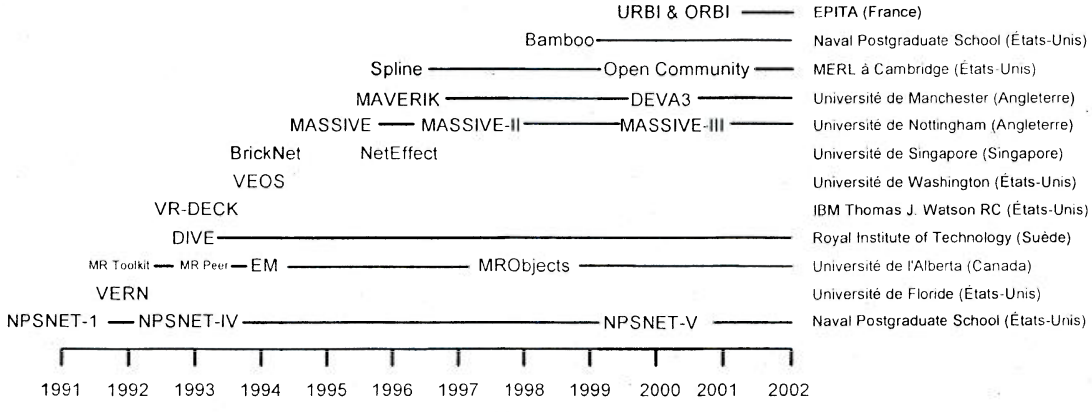

Figure 3.1 : Historique des principales architectures de MVs provenant du milieu de la recherche.

Les premiers MVs répartis apparaissent au début des années 90. Créé par la Naval Postgraduate School de Monterey en Californie, NPSNET (Macedonia *et al.*, 1994) est le premier de sa catégorie. Utilisant des stations *Silicon Graphics*, il permettait de se connecter à

une simulation interactive répartie SIMNET à moindre coût. NPSNET émulait le protocole d'échange réseau de SIMNET, qui est devenu par la suite le standard DIS. NPSNET a évolué rapidement à NPSNET-IV qui, à cause de sa complexité, imposait une longue courbe d'apprentissage et requérait beaucoup de maintenance. Puisque NPSNET visait l'exploration du domaine des MVs répartis et non leur utilisation, le groupe responsable de sa conception s'est tourné, en l'an 2000, vers une approche plus modulaire, orientée composantes et utilisant un langage plus simple et source de moins d'erreurs, le Java. Cette nouvelle architecture, NPSNET-V (Zyda & Darken, 1998), ne définit aucune interface standard d'échange entre les composantes car elle vise l'exploration des protocoles inter-modules et inter-entités. Cette approche est donc un retour à la base afin d'améliorer les concepts d'extensibilité et de composabilité qui faisaient défaut dans les versions précédentes.

VERN (Blau *et al.*, 1992) et MRToolkit (Shaw & Green, 1993) sont venus s'ajouter en 1992. DIVE (Carlsson & Hagsand, 1993) est apparu l'année suivante. Avec NPSNET, il est demeuré une architecture de référence pendant de nombreuses années. Par la suite, VR-DECK (Codella *et al.*, 1993), BrickNet (Singh *et al.*, 1995), ainsi que NetEffect (Das *et al.*, 1997) ont fait leur apparition. Ceux-ci employaient des approches similaires au VRML et à DIVE. Comme plusieurs de leurs successeurs, ils reposent sur une approche réseau ou graphique et prennent en charge l'optimisation du transfert des paquets de données entre les processus, représentant les entités, répartis sur plusieurs ordinateurs. Dans les MVs basés sur l'approche graphique, les éléments sont placés dans un graphe de scène. Il s'agit en fait d'un abus d'utilisation des librairies graphiques avec graphe de scène dans lequel des noeuds peuvent être spécialisés pour jouer le rôle d'entités virtuelles. Le graphe, avec son mécanisme de rendu, devient alors le MV. Les sections 3.2 et 3.3 préciseront pourquoi ces deux approches ne permettent pas d'obtenir des MVs *agiles*.

Conçu en 1994, VEOS (Bricken & Coco, 1994) se démarque des autres architectures de l'époque et possède encore à ce jour des fonctionnalités avancées permettant des MVs *agiles*. À partir de réflexions tenant compte de facteurs influençant les MVs, les auteurs ont conçu une architecture utilisant le langage LISP, des bases de données avec mécanismes de recherche et basée sur un paradigme biologique-environnement. Les problématiques qu'aborde VEOS sont mieux élaborées et proposent des solutions plus complètes que ce que proposent

les autres MVs. Il sera souvent question de VEOS dans les sections 3.2 et 3.3.

Démarrée en 1995, la série des MASSIVE (Greenhalgh & Benford, 1995 ; Greenhalgh & Benford, 1997 ; Greenhalgh *et al.*, 2000) s'est principalement intéressée à la résolution des problèmes reliés à la gestion spatiale et réseau des entités virtuelles. De plus, lors des dernières versions, leurs auteurs ont abordé les problèmes d'interopérabilité et à la capacité de modifier le MV en cours d'exécution.

Conçue à l'Université de Manchester, la série MAVERIK (Hubbold *et al.*, 2001) et DEVA 3 (Pettifer *et al.*, 2000) constitue, avec VEOS, l'autre architecture qui propose un paradigme différent des traditionnelles approches graphiques et réseau. De plus, il s'agit, avec VEOS, des architectures ayant le plus traité d'*agilité*. En effet, MAVERIK se distingue en offrant la composition des méthodes, des données et des entités pendant l'exécution. Basé sur MAVERIK, DEVA 3 ajoute une couche qui tient compte de la capacité du MV de supporter l'implantation de comportements physiques.

Spline (Waters *et al.*, 1996), Open Community (2006) et Bamboo (Watsen & Zyda, 1998) constituent d'autres architectures développées ces dernières années mais qui demeurent nettement incomplètes et insuffisantes, comme il sera expliqué plus loin, pour le type de MVs *agiles* recherchés.

Conçue en 2000, l'architecture URBI & ORBI (Fabre *et al.*, 2000) rejette également le VRML comme structure et paradigme d'un MV. Les auteurs veulent éviter ce qu'ils appellent le *"3D bias"*. Ils proposent plutôt l'utilisation de graphes conceptuels qui connectent les cellules entre elles avec des relations sémantiques. Cette information ne sert toutefois qu'à réduire le trafic réseau entre les cellules.

3.1.3 Origine des MVs et des jeux 3D répartis sur l'internet

Le domaine des jeux 3D a également contribué au développement des MVs. Les premiers jeux 3D n'étaient, au départ, que des engins de rendu avec un ou des personnages qui se déplaçaient sous la commande de la souris et du clavier. DOOM (2006) fait parti de cette catégorie. Par la suite, d'autres jeux sont apparus avec des scénarios plus complexes comme dans King Quest VIII (2006), des comportements physiques plus réalistes comme dans Halo

(2006) ou Flight Simulator (2006), la possibilité de créer des MVs plus facilement avec des outils puissants comme dans Warcraft III (2006).

Les MVs persistants sur l'internet tels Blaxxun (2006), Activeworlds (2006), DigitalSpace Traveler (2006) et Second Life (Rymaszewski *et al.*, 2006) font partie d'une autre catégorie d'application des MVs. Souvent désignés de *Chat3D*, ils permettent de partager de l'information, de discuter et de naviguer dans des MVs répartis sur plusieurs ordinateurs. De plus, ils fonctionnent presque sans interruption, supportent des extensions avec l'ajout d'entités en cours d'exécution et supportent un grand nombre de participants et d'entités. Toutefois, ces MVs ne s'intéressent ni ne résolvent la problématique dont il est question dans cette thèse.

Les jeux réseau persistants sont apparus il y a une dizaine d'années. Ils supportent un grand nombre de joueurs, d'où la désignation de *Massively Multiplayer Online Games* (WIKI MMOG, 2006). Ils combinent la persistance et la taille des *Chat3D* avec les comportements physiques, les décors et l'expérience de jeu des jeux 3D. Everquest (2006) et Ultima Online (2006) en sont les exemples les plus connus. Ces MVs ne résolvent pas la problématique posée dans cette thèse. De plus, presque tous les MVs présentés dans cette sous-section sont de type commercial. Ils offrent peu de documentation et aucun détail sur l'architecture n'est disponible. Par conséquent, ils seront peu cités dans les discussions à venir.

3.2 Modèles conceptuels et méta-modèles conceptuels

Le type de modélisation influence l'*agilité* des MVs. Par exemple, la granularité des modèles module la composabilité. Les niveaux de fidélité et de détails influencent également l'*agilité*. Ces niveaux peuvent varier d'un MV à l'autre et au sein d'un même MV. Par exemple, le monde réel peut être reproduit à différents niveaux de détails graphiques, permettant ainsi d'accélérer le rendu graphique 3D avec plus ou moins de polygones. Toutefois, notre étude n'aborde pas ces aspects. Elle portera principalement sur le type de modélisation.

Un grand nombre de types de modèles pour concevoir des MVs ont déjà été identifiés. Fishwick (1995) présente et classifie les plus connus. Tel que le montre la figure 3.2, le grand nombre de types de modèles peut complexifier la description d'un MV et rendre difficile la conception et la gestion de MVs *agiles*.

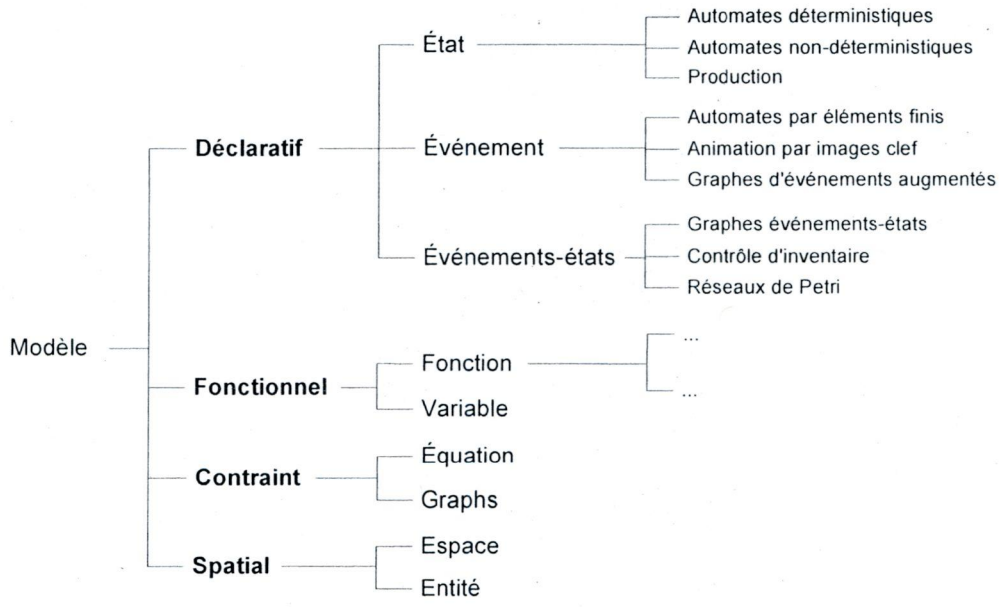

Figure 3.2 : Partie de la classification des modèles de Fishwick.

Il est toutefois possible de représenter ces modèles plus spécifiques par une approche de plus haut niveau, désignée modèle conceptuel (Fishwick, 1995).

> Un *modèle conceptuel* décrit l'essentiel de la nature des entités, des propriétés et des relations d'un système.

L'aspect générique du modèle conceptuel devrait lui permettre de représenter un grand nombre de types de modèles spécifiques. Il en existe de nombreux types. Un diagramme de classe en *Unified Modeling Language*, ou UML (Atkinson & Kühne, 2001) en constitue un exemple. Comme pour ce diagramme qui requiert du code pour s'exécuter, les modèles conceptuels requièrent des modèles spécifiques, tel qu'illustré à la figure 3.3. Quoiqu'aucune étude sur ces types de modèles spécifiques et sur la capacité des modèles conceptuels à les représenter ne sera effectuée dans cette thèse, les exemples du chapitre 5 indiqueront que le type de modèle conceptuel proposé peut contenir de nombreux types de modèles spécifiques. Il est basé sur les éléments disponibles dans le MV. De plus, le modèle conceptuel contient des parties de modèle conceptuel qui sont elles-mêmes des modèles conceptuels.

Un modèle conceptuel possède l'avantage d'être plus facilement manipulable par un scénariste qui n'est pas expert dans le domaine à modéliser, ce qui augmente sa capacité à concevoir des MVs. De plus, tous les modèles d'un MV étant basés sur un même paradigme, la gestion des modifications, des compositions et des extensions en serait facilitée, ce qui augmenterait l'*agilité*.

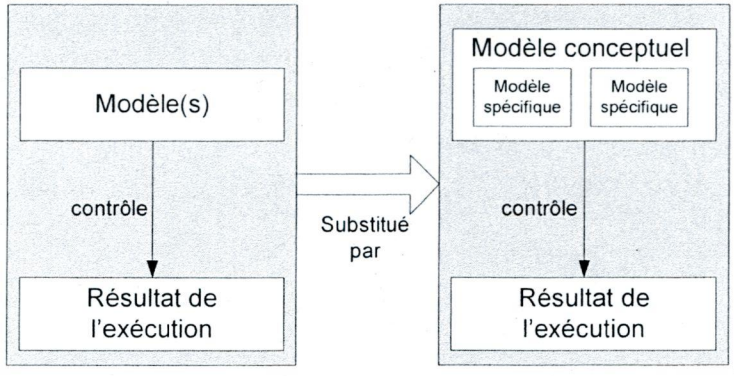

Figure 3.3 : Les modèles spécifiques devraient être encapsulés dans des modèles de plus haut niveau, des modèles conceptuels.

Tel qu'il le sera démontré ultérieurement, le type de modèle conceptuel influence l'*agilité*. Par exemple, l'UML entraîne de nombreux problèmes lors de la réutilisation. Cette thèse identifiera donc le type de modèle conceptuel sur lequel concevoir des MVs afin d'en augmenter l'*agilité*.

Il est possible d'*imposer* un type de modèle conceptuel. Ce cadre, qui se nomme *méta-modèle conceptuel* (Jaramillo *et al.*, 2003), peut également servir à ce que l'exécution soit conforme au modèle conceptuel le contrôlant.

> Un *méta-modèle conceptuel* se définit comme un modèle du cadre, des règles et des contraintes des concepts communs à un type de modèle conceptuel.

La figure 3.4 illustre que le méta-modèle encadre la conception du modèle conceptuel qui, lui-même, contrôle le résultat de l'exécution du MV.

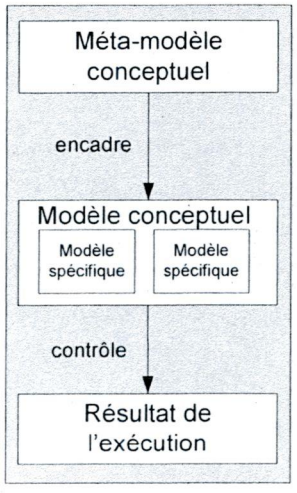

Figure 3.4 : Schéma relationnel entre le méta-modèle conceptuel, le modèle conceptuel et l'exécution du MV

Une analogie avec l'UML s'avère nécessaire. Par exemple, l'ensemble des objets d'une certaine application se conforme au diagramme de classe qui, lui-même, se conforme au langage UML. Ce langage garantit donc que chaque élément du diagramme est conforme à certaines conventions telles que l'obligation de déclarer l'accessibilité (public, privé, protégé) des attributs. Le diagramme de classes garantit à son tour qu'aucun attribut privé ne puisse être accédé directement par un autre objet lors de l'exécution. En résumé, le méta-modèle UML assure que toutes les applications développées par une équipe de développeurs n'accéderont pas certains attributs.

Le rôle du méta-modèle conceptuel est toutefois limité. Par exemple, l'UML ne peut garantir que tous les noms d'attributs privés débuteront avec une même lettre. Similairement, les méta-modèles conceptuels ne peuvent toujours garantir qu'un MV résultant de la composition de plusieurs modèles conceptuels sera cohérent, surtout lorsqu'elle est effectuée en cours d'exécution. Le prochain exemple illustre cette problématique.

> Un MV, illustré à la figure 3.5, doit permettre à un usager de mettre le feu à une branche et de la plonger dans l'eau.

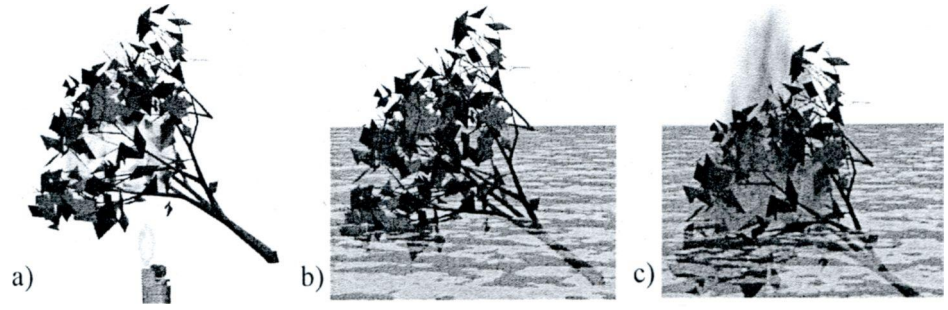

Figure 3.5 : Exemple d'un MV permettant : a) de mettre le feu à une branche avec un briquet b) de plonger une branche dans l'eau c) d'effectuer les deux actions en chaîne.

Le schéma de la figure 3.6 montre un exemple des méta-niveaux pour ce MV.

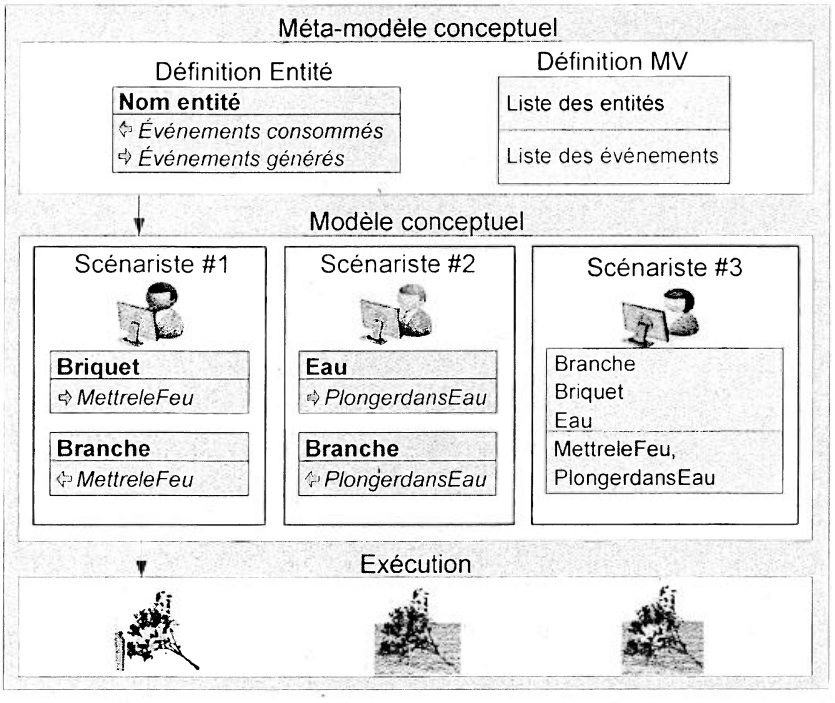

Figure 3.6 : Exemple d'un méta-modèle conceptuel, d'un modèle conceptuel et de son exécution pour l'exemple de la figure 3.5.

Dans cet exemple, le méta-modèle conceptuel force, au sein du modèle conceptuel, la définition des entités avec des champs obligatoires dont les événements consommés et générés. De plus, pour les besoins de l'exemple, il est possible de définir deux entités avec le même nom. Dans ce cas, le MV combine celles-ci en concaténant leurs événements. Ce MV est construit par trois scénaristes. Un premier scénariste conçoit un modèle conceptuel

> de briquet qui génère l'événement *MettreleFeu* ainsi qu'un autre modèle de branche qui consomme ce type d'événement. Un second scénariste conçoit un modèle conceptuel de branche qui consomme l'événement *PlongerdansEau* et un autre d'eau qui génère le même événement. Le troisième scénariste combine le tout avec la création de la liste des entités ainsi que des événements cités précédemment. Que se passe-t-il lorsque la branche est plongée dans l'eau ? Dans ce cas, la branche continue à bruler. S'il n'y avait eu qu'un seul scénariste pour la branche, il aurait probablement considéré la concaténation de deux événements et aurait implanté un comportement qui éteint le feu lorsque plongé dans l'eau. Ce méta-modèle conceptuel ne peut donc pas garantir la composition à tous les coups. Une façon de remédier à ce problème consiste à obliger par un processus, dès le départ via le méta-modèle conceptuel, qu'un scénariste définisse séparément la liste des événements possibles dans un MV. Les scénaristes pourraient ensuite créer les entités et celui en charge de la branche tiendrait compte des effets combinés du feu et de l'eau. Un processus de conception peut imposer cette approche mais le méta-modèle conceptuel doit d'abord permettre la définition séparée des événements.

L'exemple précédent montre la difficulté de garantir la cohérence d'un MV, c'est-à-dire que la composition de l'ensemble des parties de modèles conceptuels fonctionnera toujours, surtout si celles-ci sont modifiées en cours d'exécution. Toutefois, il est possible d'améliorer la cohérence nécessaire à l'*agilité* en facilitant la composition de parties de modèles conceptuels dans certains cas.

L'exemple précédent montre que la composition, ou toute autre action, peut mener à des MVs incohérents si elle n'est pas effectuée correctement. Comme autre exemple, si la gravitation est, par inadvertance, ajoutée deux fois à un même MV, les entités qui y évoluent ne doivent pas subir deux forces gravitationnelles. La gestion des actions effectuées sur un MV est encore plus complexe lorsque celui-ci change fréquemment, surtout lors de son exécution. Cette tâche, normalement effectuée par le scénariste lors de la conception, doit être prise en charge automatiquement par un gestionnaire lorsque ces actions surviennent en cours d'exécution. Ce gestionnaire ne pourra assurer la cohérence que s'il possède de l'information pour effectuer cette tâche. La gestion des actions sur le MV peut être facilitée par l'ajout d'information qui décrit l'intégration des parties d'un MV. Cette information, définie au niveau des modèles conceptuels, peut être qualifiée de métadonnée ou même de méta-modèle. Puisque le terme méta-modèle est déjà réservé, cette information sera intégrée au modèle conceptuel et

désignée comme tel afin de ne pas accroître ou mélanger inutilement les termes. La gestion des actions qui permettent l'*agilité* s'avère primordiale afin d'assurer le bon fonctionnement du MV. Elle occupera donc une part importante de cette thèse.

Une grande *agilité* peut rendre des MVs complexes à gérer, principalement à cause d'une fine granularité des éléments le constituant et des nombreuses modifications qui y surviennent, dont certaines en cours d'exécution. Des optimisations s'avèrent alors nécessaires. L'optimisation de l'application des modèles sera considérée dans cette thèse.

Tous les MVs sont basés sur une structure, un langage ou une architecture et dans de rares occasions des méta-modèles conceptuels. Néanmoins, ces langages, architectures et structures reposent implicitement sur des méta-modèles conceptuels. Le tableau 3.1 montre une partie des ceux qui ont retenu l'attention.

Niveau	OO	HLA	VRML
Méta-modèle conceptuel (M2)	UML	OMT	Standard VRML
Modèle conceptuel (M1)	Diagramme UML	SOM + FOM	Diagramme VRML Événements
Instanciation (M0)	Exécution du programme	RTI + Fédérés	Instanciation du diagramme VRML

Tableau 3.1 : Méta-modèles conceptuels pour concevoir des MVs.

Les sections suivantes discuteront de ces structures et méta-modèles retenus en regard de leur influence sur l'*agilité*.

3.2.1 Modèles conceptuels sans structure

Une bonne partie des MVs placent les entités sur un même niveau, et ce, dans un monde unique. Ils laissent aux scénaristes la tâche de gérer les relations entre les entités, de créer des catégories, etc. Ce manque de structure pose un problème car les éléments conçus par différents scénaristes ont peu en commun. L'architecture ne peut prendre en charge la cohérence du MV et en garantir l'*agilité*.

3.2.2 Structure centrée sur le rendu graphique

La structure centrée sur le rendu graphique, assez répandue, place les entités en relation parents-enfants afin de créer des graphes de scène. La faiblesse de cette approche provient de ce que les entités ne sont reliées qu'avec une relation de référentiel et de composition. Un exemple serait une entité *Automobile* composée de quatre entités *Roues*. Cette structure, utilisée dans VRML, OpenInventor (Wemecke, 1994) et de nombreuses boîtes à outils graphiques 3D, convient très bien pour des MVs purement graphiques. Toutefois les relations et les interactions entre les entités du monde réel ne sont pas limitées à la géométrie. Par conséquent, les MVs construits sur ce paradigme sont seulement *agiles* et cohérents d'un point de vue graphique.

Dans ce paradigme, les comportements sont représentés sous la forme d'une méthode virtuelle devant être surchargée (*overriding*) ou par des procédures de rappel (*callbacks)*. L'entité qui contient le comportement souscrit et publie des événements. Il s'agit d'une approche orientée entité. Le MV gère alors ces événements selon une technique qui ne favorise pas l'*agilité*. Ce sujet sera abordé dans la section 3.4 sur l'ordonnancement.

Dans une telle approche, la réutilisabilité, l'extensibilité, la modifiabilité et la composabilité sont généralement limitées à l'aspect graphique. Les comportements sont généralement conçus que pour une entité car ils y sont inclus. Il est donc impossible de garantir la réutilisation des comportements dans d'autres entités à cause d'un manque de généralisation de l'application du comportement.

Les entités interagissent par les événements. Deux techniques ont été identifiées pour cette approche : laisser chaque partie du MV établir la sémantique du dialogue ou imposer un langage commun à toutes les parties. La première laisse aux scénaristes une grande liberté. Toutefois, le sens des interactions entre les entités n'étant pas connu d'avance, il n'est pas possible de garantir la composabilité de MVs conçus par différents scénaristes. Dans la deuxième approche, l'ensemble des événements est déjà défini et les scénaristes s'y réfèrent lors de la conception, facilitant ainsi la composition. Cette technique ne pose pas de problèmes pour les extensions internes mais lorsqu'il s'agira d'une extension externe, celle-ci sera limitée au contexte des événements déjà existants.

3.2.3 Structure centrée sur les communications réseaux (*Locales*)

Pour des raisons de performance réseau, certaines architectures de MVs réseautés (*networked virtual environnement* ou *distributed virtual world*), tels que Splines et NetEffect, reprennent la structure centrée sur le rendu graphique et en y ajoutant le *locale*. Le *locale* divise le MV en blocs exclusifs. Cette structure permet de gérer les entités par régions spatiales, limitant les interactions possibles d'une entité. Par exemple, un avatar ne peut voir une entité que si elle est située dans le même *locale* ou dans le *locale* voisin. Open Community et MASSIVE3 utilisent une version améliorée de cette approche avec des entités composées.

Il devient ici possible sauver du temps d'exécution avec l'utilisation de relations d'inclusion topologique (ex.: une automobile dans le garage). Toutefois, l'inclusion d'un comburant comme l'oxygène dans une scène ne sera pas considérée car il s'agit d'une composition. De plus, cette approche ne partage que la position comme attribut commun des entités. La gestion des régions d'intérêts est limitée à une seule granularité. Finalement, les comportements étant limités aux entités, cette méthode ne permet pas de gérer automatiquement des comportements communs à toutes les entités comme la gravitation.

3.2.4 Structure centrée sur les environnements hiérarchiques

Pour résoudre le problème précédent, certaines architectures utilisent des structures centrées sur les environnements permettant de partager des caractéristiques communes aux entités. Cette approche, utilisée dans DEVA 3, NPSNET et Bamboo, fournit un milieu pour retrouver des attributs ou pour contenir les comportements qui s'appliquent sur l'ensemble des entités. DEVA 3 permet de définir des comportements propres à l'entité, communs à un groupe ou à toutes les entités et qui peuvent être ajoutés en cours d'exécution. Toutefois, les comportements de groupe ne peuvent être changés en cours d'exécution. Cette approche facilite le travail de gestion du MV pour y maintenir la cohérence. Par exemple, dans DEVA 3, un MV peut définir la gravitation sur la Terre alors qu'un autre MV définit une entité qui évoluera sur cette Terre. Lorsque les MVs sont assemblés et exécutés, l'entité subit alors la force gravitationnelle. Cette gestion de la cohérence n'est possible que dans le cas d'une relation environnement-entité. Cette approche ne gère toutefois pas les conditions d'interaction qui dépendent d'autres types de relations entre entités.

Ces MVs sont plus extensibles que les précédents. Dans DEVA 3, par exemple, un comportement ainsi que les attributs requis à ce comportement peuvent être ajoutés en cours d'exécution en utilisant des composantes. Toutefois, ce comportement ne s'applique que sur une entité et non sur un groupe ou entre un groupe d'entités. Ceci limite le type d'extensions possibles.

L'approche orientée environnement reste, comme ses prédécesseurs, toujours centrée sur l'entité. Elle suppose que l'entité la plus importante, l'environnement, est responsable d'agir et de modifier des entités moins importantes, c'est-à-dire les entités qui y sont incluses. Toutefois, pourquoi serait-ce le rôle de la terre d'appliquer la gravitation sur une entité alors que l'attraction gravitationnelle implique autant les deux entités? L'approche centrée sur l'entité pose de nombreux problèmes de modélisation, surtout lorsqu'il est question de modéliser des interactions symétriques telles que les collisions et la friction.

3.2.5 Structure orientée objet

Un scénariste peut utiliser l'OO pour concevoir un MV, non pas pour l'implanter mais plutôt pour le décrire. L'OOMP (Fishwick, 1996) est un exemple de modélisation physique OO. Plusieurs MVs utilisent des langages OO comme le C++ et le Java. Par conséquent, ils utilisent également l'OO comme méta-modèle qui comprend la classe, l'attribut, la méthode, l'association, l'agrégation et la généralisation. L'UML est une formalisation de cette approche.

Bien que l'OO soit connu pour favoriser la réutilisation et l'extension, il ne s'agit pas d'une modélisation permettant une grande *agilité* dans le cadre des MVs. Cela ne veut pas dire qu'il ne peut pas être utilisé pour implanter un MV. Il existe de nombreuses études sur les limitations, avantages et améliorations à apporter à l'OO. Toutefois, la plupart cherchent à améliorer la conception de logiciels qui ne vise pas les mêmes objectifs que ceux de l'*agilité*. Quoiqu'il aurait été utile de revoir ces publications et de transposer leurs conclusions pour la conception de MVs, ceci dépasse le cadre de cette thèse. Par conséquent, cette section présente seulement les limites les plus connues de l'OO dans l'obtention de MVs *agiles*.

Manque d'information causé par l'agrégation

L'OO définit l'agrégation comme une relation d'inclusion d'un objet dans un autre. Il

s'agit d'une relation partie-tout entre entités ou concepts. L'OO et l'UML définissent deux relations partie-tout : l'agrégation forte appelée aussi la composition et l'agrégation faible, simplement appelée agrégation. Ces relations ne sont toutefois pas aptes à décrire un grand nombre dee types d'inclusion. Des études (Winston *et al.*, 1987) ont permis de les classer en trois groupes : classe, spatiale et méronymique. La dernière se divise en six catégories : partie-tout (pédalier-bicycle), membre-collection (bateau-flotte), position-masse (pointe-tarte), substance-objet (acier-automobile), tâche-activité (payer-magasiner) et endroit-région (Everglades-Floride). Huit de ces d'agrégations (l'inclusion de classe est enlevée du groupe) se simplifient en deux types dans l'OO. L'exemple qui suit discute de l'importance d'une bonne aggrégation.

"L'automobile est dans le garage", "Le garage est composé de bois" et "Le garage est composé d'une porte" sont trois exemples d'agrégation illustrés à la figure 3.7.

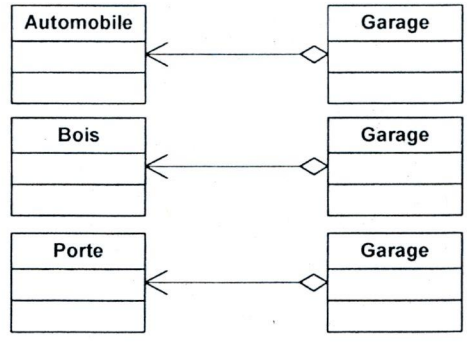

Figure 3.7 : Exemple d'agrégations ayant des significations différentes.

La première agrégation, "l'automobile est dans le garage" est une inclusion spatiale ou topologique. La seconde, "Le garage est composé de bois" est de type substance-objet. La dernière est de type partie-tout.. Tout le contenu sémantique est perdu dans une représentation simplifiée d'agrégation en OO. Cette information serait pourtant utile à la gestion d'un MV *agile*.

Par exemple, une entité Véhicule est ajoutée à un MV et se déplace dans la rue. Le MV est responsable de vérifier les collisions entre les entités. Le véhicule original étant situé dans le garage, le MV n'a besoin de vérifier les collisions qu'entre le véhicule et le couple garage-porte. Le bois et le véhicule dans le garage ne représentent pas des entités utiles pour la détection de collisions à cette étape. Toutefois, si une collision survient, la relation de

composition physique indiquera où trouver l'information sur la composition de la structure et, par conséquent, sur sa résistance à la déformation.

L'UML supporte également l'ajout de rôles à une association. Toutefois, même si l'UML permet cette solution de rechange, ni le Java ni le C++ ne conservent cette information lors de l'implantation. Toutefois, la qualification d'une relation s'avérerait possible avec une implantation adaptée.

Réutilisation et extension apparentes

L'héritage permet de créer des hiérarchies de classes apparentées, de réutiliser des classes et d'étendre des classes existantes par la spécialisation. Quoique l'héritage apporte des gains en réutilisation et en extension d'un point de vue logiciel, l'utilisation de ce mécanisme pour concevoir des MVs nuit à l'*agilité*. Le diagramme de classes final montre bien la réutilisation et l'extension mais celle-ci n'est obtenue que par l'aboutissement d'un processus long et itératif qui a probablement requis le réusinage (Opdyke & Johnson, 1990) du diagramme UML et de ses classes. Ce réusinage vise deux buts : faciliter la maintenance ainsi que les extensions et la réutilisation futures avec un remodelage minimal.

La figure 3.8 montre un diagramme de classe pouvant servir à décrire un MV. Les classes *Bateau*, *Avion* et *Automobile* sont tous des véhicules. Par conséquent, ils dérivent de la classe *Véhicule*. Cette classe renferme les méthodes et les attributs reliés à la gravitation. De plus, la classe définit la méthode virtuelle pure *Déplacer* devant être surchargée et servant à déplacer le véhicule spécialisé.

De prime abord, cet exemple facilite la réutilisation et l'extension par l'héritage. Dans cette figure, le *Bateau* est une extension de la classe *Véhicule* qui est alors réutilisée. Toutefois, le processus probable de conception de ce diagramme démontre l'inverse. En effet, au départ, il n'existait probablement que la classe *Bateau*. Par la suite, une classe *Avion* s'est ajoutée. Ce n'est probablement que lorsque la classe *Automobile* arrive que l'architecte crée la classe de base *Véhicule*. Il ne s'agit donc pas d'une réutilisation et d'une extension mais plutôt d'un réusinage.

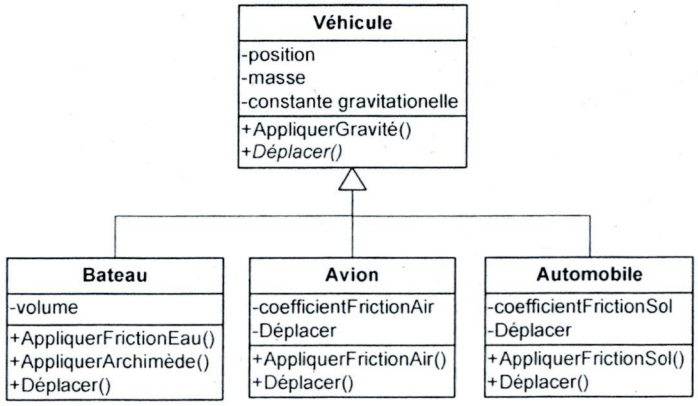

Figure 3.8 : Exemple d'héritage facilitant d'emblée la réutilisation mais provenant, le plus souvent, d'un long processus de remodelage.

Problèmes de composition et d'intégration

La composition dans des MVs basés sur l'OO s'avère difficile car les diagrammes de classes sont souvent conçus avec des objectifs de réutilisations et d'extensions différents. Les problèmes qui en découlent sont nombreux : *problème des classes de base fragiles* (Mikhajlov & Sekerinski, 1998), *incompatibilité de granularité des concepts exprimés par les classes* (Stefik & Bobrow, 1985) et réutilisation excessive de superclasses. De plus, les concepteurs des diagrammes à composer restent essentiels à toute composition. Ces problèmes sont présentés dans l'exemple qui suit.

> Deux diagrammes de classes, illustrés à la figure 3.9, ont été conçus séparément par des scénaristes. Chaque diagramme représente un MV. Le diagramme de la figure 3.9 a) met l'accent sur les environnements air, mer et sol dans lesquels évoluent des véhicules. Le diagramme de la figure 3.9 b) utilise une classification basée sur la capacité d'un corps à bouger dans l'espace et d'entrer en collision avec les autres corps. Ces diagrammes montrent déjà de la réutilisation et de l'extension avec l'héritage. Toutefois, comme pour l'exemple précédent, cette réutilisation et cette extension ont lieu *a posteriori*.

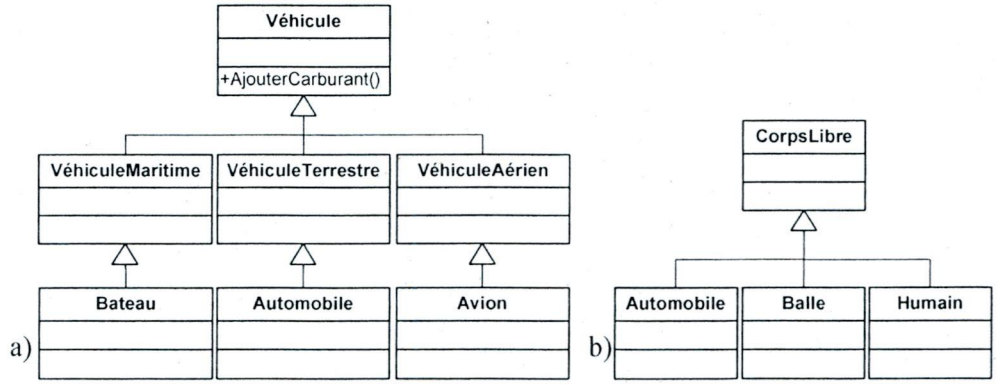

Figure 3.9 : Diagrammes de classes d'abstraction créées par des scénaristes différents et pour des MVs différents mettant l'accent sur : a) le milieu de déplacement des véhicules et b) la capacité des entités à bouger dans l'espace.

La figure 3.10 montre un résultat possible de composition de ces deux diagrammes à des fins de réutilisation et d'extension des concepts *air* et *en mouvement*..

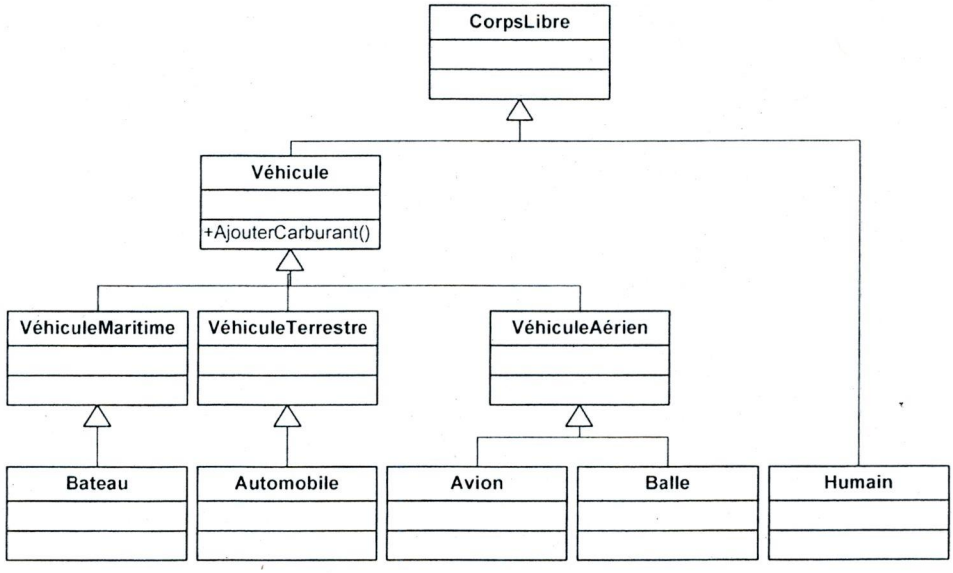

Figure 3.10 : Exemple de classe de base fragile démontrée par la classe *CorpsLibre* promue au rang de superclasse lors de la composition des deux diagrammes de la figure 3.9.

Premièrement, le diagramme intégré repose maintenant sur la classe de base *CorpsLibre*. Ce changement dans le diagramme de classe implique que la classe *Véhicule* dérive désormais de la classe *CorpsLibre*. Une telle modification du diagramme implique un remodelage de la classe *Véhicule*. Ce problème de remodelage d'une classe de base se

nomme le *problème des classes de base fragiles*. En effet, modifier une classe de base aura beaucoup plus de conséquences parce que les autres classes en dépendent.

Un autre problème survient lors de la promotion d'une classe (Stefik & Bobrow, 1985) pour régler un conflit d'héritage. Dans cet exemple, la classe *CorpsLibre* est promue afin de permettre l'intégration des deux diagrammes. Dans ce cas-ci, la promotion ne semble pas causer de problème. Toutefois, dans d'autres cas, il peut survenir des écarts sémantiques entre la classe de base et la classe dérivée. Ce problème est désigné *incompatibilité de granularité des concepts exprimés par les classes*, ou *grainsize conflict*. Il survient entre les classes dérivées et le parent qui a été promu à un niveau trop élevé dans la hiérarchie. Par conséquent, l'extension future sera plus difficile à cause de l'incompatibilité de granularité entre cette classe et les autres classes du même niveau dans l'arbre d'héritage.

Finalement, la réutilisation abusive de fonctionnalités pour la classe *Balle* crée un autre problème. Dans le diagramme intégré, le scénariste a pris la liberté d'étendre cette classe en lui ajoutant une capacité de dynamique de vol en la dérivant de la classe *VéhiculeAérien*. Cette extension peut toutefois mener à des situations étranges. Dans ce cas, il serait possible d'ajouter du carburant à la balle et celle-ci pourrait se déplacer d'elle-même. La réutilisation excessive peut mener à des situations non prévisibles ou incongrues.

Problèmes de réutilisation

Quoique l'héritage facilite la réutilisation dans certains cas, seule la classe de base est réutilisable. Il faut se rappeler que l'héritage utilise le concept de la programmation par différence (Johnson & Foote, 1988). En effet, une classe spécialisée ajoute toujours des concepts par rapport à une classe de base. Toutefois, la partie ajoutée sera difficilement réutilisable. Le prochain exemple illustre ce problème.

Une classe *Chaise* et une classe *Table* sont initialement définies dans un MV, tel qu'illustré à la figure 3.11. Si, à un moment donné, le scénariste a besoin d'une chaise à roulette, son réflexe sera de se tourner vers la classe *Chaise* et de lui ajouter des roulettes. Cette chaise à roulette ne contient que des roulettes de plus que la chaise mais elle repose sur une connaissance intime de la classe *Chaise*. Si, ultérieurement, le même scénariste veut créer une table à roulettes, il ne peut se servir des roulettes de la classe *ChaiseàRoulettes*. Le problème est causé par le couplage fort entre une classe de base et celle spécialisée qui nuit à la réutilisation de la classe spécialisée. En conséquence, la réutilisation et l'extension sont encore une fois limitées par l'héritage.

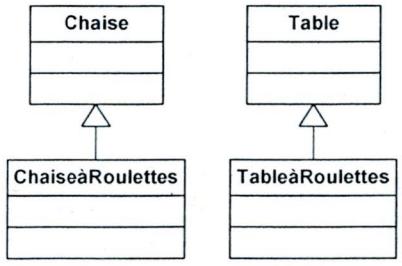

Figure 3.11 : Exemple d'une réutilisation d'une classe enfant (*ChaiseàRoulettes*) avec une table qui s'avère difficile avec l'héritage.

3.2.6 Structure centrée sur les graphes relationnels

Dans cette structure, des entités sont reliées entre elles par des liens. La structure utilisée dans URBI & ORBI s'approche des graphes relationnels : la nature des entités est imposée mais toutes les relations sont permises entre les entités. On se sert ici de cette liberté, par exemple, pour reproduire les *locales* et gérer les entités dites décoratrices auxquelles sont allouées moins de ressources. De façon similaire, BrickNet utilise les relations sémantiques pour relier les entités entre elles. Cette structure permet de reproduire des comportements environnementaux similaires à la structure orientée environnement hiérarchique.

Finalement, Daubrenet, S., Pettifer, S., & West (2000) discutent du rôle des graphes relationnels dans les MVs. Ils s'intéressent à des questions de conséquences de relations sur la cohérence du MV. Par exemple, si un groupe d'entités est qualifié à la fois de *Bois* et de *Chaise*, est-ce que, par exemple, le *Dossier* peut être qualifié de *Bois* ? Dans ce cas, l'héritage de qualificatif au *Dossier* ne pose pas trop de problème alors que l'héritage du qualificatif *Chaise* ne s'applique pas au *Dossier*. Ce problème devient important lorsque les entités sont modifiées en cours d'exécution. Évidemment, dans l'exemple précédent, la nature de la relation évite d'attribuer faussement un qualificatif d'une entité à une autre.

Dans le cas des graphes relationnels, le gestionnaire du MV peut connaître l'ensemble possible des relations. Conséquemment, il peut être programmé pour en tenir compte. Par exemple, un gestionnaire sait que, lorsqu'une entité est incluse dans un environnement, elle doit en subir les comportements. Toutefois, l'extension d'un tel MV en cours d'exécution avec une nouvelle relation s'avère difficile car il faut alors reprogrammer le gestionnaire pour qu'il

en tienne compte. Les approches précédentes sont soumises à cette contrainte.

Dans le cas où des relations peuvent être de n'importe quel type, l'extension de ce MV avec des relations peut sembler facile. Quoiqu'une nouvelle relation puisse être ajoutée à un MV, le sens et l'effet de cette relation n'est pas connu. Par exemple, un MV inclus une entité dans un environnement eau. Même s'il existe un modèle de viscosité dans l'eau, il est impossible de savoir, pour le MV en cours d'exécution, qu'il faut appliquer la viscosité sur cette entité. Pour permettre à un gestionnaire d'accomplir cette tâche, il faut définir des directives lorsqu'une relation est ajoutée. Un méta-modèle conceptuel peut contraindre la définition de telles directives lors de la conception d'un modèle conceptuel. Le gestionnaire étant conçu avec la connaissance de ce méta-modèle conceptuel, il pourra interpréter ces directives. Celles-ci sont définies par rapport aux éléments disponibles. Les MVs présentés dans ce chapitre n'utilisent toutefois pas cette stratégie. BrickNet, qui fait partie de l'approche précédente, définit cependant un mécanisme équivalent : une entité peut déclarer un comportement qui s'active lorsque certaines entités possèdent d'autres entités selon une configuration spécifique. Par exemple, lorsqu'un robot sera composé de deux jambes, un gestionnaire devra activer son comportement de marche. Dans cet exemple, le gestionnaire ne comprend pas le sens de *Jambe*, de *Composé* ou de *Marcher*. Toutefois, il comprend que si une entité est reliée à n entités de type X par une relation Y, alors il faut activer le comportement Z. Il surveillera constamment si cet énoncé est vrai et prendra les actions nécessaires. Cet énoncé, présent dans le modèle conceptuel sous forme de règle, dicte au gestionnaire comment réagir à chaque extension, modification ou composition. Si cette règle respecte la norme du méta-modèle conceptuel, le gestionnaire est en mesure de la comprendre et de l'appliquer afin d'assurer le bon fonctionnement et la cohérence du MV. Dans cet exemple, il devra intervenir chaque fois qu'une action liée à une règle est effectuée. Il n'a pas besoin d'intervenir lorsqu'un attribut est modifié car il ne possède aucune directive en rapport à cette action. Toutefois, si une relation de type Y est ajoutée, il doit vérifier la règle.

Dans cet exemple, un métalangage de définition de règles permet de gérer l'extension et la composition d'un MV avec de nouvelles relations. Tel que mentionné au début de la section, de telles règles peuvent être ajoutées au modèle conceptuel. L'analyse des règles à chaque modification du MV assure une cohérence et un respect du modèle conceptuel,

permettant ainsi une plus grande *agilité*. En conséquence, il est possible de déduire qu'un gestionnaire paramétrisable pouvant interpréter le sens des règles facilite l'*agilité* et sa gestion. L'architecture proposée dans cette thèse reprend donc cette idée. Un gestionnaire devra constamment vérifier le respect des règles et agir selon les instructions fournies.

3.2.7 Structure centrée sur le concept OMT (High Level Architecture)

Tel que mentionné au début de ce chapitre, HLA prend son origine dans le domaine militaire comme standard visant à accroître l'interopérabilité entre différents simulateurs. Cette section résume HLA sous une perspective de méta-modèle conceptuel dans la mise en place de MVs *agiles*.

Dans HLA, l'élaboration d'une simulation s'effectue par l'interconnexion de plusieurs simulateurs, dits *fédérés*. Ce réseau de fédérés se nomme *fédération*. Au point de vue du déploiement et de l'implantation, chaque simulateur est encapsulé dans un fédéré. Ceux-ci sont généralement conçus par des groupes distincts de spécialistes qui se concentrent sur une entité ou une sous-entité. L'interconnexion des fédérés est rendue possible par le *Run-Time Infrastructure* (RTI) et par une méthode de conception de simulation standardisée.

Comme le montre la figure 3.12, le RTI est responsable de transmettre les informations d'un fédéré à l'autre. Afin de maintenir la simulation cohérente, les fédérés doivent toujours communiquer entre eux par le RTI.

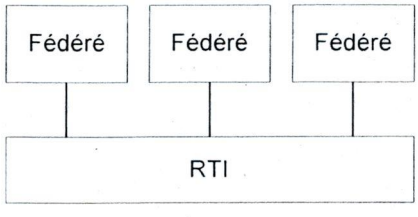

Figure 3.12 : Architecture de communication entre les fédérés.

La conception d'une simulation dans HLA comporte plusieurs étapes. Elle débute avec la définition d'un *Federation Object Model* (FOM), une spécification des données communes aux participants d'une fédération. Pour être valide, le FOM doit respecter le *Object Model*

Template (OMT). Les principales composantes de l'OMT sont les classes d'objets, synonyme d'entités, et les classes d'interactions. Celles-ci s'apparentent aux classes de l'OO parce qu'ils contiennent :

- une identité qui la distingue d'une autre entité ;
- des attributs qui la qualifient et la quantifient et qui sont accessibles par les autres entités ;
- des relations statiques (association, spécialisation, ...) ; et
- des relations dynamiques.

Toutefois, dans l'OMT, les entités n'ont pas de comportements et les attributs de classes ne correspondent pas à ceux de l'OO. De plus, quoique les entités puissent dériver les unes des autres, elles ne supportent pas l'héritage multiple. Comparativement à l'OO, l'OMT définit également des classes d'interaction à échanger sous forme d'événements entre les entités.

La deuxième étape pour concevoir une simulation consiste à définir un *Simulation Object Model* (SOM). Ce SOM est une spécification des entités et des interactions supportées par un fédéré. Tous les éléments mentionnés dans le SOM doivent se retrouver dans le FOM afin que le RTI puisse acheminer l'information aux bons fédérés.

Comme pour la plupart des autres approches, HLA utilise un paradigme inspiré de l'orienté entité. En effet, chaque entité contient les comportements qui la modifient et la fait évoluer dans le temps. Par conséquent, la plupart des comportements ne peuvent être réutilisés sans l'entité car ils ne sont pas connus et compris par l'architecture. Toutefois, il serait possible d'étendre un MV en cours d'exécution avec un nouveau comportement si celui-ci était encapsulé seul dans un fédéré. Il s'agit toutefois d'une utilisation inhabituelle et peu efficace.

HLA autorise la définition de relations entre entités. Toutefois, sauf pour la spécialisation, le RTI ne prend pas en charge les relations. Par conséquent, l'ajout ou la modification de relations doit être gérée par le scénariste lui-même, ce qui n'est pas le but recherché dans cette thèse.

HLA possède de nombreux mécanismes pour gérer la cohérence de la simulation. Parmi ceux-ci, il y a le FOM qui permet d'identifier toutes les communications entre les

fédérés. Par exemple, un MV est conçu par l'interconnexion de fédérés. Le FOM de ce MV contient une entité *Véhicule* qui possède les attributs *Position*, *Masse* et *Géométrie*. Initialement, tous les fédérés ne sont intéressés qu'à la position pour détecter les collisions avec les autres véhicules. En cours d'exécution, un scénariste décide d'étendre ce MV en créant une nouvelle entité *VéhiculeRadar* qui dérive de *Véhicule*. Le scénariste doit alors arrêter le MV et ajouter l'entité *VéhiculeRadar* au FOM comme une sous-classe de *Véhicule*. Il conçoit ensuite un fédéré qui contient l'entité *VéhiculeRadar* ainsi qu'un modèle de radar. Comme le nouveau modèle n'a besoin que de la géométrie et de la position de toutes les entités *Véhicules* et que cette information est déjà disponible dans tous les véhicules, cette extension ne nécessite toutefois pas la modification des fédérés existants. Il n'a ensuite qu'à redémarrer la simulation. Ce type d'extension reste toutefois limité et impossible en cours d'exécution.

Actuellement, une entité HLA ne peut dériver que d'une seule autre entité. Cette limitation n'est pas souhaitable dans les MVs car les entités virtuelles peuvent jouer plusieurs rôles simultanément. Dans l'exemple précédent, le *Radar* pourrait désirer de ne voir que certains sous-types de véhicules, comme par exemple des véhicules terrestres. Dans la classification de l'exemple précédent (*Véhicule* - *VéhiculeRadar*), il n'existe pas de tel type de véhicules. Par conséquent, il faudrait redéfinir toute la hiérarchie et modifier tous les fédérés en reclassant chacun des véhicules déjà présents. La modifiabilité avec une approche HLA est, par conséquent, limitée lorsqu'il est question de remettre en cause des classifications déjà établies comme pour l'OO.

Dans HLA, l'implantation des entités passe par les fédérés. En général, la granularité des fédérés est relativement grossière. En effet, chaque fédéré représente habituellement un véhicule de combat comme un blindé, un avion ou un bateau. Il a été mentionné au chapitre précédent que la granularité se doit d'être fine. Quoique la granularité d'un fédéré puisse être diminuée, le niveau de complexité pour rendre un fédéré à granularité fine conforme aux spécifications HLA est trop élevé.

Lorsqu'un fédéré s'ajoute dans une simulation, il ne reçoit pas automatiquement les attributs des entités pour lesquelles il a indiqué un intérêt dans son SOM. Ce problème est causé par l'approche centrée sur le fédéré et centrée sur l'entité. En effet, un modèle ou une interaction s'applique souvent entre les entités. Si un fédéré, qui contient un modèle de radar,

est ajouté à une fédération, il faudra qu'il demande les positions des entités qu'il peut capter. Par conséquent, l'extension demandera plus d'efforts.

HLA possède un service d'optimisation de l'aiguillage[1] de la transmission des attributs et des interactions. Ce service, nommé DDM pour *Data Distribution Management*, permet aux fédérés de définir des espaces d'aiguillage et d'associer la transmission ou non d'attributs et d'interactions d'un fédéré à l'autre selon certaines conditions. Quoique la modification soit possible, l'extension en genre des conditions d'aiguillage n'est pas possible en cours d'exécution. Premièrement, ce service nécessite la coordination des fédérés afin d'établir un espace d'aiguillage. Il faut donc recompiler au moins un autre fédéré afin que cet ajout soit effectif. Deuxièmement, HLA oblige que ces espaces d'aiguillage soient inscrits dans le FOM qui n'est chargé qu'au début de la simulation.

Comme pour les approches précédentes, les principaux problèmes de HLA nuisant à l'*agilité* proviennent d'une modélisation centrée sur l'entité. Par exemple dans HLA, la détection radar serait modélisée dans le véhicule radar. Toutefois, la détection radar des entités implique autant la capacité du radar de capter des ondes électromagnétiques d'une certaine longueur d'onde que celle des entités de réfléchir les ondes électromagnétiques. Cette interaction entre les entités est omniprésente dans les MVs. Elle implique la coordination constante entre l'élément ajouté, réutilisé ou modifié et sa paire correspondante. Par exemple, si certaines conditions atmosphériques changent, tant la capacité d'une entité de se faire voir par un radar que celle d'une entité de voir avec un radar sont modifiées. Dans ce cas, il faudrait probablement modifier les deux entités, donc deux fédérés. L'extension nécessiterait une modification des éléments déjà présents et, par conséquent, l'arrêt du MV. De plus, des tierces parties en charge de ces fédérés devraient être recontactées et remodèleraient le fédéré.

1. L'aiguillage se définit comme le processus qui achemine l'événement au(x) bon(s) destinataire(s). L'aiguilleur possède une table d'aiguillage qui contient la liste des destinataires pour un événement donnée. Ex.: L'interface usager de Windows et le VRML l'utilisent.

3.3 Rôle des éléments de base sur l'agilité

L'*agilité* est influencée par les différentes approches de conception des éléments. Afin d'identifier des pistes de solution qui aideront à accroître l'*agilité*, cette section présente une analyse de l'entité, de l'attribut et du comportement. Ils sont de bons candidats pour devenir les éléments de base du méta-modèle conceptuel recherché. Une revue des différentes approches retrouvées dans les MVs permettra d'identifier les problèmes et les meilleures stratégies pour chacun de ces éléments puis de les mettre en relation. De plus, cette analyse souligne les stratégies employées pour gérer la cohérence de MVs. Chaque section conclue avec l'approche pouvant augmenter et gérer l'*agilité* et, par conséquent, celle qui sera retenue pour bâtir l'architecture proposée dans cette thèse. Certaines sections traitent de différentes approches selon une perspective d'implantation car elles permettront de dégager des concepts liés à l'*agilité*.

3.3.1 Approches de représentation et de conception de l'entité

L'entité est la base du MV. Cette section présente les différents niveaux d'*agilité* des MVs revus dans la représentation et la conception des entités.

- **L'entité n'est pas modifiable ou extensible car toutes ses caractéristiques sont déjà définies.**

  Dans cette approche, l'ensemble des entités est déjà défini sous forme de code binaire. Certains types de MVs comme ceux persistants sur l'internet ainsi que les jeux 3D en réseaux prédéfinissent déjà l'ensemble des types d'entités. Par conséquent, ces MVs ne permettent d'instancier que les entités déjà définies avec des valeurs différentes pour les attributs. Il s'agit là d'une approche trop contraignante pour le niveau d'*agilité* visé dans la présente thèse.

- **L'entité est conçue dans un bloc monolithique (fichier binaire ou fichier textuel scripté).**

  Dans certains MVs, les entités se retrouvent définies dans un élément qui encapsule les attributs et les comportements. Les interactions entre ces entités sont déjà fixées car l'interaction avec d'autres entités s'effectue directement dans le langage de programmation ou dans un langage scripté. Ce manque d'*agilité* n'est pas dû à l'utilisation de compilateurs ou d'interpréteurs de code mais plutôt à la

prédéfinition des possibilités d'interactions du MV. La communication entre les entités peut aussi s'effectuer indirectement. Lorsque l'entité sera compilée, sa définition sera généralement séparée de son implantation. Cette méthode, combinée avec un mécanisme d'aiguillage d'événements, facilite la communication entre les entités grâce au mécanisme d'indirection qu'introduisent les événements. Toutefois, elle implique la prédéfinition de l'ensemble des événements qui seront chargés au démarrage. La nature des interactions entre les entités est alors limitée à l'ensemble des événements prédéfinis.

- **L'entité est virtuelle. Elle sert à contenir des attributs et différents comportements peuvent lui être associés.**

  Une entité ne peut être qu'un amalgame "virtuel" d'attributs dans lequel des comportements sont placés. VEOS utilise ce principe qui permet, entre autres, d'assigner un même attribut à plusieurs entités. Cette méthode permet la création de nouvelles entités temporaires en cours d'exécution. En effet, le MV peut créer des entités selon des *pattern matching* de recherche dans d'autres entités et leurs caractéristiques comme des attributs. Par exemple, une entité *TêtedeMarteau* est attachée à une autre entité *ManchedeMarteau*. Une entité virtuelle peut chercher un marteau et le système lui retournera une nouvelle entité constituée des attributs du manche et de la tête du marteau. Quoique cette méthode soit flexible, elle rend difficile la gestion de l'intégrité de l'entité.

La dernière approche sera reprise dans l'architecture car elle favorise l'*agilité*. Des mécanismes de gestion de la cohérence du MV devront toutefois être trouvés.

3.3.2 Approches de représentation et de conception de l'attribut

Une entité possède des attributs qui la qualifient et la quantifient. L'étude de plusieurs architectures a permis d'identifier une classification et l'information fournie pour les attributs ainsi que le moment et la technique d'association des attributs à l'entité.

Classification et information sur les attributs

Les attributs peuvent tous être du même type ou classés dans différentes catégories. Plusieurs architectures proposent déjà un certain nombre d'attributs pré catégorisés qui maintiennent la cohérence du MV. Toutefois, les attributs qui peuvent être définis par les

scénaristes se retrouvent, le plus souvent, sous la désignation *user-defined*.

Quelques autres architectures se distinguent en obligeant ou en gérant elles-mêmes la classification des attributs. Par exemple, VEOS prend en charge la classification des attributs en cours d'exécution dans des catégories : *bondary* contient les attributs de l'entité servant à communiquer avec son environnement, *external* contient de l'information à propos des autres entités qu'une entité perçoit et *internal* contient les attributs d'entités incluses lorsque celle-ci agit en tant qu'environnement. DEVA 3 utilise aussi cette stratégie car tous les attributs doivent être classés dans les catégories *enforced*, *innate* ou *imbued*.

Outre la classification, il est possible d'assigner de l'information supplémentaire à l'attribut. Ces informations, qui portent parfois le nom de *policies*, restreignent, par exemple, l'accès en lecture.

Il n'existe aucune classification qui semble augmenter l'*agilité* des MVs. Certaines informations ou *policies* à propos des attributs peuvent toutefois faciliter leur accès par les comportements et aider à la gestion du MV. Elles seront donc employées dans l'architecture.

Moment et technique d'association des attributs à l'entité

Les différents moments et techniques pour définir et assigner un attribut à une entité influencent l'*agilité*. L'énumération qui suit met en relation les techniques de création et d'instanciation des attributs pour étendre les MVs en cours d'exécution.

- **Aucune assignation possible de nouveaux attributs à l'entité**

  Dans ce cas, les attributs que possède l'entité sont déjà définis et il est impossible d'en ajouter comme dans la plupart des MVs de jeux 3D. Cette approche ne s'avère aucunement *agile* mais permet d'optimiser le système, de le rendre plus robuste et de prendre en considération les particularités de chacun des attributs.

- **Définition de nouveaux attributs et ajout aux entités avant l'exécution.**

  Une amélioration du cas précédent consiste à rendre possible la création de nouveaux attributs avant le début de l'exécution afin qu'ils soient compilés au sein de l'entité. MASSIVE-3, EM (Wang *et al.*, 1995) et DEVA 3 et URBI & ORBI se classent dans cette catégorie.

- **Définition de nouveaux attributs avant l'exécution et ajout aux entités en cours d'exécution.**

  Certaines architectures supportent l'ajout d'attributs en cours d'exécution même si les attributs doivent toujours être créés avant l'exécution. Pour permettre l'ajout d'un attribut à une entité en cours d'exécution, il existe deux possibilités d'implantation : créer un attribut séparé de l'entité, le compiler et le charger avec des mécanismes de librairies dynamiques comme les DLL ou utiliser des langages scriptés comme le Java (Arnold *et al.*, 2000), le TCL (Ousterhout, 1994), Goal (Fabre *et al.*, 2000) et l'OML (Green, 1994) qui émuleront l'attribut. La compilation, une étape qui vient limiter l'*agilité*, s'évite par l'utilisation d'attributs interprétés. Les architectures revues dans cette thèse qui accèdent aux attributs interprétés utilisent du code également interprété.

- **Création d'attributs et ajout aux entités en cours d'exécution.**

  Le cas le plus *agile* permet la définition et l'instanciation d'un attribut dans un MV en cours d'exécution. Cette approche impose des contraintes au format de définition, de chargement et d'instanciation. La définition de chaque attribut doit être séparée de la définition des autres attributs pour permettre son ajout individuel dans le MV. Par conséquent, la définition d'un seul attribut peut se retrouver dans un fichier textuel ou binaire. L'implantation de l'attribut peut être interprété ou compilé. VEOS est le plus avancé en ce qui a trait aux attributs. Il utilise des attributs scriptés pouvant être chargés et associés aux entités en cours d'exécution selon des patrons comme des *match* partiels, des *unbound attibute values*, des similarités et des analogies.

En résumé, la dernière approche est la plus souhaitable, ce que permettent les attributs basés sur des scripts qui facilitent la définition et ne requirent aucune implantation binaire. La granularité demeure toutefois le critère dont il faudra tenir le plus compte : granularité de définition, granularité de chargement et granularité d'attribution à l'entité.

3.3.3 Approches de représentation et de conception des comportements

Les comportements animent les MVs. Ils sont parfois appelés fonctions ou méthodes. Ils effectuent plusieurs types de modifications à l'intérieur du MV comme changer la valeur des attributs. Les comportements peuvent être analysés selon les approches de création et définition des comportements, d'exécution des comportements et, finalement, du temps et de la technique d'association des comportements à l'entité.

Création ou définition des comportements

Quatre approches de création ou de définition de comportements ont été identifiées.

- **Une ou des fonctions ou méthodes prédéfinies (*built-in*).**

  Les MVs contiennent ou des comportements pouvant être assignés aux entités. Ces comportements prédéfinis peuvent être facilement testés pour s'assurer du bon fonctionnement et de la cohérence du MV dans toutes les conditions car toutes les résultantes du comportement, quoique possiblement nombreuses, sont connues. Cette approche empêche l'extensibilité car, dès que le MV est développé, il est impossible d'y ajouter de nouveaux comportements.

- **Une méthode ou fonction assignée à une entité.**

  Les comportements s'implantent fréquemment sous une méthode ou fonction appelée périodiquement ou lorsqu'un événement est lancé. VRML utilise cette approche lors de chaque boucle de rendu avec le traitement, en cascade, des événements qui mènent à l'appel de la méthode Java *processEvent*. Si le MV avait pris lui-même en charge l'appel de la bonne méthode selon l'événement, alors VRML aurait été classé dans la prochaine approche. BrickNet exploite la même technique avec la méthode *animate*, écrit en langage Starship (Loo, 1991). Plusieurs librairies graphiques entrent également dans cette catégorie. Quoique qu'elle soit rapide à implanter, cette approche ne permet pas l'assignation de plusieurs comportements à une même entité, laisse trop de liberté qui nuit à la gestion de la cohérence et impose une charge de travail trop importante lors de la conception des comportements et des entités.

- **Plusieurs méthodes ou fonctions assignées à une entité.**

  Cette approche est très répandue car plusieurs architectures supportent l'association d'un segment de code pour chaque type d'événement reçu. Par conséquent, l'architecture prend en charge l'appel des différentes méthodes ou fonction contenues dans les entités. NPSNET-V, avec l'utilisation de *protocols*, permet l'ajout de comportements à une même entité qui seront activés selon les événements. Bamboo, avec l'association événement-procédure de rappel, VEOS avec l'association événement-section de code en LISP et EM avec l'association événement-méthode utilisent cette technique. Elle permet la conception de comportements relativement indépendants des entités. Toutefois, cette approche centre le comportement sur l'entité et ne facilite pas la gestion de la composition de comportements dans un MV *agile*.

- **Plusieurs méthodes ou fonctions entrant dans une classification.**

  Certaines architectures proposent des catégories de comportements afin d'imposer certaines contraintes lors de l'exécution ou pour faciliter l'exécution. Pour ce faire, l'architecture utilise des méthodes dans lesquelles le comportement doit être classé ou des méthodes virtuelles pures à définir obligatoirement. Par exemple, VEOS impose trois catégories de comportements : *react, interact* et *persist*. DEVA 3 classe les comportements selon *enforced* (obligatoires), *innate* (interne) et *imbued* (par défaut). Une série d'événements peuvent être également prédéfinis afin de s'assurer que les entités et le MV possèdent un minimum de caractéristiques désirées de l'usager. HLA, SIMNET et DIS se classent dans cette catégorie. Toutefois, HLA apporte une amélioration par rapport aux deux autres car il permet aux scénaristes de définir l'ensemble des événements possibles avec le FOM.

Toutes ces approches sont centrées sur l'entité. Quoique la dernière permet la gestion d'une certaine cohérence et permet des modifications en cours d'exécution, aucune ne constitue une approche satisfaisante. D'autres approches devront être trouvées.

Exécution des comportements

L'exécution ou l'appel des comportements implique presque toujours l'exécution d'un segment de code prédéfini ou pouvant être défini ultérieurement. La figure 3.13 illustre les différentes techniques d'exécution retrouvées dans les architectures incluses dans cette revue.

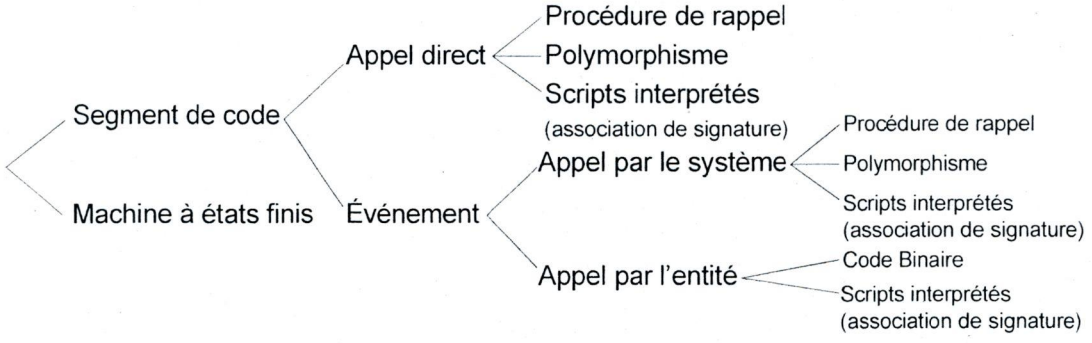

Figure 3.13 : Techniques d'exécution des comportements dans des MVs.

DIVE utilise les machines à états qui modifient les attributs d'une entité lorsqu'un

certain événement est lancé. Les autres architectures utilisent des segments de code pour exécuter les comportements et, donc, modifier les attributs. L'appel direct dans le langage de programmation ou par les événements sont les deux mécanismes possibles pour exécuter ces segments de code. L'appel direct peut prendre la forme d'une procédure de rappel avec des pointeurs de fonctions comme dans MAVERIK, de méthodes surchargées appelées par polymorphisme comme dans VRML ou par l'interprétation de scripts d'une signature donnée comme dans BrickNet. DEVA 3 utilise une technique hybride de polymorphisme et de procédures de rappel. L'utilisation d'événements permet également d'exécuter des comportements. Ils facilitent l'appel des comportements car ils s'adaptent bien aux appels asynchrones. L'appel de méthodes de façon asynchrone directement dans le langage est également possible mais implique des artifices complexes comme les *Asynchronous Method Invocation, AMI* (Object Management Group, 1998) que l'on retrouve dans le Common Object Request Broker Architecture, ou CORBA (Henning & Vinoski, 1999). L'architecture prend toujours en charge l'aiguillage de l'événement mais l'appel du segment de code du comportement peut être effectué par l'entité ou par le système d'exploitation lui-même. Dans ce dernier cas, le MV n'a besoin de connaître que le destinataire de l'événement et non pas le type de message. Bamboo utilise les événements pour lancer l'appel des procédures de rappel.

En résumé, l'utilisation d'événements est beaucoup plus fréquente car elle offre une plus grande souplesse d'association cause-effet. En effet, un événement peut créer une cascade d'autres événements dont la gestion est plus simple qu'avec des appels directs. Par exemple, des boucles peuvent être plus facilement détectées. Toutefois, cette gestion devrait être cachée au scénariste et celui-ci ne devrait se concentrer que sur la modification des attributs des entités. De plus, les événements correspondent mieux au paradigme utilisé dans le domaine de la simulation avec les DEVS (Zeigler *et al.*, 2000) et les *Distributed Discrete Event Simulation* (Fujimoto, 2000). Par conséquent, les systèmes asynchrones événementiels constituent la meilleure façon d'activer les comportements dans les MVs *agiles*. L'architecture proposée emploiera donc les systèmes asynchrones événementiels.

Temps et technique d'ajout des comportements

L'ajout d'un comportement est possible, comme pour l'attribut, soit avant l'exécution

soit en cours d'exécution. L'ajout avant l'exécution présente peu d'intérêt pour cette thèse. Quant à l'ajout d'un comportement pendant l'exécution, il est supporté par plusieurs architectures telles Massive 3 et DEVA 3, et est rendu possible par des langages scriptés tel que Goal ou EM. Des comportements compilés peuvent également être ajoutés en cours d'exécution en utilisant des procédures de rappel. Cette dernière approche, retenue pour cette thèse, s'avère la plus rapide et favorise la réutilisation de nombreux modèles existants, un des objectifs visés.

3.4 Ordonnancement de MVs agiles

L'architecture proposée devra gérer l'avancement du temps dans des MVs *agiles*. Il existe de nombreuses approches d'ordonnancement. Certaines, plus dynamiques que d'autres, appartiennent au domaine des systèmes d'exploitation ou d'aiguillage réseau. Elles reposent toutefois sur la gestion de files multiples de tâches, ce qui ne semble guère approprié aux besoins des MVs.

Les MVs et le domaine de la simulation utilisent de nombreux autres types d'ordonnanceurs génériques (Fujimoto, 2000) : boucles (*time-stepped* ou *frame-based*), événements (*event-driven* ou *event oriented*) ou les deux combinés. Dans l'approche par boucle, l'ordonnanceur appelle tous les éléments présents dans le MV et recommence ainsi de suite dans une boucle infinie. Parfois, les ordonnanceurs doivent suivre un ordre d'appel défini par la structure du MV, tel un graphe de scène. Certains ordonnanceurs comme ceux rencontrés dans les graphes de scène appellent chaque noeud du graphe de scène selon un ordre précis, ce qui s'avère inefficace.

L'ordonnancement par boucle ou hybride est trop limitatif pour le niveau d'*agilité* recherché dans le MV car il impose un incrément temporel variable en synchronisation avec le rendu. Il est alors difficile de contrôler l'avancement du temps. La modélisation de certains comportements physiques implique le contrôle fin de l'avancement du temps. Cette approche ne sera pas retenue.

Les approches par événements discrets sont plus flexibles que les boucles ou les approches hybrides parce qu'elles sont indépendantes des tâches à exécuter. Les événements ont deux attributs principaux : un temps et une valeur spécifique à l'application. Cette approche

est à la base des simulations à événements discrets. Pour des raisons de complexité, elle n'est toutefois pas employée telle quelle. Elle est toujours encapsulée dans une couche de plus haut niveau. L'exemple le plus connu est l'orienté processus, où chaque processus logique représente un modèle d'une entité ou d'un comportement. L'orienté processus est employée dans HLA. Il a le désavantage de ne pas pouvoir automatiser suffisamment la gestion des événements d'un point de vue scénariste, tel qu'il serait requis dans des MVs *agiles*. De plus, la granularité des processus logiques est relativement grossière.

Par conséquent, l'architecture proposée se basera sur l'ordonnancement orienté événement et lui ajoutera une couche qui publiera et consommera les événements de façon transparente pour le scénariste.

3.5 Composantes d'une architecture

La présentation des caractéristiques sous-jacentes à l'*agilité* ainsi que l'état de l'art ont permis d'identifier trois composantes essentielles pour que l'architecture retenus améliore l'*agilité* des MVs :

- un méta-modèle conceptuel ;
- un processus ;
- des gestionnaires.

La première composante, le méta-modèle conceptuel, sera centré sur l'interaction et non sur l'entité. De plus, il devra supporter autant des modélisations d'interactions dites symétriques comme la détection de collision et la friction que de modélisations asymétriques comme le lancer d'une flèche par un arc qui contient clairement un instigateur. L'entité sera virtuelle ; des attributs et des comportements pourront lui être ajoutés ou retirés en cours d'exécution. La granularité de définition, la granularité de chargement et la granularité d'attribution de l'attribut et des comportements aux entités s'avérera importante. Les comportements utiliseront les événements asynchrones. La gestion des événements sera toutefois cachée à l'usager afin que celui-ci se concentre sur la modification des attributs. Le méta-modèle conceptuel imposera un cadre de définition des règles d'intégration des parties d'un MV, permettant d'en gérer l'extension, la composition et la modification.

La deuxième composante consiste en un processus qui encadre l'utilisation du méta-

modèle conceptuel et assure l'*agilité* présente et future. Ce processus contient les actions qui concrétisent les caractéristiques définies au chapitre 2 et s'assure que les modèles conceptuels soient conformes au méta-modèle conceptuel.

Troisièmement, des gestionnaires assureront le bon fonctionnement et la cohérence du MV tout au long des modifications qui pourront y survenir. Cette thèse s'inspire de DEVA 3 qui définit un MV comme l'intégration de trois modules de gestion : gestion spatiale et graphique, exécution et répartition. Les objectifs de cette thèse et de DEVA 3 diffèrent sur plusieurs points. Par conséquent, à partir des observations précédentes et des objectifs, un MV *agile* requiert une gestion des modèles conceptuels, une gestion du temps, une gestion des ressources et une gestion de la répartition. La relation entre ces modules est illustrée à la figure 3.14.

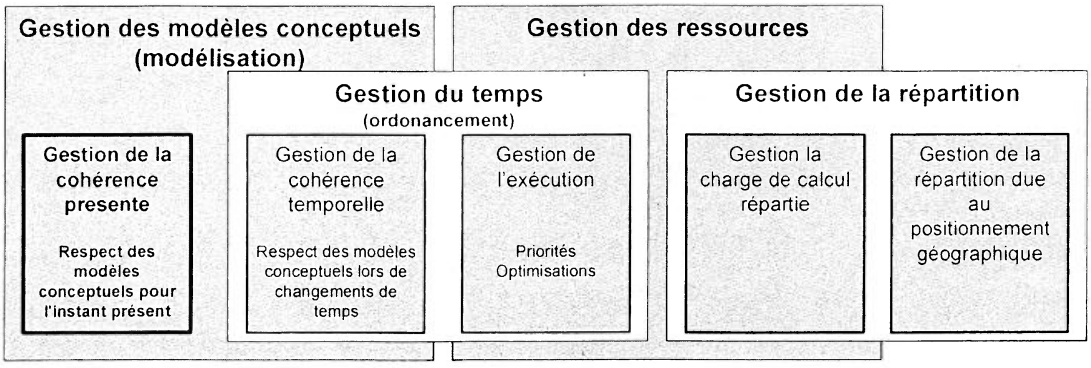

Figure 3.14 : Modules interdépendants à la base d'un MV.

La gestion des modèles conceptuels s'assure que le MV se déroule selon ce qui est décrit dans les modèles conceptuels. Cette gestion comprend deux modules. Le premier, la *gestion de la cohérence présente*, s'assure qu'à chaque instant le MV soit cohérent et que son déroulement s'effectue en conformité à ce qui est décrit dans les modèles conceptuels. Par exemple, il peut vérifier que la gravitation ne s'applique pas deux fois sur une même entité. Ce module vérifiera les règles d'intégration des parties d'un MV, donc des modèles conceptuels, afin de maintenir la cohérence. La gestion du temps comprend la *gestion de la cohérence temporelle* qui s'assure que le déroulement temporel du MV obéisse à ce qui est décrit dans les modèles conceptuels et qu'il soit cohérent d'un point de vue temporel. Elle garantit que les

modèles seront appelés dans le bon ordre et à des intervalles de temps qui garantissent leur validité. Le second module, la gestion de l'exécution permet d'optimiser l'application des modèles selon les priorités ou autres approches. Les autres modules ne seront pas considérés dans l'architecture car ils ne sont pas essentiels à l'atteinte de l'*agilité*. L'implantation, décrite à l'annexe A, propose néanmoins une ébauche de solution pour ces modules.

La *gestion de la cohérence présente* a été identifiée comme le module le plus critique afin d'améliorer l'*agilité*. La majorité des efforts porteront sur celui-ci. Le développement de la *gestion de la cohérence temporelle* et de l'exécution, ne pouvant être évité, sera juste assez élaboré afin de démontrer l'*agilité* apportée par l'architecture.

3.6 Conclusion sur l'état de l'art

Le chapitre 3 a présenté une revue des différentes architectures de MVs. Aucune n'atteint le niveau d'*agilité* visé. L'identification des avantages et limites a révélé des pistes de solution. L'analyse du rôle des éléments des MVs a permis de mieux comprendre comment ils peuvent améliorer l'*agilité*.

À partir des leçons ainsi acquises, l'architecture proposée comprendra trois composantes : un méta-modèle conceptuel, un processus et des gestionnaires. Le chapitre 4 décrit cette architecture.

Chapitre 4

Architecture Propriété Interaction Acteur (APIA)

Les chapitres précédents ont identifié plusieurs moyens afin d'améliorer l'*agilité* des MVs. La présentation détaillée des caractéristiques de l'*agilité* au chapitre 2 a permis d'identifier certaines pistes de solutions telles que décrire le MV par l'assemblage de nombreux éléments, permettre une granularité plus fine, améliorer l'*agilité* du point de vue usager et scénariste, faciliter l'ajout de comportements de façon générique et permettre la modification des éléments en cours d'exécution.

La revue de littérature et l'analyse des MVs existants selon plusieurs critères a permis également d'identifier plusieurs pistes de solutions : éviter l'héritage, exploiter la sémantique d'apportent les relations, prendre en charge l'intégration automatique de parties de modèles conceptuels et centrer la modélisation sur l'interaction.

Le constat de cette analyse permet de proposer une architecture (Bernier & Poussart, 2000 ; Bernier *et al.*, 2002), désignée Architecture Propriété Interaction Acteur (**APIA**). Cette architecture se doit d'être à la fois rigide pour empêcher des actions nuisibles et limitatives sur l'*agilité* future et suffisamment souple pour permettre des actions qui représentent elles-mêmes cette *agilité*. L'architecture doit être également en mesure de gérer cette *agilité*. Pour ce faire **APIA** repose sur trois composantes

- Éléments du méta-modèle conceptuel — Ils constituent les pièces fondamentales du MV qui seront réutilisables et modifiables. De plus, l'information de gestion de la cohérence y est incluse. Il s'agit ici des pièces de conception des modèles conceptuels. On y retrouve l'approche centrée sur l'interaction.

- Processus — L'ensemble des actions posées par les scénaristes, les usagers et les interactions sur le MV forment un processus. Ces actions concrétisent l'extensibilité, la modifiabilité, la réutilisabilité et la composabilité.
- Gestionnaires — Deux gestionnaires gèrent les actions et les éléments. Ils appliquent des algorithmes afin de maintenir le MV cohérent.

Ce chapitre décrit chacune de ces composantes. Quoique le chapitre 5 vienne démontrer l'*agilité* des MVs basés sur **APIA**, les nombreuses explications et exemples de ce chapitre en fournissent tout de même une illustration.

4.1 Éléments du méta-modèle conceptuel

Cette section présente les éléments du méta-modèle conceptuel, soit les pièces avec lesquelles le MV peut être créé. Ces éléments sont l'acteur, la propriété, le personnage, la relation et l'interaction.

4.1.1 Acteur

Le terme *acteur* se définit à tout ce qui peut être circonscrit dans l'espace cartésien ou dans un autre espace.

L'acteur est l'entité dans APIA, la "chose" à simuler. Un sous-marin, un câble, des mailles de câble, une tumeur, une souris ou un pilote virtuel en sont des exemples. La figure 4.1 illustre une représentation graphique d'un acteur.

Figure 4.1 : Représentation graphique d'un acteur.

Comparativement à une classe en OO et des entités dans la plupart des MVs, l'acteur ne possède ni attribut ni méthode lors de son instanciation. Il ne peut être spécialisé comme en

OO. Il peut, comme tous les éléments, être créé et détruit en cours d'exécution. L'acteur n'encapsule ni ne protège aucune donnée. Il contiendra des instances de propriétés, jouera des personnages et des interactions agiront sur lui. Ce sera au *gestionnaire de la cohérence présente* et *temporelle* de gérer les accès aux propriétés.

4.1.2 Propriété

> La *propriété* se définit comme une ou des données qui établit une relation d'attribution de l'acteur.

La figure 4.2 montre le symbole graphique de la propriété.

Figure 4.2 : Représentation graphique d'une propriété.

Les propriétés servent de quantitatifs ou de qualificatifs de l'acteur. La masse, la vitesse, la position, le nombre de Reynolds, le coût ou les coordonnées de la souris en sont des exemples. Comparativement aux attributs de l'OO, l'acteur n'encapsule pas de propriétés. Il agrège plutôt des instances de propriétés. Cette distinction est importante car la propriété sert à définir les personnages alors que l'instance de propriété possède une valeur et quantifie ou qualifie un acteur. De plus, l'acteur ne modifie pas les propriétés. Les interactions jouent ce rôle. D'autres détails seront donnés à la section 4.1.5 à cet égard.

Les propriétés dépendent souvent les unes des autres. Par exemple, la masse, le volume et la densité d'un acteur sont inter-reliés. D'un point de vue physique, toutes les propriétés finissent par être inter-reliées. Quoique pertinente dans certaines situations, la problématique de la gestion des dépendances entre les propriétés ne sera pas abordée.

Le temps représente une propriété particulière car il ne respecte pas entièrement la définition précédente. Dans la version actuelle, il est omniprésent et n'appartient pas à un acteur spécifique. Les interactions ne peuvent changer le temps. Le *gestionnaire de la cohérence temporelle* se charge de cette tâche.

4.1.3 Personnage

Le dictionnaire Le Petit Robert définit un personnage comme : "Chacune des personnes qui figure dans une œuvre théâtrale"[1]. La définition qui suit s'en inspire.

> Un *personnage* se définit comme l'ensemble des propriétés qui permettent à un acteur de jouer un rôle dans une interaction.

Dans l'exemple de la figure 2.2, un arc joue le rôle d'un personnage pouvant lancer des flèches et, dans la figure 2.3, ce même arc joue un personnage pouvant prendre feu parce qu'il est constitué de bois. L'acteur *Arc* joue deux rôles, donc deux personnages.

Un personnage définit les propriétés obligatoires pour qu'un acteur joue un rôle dans le MV. Il regroupe une ou des propriétés. Un acteur ne peut jouer un personnage que s'il en possède les instances de propriétés. Tel qu'illustré à la figure 4.3, des personnages différents regroupent parfois des propriétés communes. La lettre C, pour *character*, sera parfois utilisée comme abrégé.

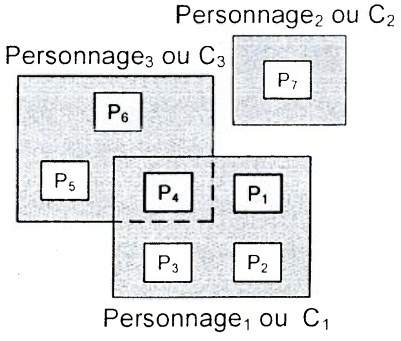

Figure 4.3 : Représentations graphiques de personnages qui regroupent certaines propriétés.

Un personnage peut grouper entre une seule propriété et l'ensemble des propriétés d'un acteur. Son nom peut donc varier entre celui normalement réservé à la propriété et celui de l'acteur. La figure 4.4 illustre la variation du nom du personnage avec le nombre de propriétés.

---

1. Rey, A, Rey-Debove, J., & Robert, P. *Le petit Robert*. Dictionnaires Le Robert. 2841 p.

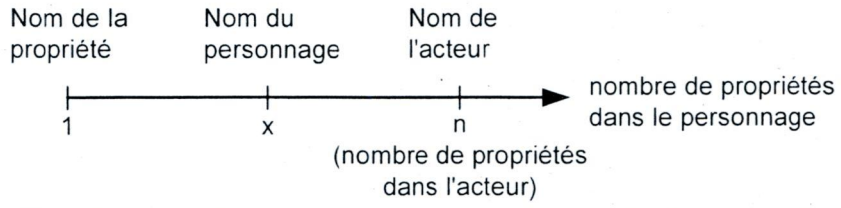

Figure 4.4 : Différentes alternatives pour nommer le personnage selon le nombre de propriétés qu'il regroupe.

L'exemple qui suit discute des différentes alternatives. De tels exemples amèneront des précisions et des explications sur les concepts tout au long de ce chapitre.

> Un scénariste désire concevoir un MV tel qu'illustré à la figure 2.5. Pour mettre le feu à un acteur, le comportement de combustion disponible requiert que celui-ci possède une masse combustible. De plus, tel que mentionné précédemment, l'acteur qui sera brûlée doit jouer un rôle, un personnage. Cette capacité de jouer un personnage peut être abordée de plusieurs façons. Premièrement, un personnage *MasseCombustible* pourrait être défini et contiendrait la propriété *MasseCombustible*. Cette approche permettrait au briquet de mettre le feu à tous les acteurs qui possèdent cette propriété. Par exemple, un arc constitué de bois et possèdant une masse combustible pourrait interagir avec le briquet. Cette façon de concevoir les personnages permet une certaine réutilisabilité, extensibilité et modifiabilité à court terme. Toutefois, comme il sera vu plus tard, un personnage du même nom que la propriété unique qu'il représente posera des problèmes pour établir les relations entre l'interaction et l'ensemble de ses personnages. À l'opposé, un personnage pourrait prendre le nom de l'acteur. En effet, s'il n'existe que des arbres dans le MV, le briquet n'a besoin que d'un personnage *Arbre*. Toutefois, ce cas ne faciliterait pas l'ajout d'un arc pouvant blûler. Entre ces deux extrêmes, il existerait un personnage *Combustible*. Ce personnage ne regrouperait que la masse combustible mais pourrait s'appliquer tant à l'arbre qu'à l'arc.

Dans certaines situations, les propriétés appartenant au personnage peuvent être optionnelles. Par exemple, pour être visible, un personnage doit posséder une géométrie dans le cas d'un objet ou une intensité de la lumière dans le cas d'une ampoule. Une seule de ces propriétés optionnelles sera suffisante pour que l'acteur soit visible, donc joue un personnage *Visible*.

4.1.4 Relation

Une *relation* se définit comme la liaison orientée et qualifiée entre deux acteurs.

La figure 4.5 illustre une relation

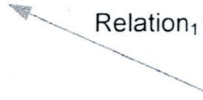

Figure 4.5 : Représentation graphique d'une relation.

La relation entre deux acteurs doit toujours être qualifiée par un nom. L'exemple qui suit présente quelques cas d'utilisation de la relation.

> Un MV est composé d'un garage, d'une porte, de bois et d'une automobile. Ces acteurs établissent des relations entre eux. La forme la plus simple consiste à la composition. Le garage est composé d'une porte, de bois et d'une automobile. Toutefois, la composition a une signification différente dans chacun des cas comme il a été présenté dans le chapitre précédent. L'automobile est incluse topologiquement (inclusion topologique) dans le garage, le garage est composé physiquement de bois (composition physique) et le garage est composé d'une porte (relation membre collection ou structurelle) serait une meilleure représentation du monde réel. Cette information s'avère utile parce qu'elle permet la gestion du processus d'interaction entre les acteurs. Elle solutionne le problème du véhicule dans le garage tel qu'expliqué dans le chapitre 3.

Plusieurs relations peuvent exister entre deux acteurs. Toutefois, seulement une relation d'un même type peut s'appliquer entre les acteurs parce qu'une seconde n'apporte aucune information utile. En plus d'un nom, les relations possèdent un sens. À cause de ce sens, une relation en sens inverse, ou réciproque, est automatiquement créée pour élaborer les règles relationnelles qui permettront de définir les conditions d'interaction. Tel que présenté dans le chapitre 3, plusieurs relations méronymiques de Winston *et al.* seront utilisées comme ensemble initial de relations. Actuellement, il n'est pas possible de créer des relations liant plus de deux acteurs.

4.1.5 Interaction

> L'*interaction* se définit comme tout lien de cause à effet qui agit entre les instances de personnages incarnés par les acteurs.

APIA base le nouveau paradigme du MV sur l'interaction. Elle agit sur les propriétés des personnages. Elle ne connaît donc pas les acteurs, similairement au mécanisme de polymorphisme en OO qui ne demande pas de connaissance de la classe concrète. La figure 4.6 illustre une interaction générique, sans ses liens avec les personnages.

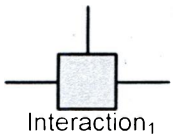

Figure 4.6 : Représentation graphique de l'interaction entre trois personnages (non représentés).

L'interaction possède l'avantage de n'appartenir à aucun acteur ; elle peut donc agir autant de façon symétrique comme les collisions entre deux acteurs que de façon asymétrique comme lancer une flèche avec un arc ou mettre le feu à un arbre avec un briquet. Les interactions sont généralement désignées avec des verbes ou des noms de processus physiques. Dans l'exemple précédent, *Brûler* ou *Combustion* aurait pu être employé.

Les données ne se retrouvent pas dans l'interaction mais plutôt dans les propriétés localisées dans les acteurs. La figure 4.7 illustre une représentation d'une interaction avec les divers paramètres à définir. Cet exemple fait interagir trois personnages selon des multiplicités différentes. Il fournit également des règles relationnelles et de l'information d'accès aux propriétés.

L'interaction est définie par un nom qui l'identifie de façon unique et par les liens, et non les relations, qu'elle établit avec les personnages qui interagissent. Par exemple, un MV réunit une source de chaleur, un combustible et un comburant. Le processus pour mettre le feu requiert justement ces trois personnages car ils contiennent les propriétés nécessaires à la combustion.

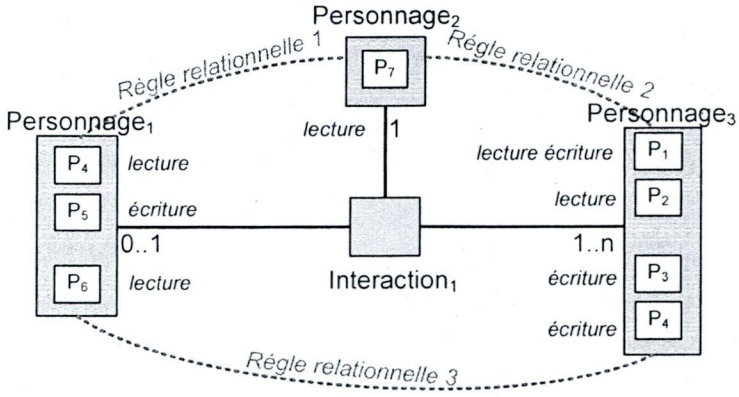

Figure 4.7 : Définition de l'interaction avec ses personnages, ses règles de multiplicité et relationnelles inter-personnages et l'accès aux propriétés.

Dans un but de réutilisation, de modification, d'extension et de composition, l'interaction ne voit pas l'acteur mais plutôt le personnage qu'il incarne. Cette partie est désignée instance de personnage. Il en découle que l'acteur peut contenir plusieurs instances de personnages.

Dans une approche centrée sur l'entité, le comportement est instancié en même temps que l'entité. Dans une approche centrée sur l'interaction, il existe autant d'instances d'interactions que de combinaisons d'acteurs qui interagissent. L'instanciation manuelle des interactions, par le scénariste ou l'usager, est toutefois impossible dans des MVs qui sont fréquemment modifiés. Il est donc nécessaire d'automatiser ces instanciations. **APIA** contraint donc la définition de règles d'interaction, qui, lorsqu'elles sont respectées en cours d'exécution du MV, donnent lieu à l'instanciation ou la modification des interactions par le *gestionnaire de la cohérence présente*. Cette automatisation de l'instanciation des interactions selon des règles d'instanciation prédéfinies par le scénariste constitue la clef qui rend possible l'approche centrée sur l'interaction, permet la gestion du MV et accroît l'*agilité*. Les points suivants doivent être définis lors de la création d'une interaction :

- règle relationnelle entre l'acteur d'ancrage et les instances de personnages qui participent à l'interaction ;
- règles de multiplicité des personnages ;
- règles relationnelles inter-personnages ;
- conditions d'appel de l'interaction et information passée ; et

- modes d'accès aux propriétés.

Chacun de ces points sera détaillé dans les sections suivantes.

Règle relationnelle entre l'acteur d'ancrage et les instances de personnages qui participent à l'interaction

Si les interactions étaient présentes dans tout le MV, il en résulterait des MVs homogènes au niveau comportemental mais la charge de calcul imposée par le grand nombre d'instances d'interactions s'avérerait trop importante. De plus, certaines interactions lourdes en charge de calcul n'ont pas besoin de s'appliquer dans tout le MV. La première règle relationnelle sert donc à restreindre la zone d'applicabilité des interactions.

Une interaction est ajoutée à un MV par son ancrage à un acteur. À ce moment, la règle relationnelle entre l'acteur d'ancrage et un ou plusieurs personnages participants à l'interaction est donnée au *gestionnaire de la cohérence présente*. Si ce dernier, avec la règle relationnelle, peut trouver un des personnages, il passe alors à la vérification des règles de multiplicité. Dans le cas contraire, l'interaction ne s'applique pas.

Règles de multiplicité des personnages

Les règles de multiplicité définissent les conditions de combinaison entre toutes les instances de personnages pour tous les types de personnages. La multiplicité d'**APIA** s'inspire des multiplicités de l'UML.

- 0..1— Ce personnage est optionnel et l'interaction gère au plus une instance de personnage.
- 1 — Ce personnage est obligatoire et l'interaction gère une instance de personnage.
- 0..n— Ce personnage est optionnel et l'interaction gère autant d'instances de personnage qu'il en existe.
- 1..n— Ce personnage est obligatoire et l'interaction gère autant d'instances de personnage qu'il en existe.

Les exemples qui suivent expliquent leur fonctionnement. Ils introduisent également la représentation désignée APR pour acteur, character (personnage) et relation. Cette représentation consiste en une photo instantanée du MV sans les propriétés, les événements en

transition et les instances d'interactions.

Un MV, illustré la figure 4.8, comprend un combustible, une source de chaleur et un comburant.

Figure 4.8 : Exemple de MV avec des acteurs combustibles et des sources de chaleur.

Afin de gérer l'évolution du MV et la vérification des règles d'interaction, le *gestionnaire de la cohérence présente* utilise la représentation APR illustrée à la figure 4.9 Les personnages assignés se retrouvent sous le nom de l'acteur dans des crochets "[]".

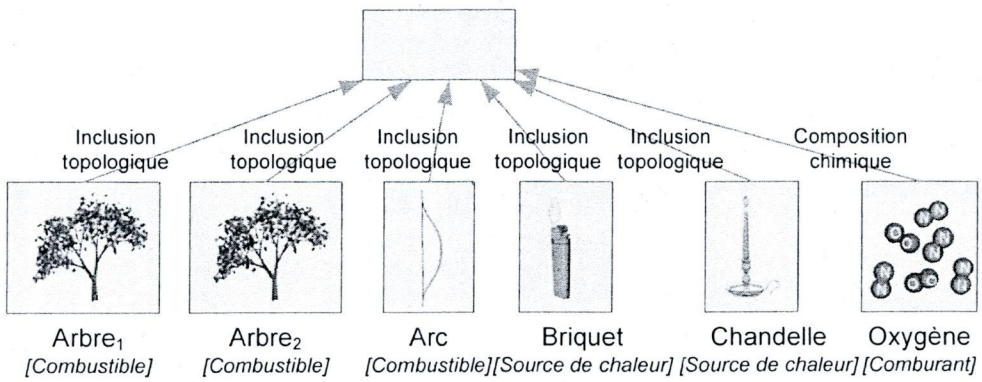

Figure 4.9 : Diagramme AOR (Acteur, Personnage, Relation) d'un MV comprenant tous les éléments pour mettre le feu.

L'oxygène est un acteur non-visible du MV. En observant ce diagramme, on constate qu'il existe la possibilité de mettre le feu. Une source de chaleur (*Briquet*, *Chandelle*) peut mettre le feu à un combustible (*Arbre*$_1$, *Arbre*$_2$ ou *Arc*) grâce à la présence de comburant (*Oxygène*). Ce MV est déjà en exécution avec ses acteurs mais sans interaction de combustion. Il existe, en dehors du MV, un modèle de combustion pouvant y être ajouté. Son concepteur n'a pas pris en considération qu'il s'appliquerait entre un briquet et un arbre, entre un briquet et un arc, etc. Toutefois, il le connaît suffisamment pour établir des règles qui permettent de formuler les conditions d'interaction de la combustion, garantissant ainsi une utilisation adéquate de son modèle entre les bons acteurs.

La figure 4.10 illustre la représentation de l'interaction *Combustion* avec les personnages qu'elle peut mettre en interaction. Les variables m_1, m_2 et m_3 représentent la multiplicité des personnages pour cette interaction.

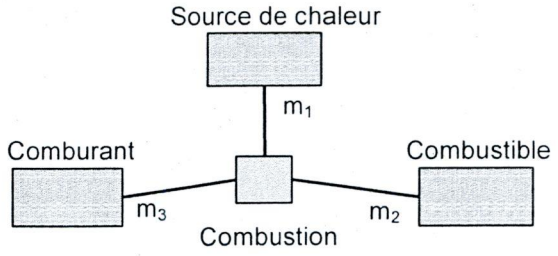

Figure 4.10 : Schéma de l'interaction *Combustion* avec multiplicité des personnages.

Étant donné qu'il existe 4 multiplicités possibles par personnage et 3 personnages différents, il en résulte 64 règles différentes possibles. Toutes ces combinaisons peuvent avoir un sens selon l'implantation de l'interaction. Quelques cas illustreront l'utilité de la multiplicité.

Multiplicité: m_1= 1, m_2= 1 m_3= 1

Il s'agit du cas le plus simple. La combustion requiert exactement un acteur de chaque personnage. Elle utilisera les propriétés de ces acteurs pour activer la combustion et mettre le feu. La figure 4.11 présente un exemple de code d'interaction pouvant mettre le feu à l'arbre.

```
Si ((position_SourceChaleur - position_Combustible < distanceMinimale) et
(intensité_SourceChaleur > intensitéMinimale)) alors
{
    enCombustion_Combustible = vrai
    Quantité_Combustible = Quantité_Combustible -1 ;
}
```

Figure 4.11 : Exemple minimaliste d'implantation de l'interaction *Combustion*.

Les variables représentent les propriétés et les indices représentent les personnages. Cette implantation ne tient pas compte du comburant. Une telle omission signifie que l'existence du comburant, représenté par *Oxygène*, est requise pour instancier l'interaction mais que ses propriétés, comme la quantité de comburant, ne sont pas considérées même si il est présent dans le MV. Par conséquent, le processus de combustion décrit ici ne consume pas d'oxygène.

Cette interaction impose que la source de chaleur reste près du combustible pour que le processus de combustion se poursuive. Dans le monde réel, peu de temps après qu'un combustible débute sa combustion, la chaleur générée suffit à maintenir le processus. Toutefois, dans ce cas, l'interaction implante plutôt un modèle très simplifié.

La figure 4.12 illustre toutes les combinaisons possibles selon les multiplicités (1, 1, 1). Pour la suite des exemples, les combinaisons seront représentées par un vecteur de vecteurs dont chaque membre du premier niveau d'accolades représente un personnage et le deuxième niveau, un vecteur d'instances de personnage. Le deuxième niveau n'est présent que s'il y a plus d'une instance de personnage.

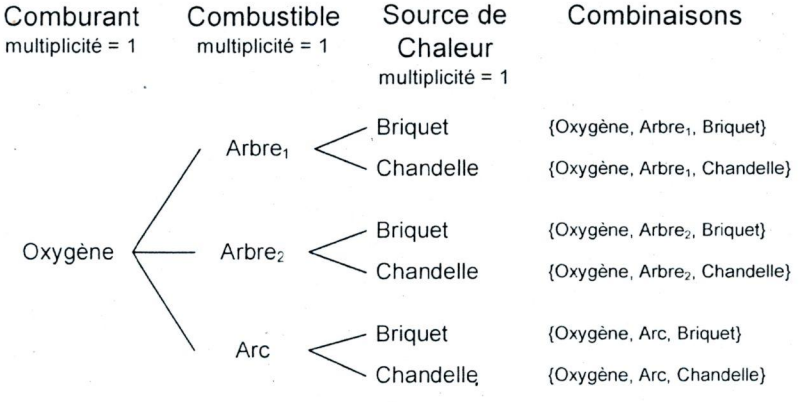

Figure 4.12 : Combinaisons générées pour l'interaction *Combustion* selon des règles de multiplicité (1, 1, 1) et les acteurs du MV de la figure 4.9.

Le *gestionnaire de la cohérence présente* instanciera 6 interactions et fournira les bonnes propriétés à chaque instance, donc combinaison, de la figure 4.12. Ce processus est transparent pour l'interaction. De plus, celle-ci n'a pas à se préoccuper des acteurs qui incarnent les personnages.

Les multiplicités (1, 1, 1) résultent parfois en la somme des interactions, donc en une somme d'instances d'interactions sur une instance de personnage. Par exemple, plusieurs instances interactions appliquent une force sur une instance d'interaction. La propriété force sera alors implantée par un vecteur dont chacun des éléments représentera la force appliquée par chaque instance d'interaction. Une autre interaction sommera les forces du vecteur et appliquera les lois de la dynamique à l'instance de personnage.

Dans l'exemple précédent, les multiplicités (1, 1, 1) ne tiennent pas compte de l'effet additif des sources de chaleur. En effet, il est possible que les sources de chaleur ne puissent mettre le feu individuellement mais, lorsque leur effet est combiné, elles émettront une intensité de chaleur suffisante pour mettre le feu. Ce phénomène est modélisé avec la prochaine combinaison de multiplicité.

Multiplicité: $m_1 = 1..n$, $m_2 = 1$ $m_3 = 1$

Ces règles de multiplicité, combinées avec l'interaction de la figure 4.13, permettent de tenir compte de l'effet additif des sources de chaleur. Il s'agit donc de l'interaction de la somme. Le processus de consommation de comburant y est ajouté pour préparer la comparaison avec la prochaine combinaison de multiplicités.

```
Si ((Moyenne (position_SourceChaleurs) - position_Combustible < distanceMinimale) et
(Somme(intensité_SourceChaleurs) > intensitéMinimale)) alors
{
    enCombustion_Combustible = vrai
    Quantité_Combustible = Quantité_Combustible -1 ;
    Quantité_Comburant = Quantité_Comburant -1
}
```

Figure 4.13 : Exemple d'implantation de l'interaction *Combustion* tenant compte de l'effet additif des sources de chaleur.

L'interaction utilise la multiplicité 1..n en effectuant la somme des intensités pour évaluer si le combustible prendra feu. La figure 4.14 présente toutes les combinaisons possibles résultant de ces trois multiplicités. Comme il n'existe que trois combinaisons possibles, il y aura trois instances de l'interaction *Combustion*. Chaque instance recevra l'ensemble des instances de personnage *Source de chaleur*.

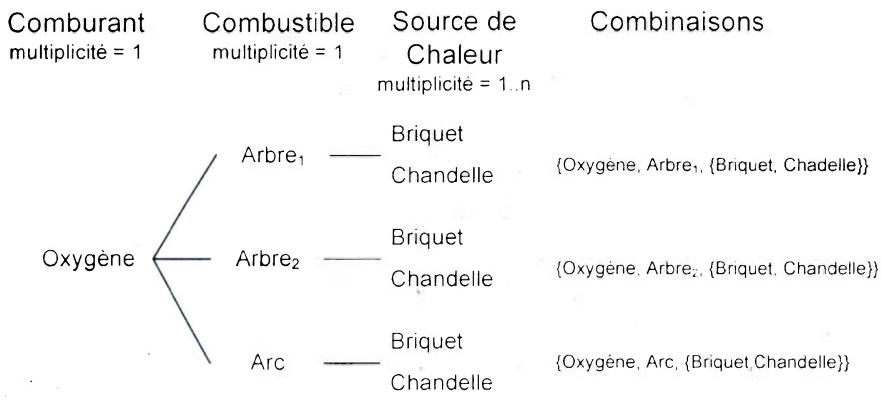

Figure 4.14 : Combinaisons générées pour l'interaction *Combustion* selon des règles de multiplicité (1..n, 1, 1) dans un MV comprenant les acteurs de la figure 4.9.

Les deux exemples précédents requièrent la présence de comburant pour activer la combustion. Cette obligation peut limiter la réutilisation de l'interaction. Le prochain exemple de multiplicités rend le comburant optionnel pour contrer cette limitation.

$m_1 = 1..n,\ m_2 = 1\ m_3 = 0..1$

Ces règles résultent en trois instances d'interaction qui recevront ou non du comburant. La figure 4.15 présente un exemple d'une telle interaction.

```
Si (Moyenne (position_SourceChaleurs) - position_Combustible < distanceMinimale) et
(Somme(intensité_SourceChaleurs) > intensitéMinimale)) alors
{
   Quantité_Combustible = Quantité_Combustible -1
   Si comburant alors Quantité_Comburant = Quantité_Comburant -1
}
```

Figure 4.15 : Exemple d'implantation de l'interaction *Combustion* tenant compte de l'effet additif des sources de chaleur et de la possibilité que le comburant ne soit pas défini.

Cette interaction vérifie si le comburant est disponible et en réduit la quantité. Cette interaction accroît la réutilisation parce qu'elle nécessite moins de conditions d'utilisation. Toutefois, si un combustible comme un arc était plongé dans l'eau, le manque de comburant (l'air) n'empêcherait pas l'acteur de brûler.

À noter que, lorsqu'un combustible prend feu, il peut devenir lui-même une source de chaleur. Ceci est obtenu avec l'interaction *Combustion* qui assigne un personnage *Source*

de chaleur à l'acteur prenant feu. Dès ce moment, le *gestionnaire de la cohérence présente* crée de nouvelles instances de l'interaction *Combustion*. Si cet acteur était $Arbre_1$, avec la règle m_1= 1, m_2= 1 m_3= 1, il se créerait trois instances. La première fait interagir les acteurs {*Oxygène*, $Arbre_1$, $Arbre_1$}, une deuxième les acteurs {*Oxygène*, $Arbre_2$, $Arbre_1$} et une troisième les acteurs {*Oxygène, Arc*, $Arbre_1$}. Pour la règle m_1= 1..n, m_2= 1 m_3= 1, les trois instances de *Combustion* existantes seraient modifiées et recevraient une nouvelle instance de personnage, $Arbre_1$. La première combinaison passerait de {*Oxygène*, $Arbre_1$, {*Briquet, Chandelle*}} à {*Oxygène*, $Arbre_1$, {*Briquet, Chandelle*, $Arbre_1$}}. La proximité de la source de chaleur et du combustible produirait un feu plus intense, ce qui correspond aux observations dans le monde réel. De plus, mettre le feu au premier arbre de la forêt pourrait créer un feu de forêt si la distance entre les arbres n'est pas trop grande. Cette interaction n'a pas été prévue pour cette utilisation mais la conception d'interaction avec les personnages permet aux usagers d'un tel MV d'observer certains comportements non prévus explicitement par les scénaristes mais tout de même réalistes.

La multiplicité optionnelle (0..1, 0..n) peut également servir à optimiser la gestion de l'interaction à l'égard d'un personnage. La figure 4.16 montre un cas d'interaction avec une multiplicité optionnelle.

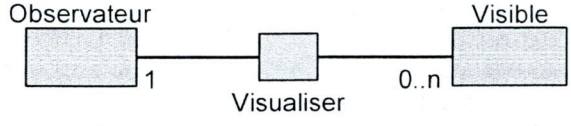

Figure 4.16 : Exemple de multiplicité optionnelle pour le personnage *Visible* résultant en l'instanciation automatique de l'interaction *Visualiser* dès qu'une instance de personnage *Observateur* est créée.

Un acteur *Caméra* pourrait jouer le rôle d'un personnage *Observateur* de cet exemple. Dès l'ajout de la caméra au MV, il se créera une instance d'interaction *Visualiser* qui enverra cette information à plusieurs stations d'affichage 3D. Ce processus exige généralement un temps d'initialisation non négligeable qui peut nuire à l'exécution du MV. L'utilisation d'une multiplicité optionnelle peut éviter de recommencer ce processus même si toutes les instances de personnage *Visible* disparaissent temporairement du champ de vue de l'instance de personnage *Observateur*.

Règles relationnelles inter-personnages

Les règles relationnelles sont des conditions de relation des acteurs qui jouent les

personnages pour que l'interaction ait lieu entre ces acteurs. Autrement dit, il s'agit de l'ensemble des configurations relationnelles possibles entre les acteurs jouant certains personnages qui créent une instance d'interaction.

Les règles relationnelles visent deux buts :

- établir les conditions de relation entre les personnages d'une interaction ; et
- optimiser l'application des interactions.

Les exemples suivants décrivent le fonctionnement des règles relationnelles.

Dans le MV illustré à la figure 4.17, deux véhicules se déplacent dans la rue et peuvent entrer en collision l'un avec l'autre, avec le garage ou le véhicule qui y est inclus.

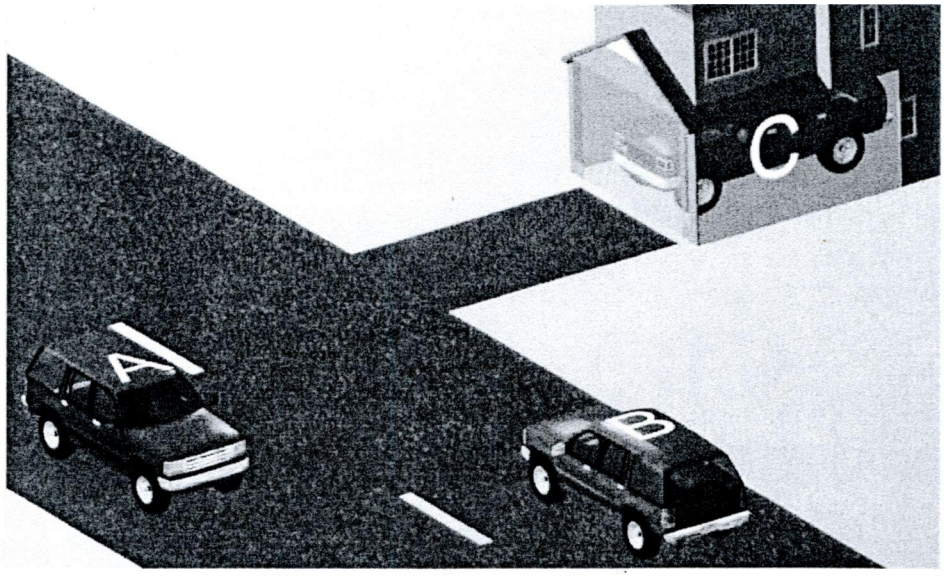

Figure 4.17 : Exemple d'un MV avec deux véhicules se déplaçant dans la rue et pouvant entrer en collision entre eux, avec le garage ou le véhicule qui y est inclus.

La figure 4.18 illustre le diagramme APR de ce MV.

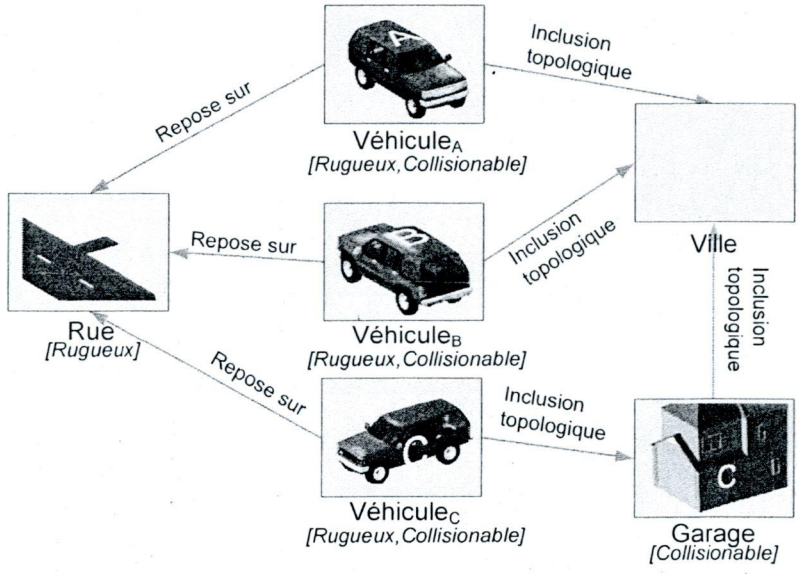

Figure 4.18 : Diagramme APR pour le MV de la figure 4.17.

Cette figure montre les trois instances d'acteurs *Véhicule*. L'inclusion topologique entre les véhicules et la ville montre qu'ils sont tous situés dans la même ville, donc peuvent entrer en collision les uns avec les autres. Le garage peut également entrer en collision. Finalement, le diagramme montre que les véhicules reposent sur la rue.

Une interaction de détection de collisions est ajoutée au MV. La figure 4.19 illustre la définition de cette interaction avec ses règles de multiplicité.

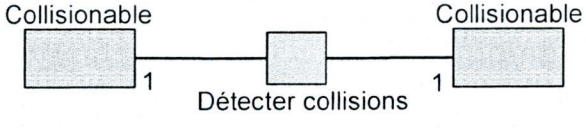

Figure 4.19 : Définition de l'interaction *Détecter collisions* avec ses règles de multiplicité.

L'interaction *Détecter collisions* dépend de deux personnages *Collisionables*. La figure 4.20 présente toutes les combinaisons possibles d'interactions en ne tenant compte que des règles de multiplicité :

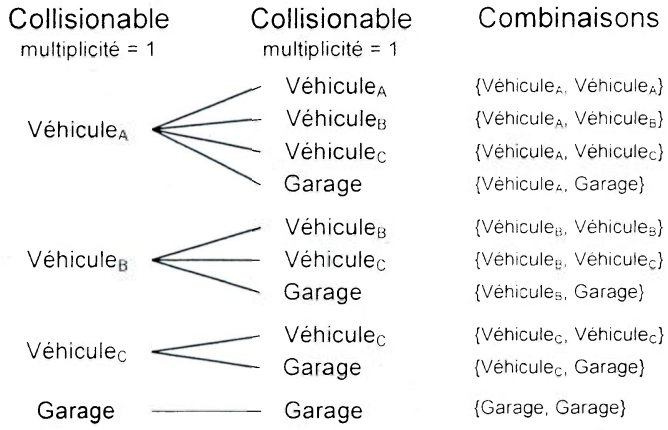

Figure 4.20 : Liste des combinaisons de l'interaction *Détecter collisions* dans le MV de la figure 4.18.

Quoique le MV décrit par la figure 4.18 soit très simple, il génère quand même 10 combinaisons dont certaines n'ont aucun sens. Par exemple, un véhicule ne peut entrer en collision avec lui-même. Par conséquent, les combinaisons {*Véhicule$_A$*, *Véhicule$_A$*}, {*Véhicule$_B$*, *Véhicule$_B$*}, {*Véhicule$_C$*, *Véhicule$_C$*} et {*Garage*, *Garage*} doivent être éliminées. De plus, certaines combinaisons sont inutiles. En effet, l'acteur *Véhicule$_C$*, situé dans l'acteur *Garage*, ne devrait pas entrer en collision avec les autres véhicules. Par conséquent, les combinaisons {*Véhicule$_A$*, *Véhicule$_C$*}, {*Véhicule$_B$*, *Véhicule$_C$*} doivent être également éliminées.

L'ajout de règles de relation permettra d'éliminer les combinaisons indésirables. La figure 4.21 montre l'ajout de deux règles relationnelles à l'interaction de la figure 4.19.

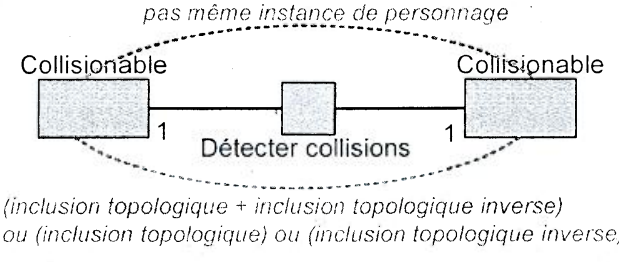

Figure 4.21 : Définition de l'interaction *Détecter collisions* avec les règles de multiplicité et relationnelles.

Deux règles relationnelles y sont définies. Premièrement, les deux personnages ne peuvent faire partie du même acteur. Cette règle élimine les combinaisons qui n'ont aucun

sens pour cette interaction : {*Véhicule*$_A$, *Véhicule*$_A$}, {*Véhicule*$_B$, *Véhicule*$_B$}, {*Véhicule*$_C$, *Véhicule*$_C$} et {*Garage*, *Garage*}. Deuxièmement, toutes les instances du personnage *Collisionable* doivent être séparées par une suite de relations *Inclusion topologique* et *Inclusion topologique inverse* ou par une relation *Inclusion topologique* ou de son inverse. Cette règle signifie que les deux acteurs ne peuvent interagir que s'ils possèdent le même parent d'inclusion topologique ou s'ils établissent une relation entre celui qui enferme et celui qui est inclus. Dans ce cas-ci, les acteurs *Garage*, *Véhicule*$_A$ et *Véhicule*$_B$ sont inclus topologiquement dans l'acteur *Ville*. *Véhicule*$_C$ est toutefois inclus dans *Garage* et non directement dans *Ville*. Il ne peut interagir qu'avec *Garage*. Cette règle élimine donc les combinaisons {*Véhicule*$_A$, *Véhicule*$_C$} et {*Véhicule*$_B$, *Véhicule*$_C$}.

Une deuxième interaction peut également s'appliquer dans le MV décrit à la figure 4.18. Il s'agit du processus physique de friction qui s'applique entre deux corps, illustré à la figure 4.22.

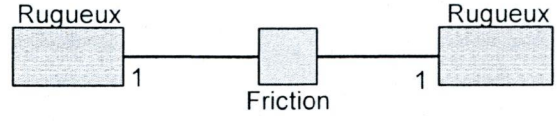

Figure 4.22 : Définition de l'interaction *Friction* entre les corps rugueux.

La définition de l'interaction *Friction* ci-dessus produit les combinaisons de la figure 4.23.

Rugueux multiplicité = 1	Rugueux multiplicité = 1	Combinaisons
Véhicule$_A$	Véhicule$_A$	{Véhicule$_A$, Véhicule$_A$}
	Véhicule$_B$	{Véhicule$_A$, Véhicule$_B$}
	Véhicule$_C$	{Véhicule$_A$, Véhicule$_C$}
	Rue	{Véhicule$_A$, Rue}
Véhicule$_B$	Véhicule$_B$	{Véhicule$_B$, Véhicule$_B$}
	Véhicule$_C$	{Véhicule$_B$, Véhicule$_C$}
	Rue	{Véhicule$_B$, Rue}
Véhicule$_C$	Véhicule$_C$	{Véhicule$_C$, Véhicule$_C$}
	Rue	{Véhicule$_C$, Rue}
Rue	Rue	{Rue, Rue}

Figure 4.23 : Combinaisons possible pour l'interaction *Friction* dans le MV de la figure 4.18.

Comme pour l'interaction précédente, certaines combinaisons n'ont aucun sens : $\{Véhicule_A, Véhicule_A\}$, $\{Véhicule_B, Véhicule_B\}$, $\{Véhicule_C, Véhicule_C\}$ et $\{Rue, Rue\}$. En effet, un corps rigide ne peut entrer en friction avec lui-même. De plus, même si les véhicules possèdent des caractéristiques de friction, elles ne doivent s'appliquer que lorsqu'ils sont en contact. Les combinaisons $\{Véhicule_A, Véhicule_B\}$, $\{Véhicule_A, Véhicule_C\}$ et $\{Véhicule_B, Véhicule_C\}$ doivent donc être éliminées. Les règles relationnelles permettent d'éliminer de tels cas. La figure 4.24 illustre la définition de l'interaction *Friction* avec les règles relationnelles.

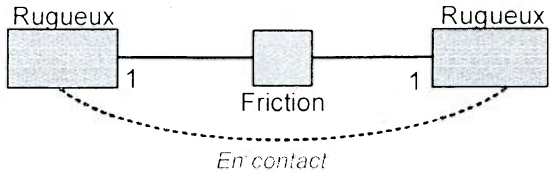

Figure 4.24 : Définition de l'interaction *Friction* avec les règles de multiplicité et relationnelles.

La règle relationnelle permet donc de rejeter les combinaisons qui n'ont aucun sens pour cette interaction ou, inversement, faire interagir seulement les combinaisons valides. Il faut noter que la relation de contact peut être générée par l'interaction de collision présentée précédemment. Ce changement de statut de l'acteur est rendu possible par la connaissance, *a priori*, de la relation *EnContact* par l'interaction *Détecter collisions*. Il s'agit d'un exemple de composabilité. Deux interactions, conçues indépendamment, n'ont besoin que de connaître la relation *EnContact* et quelques propriétés pour être intégrées dans un même MV.

Conditions d'appel de l'interaction et information passée

Les mécanismes d'appel de l'instance d'interaction font partie de la gestion du temps et des ressources. L'explication du fonctionnement des mécanismes d'appel sera donnée ultérieurement. Toutefois, il faut mentionner dès cette étape que les types d'appel possibles doivent être définis lors de la conception de l'interaction.

Les instances d'interactions modifient les instances de propriétés lorsque surviennent d'autres modifications dans le MV. Il existe trois types de modification, donc trois types d'appel (ou de notification, de message) que doit gérer l'interaction. Ils sont décrits ci-dessous.

- Initialisation et terminaison — Le démarrage et la terminaison d'une instance d'interaction constituent des moments privilégiés pour effectuer des initialisations ou d'autres processus requérant un temps relativement long. Si l'interaction ne dépend que de personnages de multiplicité 1, il est possible de ne considérer que les initialisations et les terminaisons parce qu'aucune instance de personnage ne peut s'ajouter ou s'enlever sans créer ou détruire l'interaction.
- Ajout et retrait d'instances de personnages — Lorsque la multiplicité est de 0..1, 0..n ou 1..n, il se peut que l'interaction doive gérer des instances de personnage non présentes au moment de l'initialisation ou que celles-ci disparaissent. L'interaction exige alors d'être avertie lorsque des instances de personnages s'ajoutent ou se retirent afin de prendre les actions qui s'imposent. L'exemple qui suit explique le concept d'ajout et de retrait de personnages.

Dans le MV illustré à la figure 4.25, un acteur *Caméra* est placé dans la $Pièce_A$ et observe la pièce elle-même et tout ce qui s'y trouve, l'acteur *Chaise* dans ce cas-ci. Dans la $Pièce_B$ se trouvent deux autres acteurs : un *Humain* et un *Bureau*.

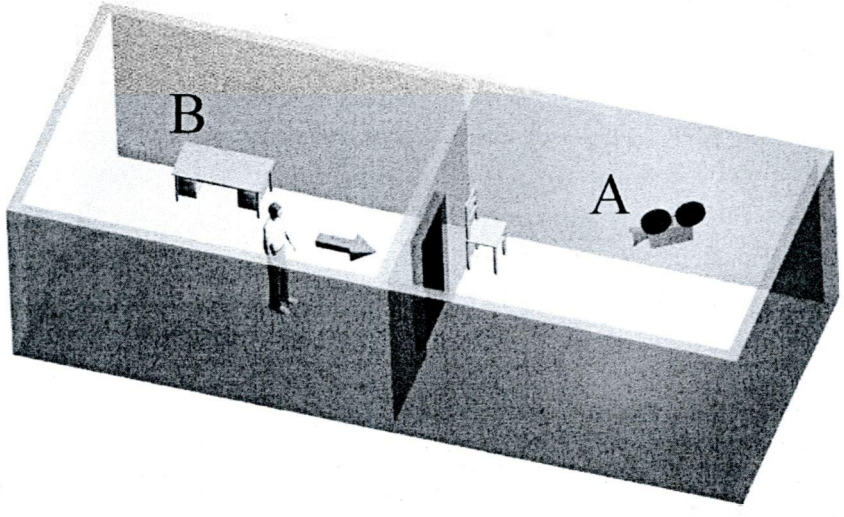

Figure 4.25 : Exemple d'un MV où un observateur ne voit que ce qui est visible dans sa pièce.

Il serait intéressant que la caméra ne voit que les acteurs situés dans la même pièce qu'elle. Comme pour les exemples précédents, il est possible d'utiliser les règles de multiplicité et les règles relationnelles pour limiter le contexte de l'interaction de

visualisation. La figure 4.26 illustre un diagramme APR possible de ce MV. Sept acteurs sont interreliés avec des relations *Inclusion topologique* et *même pièce.*

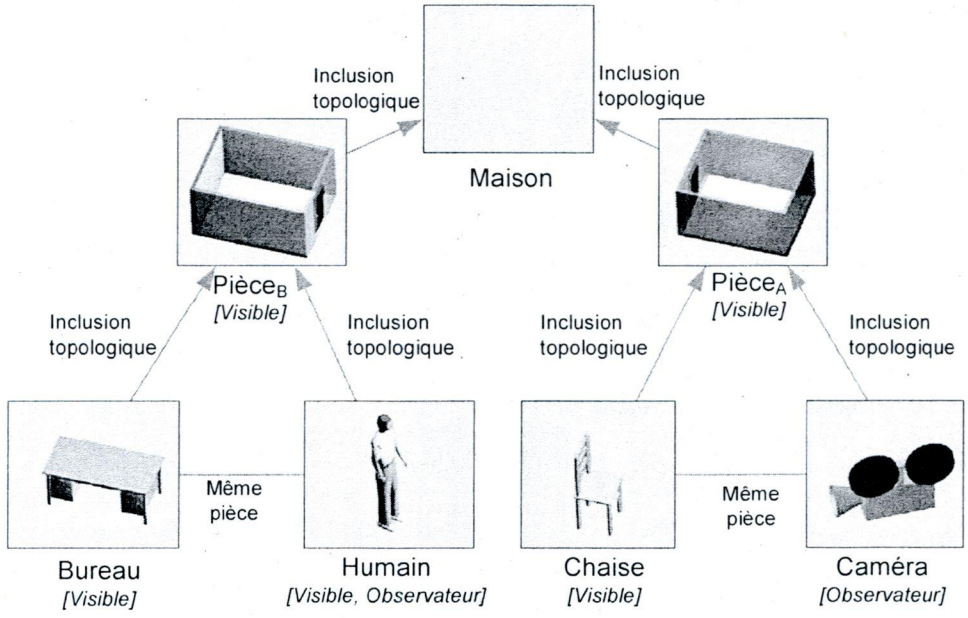

Figure 4.26 : Diagramme APR du MV de l'exemple de la figure 4.25.

La caméra ne verra que ce qui est dans la pièce si des règles limitent la visibilité de l'interaction. La figure 4.27 illustre l'interaction *Visualiser*.

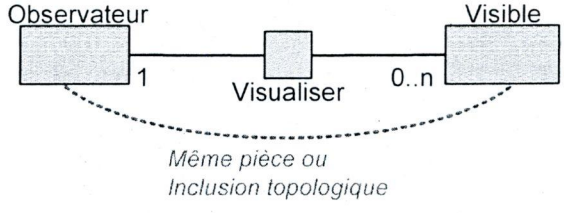

Figure 4.27 : Interaction *Visualisation* qui restreint la visibilité à une seule pièce.

Cette interaction limite la visualisation de la caméra et de l'humain à la pièce elle-même et aux acteurs qui y sont inclus. Avec le diagramme APR de la figure 4.26, le *gestionnaire de la cohérence présente* crée deux instances de cette interaction. Chacune des instances reçoit les ensembles respectifs suivants : { *Caméra*, {*Chaise, Pièce$_A$*}} et { *Humain*, {*Bureau, Pièce$_B$*}}. Lors de la création des instances d'interaction, le gestionnaire fourni à celles-ci la liste des instances de personnages qui se sont ajoutées, dans ce cas-ci,

toutes celles qui participent à l'interaction.

Si l'humain se déplace et change de pièce, il modifie le diagramme APR qui devient celui de la figure 4.28.

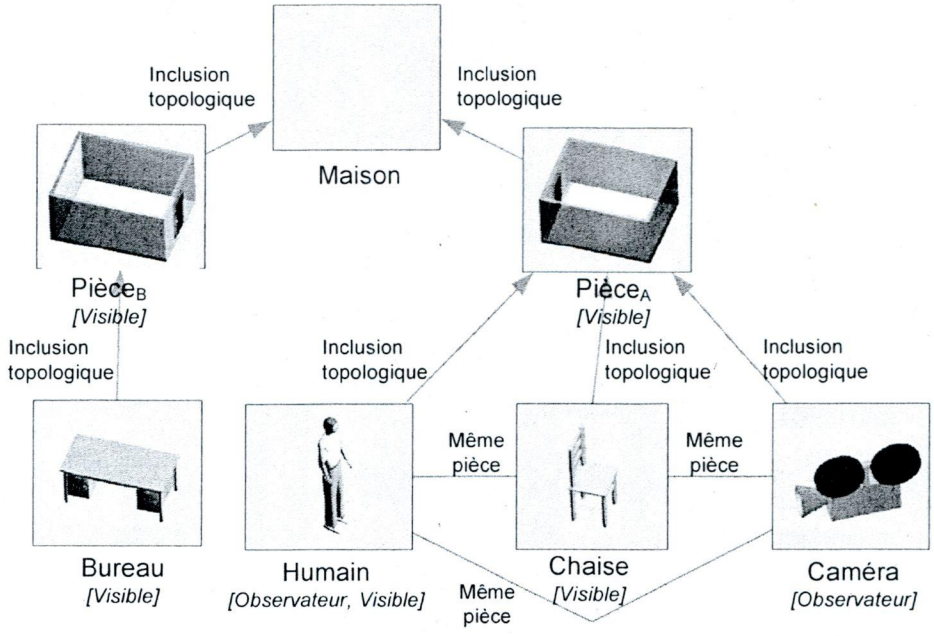

Figure 4.28 : Nouveau diagramme APR lorsque l'humain change de pièce.

Avec ce changement, le *gestionnaire de la cohérence présente* modifie les ensembles initiaux (pour chacune des instances de l'interaction) des instances de personnages en { *Caméra*, {*Chaise, Pièce$_A$, +Humain*[1]}} et en { *Humain*, {*+Chaise, +Pièce$_A$, -Bureau, -Pièce$_B$*}}. Les instances d'interaction reçoivent les instances de personnage ajoutées et retirées. Pour la première instance d'interaction, seule l'instance *Humain* a été ajoutée. Cette instance de l'interaction *Visualiser* ajoutera *Humain* à sa liste d'objets dans sa librairie graphique de façon à ce qu'il soit représenté graphiquement. La liste des instances de personnages de la seconde instance d'interaction change de façon plus importante. Deux instances de personnages y ont été ajoutées : *Chaise* et *Pièce$_A$*. Quant aux instances de personnage *Bureau* et *Pièce$_B$*, l'interaction devra gérer leur retrait. Pour la visualisation, l'interaction enlèvera cet acteur du graphe de scène de la librairie graphique.

1. Le signe + signifie que le personnage a été ajouté alors que le signe - signifie que le personnage sera enlevé.

- Modification de propriétés — L'interaction recevra la liste des propriétés modifiées afin de prendre les actions qui s'imposent. La section 4.3.2 sur la *gestion de la cohérence temporelle* en détaillera le fonctionnement.

Modes d'accès aux propriétés

Les interactions accèdent aux propriétés en lecture, en écriture ainsi que, occasionnellement, dans les deux modes à la fois. APIA contraint chaque interaction à définir les modes d'accès aux propriétés. L'exemple qui suit en présente un cas.

La figure 4.29 illustre une interaction de combustion avec ses modes d'accès.

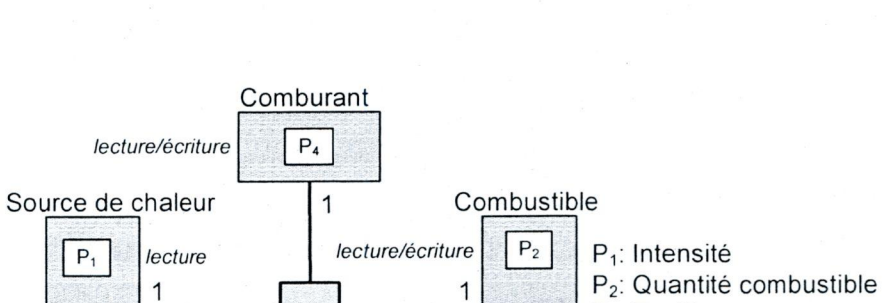

Figure 4.29 : Définition des modes d'accès des propriétés de l'interaction *Combustion*.

Dans cet exemple, l'interaction *Combustion* dépend de trois personnages et de leurs propriétés. La figure 4.30 illustre une implantation de cette interaction en reprenant celle de la figure 4.15. Elle lui ajoute une dépendance par rapport à la quantité de comburant et met en évidence l'utilisation des propriétés en lecture et en écriture. Toutes les multiplicités de personnages de cet exemple sont de 1 et l'ajout ou le retrait d'instances de personnages se simplifie à l'initialisation ou à la terminaison de l'interaction. Dans le cas contraire, l'ajout ou la rétractation d'instances de personnages demanderait l'application d'une équation du type des figures 4.13 ou 4.15.

Si ($position_{SourceChaleur}$ - $position_{Combustible}$ < distanceMinimale) et ($intensité_{SourceChaleur}$ > intensitéMinimale)) alors
{
    $\boldsymbol{Quantité_{Comburant}}$ = $Quantité_{Comburant}$ - 1
    $\boldsymbol{Quantité_{Combustible}}$ = $Quantité_{Combustible}$ - 1
}

Figure 4.30 : Détail de l'interaction *Combustion* mettant en évidence les accès en écriture en gras italique et en lecture en italique régulier.

Certaines propriétés comme la position de la source de chaleur et du combustible sont accédées en lecture alors que d'autres propriétés comme la quantité de comburant et de combustible sont accédées tant en lecture qu'en écriture.

Premièrement, l'interaction doit vérifier les conditions de combustion dès son instanciation ou lors du retrait ou l'ajout d'instances de personnages. Par la suite, l'interaction réévaluera les conditions de distance minimale si la position de la source de chaleur ou du combustible change, donc si une autre interaction modifie la position.

Le mode d'accès aux propriétés sert au *gestionnaire de la cohérence présente* pour générer les événements qui causeront éventuellement l'appel des instances d'interactions. L'exemple qui suit explique ce mécanisme.

Une interaction se sert de la position des deux instances de personnage *Collisionable* pour détecter la collision et, lorsque qu'elle survient, l'interaction change la position des acteurs. Cet accès en mode mixte est montré à la figure 4.31.

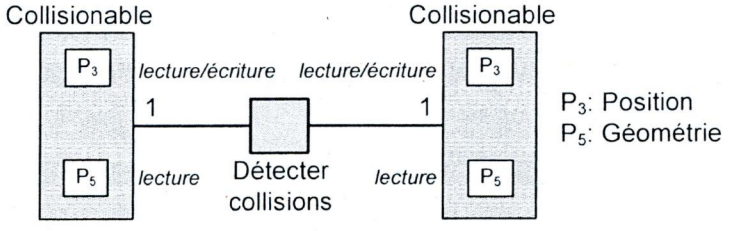

Figure 4.31 : Interaction *Détection de collisions* utilisant la propriété Position (P_3) en mode mixte, elle-même déjà utilisée par l'interaction *Combustion* à la figure 4.29.

Si les deux interactions *Combustion* et *Détecter collisions* se retrouvent dans le même MV et, si les acteurs jouent des rôles doubles, il survient un phénomène en chaîne important qui se doit d'être expliqué avec un exemple.

La figure 4.32 illustre un MV composé d'un briquet, d'une chandelle et d'une chaise. La chaise est constituée de bois et le bureau de métal. Le briquet peut mettre le feu à tout ce qui est fait de bois et il peut entrer en collision avec le bureau et la chaise.

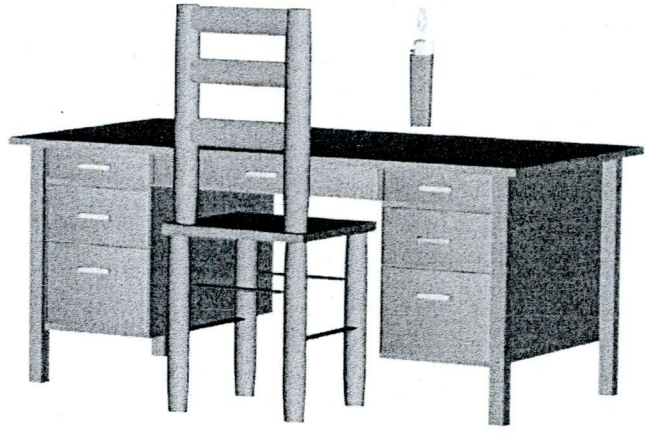

Figure 4.32 : Exemple d'un MV avec une chaise en bois, un bureau en métal et un briquet.

La figure 4.33 montre deux instances d'interactions qui agissent sur le briquet. L'omission de l'instance d'interaction *Détecter collisions* entre les acteurs *Briquet* et *Chaise* n'a pour but que de simplifier l'illustration. Dans cet exemple, l'instance de propriété *Position* sera modifiée (écriture) lors de la détection d'une collision. L'interaction *Combustion*, illustrée à la figure 4.29, dépend (lecture) de la distance entre les acteurs. Par conséquent, la modification de *Position* par *Détecter collisions* causera l'appel de *Combustion*.

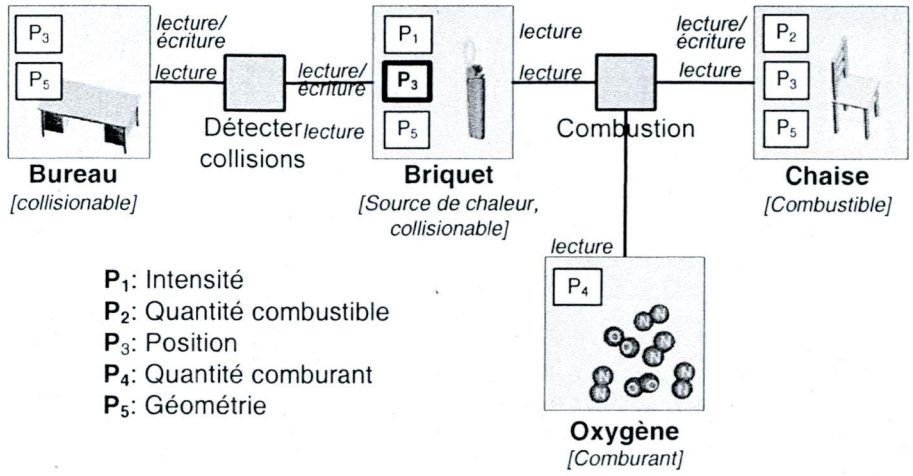

Figure 4.33 : Représentation des instances d'interactions *Détecter collisions* et *Combustion* accédant la propriété *Position* P_3.

Le *gestionnaire de la cohérence temporelle* construira, pour chacune des instances de propriété en écriture, la liste des instances d'interactions qui en dépendent. La figure 4.34 montre une vue détaillée de l'effet de la modification de l'instance de propriété *Position*. L'instance d'interaction *Visualiser* y a été ajoutée.

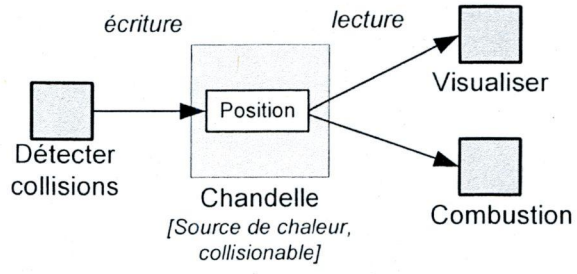

Figure 4.34 : Exemple de déclenchement de l'appel de plusieurs instances d'interactions par la modification d'une instance de propriété par une instance d'interaction.

4.1.6 Discussion sur les éléments du méta-modèle conceptuel

Cette section a présenté les éléments du modèle conceptuel et, par conséquent, qui serviront à construire le MV. Toutefois, la connaissance des éléments n'est pas suffisante pour obtenir des MVs *agiles*. La section 4.2 présente les actions qui concrétisent l'*agilité*.

4.2 Processus de modification d'un MV

Un MV est composé, réutilisé, modifié ou étendu par des actions que posent les scénaristes, les usagers ou les interactions. L'ensemble de ces actions doit être bien compris et regroupé dans un processus. Cette section présente ce processus avec les conséquences et les considérations sur l'*agilité*. Ces actions peuvent être effectuées autant avant l'exécution qu'à n'importe quel moment en cours de l'exécution du MV. L'ordre des actions doit néanmoins respecter le diagramme de dépendances illustré à la figure 4.35. Par exemple, un personnage peut être créé à n'importe quel moment lors de l'exécution à condition que les propriétés qu'il utilise aient été préalablement définies.

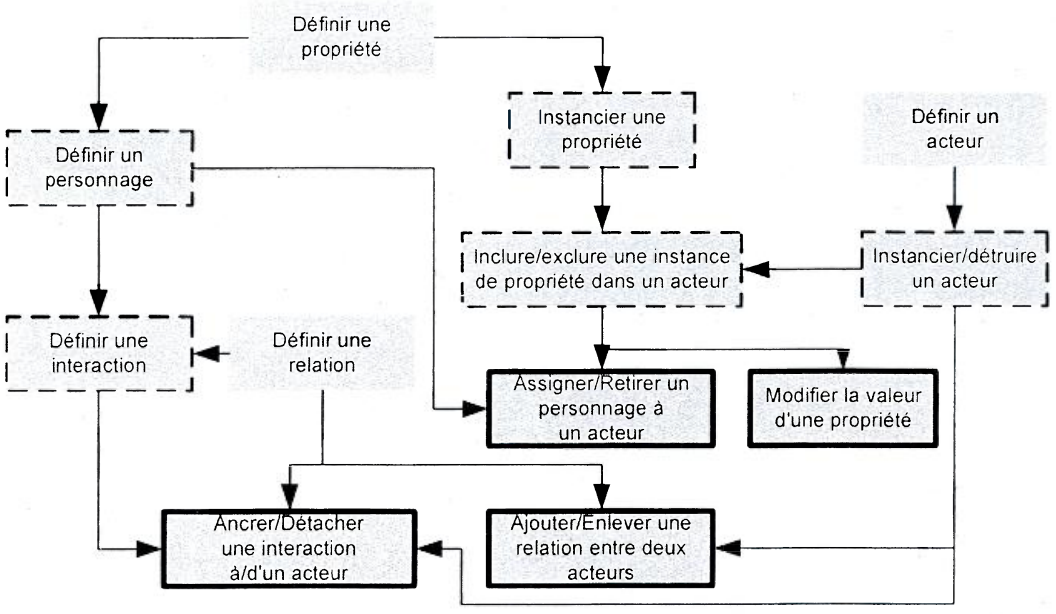

Figure 4.35 : Dépendance des différentes actions qui modifient un MV.

La figure 4.35 montre trois groupes d'actions du scénariste. Le premier, indiqué par l'absence de cadre noir, regroupe les actions qui ne dépendent d'aucune autre : définir une propriété, définir une relation et définir un acteur. Ces actions peuvent être effectuées à n'importe quel moment (avant ou en cours d'exécution) mais leur effet n'est pas immédiat sur le niveau de réutilisabilité, de composabilité, d'extensibilité et de modifiabilité. Toutefois, les autres actions qui permettent d'obtenir des MVs *agiles* en dépendent fortement. Par exemple,

un scénariste décide de définir des propriétés en cours d'exécution. Au moment de la définition, il n'y aura aucun changement dans le MV car aucun personnage ou interaction ne connaît cette propriété. En effet, le scénariste qui ajoutera cette propriété l'utilisera probablement exclusivement au début car les autres scénaristes ne la connaissent pas encore. Toutefois, avec le temps, des éléments (personnages, interactions, acteurs) conçus par d'autres scénaristes se mettront à utiliser cette propriété et son influence deviendra importante sur l'*agilité* car beaucoup d'éléments reposeront sur elle. Par exemple, une propriété définie comme *CaractéristiquesPhysiquesdelaFlèche* nuirait à la réutilisation.

Le deuxième groupe, indiqué par un cadre pointillé à la figure 4.35, réunit des actions qui dépendent des autres et dont d'autres dépendent également : définir un personnage, définir une interaction, instancier une propriété, instancier et détruire un acteur et inclure et exclure une instance de propriété d'un acteur. Comme toutes les actions, elles peuvent être posées en cours d'exécution. Elles devraient maximiser l'utilisation d'éléments déjà définis. Il serait en effet peu utile de créer un acteur qui joue des personnages n'étant reconnus par aucune interaction. Cet acteur ne pourrait interagir avec son environnement et serait, par conséquent, inutile à ce moment. Toutefois, il pourrait se mettre à interagir dans le futur lorsque ce personnage deviendrait connu et utilisé.

Le troisième groupe, indiqué par un cadre large noir à la figure 4.35, réunit des actions dont aucune autre ne dépend : ancrer/détacher une instance d'interaction à/d'un acteur, lier/délier deux acteurs et assigner ou retirer un personnage d' un acteur. Ces actions produisent un maximum d'effets dans le MV et incarnent les caractéristiques sous-jacentes à l'*agilité*.

Effectuer des actions représente en soit l'*agilité* (modification, extension, composition, liberté d'action) mais influence également celle-ci dans le future. Il faut tenter de maximiser l'*agilité* présente sans nuire à celle-ci dans le future ni rendre le MV incohérent. Par exemple, une propriété mal définie peut limiter la réutilisabilité et l'extensibilité. Tel que présenté au chapitre 3, le méta-modèle conceptuel définit les règles de conception et de gestion du MV. Il encadre ce que peut faire le scénariste. Cette limitation s'avère essentielle afin de garantir l'intégration entre les parties de modèles conceptuels et ainsi obtenir des MVs cohérents. Toutefois, lorsque la limitation est trop grande, le scénariste peut ne pas être capable de créer le MV qu'il désire. **APIA** possède une approche de groupes d'actions dont les conséquences

sur l'*agilité* future sont progressives. Certaines actions, principalement du groupe 1, impliquent un niveau de responsabilité important pour le scénariste alors que d'autres, du groupe 3, influencent peu l'*agilité* future et peuvent être effectuées facilement en cours d'exécution. En fait, ces dernières sont tellement bien encadrées et comprises grâce au méta-modèle conceptuel que les interactions elles-mêmes pourront les effectuer. Les gestionnaires seront en mesure d'en gérer les conséquences pour maintenir le MV cohérent.

La figure 4.36 illustre que le rôle du scénariste sur l'*agilité* future du MV décroît lorsque le méta-modèle conceptuel, à travers des contraintes de conception et sa gestion du MV, prend en charge de plus en plus responsabilités. Cette approche est équivalente à celle des FOM dans HLA mais est plus granulaire et comprend plus d'actions.

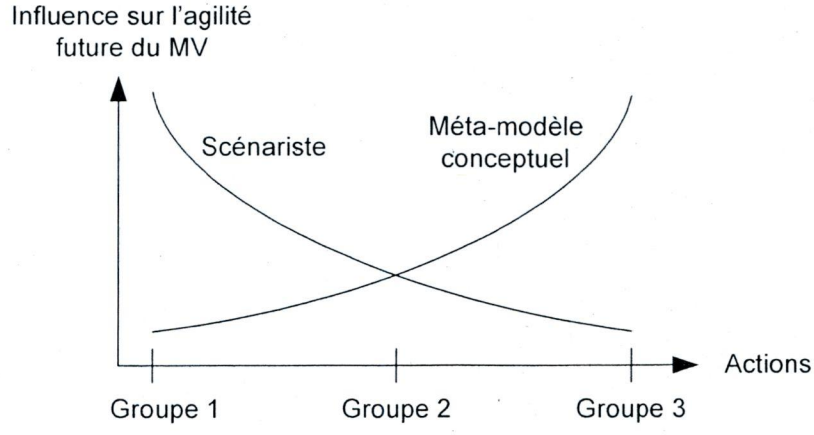

Figure 4.36 : Influence du scénariste vs méta-modèle conceptuel selon les actions que pose le scénariste.

Cette figure montre que les actions du groupe 1 sont peu encadrées par le méta-modèle conceptuel et il est de la responsabilité du scénariste d'effectuer ces actions correctement. Les actions de ce groupe ont une simplicité apparente causée par une grande liberté d'action du scénariste. Cette liberté donne la possibilité au scénariste d'effectuer des actions de grande influence sur l'*agilité* future. Les actions du groupe 2, telle la définition d'une interaction, semblent complexes mais laissent moins de décisions critiques au scénariste. Il risque donc moins de nuire à l'*agilité* future. Finalement, les actions du groupe 3 sont plus contraintes par le méta-modèle conceptuel et comprises par les gestionnaires. Ces actions ont moins d'impacts négatifs sur l'*agilité* future. Par conséquent, ces actions, seront plus faciles à effectuer.

Les dépendances entre les actions influencent le moment où ces actions peuvent être posées. Si une action est posée en cours d'exécution, toutes celles qui en dépendent doivent être effectuées en cours d'exécution. Par exemple, si un scénariste assigne un personnage avant l'exécution du MV, il est évident que le personnage est également défini avant l'exécution. Toutefois, ce personnage pourrait également être assigné en cours d'exécution. Cette observation implique que les actions totalement indépendantes, soit celles du groupe 1, seront effectuées moins fréquemment et moins à l'improviste, surtout en cours d'exécution. Celles de groupe 3 ont plus de chances d'être posées en toute liberté en cours d'exécution et ainsi de concrétiser l'*agilité*. Les actions du groupe 2 se situent entre les deux.

4.2.1 Définir une propriété

L'étape la plus critique consiste à définir les propriétés. Trouver des propriétés utiles à un grand nombre d'interactions est un défi. Puisque le méta-modèle conceptuel impose peu de contraintes au scénariste par rapport à la propriété, il est difficile, avec **APIA**, de garantir des propriétés qui résulteront en des MVs *agiles*. Par conséquent, la responsabilité du scénariste est grande à cette étape. Il faut donc privilégier des propriétés reconnues comme fondamentales (*masse*, *position* et *géométrie* au lieu de *PropriétéArc*).

Les propriétés doivent être définies à un niveau approprié de granularité. Si elle s'avère trop fine, elle crée une complexité inutile car les gestionnaires utilisent la majorité des ressources disponibles à la gestion des propriétés et la conception des interactions en est complexifiée. À l'opposé, une granularité grossière nuit à l'*agilité*. Par exemple, la conception d'un acteur *Flèche* dans un MV requiert une masse, une position, une orientation, une vitesse angulaire et linéaire. Toutes ces données peuvent être regroupées dans une même propriété appelée *PropriétésPhysiques*, ou même pire, *PropriétésdeLancementdeFlèche*. Il serait difficile de réutiliser, d'étendre ou de modifier la flèche, donc d'interagir à un autre niveau que le lancement de flèches avec un tel acteur. De plus, ses propriétés ne seraient guère réutilisables. À l'opposé, la création d'une propriété pour chacune des coordonnées de la position (position X, position Y, position Z), de la vitesse, etc., viendrait alourdir la conception et la gestion dans APIA. Le méta-modèle conceptuel décourage toutefois les trop fines granularités avec des définitions obligatoires lors de la conception des interactions. Un juste

milieu doit être recherché par le scénariste. Ce juste milieu correspond probablement aux données utilisées dans les modèles physiques d'ingénierie.

Le choix entre la création d'une propriété ou d'un acteur avec cette propriété s'avère difficile. Dans l'exemple précédent, les flèches ne devraient pas être conçues comme des propriétés d'un arc mais plutôt comme des acteurs. Ici encore, le scénariste doit effectuer ce choix en tenant compte des caractéristiques de l'*agilité*. Néanmoins, **APIA** offre ce choix.

4.2.2 Définir un personnage

La définition des personnages repose sur la définition de propriétés. Le choix entre un personnage très spécifique défini avec le nom d'un acteur et un autre général défini avec le nom d'une propriété restera la problématique principale du scénariste. Comme pour la propriété, **APIA** permet la définition séparée de chaque personnage afin d'assurer la composabilité.

4.2.3 Définir une relation

Les relations lient les acteurs. Avec les personnages, elles permettent de concevoir des MVs sémantiques. Elles permettent ainsi de structurer le MV et d'élaborer les règles d'interaction. Par conséquent, le scénariste devrait privilégier l'utilisation de relations universellement reconnues, telles les relations d'agrégation de Winston *et al.*, afin de maximiser leur utilité.

Les relations jouent un rôle dans l'ordonnancement et, éventuellement, dans la répartition du MV. Par exemple, deux acteurs se trouvant sur des planètes différentes ont peu de chance d'interagir. Elles pourraient donc être réparties sur des ordinateurs différents. Cependant, ces optimisations de la répartition n'ont pas été considérés dans cette thèse.

4.2.4 Définir une interaction

La définition d'une interaction implique de nombreuses étapes : définition des règles relationnelles entre l'instance d'interaction et ses instances de personnages, définition des règles de multiplicité des personnages, définition des règles relationnelles entre les personnages, conditions d'appel de l'interaction et définition des modes d'accès aux propriétés.

Quoique le nombre de décisions qui doivent être prises lors de la définition de l'interaction soit supérieur à celui des autres éléments, ces décisions influencent moins l'*agilité* future. En effet, cette *agilité* repose plus sur les éléments qu'elle utilise.

La conception d'une interaction peut s'effectuer de zéro ou sur la base d'un modèle existant. Parce que l'interaction permet l'encapsulation de plusieurs types de modèles avec les règles de multiplicité et relationnelles, **APIA** facilite l'utilisation d'un modèle existant et, par conséquent, l'extensibilité et la réutilisabilité.

4.2.5 Instancier une propriété

Instancier une propriété implique de lui assigner une valeur. Cette valeur quantifiera ou qualifiera éventuellement un acteur. Par conséquent, le scénariste doit déjà penser à quel acteur cette instance de propriété sera ajoutée. L'impact sur l'*agilité* reste toutefois minime.

En plus de la méthode conventionnelle, une instance de propriété peut être initialisée à partir d'une autre instance de propriété ou avec une valeur par défaut. Ces deux façons d'initialiser une instance de propriété sont utiles en cours d'exécution lorsqu'un usager ou un scénariste doit effectuer cette opération comme dans l'exemple de la copie de la masse de la section 2.3.5.

4.2.6 Instancier ou détruire un acteur

L'instanciation d'un acteur implique peu de conséquences pour les autres actions qui en dépendent. En effet, l'assignation des propriétés ne dépend pas des acteurs dans lesquels elles se retrouvent. Il en est de même pour les personnages. Les acteurs doivent tous porter un nom unique car une interaction pourrait utiliser les noms d'acteurs pour différentier les instances de personnages dans une liste quand la multiplicité est de 0..n ou 1..n. Quant à la destruction de l'acteur, elle implique le retrait des personnages et des instances de propriétés qu'il possède ainsi que des instances de relations qui le relie à d'autres acteurs.

APIA ne distingue pas un acteur de son instanciation car chaque définition d'acteur ne donne lieu qu'à une instanciation afin que chaque instance possède un nom unique. Par conséquent, le terme acteur désignera également son instance.

4.2.7 Inclure ou exclure une instance de propriété dans un acteur

Chaque instance de propriété est assignée à un acteur. La figure 4.37 en illustre un exemple. Cette action ne pose aucune difficulté tant du point de vue du scénariste que du *gestionnaire de la cohérence présente*. Évidemment, plus un acteur contiendra de propriétés, plus il sera en mesure de jouer des personnages et d'interagir dans le MV.

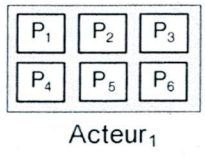

Figure 4.37 : Représentation graphique de l'assignation des propriétés à un acteur.

4.2.8 Assigner ou retirer un personnage à un acteur

L'assignation d'un personnage à un acteur n'est possible que si ce dernier possède l'ensemble des propriétés obligatoires du personnage. La figure 4.38 illustre un exemple d'assignation de deux personnages à un acteur.

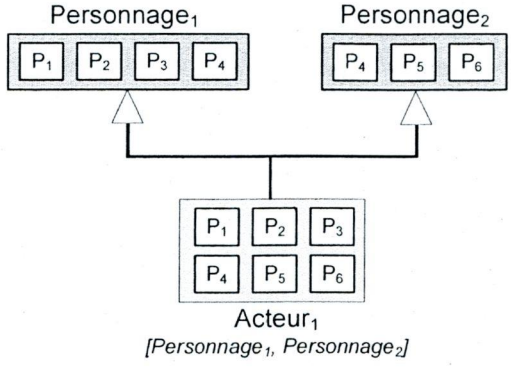

Figure 4.38 : Représentation graphique de l'assignation des personnages à un acteur.

L'assignation est représentée par le symbole de l'héritage en l'UML ou, tel que présenté précédemment dans le diagramme APR, par l'ajout du nom du personnages entre crochets "[]" sous l'acteur. L'assignation de personnages à un acteur qui possède l'ensemble des propriétés requises n'est pas automatique. Le scénariste doit assigner lui-même le personnage à l'acteur.

Une assignation automatique des personnages serait possible mais elle n'a pas été évaluée au cours des présents travaux.

Lorsqu'un personnage est assigné ou retiré d'un acteur, le *gestionnaire de la cohérence présente* applique les algorithmes afin que toutes les règles définies dans les modèles conceptuels soient respectées. Par exemple, si un acteur cesse de jouer le rôle de personnage *Visible*, l'interaction *Visualiser* verra cet acteur retiré de sa liste. Les algorithmes appliqués dans ce cas sont présentés à la section 4.3.1.

4.2.9 Ajouter et enlever une relation entre deux acteurs

Un acteur sans relation avec au moins un autre acteur ne participe à aucune interaction. Les relations sont donc essentielles dans APIA. Cette interconnexion d'acteurs génère des graphes relationnels. Tel que mentionné au chapitre précédent, contrairement à certaines autres structures comme celles orientés graphiques, les graphes relationnels offrent un plus grand choix de description pour les MVs. L'ajout ou le retrait d'une instance de relation requiert l'intervention du *gestionnaire de la cohérence présente*. La section 4.3 explique les mécanismes du gestionnaire pour effectuer une telle tâche.

4.2.10 Ancrer ou détacher une interaction à un acteur

L'ancrage ou le détachement d'une interaction active la vérification des règles d'interaction par le *gestionnaire de la cohérence présente*. La figure 4.39 montre plusieurs acteurs interreliés avec l'ancrage d'une interaction à un acteur. L'ancrage utilise la représentation UML de l'agrégation. Il définit une zone d'action qui représente la limite où les acteurs participeront à une interaction à cause de cet ancrage. La zone est définie par la règle relationnelle entre l'instance d'interaction et ses instances de personnages, les règles relationnelles inter-personnages et les règles de multiplicité. Avec un rayon d'action trop petit, illustré par la zone 1, aucun acteur n'interagit alors qu'avec un rayon d'action trop grand, illustré par la zone 3, l'interaction devient omniprésente. Lorsque bien conçues, les interactions agissent dans une zone d'action désirée par le scénariste, représentée par la zone 2.

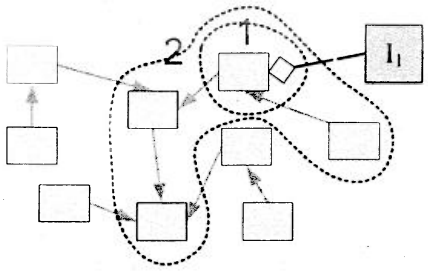

Figure 4.39 : Ancrage d'une interaction dans un MV résultant en trois zones d'action possibles pour l'interaction selon les règles d'interaction.

4.2.11 Résultat global

Pour concevoir un MV, le scénariste utilise les éléments et les actions du processus APIA. La figure 4.40 illustre un exemple de modèle conceptuel.

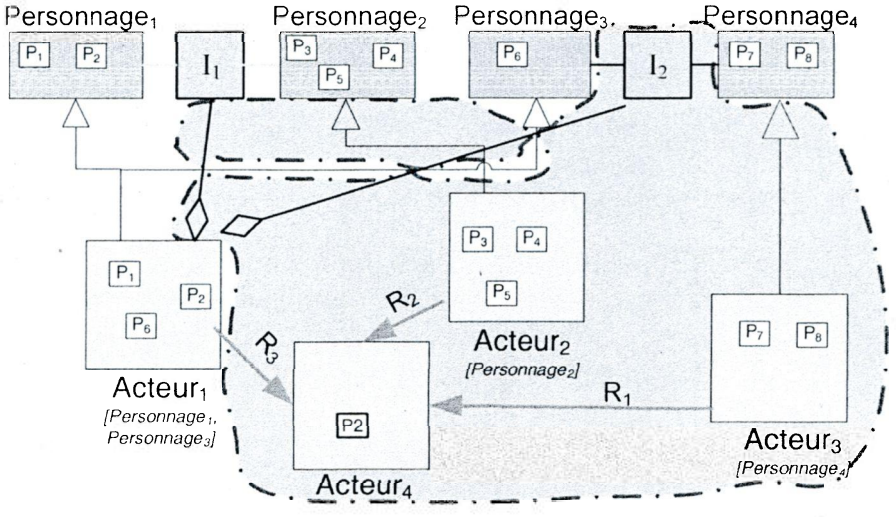

Figure 4.40 : Représentation graphique d'un modèle conceptuel avant et après (illustré par les zones foncées entourées d'un trait pointillé) le début de l'exécution d'un MV.

Les zones foncées entourées d'un trait pointillé correspondent aux extensions par rapport à l'état précédent du MV. Dans cet exemple, l'état précédent correspond à l'état avant l'exécution du MV et les ajouts sont effectués après le début de son exécution. Les actions groupées dans cette zone obéissent aux dépendances des actions du processus. En effet, il n'aurait pas été possible de définir l'interaction I_2 avant l'exécution tout en définissant le

Personnage$_4$ en cours d'exécution. Parce qu'il s'avère difficile de montrer les ajouts dans tout le modèle conceptuel, les prochains exemples montreront les extensions et les retraits dans ce diagramme mais sans les boites de personnages, ce qui correspond au diagramme APR + Interaction (APRI).

4.3 Gestionnaires

Une plus grande *agilité* requiert une gestion de la cohérence du MV, donc de son modèle conceptuel et du déroulement, surtout lorsque des actions sont posées en cours d'exécution. Les deux prochaines sections expliquent le rôle et le fonctionnement des *gestionnaires de la cohérence présente* et *temporelle*.

4.3.1 Gestion de la cohérence présente

Le *gestionnaire de la cohérence présente* gère la conformité du déroulement du MV avec son modèle conceptuel et effectue des opérations afin de le maintenir cohérent à l'instant présent. Ce gestionnaire se charge d'appliquer les conséquences des actions du processus à la fois sur le reste du modèle conceptuel et sur le déroulement. Par exemple, il gérera les instances d'interactions qui, elles-mêmes, modifieront ou étendront le MV en effectuant les opérations dont devra tenir compte ce gestionnaire. Comme autre exemple, le retrait d'une instance de propriété d'un acteur impliquera le retrait des personnages qui en dépendent.

Le MV peut être modifié par différentes actions posées par des scénaristes, des usagers et même des interactions. Certaines de ces actions n'influencent pas immédiatement le MV autrement que par l'effet de l'action elle-même. Lorsque ces actions sont posées, le gestionnaire vérifie l'état du MV pour l'instant présent. Il s'assure, entre autres, qu'un acteur ne joue pas deux fois le même rôle. Comme autre exemple, si une propriété est modifiée, cette action implique l'appel de toutes les interactions qui en dépendent. Le *gestionnaire de la cohérence présente* se charge de vérifier ces règles ainsi que les règles d'interaction définies par le scénariste.

La figure 4.41 illustre une classification des actions selon les conséquences sur les instances d'interactions qui doivent être gérées par le *gestionnaire de la cohérence présente*.

Six de ces actions n'influencent aucunement le MV autrement que leur effet immédiat. Deux actions peuvent avoir une conséquence indirecte sur un MV. Finalement, sept actions ont, en plus de leur effet immédiat, un effet possible sur les instances d'interactions.

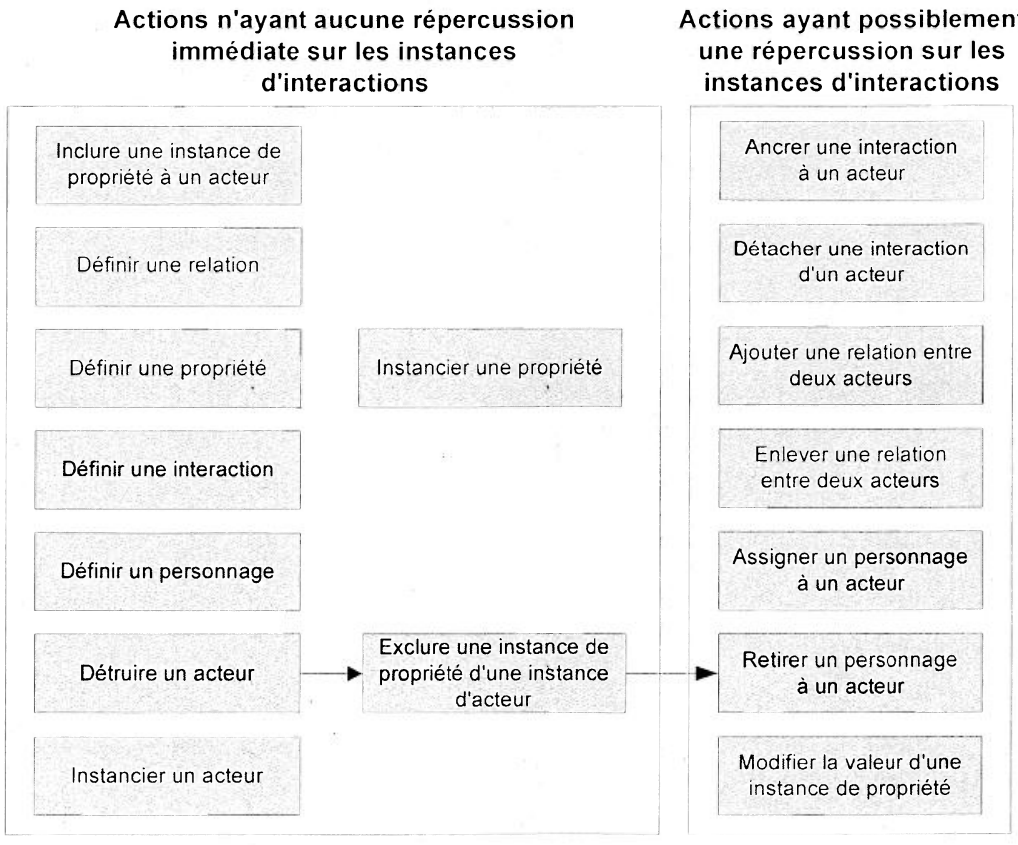

Figure 4.41 : Catégorisation des actions selon leur répercussion sur les instances d'interactions.

Tel que le montre la figure 4.42, le *gestionnaire de la cohérence présente* applique les conséquences de ces sept actions sur les instances d'interactions. Ces actions peuvent résulter en une ou plusieurs des quatre conséquences suivantes :

- création ou destruction d'une ou de plusieurs instances d'interactions ;
- destruction d'une ou de plusieurs instances de personnages utilisées par une ou plusieurs instances d'interactions ;
- appel d'une ou de plusieurs instances d'interactions ; et
- aucune modification.

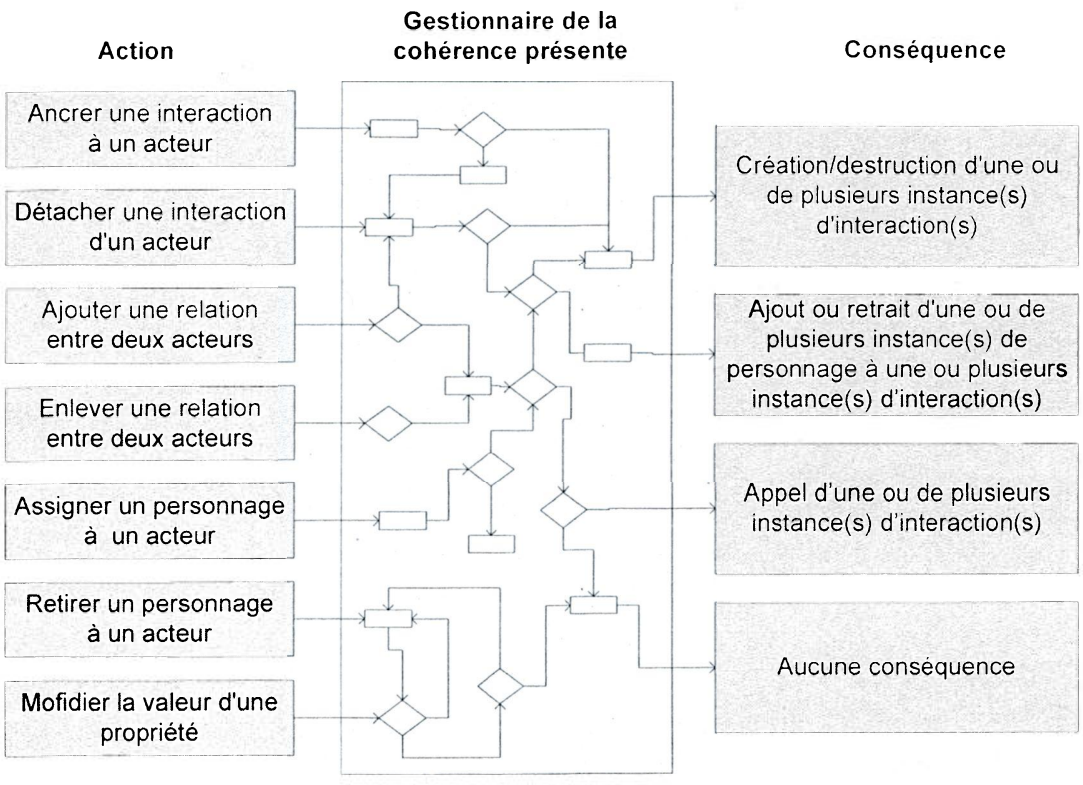

Figure 4.42 : Correspondance entre les actions posées sur le MV et les conséquences qui en découlent par le *gestionnaire de la cohérence présente*.

Une instance d'interaction gère plusieurs instances de personnages selon des conditions de relation et de multiplicité données. Une représentation concise de ces personnages s'avère essentielle car le *gestionnaire de la cohérence présente* peut parfois créer et comparer des milliers de ces conditions. Le tableau des instances de personnages (TIP) et la liste des chemins relationnels (LCR) ont été créés dans ce but. Chaque instance d'interaction contiendra un TIP et une LCR qui servent à la différentier des autres instances d'interaction. Les prochaines sections décrivent les algorithmes appliqués par le *gestionnaire de la cohérence présente* pour chacune des actions. Ils utilisent le TIP et la LCR. L'exemple qui suit présente un TIP et une LCR pour un modèle conceptuel simple.

La figure 4.43 illustre un exemple générique de modèle conceptuel.

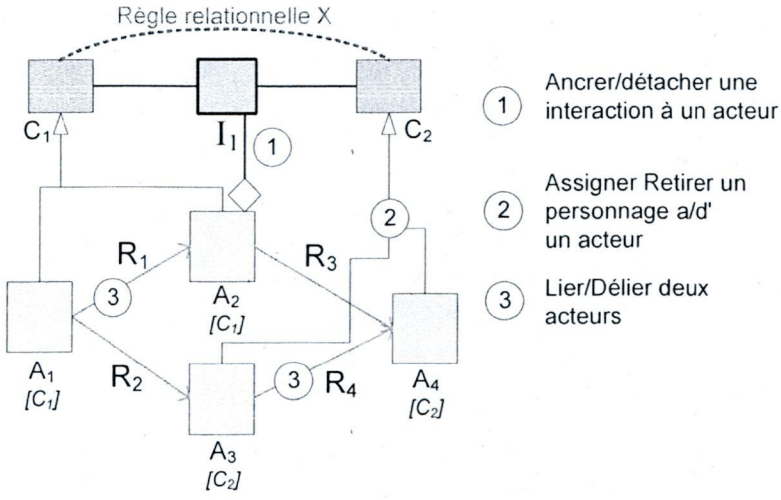

Figure 4.43 : Représentation graphique d'un modèle conceptuel d'un MV pour expliquer le fonctionnement interne du *gestionnaire de la cohérence présente*.

L'instanciation de I_1 dépend des règles de relation entre acteurs et des règles de multiplicité. Pour ce faire, le *gestionnaire de la cohérence présente* utilise d'abord le TIP qui contient l'ensemble des instances de personnages de façon structurée. La figure 4.44 en illustre un exemple.

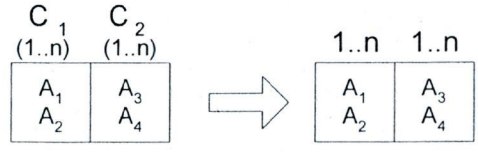

Figure 4.44 : Tableau des instances de personnages (TIP) pour l'interaction I_1 de la figure 4.43.

Le tableau de gauche montre les instances de personnages qui entrent en interaction. Il existe deux instances de personnage C_1 $\{A_1$ et $A_2\}$ ainsi que deux instances de personnage C_2 $\{A_3$ et $A_4\}$. Chaque colonne représente un personnage alors que les éléments de la colonne représentent les instances de ces personnages. En posant que C_1 est toujours le premier personnage et que C_2 est toujours le deuxième personnage dans l'interaction, le tableau de gauche se simplifie à celui de droite en ne conservant que les multiplicités.

La LCR constitue la deuxième forme d'information nécessaire au *gestionnaire de*

la cohérence présente. Elle permet de garder une trace de toutes les instances de relations permettant à chaque paire d'instances de personnages, d'entrer en interaction. La figure 4.45 illustre une LCR pour l'exemple précédent.

A_1->A_3 : (R_2) ou (R_1 et R_3 et R_4^{-1})
A_1->A_4 : (R_1 et R_3) ou (R_2 et R_4)
A_2->A_3 : (R_3 et R_4^{-1}) ou (R_1^{-1} et R_2)
A_2->A_4 : (R_3) ou (R_1^{-1} et R_2 et R_4)

Figure 4.45 : Liste des chemins relationnels (LCR) associés au tableau des instances de personnages (TIP) de la figure 4.44.

La figure 4.45 montre que A_1 et A_3 sont séparés par R_2. Un autre chemin de relations entre ces deux acteurs passe par R_1, R_3 et R_4^{-1}. Si R_2 disparaît, la règle relationnelle tient toujours car le chemin d'instances de relations entre A_1 et A_3 devient {R_1 et R_3 et R_4^{-1}}. Toutefois, si plus d'une instance de relation disparaît, par exemple R_2 et R_1, le TIP devient invalide et l'instance d'interaction disparaît.

Ancrer une interaction à un acteur

L'ancrage d'une interaction à un acteur est essentielle à son instanciation. Lorsqu'une interaction est ancrée, le gestionnaire intercepte cette action et vérifie si les règles d'interaction sont réunies. Il en résultera une instanciation ou une modification d'une ou des instances d'interaction. La figure 4.46 montre l'ancrage (symbole de l'agrégation dans UML) de trois interactions avec leur zone des acteurs qu'elles font interagir.

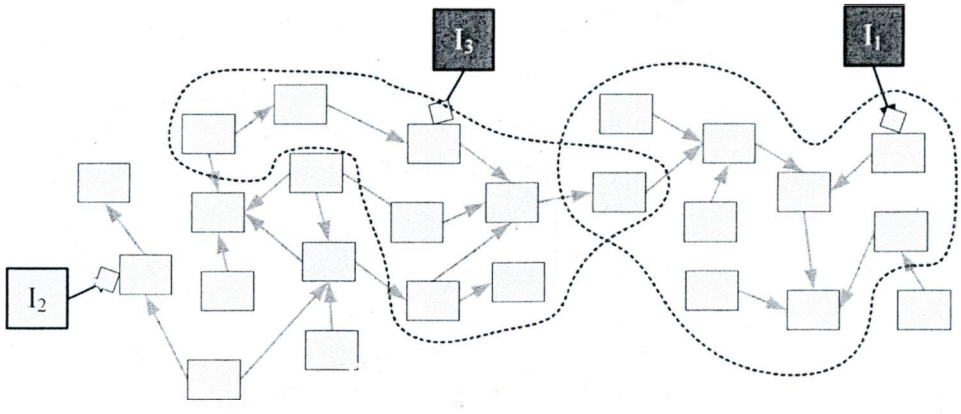

Figure 4.46 : Zones couvertes par l'ancrage des interactions.

Si les trois interactions étaient du même type ($I_1 = I_2 = I_3$), il y aurait alors recoupement de ces zones. Le scénariste étant libre de poser l'action de son choix, le gestionnaire doit gérer ces recoupements. Lorsque ce cas se présente, le gestionnaire détecte ces recoupements et ne crée qu'une seule instance d'interaction pour un même groupe d'instance de personnages (même TIP). Si les zones se recoupent partiellement, il se peut qu'une même instance soit causée par des ancrages différents. Si une zone en englobe une autre, toutes les instances causées par l'ancrage de l'interaction de la zone intérieure seront aussi causées par l'ancrage de la zone extérieure. Inversement, si deux zones ne se recoupent pas, chaque instance sera causée par un seul ancrage d'interaction.

La figure 4.47 montre le pseudo-code de l'algorithme utilisé lors de l'ancrage d'une interaction à un acteur.

- Si (interaction déjà ancrée) terminer ;
- Ancrer l'interaction à cette instance d'acteur ;
- Créer le tableau des instances de personnages (TIP) ;
- Diviser le TIP initial pour respecter les règles de multiplicité ;
- Pour chaque TIP_{valide} résultant de la division :
  - Diviser le TIP_{valide} pour respecter les règles relationnelles ;
  - Pour chaque TIP_{valide} résultants de la division :
    - Compléter la liste des chemins relationnels (LCR_{valide}) entre toutes les paires d'acteurs ;
  - Pour chaque instance de cette interaction :
    - Appeler l'algorithme de comparaison entre TIP_{valide} & LCR_{valide} et $TIP_{instance}$ & $LCR_{instance}$;
  - Si aucune instance d'interaction identique ou similaire :
    - Créer une instance d'interaction et ajouter cette instance à la liste de l'interaction ;
    - Créer un $TIP_{instance}$ et une $LCR_{instance}$ vides ;
    - Appeler l'algorithme de comparaison entre TIP_{valide} & LCR_{valide} et $TIP_{instance}$ & $LCR_{instance}$;

Figure 4.47 : Algorithme exécuté par le *gestionnaire de la cohérence présente* lorsqu'une interaction est ancrée à un acteur.

La première étape consiste à construire le TIP. La seconde étape divise le TIP initial pour respecter les règles de multiplicité. L'exemple suivant explique le fonctionnement de la suite de cet algorithme.

Pour les besoins de l'explication, la multiplicité sera désormais de 1 pour les deux personnages. Le *gestionnaire de la cohérence présente* doit donc diviser le tableau jusqu'à ce que les règles soient respectées. La figure 4.48 illustre un tel processus.

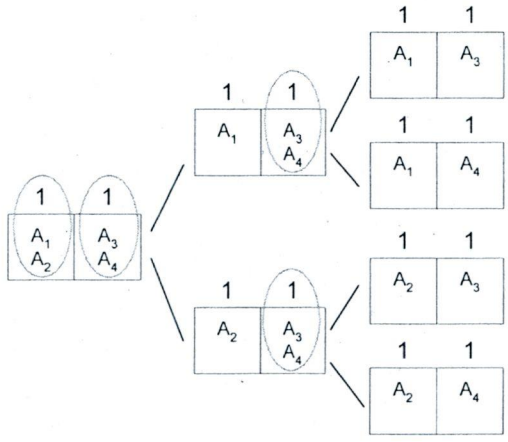

Figure 4.48 : Illustration de l'algorithme de division d'un TIP pour respecter les règles de multiplicité.

Le TIP initial contient plus d'une instance par personnage ce qui ne respecte pas les règles de multiplicité. Cette infraction aux règles de multiplicité est encerclée à la figure 4.48. L'algorithme divise d'abord le TIP initial en séparant les instances de la première colonne. S'il y avait eu trois instances pour le premier personnage, trois TIPs auraient été créés. Par la suite, l'algorithme divise encore les TIPs afin de respecter les règles de multiplicité pour la deuxième colonne. Cette division s'effectue en appelant l'algorithme de façon récursive.

Une fois que les règles de multiplicité sont respectées par tous les TIPs, il faut s'assurer que les règles relationnelles le soient également. Pour les fins de cet exemple, la règle relationnelle entre C_1 et C_2 de l'interaction I_1 sera $\{R_2$ ou $R_3\}$ et les règles de multiplicité seront de 1..n pour les deux personnages. Le TIP initial, illustré à gauche de la figure 4.49, respecte déjà les règles de multiplicité de 1..n. L'algorithme de vérification des règles de multiplicité est alors complété. Par la suite, le gestionnaire applique l'algorithme de vérification des règles relationnelles. Parce que la règle relationnelle est $\{R_2$ ou $R_3\}$, il est possible, à partir de la figure 4.49, de voir que les acteurs A_1 et A_4 ne peuvent faire partie de la même interaction. Il en est de même pour les acteurs A_2 et A_3. La figure 4.49 montre, par un trait large, les paires de personnages en défaut de règle relationnelle.

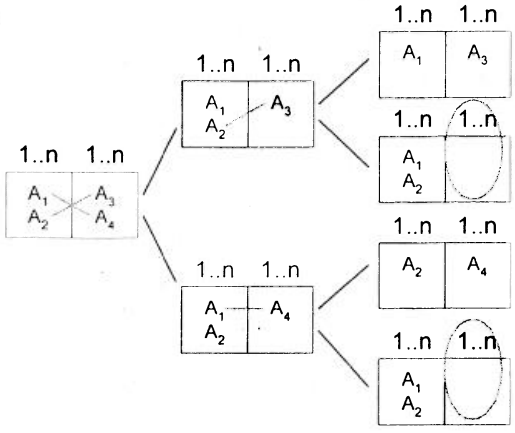

Figure 4.49 : Illustration de l'algorithme de division des TIPs pour respecter les règles relationnelles.

Pour régler ce problème, l'algorithme divise le TIP initial en deux TIPs, un avec le premier acteur de la paire et un autre avec le deuxième acteur de la paire. Le processus de poursuit récursivement jusqu'à ce que toutes les règles soient respectées ou que le TIP trouvé ne respecte plus les règles de multiplicité. Dans cet exemple, seulement le premier et le troisième TIP sont valides.

Une instance d'interaction doit exister pour tous les TIPs valides résultants. Il faut toutefois vérifier, pour chaque TIP valide obtenu, si une instance d'interaction n'existe pas déjà avec ce TIP. Par conséquent, chaque TIP sera comparé avec toutes les instances de l'interaction afin de prendre les actions qui s'imposent. Cette opération, effectuée par un autre algorithme, sera utilisée lors de la majorité des actions posées sur le MV. La figure 4.50 en montre le pseudo-code.

- Si (TIP_{valide} != $TIP_{instance}$) Terminer;
- Si (lien de cause à effet entre l'ancrage de l'interaction et cette instance d'interaction) :
  - Modifier la LCR_i
    - Pour chaque nouveau chemin relationnel (CR) de la LCR_i, créer un lien de cause à effet avec cette instance ;
    - Pour chaque ancien CR de la LCR_i, enlever le lien de cause à effet avec cette instance ;
- Sinon si (($TIP_{valide} \subset TIP_{instance}$) ou ($TIP_{instance} \subset TIP_{valide}$)) :
  - Vérifier le lien de cause à effet entre l'ancrage de l'interaction et cette instance d'interaction
  - Pour chaque nouvelle instance de propriété impliquée dans cette instance d'interaction :
    - Ajouter les implications en rapport aux propriétés : déclenchement, accès ;
    - Pour chaque propriété plus impliquée dans cette instance d'interaction :
      - Enlever les implications en rapport aux propriétés : déclenchement, accès ;
  - Pour chaque nouveau CR de la LCR :
    - Ajouter un lien de cause à effet avec cette instance ;
  - Pour chaque ancien CR de la LCR:
    - Enlever le lien de cause à effet avec cette instance ;
  - Créer la liste des instances de personnages ajoutées et détruites ;
  - Faire l'appel initialisation, modification des instances de personnages ou terminaison ;
  - Si (terminaison) retirer le lien de cause à effet entre l'ancrage de l'interaction et cette instance ;

Figure 4.50 : Algorithme de comparaison du TIP et de la LCR.

Cet algorithme effectue la comparaison entre le $TIP_{instance}$, celui de l'instance d'interaction existante, et le TIP_{valide}, celui provenant de l'action, pour déterminer si l'interaction s'applique entre un groupe identique ou similaire d'instances de personnages. La comparaison entre le TIP valide et celui de l'instance donne un des trois résultats suivants : identiques, similaires ou différents.

Dans le premier cas, les deux TIPs sont identiques. L'interaction actuelle agit donc entre les mêmes instances de personnages. Il suffit de mettre à jour la LCR parce que le TIP peut être causé par des règles relationnelles différentes.

Dans le second cas, un des TIPs est similaire à l'autre. Pour être similaires, toutes les colonnes du TIP doivent être identiques ou similaires. Il suffit d'une colonne différente pour que l'algorithme déclare les TIPs différents. La similarité dépend des multiplicités des personnages. Pour un personnage de multiplicité 1, la similarité équivaut à être identique. Toutefois, pour les multiplicités 0..1, 1..n et 0..n, la similarité a plusieurs conséquences possibles. L'exemple de la figure 4.51 illustre de tels cas.

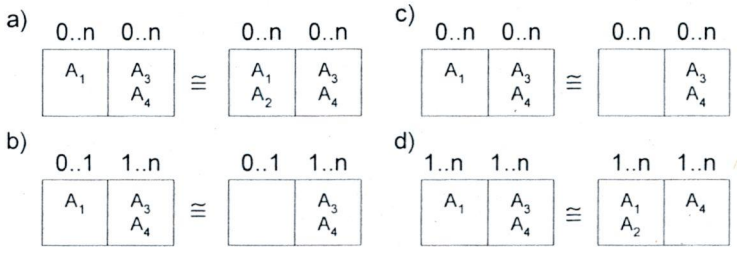

Figure 4.51 : Quatre exemples de TIP similaires.

Les TIPs de la figure 4.51 a) possèdent deux multiplicités 0..n. Une instance de personnage est ajoutée dans la première colonne du tableau de droite. Puisque la multiplicité est de 0..n, il s'agit d'une interaction entre les mêmes instances de personnages mais seulement une instance s'y est A_2 ajoutée. La deuxième colonne est identique. Par conséquent, ces deux TIPs sont similaires. L'exemple de la figure 4.51 b) compare les mêmes TIPs mais avec une instance de personnage de moins dans la colonne de gauche du tableau de droite. Les deux TIPs sont également similaires. L'exemple de la figure 4.51 c) utilise les mêmes TIPs mais avec une multiplicité différente. Dans ce cas, l'algorithme de comparaison déclare les deux TIPs comme étant différents. Par conséquent, l'instance d'interaction agit entre des instances de personnages différentes de celles du TIP valide. Il se peut que certaines colonnes des TIPs, donc les instances de personnages, soient similaires ou même identiques mais il ne suffit que d'un personnage pour déclarer les TIPs différents. Finalement, l'exemple de la figure 4.51 d) fait intervenir un ajout et un retrait d'instance de personnages dans le même tableau. Il s'agit encore de deux TIPs similaires.

Quoique toutes les actions d'ajout de relation, d'ancrage d'interaction et d'assignation de personnages passent par le même chemin, l'ancrage d'une interaction ne modifie pas les TIPs des instances de personnage existantes. L'ancrage d'une interaction ne résulte qu'en la création d'instances d'interactions ou en l'ajout d'une dépendance entre une instance existante et un ancrage. Les algorithmes de division du TIP seront exécutés lors de l'assignation ou du retrait d'un personnage et lors de l'ajout ou du retrait d'une liaison entre des acteurs.

Détacher une interaction d'un acteur

Cette opération s'effectue de manière totalement différente de la précédente. La figure 4.52 montre l'algorithme appliqué par le *gestionnaire de la cohérence présente* lorsque qu'une

interaction est détachée d'un acteur.

- Si (interaction pas ancrée) terminer ;
- Pour chaque instance de l'interaction :
  - Enlever le lien de cause à effet entre l'ancrage de cette interaction et cette instance d'interaction ;
  - Si l'instance ne dépend d'aucun autre ancrage :
    - Créer un TIP et une LCR vides (TIP_{vide} et la LCR_{vide}) ;
  - Appeler l'algorithme de comparaison du TIP_{vide} et de la LCR_{vide} avec le $TIP_{instance}$ et la $LCR_{instance}$;

Figure 4.52 : Algorithme exécuté par le *gestionnaire de la cohérence présente* lorsqu'une interaction est détachée d'un acteur.

Cet algorithme s'avère plus simple et plus rapide que celui de l'ancrage de l'interaction. Lors de la création de l'instance d'interaction, le TIP et la LCR de l'instance sont vides. La comparaison avec un TIP valide produit des connexions entre les instances de personnages, de propriétés et de relations avec cette instance d'interaction. Pour détruire cette instance et annuler ces connexions, il suffit d'utiliser un TIP et d'une LCR vides.

Assigner un personnage à un acteur

La gestion de cette action introduit deux boucles supplémentaires à l'algorithme de la figure 4.47. La première tient compte de toutes les interactions qui dépendent de ce personnage. La deuxième tient compte de tous les ancrages de l'interaction. La figure 4.53 illustre ces deux boucles supplémentaires.

- Si (personnage déjà assigné à l'acteur) terminer ;
- Assigner un personnage à l'acteur et créer une instance de personnage ;
- Pour chaque interaction dépendante du personnage :
  - Pour chaque point d'ancrage de l'interaction :
    - CONTINUER À LA TROISIÈME LIGNE DE L'ALGORITHME DE L'ANCRAGE ;

Figure 4.53 : Algorithme exécuté par le *gestionnaire de la cohérence présente* lorsqu'un personnage est assigné à un acteur.

L'exemple suivant montre le fonctionnement de cet algorithme.

Les TIPs de la figure 4.54 sont basés sur l'exemple de la figure 4.43 dans lequel tous les personnages ont déjà été assignés aux acteurs sauf l'assignation de C_1 à A_2. La seule instance d'interaction de ce MV possède un TIP avec un acteur qui joue C_1 et deux acteurs qui jouent C_2. Cette instance respecte toutes les règles de multiplicité et de relation entre acteurs.

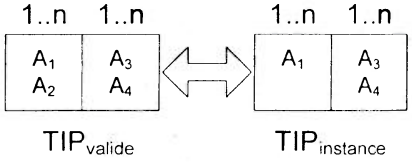

Figure 4.54 : Comparaison d'un TIP valide (TIP_{valide}) avec le TIP de l'instance d'interaction ($TIP_{instance}$) lors de l'assignation du personnage C_1 à l'acteur A_2.

Si un personnage C_1 est ajouté à l'acteur A_2, le gestionnaire se retrouve avec un seul TIP valide, soit celui de gauche à la figure 4.54. Le gestionnaire compare donc ce TIP avec ceux de toutes les instances d'interaction. Dans ce cas, il en existe une seule et son TIP est illustré à droite dans la figure 4.54. L'algorithme de comparaison déduit que les deux TIPs sont similaires et il ajoute l'instance de personnage A_2 à l'instance d'interaction actuelle. Le gestionnaire prend également en charge les conséquences de l'ajout de ces instances de personnages. Par exemple, il ajoute les implications en rapport aux accès aux propriétés pour cette instance de personnage. En effet, la modification d'une instance de propriété de cette nouvelle instance de personnage causerait l'appel de l'instance d'interaction. De plus, le gestionnaire envoie un message d'ajout d'instance de personnages à l'instance d'interaction pour que celle-ci prenne les mesures qui s'imposent. Par exemple, si l'instance de personnage ajoutée était visible, alors l'interaction de visualisation créerait un nouvel élément graphique avec la géométrie appropriée et une position initiale. Par la suite, si la position changeait, d'autres mécanismes du gestionnaire enverraient un message à l'instance d'interaction.

Il arrive aussi que l'assignation d'un personnage à un acteur permette à une interaction de respecter ses règles. En effet, lorsque les TIPs des toutes les instances d'interaction ne sont pas similaires ou identiques au TIP valide, alors le gestionnaire instancie une nouvelle interaction.

Cet algorithme ajoute deux niveaux de complexité par rapport à celui de l'interaction.

Il s'avère donc plus lent d'un facteur qui dépend du produit du nombre de personnages par interaction par le nombre d'ancrages par interaction.

Retirer un personnage à un acteur

Le retrait d'une instance de personnage prend plus de temps que le détachement d'une interaction. La figure 4.55 illustre l'algorithme employé par le *gestionnaire de la cohérence présente*.

- Si (l'acteur ne joue pas ce rôle) terminer ;
- Pour chaque interaction dépendant de ce personnage :
  - Pour chaque instance de cette interaction :
    - Créer un $TIP_{modifié}$ à partir du $TIP_{instance}$ en enlevant toutes références à l'instance de personnage concernée ;
    - Créer le $LCR_{modifée}$ à partir du $LCR_{instance}$ en enlevant tous liens dont l'extrémité comprend l'instance d'interaction ;
    - Appeler l'algorithme de comparison du $TIP_{modifié}$ et de la $LCR_{modifié}$ avec le $TIP_{instance}$ et la $LCR_{instance}$;
- Détruire l'instance de personnage ;
- Retirer le personnage à l'instance d'acteur ;

Figure 4.55 : Algorithme exécuté par le *gestionnaire de la cohérence présente* lorsqu'un personnage est retiré d'un acteur.

Pour chaque interaction dépendante de ce personnage, le gestionnaire doit vérifier toutes les instances d'interactions. Le gestionnaire crée un TIP en enlevant toutes les instances de personnage du TIP de l'instance.

La figure 4.56 illustre un exemple de retrait d'instance de personnage.

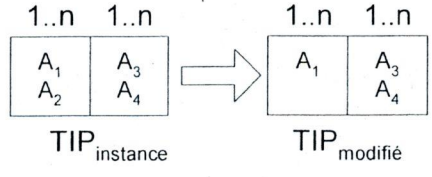

Figure 4.56 : Création d'un TIP modifié à partir du TIP de l'instance d'interaction lors du retrait du personnage C_1 à l'acteur A_2.

Dans cet exemple, C_1 est retiré A_2. Cette opération ne fait qu'enlever des instances de personnages au TIP. S'il advenait le retrait du personnage C_1 à l'acteur A_1, alors il n'y aurait plus aucune instance de personnage pour un personnage de multiplicité 1..n et le gestionnaire ferait disparaître cette instance d'interaction.

Comme pour l'ajout d'un personnage à un acteur, le retrait d'un personnage a deux conséquences possibles : laisser l'instance d'interaction en place ou causer sa disparition. Dans le premier cas, le gestionnaire enverra un message de retrait d'instance de personnage à l'instance d'interaction. Dans le second cas, il enverra le message approprié au type d'appel demandé. Si l'interaction demande à être notifiée des ajouts ou des retraits des instances de personnages, alors l'ensemble des instances de personnages présents dans le TIP sera envoyé à l'instance d'interaction pour qu'elle se termine. Autrement, le message de terminaison sera appelé sur cette instance d'interaction.

Ajouter une relation entre deux acteurs

Pour chaque ajout de relation entre deux acteurs, le gestionnaire doit vérifier toutes les interactions qui dépendent de ce lien. La figure 4.57 illustre cet algorithme.

```
- Si (lien identique existe entre les deux acteurs) terminer ;
- Pour chaque interaction qui dépend de ce lien :
    - CONTINUER À LA TROISIÈME LIGNE DE L'ALGORITHME DE l'ANCRAGE ;
```

Figure 4.57 : Algorithme exécuté par le *gestionnaire de la cohérence présente* lorsqu'une relation est ajoutée entre deux acteurs.

Cette action résulte en les mêmes conséquences que l'assignation de personnages. Toutefois, il se peut que, pour deux TIPs identiques, les LCRs soient différents. Cette différence est gérée par l'algorithme de comparaison montré à la figure 4.50 aux lignes 12 à 15. Cette information sera utilisée lors du retrait des relations entre acteurs.

Enlever une relation entre deux acteurs

Cette action s'avère la plus complexe du point de vue la de conception algorithmique. Elle est toutefois plus rapide à s'exécuter que la majorité des autres actions car toutes les

informations nécessaires à la gestion de cette action ont été générées sous une forme optimale lors de la création de l'instance de l'interaction. La figure 4.58 montre l'algorithme utilisé lorsqu'une instance de relation est enlevée entre deux acteurs.

- Si (acteurs non liés par ce lien) terminer ;
- Pour chaque instance d'interaction dépendante de ce lien :
  - Trouver toutes les paires d'acteurs dont le CR est coupé dans la $LCR_{instance}$;
  - Diviser le $TIP_{instance}$ récursivement jusqu'à ce que les paires d'acteurs n'y soient plus présent :
    - S'assurer que les règles de multiplicité sont respectées ;
    - Créer la nouvelle LCR ;
  - Si (nombre de TIP obtenu == 0) enlever cette instance d'interaction puis tErminer ;
  - Si (nombre de TIP obtenu >= 1) appeler l'algorithme de comparaison avec les premiers TIP et LCR ;
  - Si (nombre de TIP obtenu > 1) créer des instances d'interaction avec les autres TIPs et LCRs ;

Figure 4.58 : Algorithme exécuté par le *gestionnaire de la cohérence présente* lorsqu'une relation est enlevée entre deux acteurs.

Le retrait d'une instance de relation doit être communiquée à toutes les instances des interactions qui en dépendent. Le TIP et la LCR de chacune de ces instances seront donc modifiés. Il existe quatre résultats du retrait d'une instance de relation : la destruction de l'instance, la création de nouvelles instances, modification d'une instance par le retrait d'instances de personnages et, finalement, la modification de la LCR. L'exemple suivant permettra d'expliquer chacun de ces cas.

La figure 4.59 a) montre une interaction I_1 avec trois personnages. Les deux premiers personnages ont une multiplicité de 1 alors que le troisième a une multiplicité de 1..n. Les règles relationnelles sont définies entre C_1 et C_3, entre C_1 et C_2 et de C_3 avec lui-même. L'exposant n désigne une succession illimitée d'instances de relation R_2 comme chemin de relation valide. La figure 4.59 b) montre les acteurs avec les instances de relations. Les exposants ' et " servent à distinguer des instances différentes d'un même type de relation.

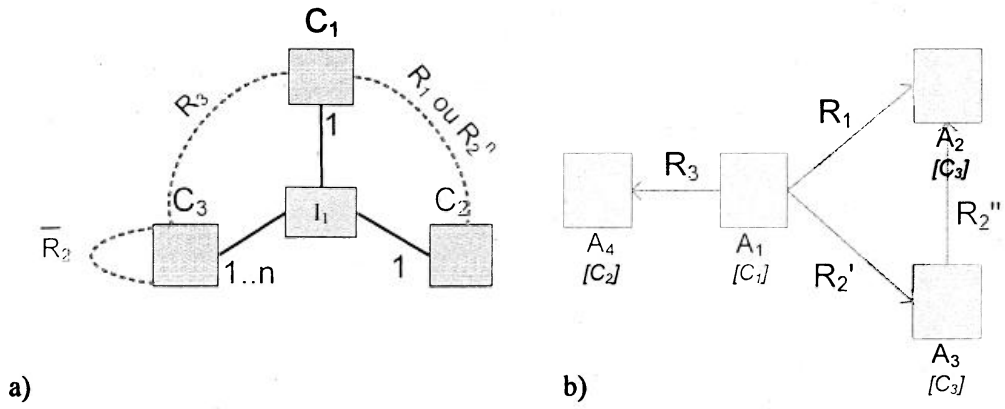

Figure 4.59 : Représentation graphique d'un modèle conceptuel pour expliquer l'algorithme utilisé par le gestionnaire lors du retrait d'une instance de relation entre deux acteurs.

La figure 4.60 donne le TIP et la LCR pour le MV décrit précédemment.

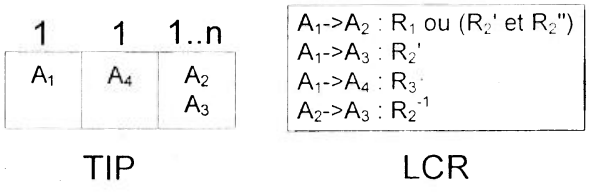

Figure 4.60 : TIP et LCR générés par le *gestionnaire de la cohérence présente* pour le MV de la figure 4.59.

Premièrement, le TIP peut devenir invalide lorsqu'une instance de relation entre deux acteurs est enlevée. Si, par exemple, l'instance de relation R_3 est retirée entre les acteurs A_1 et A_4, la LCR permet d'affirmer que la règle relationnelle entre A_1 et A_4 n'est plus respectée. Par conséquent, l'algorithme de division des TIPs doit s'appliquer pour respecter les règles relationnelles. La figure 4.61 montre le résultat de la division. Il n'existe aucun TIP conforme aux règles relationnelles et de multiplicité. Par conséquent, le retrait de R_3 conduit à la destruction de cette instance d'interaction.

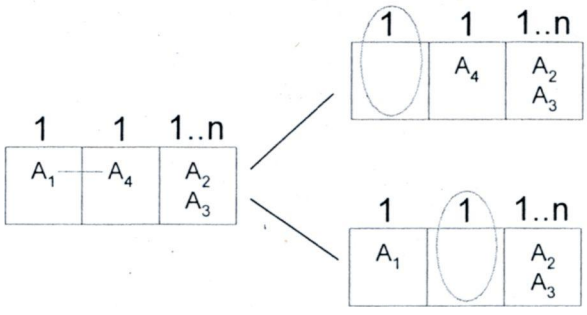

Figure 4.61 : Impossibilité de respecter les règles relationnelles et de multiplicité lors de la division du TIP.

Comme deuxième conséquence du retrait d'une instance de relation, il se peut que cette action crée plusieurs instances d'interactions. Si, par exemple, l'instance de relation R_2" est enlevée, le chemin des relations entre les acteurs A_1 et A_2 ne respecte plus la règle relationnelle pour ces personnages. Par conséquent, le *gestionnaire de la cohérence présente* doit séparer ces instances de personnages de la même instance d'interaction. Cette division est montrée à la figure 4.62.

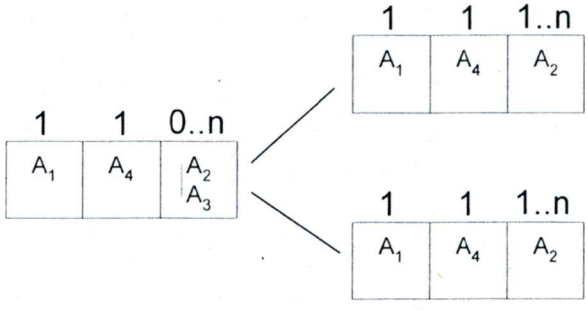

Figure 4.62 : Division du TIP original en deux TIPs, donc deux instances d'interactions, due à l'impossibilité d'avoir les acteurs A_2 et A_3 dans le même tableau.

Lorsqu'une instance d'interaction disparaît pour laisser la place à plusieurs autre instances, celle existante est d'abord modifiée par l'ajout et le retrait d'instances de personnages. Les autres instances d'interactions sont créées avec les TIPs restants.

Comme troisième conséquence du retrait d'une instance de relation, il arrive parfois que le retrait d'une instance de relation ne fasse que modifier un TIP d'une instance d'interaction existante. Par exemple, si l'instance de relation R_2' est retirée à la figure 4.63, ce retrait oblige la séparation de A_1 et A_3 de la même instance d'interaction.

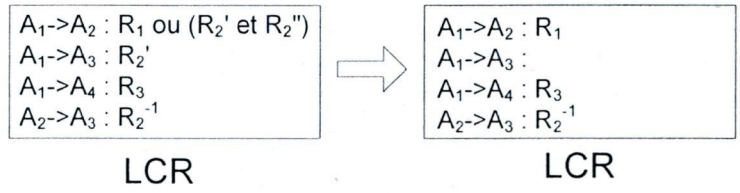

Figure 4.63 : Retrait de l'instance de relation R_2' menant à l'impossibilité d'avoir les deux instances de personnages dans la même instance d'interaction.

La figure 4.64 montre la division du TIP en enlevant un des deux acteurs à tour de rôle. Le deuxième TIP ne respecte pas les règles relationnelles. Il ne reste donc que le premier TIP.

Il reste un TIP résultant du retrait de l'instance de relation R_2'. Le gestionnaire ne détruira donc pas l'instance d'interaction en cours pour en créer une autre avec le nouveau TIP. Il modifiera l'instance actuelle et appellera la méthode ajout ou retrait de personnages de l'interaction.

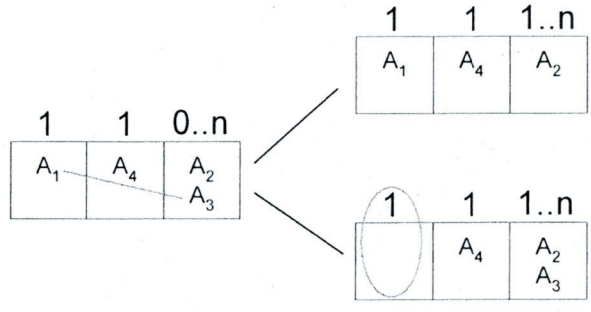

Figure 4.64 : Division d'un TIP résultant en un seul TIP valide à cause du non respect de la règle de multiplicité 1 dans le second TIP.

Comme quatrième possibilité, le retrait d'une instance de relation peut résulter en la modification de la LCR d'une instance d'interaction, sans modifier le TIP. Par exemple, l'instance de la relation R_1 est retirée. Cette action cause la modification de la LCR, comme il est possible de le voir à la figure 4.65, mais ne perturbe aucunement le TIP.

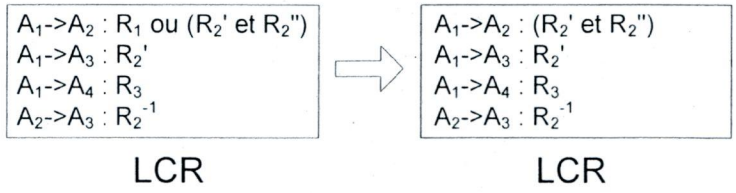

Figure 4.65 : Retrait de l'instance de la relation R_1 ne modifiant en rien les TIPs de cette instance d'interaction car toutes les règles relationnelles sont respectées.

Par conséquent, une telle action laisse l'instance d'interaction intacte. Le gestionnaire n'enverra pas de message à l'instance d'interaction.

Modifier la valeur d'une propriété

La dernière action à gérer le *gestionnaire de la cohérence présente* est la modification des instances de propriétés par une instance d'interaction. Puisque cette action dépend du temps, son fonctionnement est expliqué à la section sur la *gestion de la cohérence temporelle*.

4.3.2 Gestion de la cohérence temporelle

Jusqu'à maintenant, les MVs présentés étaient statiques. Toutefois, un MV évolue dans le temps à partir de l'état présent et des entrées. Il est de la responsabilité du *gestionnaire de la cohérence temporelle*, désigné *ordonnanceur* pour la suite, de faire progresser le MV dans le temps de façon cohérente. Le rôle de l'*ordonnanceur* est de faire évoluer le MV dans une séquence conforme au modèle conceptuel. Toutefois, ce rôle ne sera pas abordé dans sa totalité. Le classement des priorités et l'optimisation d'exécution du MV, autre rôle possible, ne sera pas abordé car sa complexité dépasse le cadre de cette thèse. L'*ordonnanceur* permettra de concevoir une implantation de l'architecture **APIA** afin de démontrer les concepts d'*agilité* présentés à travers des exemples d'applications.

L'*ordonnanceur* est basé sur les simulations par événements discrets. Toutefois, utilisé comme telles, celles-ci ne permettent pas d'atteindre le niveau de dynamisme et d'*agilité* recherché dans cette thèse. Des champs d'information, incluses dans le modèle conceptuel, permettent à l'*ordonnanceur* de supporter les caractéristiques de l'*agilité* en cours d'exécution.

Dans **APIA**, chaque action est convertie en un événement que gère l'*ordonnanceur*. Tel

que présenté à la figure 4.41, certaines actions ne sont pas prises en charge par le *gestionnaire de la cohérence présente*. Elles n'impliqueront également aucun traitement par l'*ordonnanceur* sauf si elles causent une autre action gérée par le *gestionnaire de la cohérence présente*. Pour les actions prises en charge par ce dernier, l'*ordonnanceur* les classe en deux catégories telles qu'illustrées à la figure 4.66 : les actions qui ne dépendent pas du temps, ou, qui, du moins, s'effectuent toujours à l'instant présent et l'action visant la modification de propriété qui dépend du temps. Cette dernière peut s'effectuer à l'instant présent ou à un moment dans le futur. Les actions de la première catégorie ont toujours priorité sur l'action de modifier une propriété.

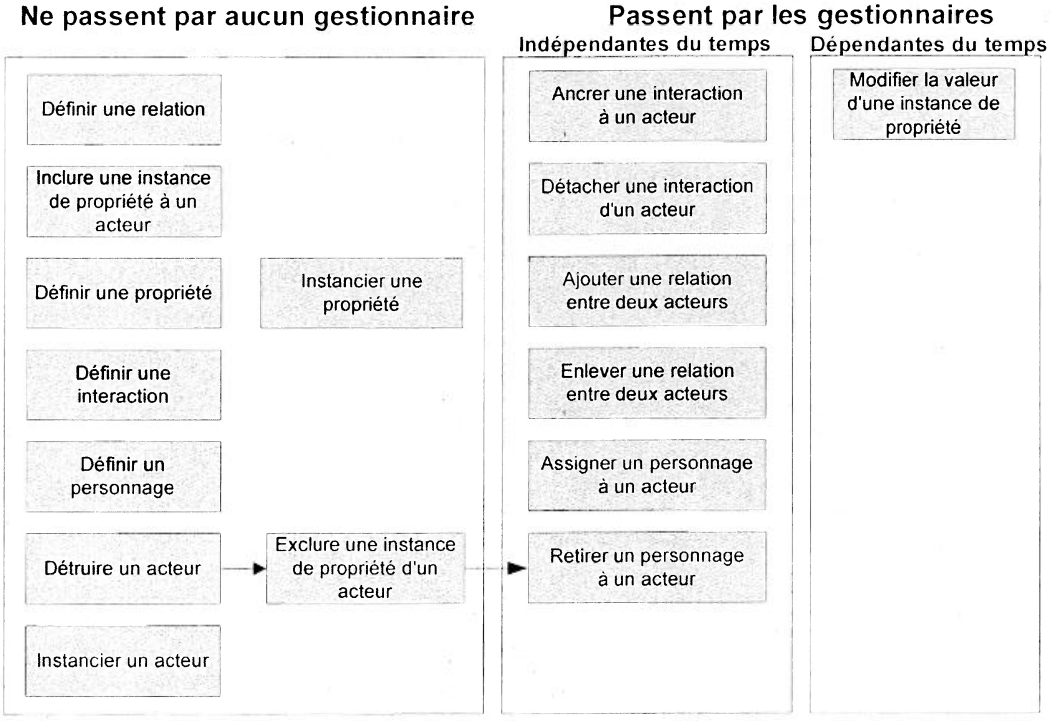

Figure 4.66 : Classement des actions en regard de leur gestion par l'*ordonnanceur*.

Rôle de l'ordonnanceur pour les actions indépendantes du temps

Lorsqu'une action indépendante du temps est posée, le *gestionnaire de la cohérence présente* intercepte cette action et crée un événement qu'il ajoute à une file, illustrée à la figure 4.67. Cet événement sera traité ultérieurement.

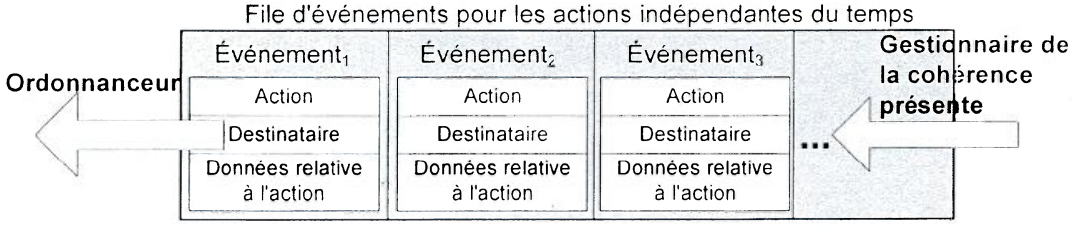

Figure 4.67 : Ajout d'un événement à une file par le gestionnaire suite à une action suivit par son traitement ultérieur par l'*ordonnanceur*.

Si l'événement est le seul dans la file, l'*ordonnanceur* retirera cet événement dès qu'il en aura fini avec l'événement en cours et appellera le *gestionnaire de la cohérence présente* pour qu'il le traite. Une telle file est nécessaire car une action peut causer d'autres actions en cascade et l'utilisation d'une pile, par exemple, causerait de l'incohérence dans le MV. L'exemple suivant donne plus de détails.

> La figure 4.68 illustre une instance d'interaction I_1 en cours d'exécution, représentée par le numéro 1. Ensuite, I_1 assigne un personnage C_3 à l'acteur A_1, représenté par le numéro 2 à la figure 4.68. À ce moment, le *gestionnaire de la cohérence présente* crée une instance d'interaction I_2 avec l'instance de personnage A_1, représenté par le numéro 3 à la figure 4.68, parce que toutes les règles d'interaction sont respectées et qu'aucune instance d'interaction ne fait déjà interagir ces mêmes instances de personnages. Si l'interaction I_2, lors de son premier appel, enlève l'instance de relation R_1 (numéro 4), le gestionnaire se rend compte que I_1 ne respecte plus les règles de l'interaction dont elle est l'instance et détruit celle-ci.

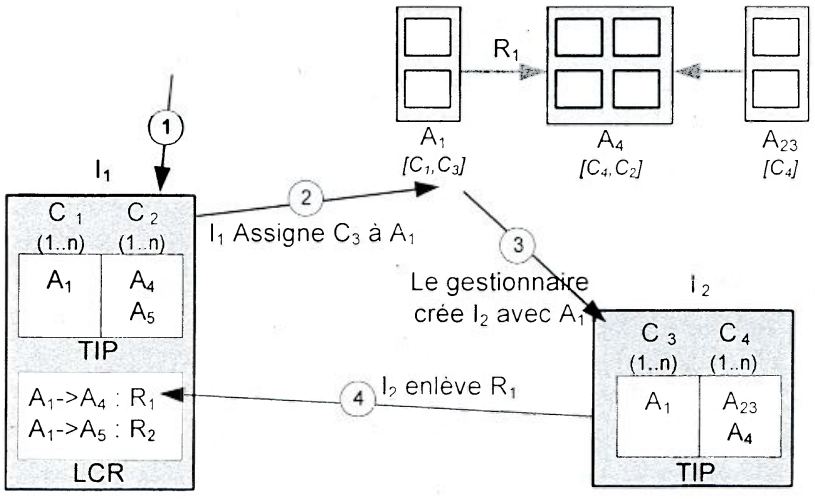

Figure 4.68 : Diagramme d'instances d'éléments qui montre que la séquence d'actions circulaire est résolue par l'utilisation d'une file d'événements.

Lorsqu'une action indépendante du temps est posée, elle est ajoutée à une file. Quand l'*ordonnanceur* a terminé avec l'événement en cours, il traite l'événement suivant dans la file. La figure 4.69 montre les cinq états par lesquels passe la file d'événements pour l'exemple précédent. L'exécution du MV débute avec un événement E_0. L'*ordonnanceur* retire E_0. Il ne reste donc plus d'événements dans la file. L'*ordonnanceur* demande alors au *gestionnaire de la cohérence présente* de traiter cet événement. Le processus se poursuit tel que décrit précédemment.

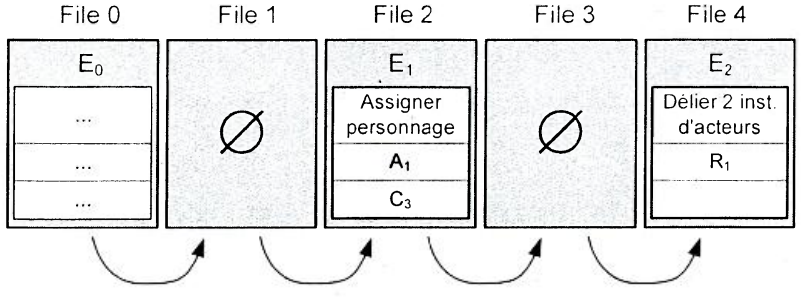

Figure 4.69 : Évolution de la file d'événements pour l'exemple de la figure 4.68.

La file d'évènements permet aux interactions d'effectuer l'ensemble de leurs actions et ensuite de voir ces actions se concrétiser. Lorsque le *gestionnaire de la cohérence présente* traite un événement, il peut causer l'appel ou la création de nombreuses instances

d'interactions. Ces conséquences de l'action ne passent pas par l'*ordonnanceur* car le *gestionnaire de la cohérence présente* les prend en charge immédiatement. La conséquence d'une action est toujours, à une exception près, en rapport avec les instances d'interactions. Les conséquences possibles sont la création d'une instance, l'ajout ou le retrait d'instances de personnages ou la destruction de l'instance. Toutefois, si une instance d'interaction pose une action lors de son initialisation suite à une autre action, celle-ci est placée dans la file.

Rôle de l'ordonnanceur pour l'action dépendante du temps

Seule l'action de modification de la valeur d'une instance de propriété dépend du temps. Si une ou plusieurs propriétés sont modifiées par une interaction, le *gestionnaire de la cohérence présente* détecte ces modifications, crée un événement marqué du temps de modification de la propriété(s) et place cet événement dans une liste ordonnée selon le temps de l'événement. À chaque ajout d'événement, la liste est triée par ordre temporel croissant. L'*ordonnanceur* traitera cet événement lorsqu'il sera celui avec le temps de valeur la plus basse. Cette liste est montrée à la figure 4.70.

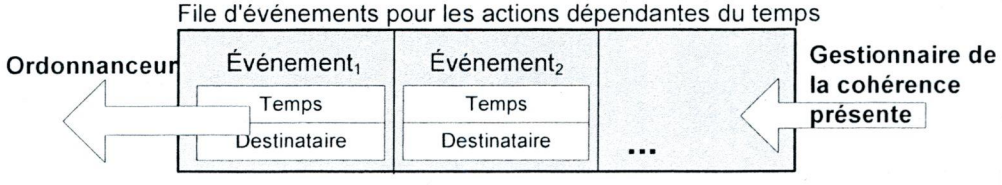

Figure 4.70 : Ajout, par le *gestionnaire de la cohérence présente*, d'un événement à une liste ordonnée temporellement lors de la modification d'une propriété.

Lorsque non spécifié, le temps de modification d'une propriété, donc de l'événement, est celui du temps actuel dans le MV. Si une instance d'interaction modifie une instance de propriété, le *gestionnaire de la cohérence présente* crée autant d'événements qu'il y a d'instances d'interactions qui utilisent cette même instance de propriété en lecture. L'exemple qui suit explique le fonctionnement de l'*ordonnanceur* lorsque des instances de propriétés sont modifiées.

La figure 4.71 montre trois instances d'interactions qui modifient des instances de propriétés. Pour le reste de la thèse, la lecture sera illustrée par un trait fin alors que l'écriture, accompagnée d'une lecture ou non, le sera par un trait gras.

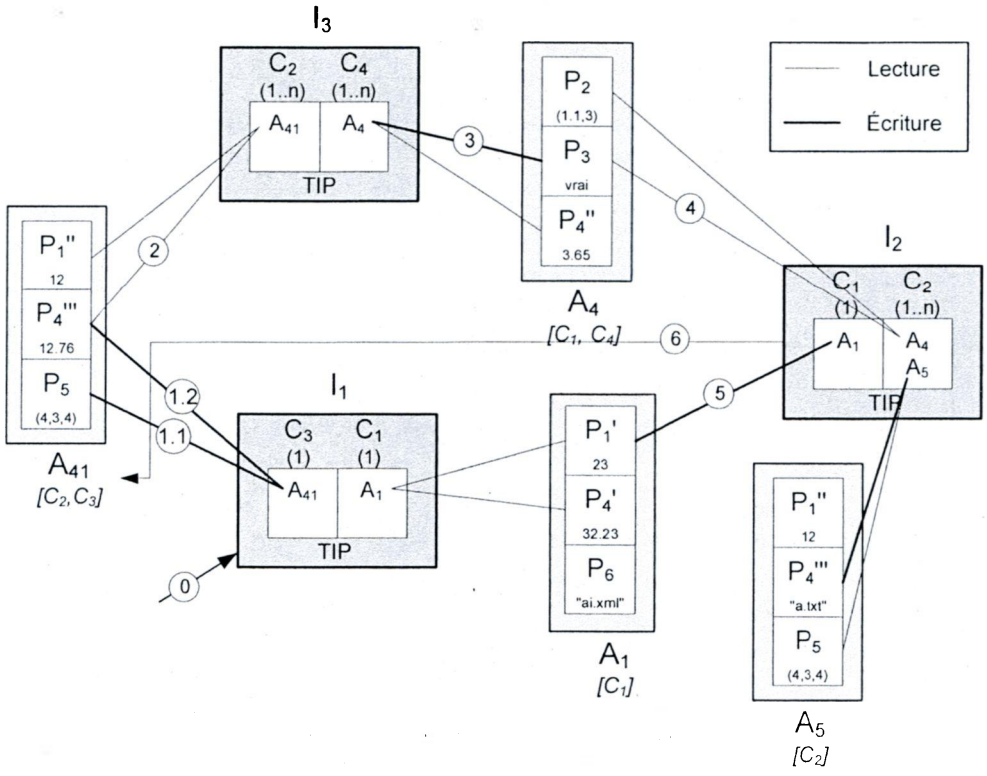

Figure 4.71 : Diagramme de dépendance entre instances de propriétés et d'interactions.

Pour simplifier l'exemple, toutes les propriétés sont modifiées à l'instant présent (T_0) sauf pour P_1" qui est modifiée 0.01 seconde dans le futur. La figure 4.72 montre les étapes, identifiées par un numéro, par lesquelles passe la liste des événements pour cet exemple.

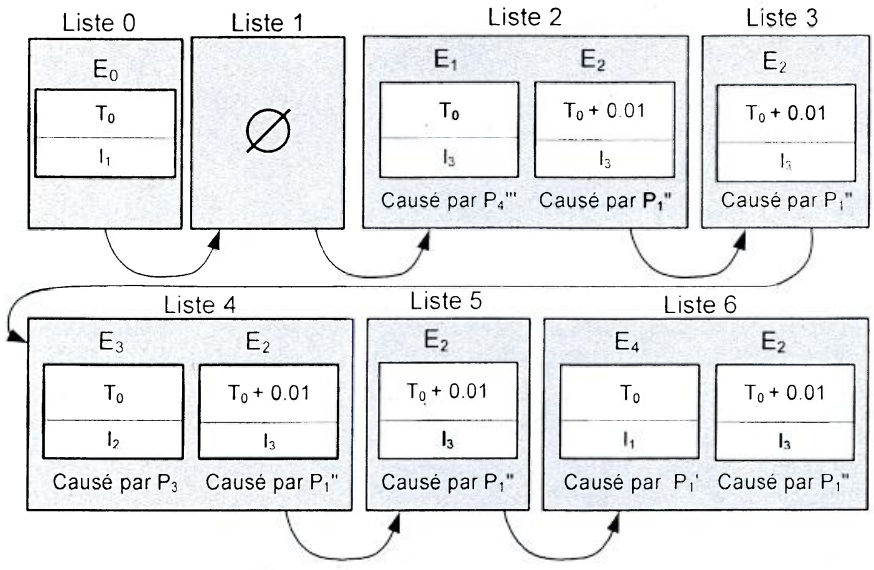

Figure 4.72 : Évolution de la liste d'événements pour l'exemple de la figure 4.71.

Initialement, l'*ordonnanceur* retire l'événement E_0 de la liste 0 et appel l'instance d'interaction I_1. Tel qu'indiqué par l'action 1.1 sur la figure 4.71, celle-ci modifie l'instance de propriété P_4''' de l'acteur A_{41} en faisant passer la valeur de 12.76 à 12.87 et P_1'' en modifie (action 1.2) P_1'' en modifiant la valeur de 12 à 16. Lorsque l'appel de I_1 se termine, le *gestionnaire de la cohérence présente* regarde les modifications des instances de propriétés et les instances d'interactions qui dépendent des instances de propriétés de l'acteur A_{41}. Il génère alors deux événements contenues dans *Liste 2* : E_1 destiné à l'instance d'interaction I_3 avec un temps T_0 et E_2 destiné également à I_3 mais avec un temps $T_0 + 0.01$. Une fois ces événements ajoutés, le *gestionnaire de la cohérence présente* redonne le contrôle à l'*ordonnanceur*. Celui-ci prend le prochain événement dans sa file, c'est-à-dire E_1. Cet événement cause l'appel de I_3 (action 2). L'*ordonnanceur* appelle cette instance d'interaction qui modifie l'instance de propriété P_3 de vrai à faux (action 3). Lorsque I_3 a terminé, le *gestionnaire de la cohérence présente* détecte que cette modification génère un événement E_3 pour l'instance d'interaction I_2 (action 4). Cet événement est ajouté à *Liste 3*, ce qui donne *Liste 4*. Le *gestionnaire de la cohérence présente* redonne à nouveau le contrôle à l'*ordonnanceur* qui prend le prochain événement dans *Liste 4*, c'est-à-dire E_3. Cet événement cause l'appel de I_2 qui modifie P_1' (action 5) et retire le personnage C_3 à l'acteur A_{41} (action 6). L'*ordonnanceur* ajoute alors deux événements : E_4 pour la modification de l'instance de propriété P_1' et E_5 pour le retrait du personnage C_3 à A_{41}. Le premier événement est ajouté à la liste des événements de la modification des propriétés et le second à la file des événements pour les actions non dépendantes du temps. Ce dernier événement

n'est pas montré à la figure 4.72. Ensuite, le *gestionnaire de la cohérence présente* redonne le contrôle à l'*ordonnanceur*. Si l'*ordonnanceur* considérait toutes les actions de la même manière, ce dernier traiterait l'événement E_3 et ensuite E_4. Toutefois, comme il a été mentionné précédemment, l'*ordonnanceur* traite les actions indépendantes du temps (E_4) puis celles qui dépendent du temps (E_3). Dans ce cas, le retrait du personnage C_3 à l'acteur A_{41} cause la destruction de l'instance d'interaction I_1. La destruction de cette interaction est effective immédiatement, ce qui cause l'appel de I_1 pour sa terminaison. L'événement E_4 ne se rendra jamais à l'interaction I_1. Le *gestionnaire de la cohérence présente* se chargera, avec l'*ordonnanceur*, de détruire cet événement. Par la suite, l'événement E_1 sera traité lorsque le temps de simulation atteindra $T_0 + 0.01$.

Simulations à événements discrets et temps logique

Le modèle d'avancement du temps est inspiré des simulations à événements discrets basés sur plusieurs types temps : le temps physique (*physical time*), le temps de simulation (*simulation time*) et le temps de l'horloge (*wallclock time,* T_w). Selon le but visé, ces temps seront plus ou moins égaux. Idéalement, dans les simulations interactives tels que les MVs, le temps physique devrait toujours égaler celui de l'horloge. Cependant, une telle synchronisation s'avère impossible. Le temps de simulation sert justement à régler ce problème. Le temps physique est remplacé par une série ordonnée de valeurs temporelles et l'avancement d'un temps à l'autre s'effectue lorsque le temps de l'horloge est près du temps de simulation. Ce temps qui avance se nomme temps logique. Par conséquent, ce temps logique ne peut prendre que des valeurs de la série ordonnée du temps de simulation. Cette série ordonnée correspondra à des événements.

Premièrement, il est impossible d'avancer le temps logique au moment précis où le temps de l'horloge rejoint celui de l'événement. Cette impossibilité tient à la limite des ordinateurs et des systèmes d'exploitation, même ceux dits temps réel, à contrôler leur temps avec précision. Le temps de l'horloge est une variable qui avance de façon continue avec une résolution infinie. Dans une simulation, le temps de l'horloge est représenté par l'horloge de l'ordinateur qui possède une certaine résolution. Deuxièmement, le temps de traitement d'un événement peut nécessiter plus de temps que celui qui le sépare du suivant. Finalement, même si un événement n'était pas traité exactement à l'instant où le temps de l'horloge passe par celui de l'événement, un usager dans le monde réel ne verrait pas la différence en-dessous d'un

certain seuil. Par exemple, dans l'hypothèse que la vision humaine ne perçoit pas un délai de moins de 30 millisecondes (ms), le traitement à plus ou moins la moitié de cet intervalle ne serait pas perceptible.

Pour toutes ces raisons, une valeur d'écart entre le temps logique et le temps de l'horloge est requise. Cette valeur correspond au plus petit écart de temps qui ne perturbe pas la communication avec le monde réel. Cette valeur sera désignée T_R. Pour la vision humaine, elle serait de 30 ms.

L'exemple qui suit illustre le mécanisme d'avancement du temps dans APIA.

La figure 4.73 illustre une série d'événements. Le temps logique y prend la valeur du temps de l'événement en cours, soit E_1.

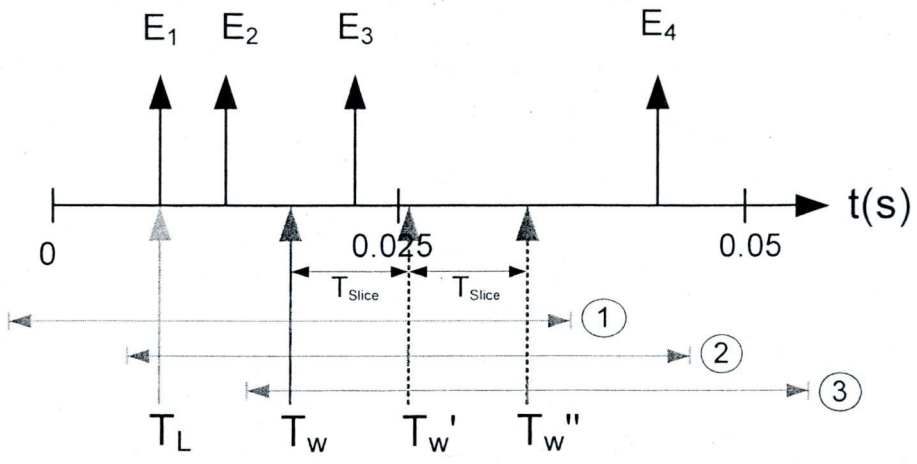

Figure 4.73 : Exemple de série d'événements avec le temps logique et le temps de l'horloge lorsque E_1 vient tout juste d'être traité.

Ce temps logique est valide car il est inclus dans l'intervalle $T_w - ½T_R < T_E < T_w + ½T_R$. Cet intervalle assure qu'un usager, par exemple, ne percevra pas deux événements distants de plus de T_R, dans l'hypothèse où les deux événements devraient être traités simultanément. Cet intervalle est identifié par l'axe 1 dans la figure 4.73. Lorsque cet événement est traité, l'*ordonnanceur* doit décider s'il traite le prochain événement immédiatement ou s'il attend en faisant une pause. Tant que $T_L < T_w$, l'*ordonnanceur* traite les événements. Toutefois, jusqu'à quels événements dans le futur doit-il traiter ? Combien de temps doit-il attendre s'il ne peut traiter le prochain événement ?

Dans un système d'exploitation, un *ordonnanceur* ne peut se synchroniser que sur un multiple du temps base. Ce temps correspond au *time slice* qui provient de l'exécution *multithread*. Il est présent tant sur les systèmes d'exploitation grand public que sur ceux dits temps réel. Même avec l'utilisation des événements du système d'exploitation, le temps minimal entre deux événements correspond au *time slice*. Cet intervalle que peut attendre l'*ordonnanceur* avant de reprendre son exécution sera désigné T_{slice}.

Dans la figure 4.73, une pause de T_{slice} mènerait le temps de l'horloge à T_w' (axe 2). Deux pauses le mèneraient deux fois plus loin (axe 3). Cependant, il ne pourrait se rendre jusqu'à T_w'' sans traiter E_2 car cet événement serait à un temps inférieure à $T_w - \frac{1}{2}T_R$. S'il n'était pas traité jusqu'à ce que l'horloge atteigne T_w'', le delai dans son traitement ne serait pas perçu dans le monde réel. Cependant, il le serait si le temps logique venait à avancer subitement parce qu'il viendrait de traiter un événement à $T_w - \frac{1}{2}T_R$. Dans cet exemple, on ne pourrait avancer le temps logique car il existe des événements à traiter entre ces deux temps. Par conséquent, le temps logique avance au temps de E_2 et ensuite à celui de E_3. La figure 4.74 montre le temps logique et d'horloge lorsque l'événement vient tout juste d'être traité.

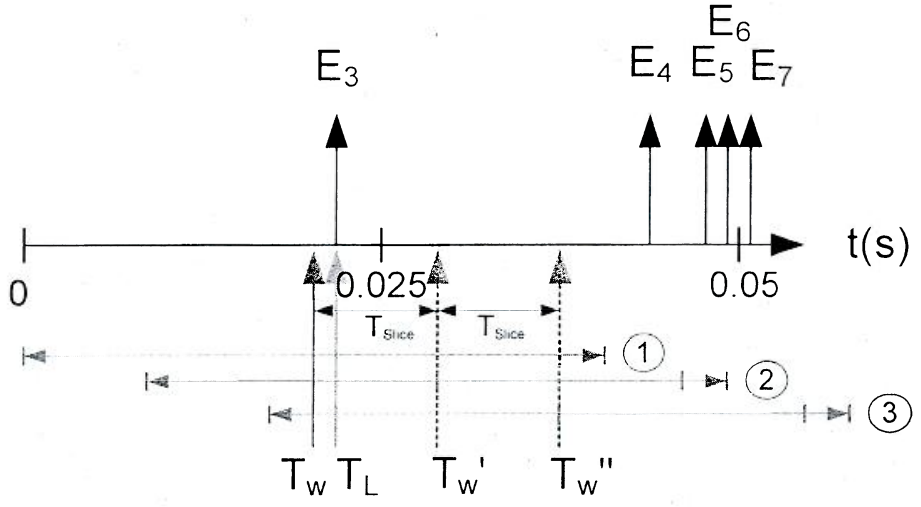

Figure 4.74 : Exemple de série d'événements avec le temps logique et le temps de l'horloge lorsque E_3 vient tout juste d'être traité.

Le temps de l'horloge a progressé pendant la période de temps de traitement de ces deux événements. Il faut encore décider si le temps logique doit être avancé au temps de l'événement suivant afin de le traiter ou s'il faut attendre pour avancer le temps logique.

Dans cet exemple, l'événement E_4 ne peut être traité immédiatement car son traitement serait perçu, dans le monde réel, comme désynchronisé avec le temps réel. En effet, $T_{E4} > T_w + \frac{1}{2}T_R$. Le simulateur fera donc une pause qui le mènera à T_w' (axe 2). Lorsque le temps logique sera rendu à T_w', il pourra traiter l'événement E_4 car $T_{E4} < T_w + \frac{1}{2}T_R$. Dans cet exemple, le simulateur pourrait attendre jusqu'à T_w'' (axe 3). Toutefois, parce qu'il peut survenir des séquences d'événements rapprochés à traiter dans un court intervalle de temps, par exemple E_5, E_6 et E_7, il est préférable de toujours traiter l'événement le plus éloigné pour bénéficier d'un temps tampon (égal à T_R) afin de compenser pour les périodes où il faudra traiter de nombreux événements rapprochés.

L'*ordonnanceur* doit donc traiter tout événement dont la valeur du temps est inférieure à T_W et, dans les autres cas, décider de traiter l'événement ou d'attendre un multiple de temps T_{slice}. L'équation suivante donne le nombre de périodes (N) d'attente en fonction du temps du prochain événement, du temps de l'horloge, du temps de repos minimum (T_{slice}) et du temps logique (T_L).

$$N = \left\lceil \frac{T_E - T_R/2 - T_w + T_{slice}}{T_{slice}} \right\rceil \qquad \text{(eq. 1)}$$

L'équation 1 sera évaluée à chaque événement. Si N = 0, alors l'*ordonnanceur* doit traiter l'événement suivant.

Problèmes d'ordonnancement

Modifier une instance de propriété pour l'instant présent pose de nombreux problèmes. Premièrement, si toutes les instances d'interactions modifient les instances de propriétés pour l'instant présent, il se créera une boucle qui empêchera le temps logique d'avancer. Deuxièmement, un temps d'écriture égal au temps de lecture d'une propriété ne correspond à aucune observation physique. Il s'agirait du principe d'instantanéité. La figure 4.75 illustre un premier exemple de boucle sans fin. Si, dans cet exemple, toutes les interactions sont instantanées, il se crée une boucle sans fin qui empêchera le MV de progresser dans le temps. L'exemple suivant présente un tel cas.

La figure 4.75 illustre, à partir du briquet des exemples précédents, l'ensemble des instances d'interactions qui utilisent, en lecture ou en écriture, les propriétés de l'acteur *Briquet*.

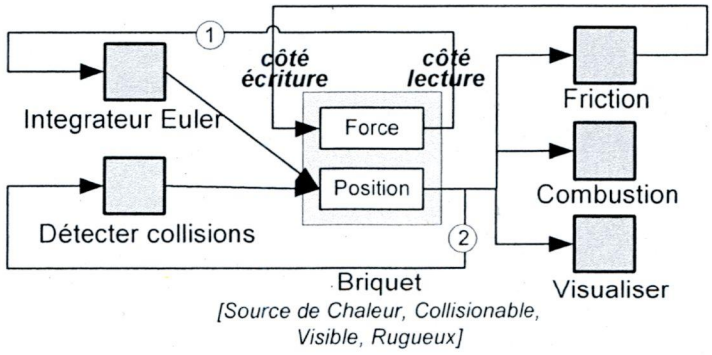

Figure 4.75 : Exemple d'un acteur "prenant vie" grâce à l'ensemble des interactions qui agissent sur lui à condition que cet ensemble soit coordonné temporellement.

Si toutes ces instances d'interactions étaient instantanées, le briquet ne subirait aucune évolution temporelle. Par exemple, l'interaction *Détecter collisions* détecte une collision et modifie la position du briquet. Cette modification cause l'appel de quatre instances d'interactions. Si une de ces interactions est instantanée, l'évènement ainsi généré cause l'appel de *Integrateur Euler,* identifié par le numéro 1 dans la figure 4.75, ou de *Détecter collisions*, identifié par la numéro 2 dans la figure 4.75. Toutefois, dans un même temps, la détection de collisions peut s'appliquer à nouveau et causer l'appel des interactions du groupe 2 en même temps que l'intégrateur ce qui résulte en une boucle sans fin.

4.4 Implantation

L'architecture **APIA** ne peut se concrétiser et montrer son plein potentiel qu'avec une implantation qui supporte ses concepts. Par exemple, si APIA supporte conceptuellement l'ajout de l'interaction séparément des autres éléments, l'implantation doit permette cet ajout via des mécanismes de scripts ou de librairies dynamiques. Un *framework* a été identifié comme la meilleure approche logicielle pour l'implantation d'**APIA,** présentée à l'annexe A, car il permet d'accroître la réutilisabilité et la modularité des applications.

4.5 Conclusion sur l'architecture

En résumé, ce chapitre a présenté **APIA**, une architecture pour concevoir des MVs plus *agiles*. **APIA** repose sur un méta-modèle conceptuel, un processus et des gestionnaires. Le méta-modèle conceptuel comprend de nombreux éléments pour une fine granularité, est centré sur l'interaction et permet des graphes relationnels. De plus, il oblige la définition de règles d'interaction, ce qui permet de gérer les extensions, les compositions et les modifications en cours d'exécution. Le processus contient les actions possibles pouvant être posées par des scénaristes, des usagers ou des interactions. Tel qu'il a été mentionné précédemment, le fait d'effectuer des actions représente en soit une *agilité* mais influence aussi le niveau d'*agilité* future. Le deuxième et surtout le troisième groupe d'actions (ancrer/détacher une instance d'interaction à/d'un acteur, lier/délier deux acteurs et assigner/retirer un personnage à/d' un acteur) représentent un avancement par rapport aux autres MVs qui sont peu *agiles*. Finalement, deux gestionnaires s'assurent de la cohérence du MV avec son modèle conceptuel à chaque action posée. L'automatisation de la gestion du MV avec l'instanciation des interactions selon des règles d'instanciation prédéfinies par le scénariste constitue la clef qui rend possible l'approche centrée sur l'interaction et accroît l'*agilité*.

L'architecture **APIA** laisse également certains problèmes non résolus. Par exemple, le scénariste doit penser aux conséquences de ses choix, donc ses actions, sur l'*agilité* future du MV résultant lorsqu'il est question de créer des propriétés, des personnages et des relations. Toutefois, pour créer des interactions ou effectuer d'autres actions, les conséquences des choix du scénariste seront moindres. Finalement, pour certaines actions, les conséquences seront presque nulles. Il est donc facile d'effectuer ces actions sans que leurs conséquences diminuent l'*agilité* future ou créent des MVs incohérents. Un autre problème possible touche les propriétés qui peuvent parfois être dépendantes les unes des autres. **APIA** ne gère actuellement pas ces dépendances. Toutefois, il s'agit d'un problème qui n'a pas été rencontré dans les applications développées avec **APIA** jusqu'à maintenant.

Les règles de multiplicité et relationnelles permettent de définir des règles d'interaction entre les personnages. La vérification de ces règles requiert une part non négligeable du temps de calcul lors de l'exécution du MV. Il faut donc limiter la complexité des règles d'interaction. Les deux dernières sections du chapitre 5 discutent de ces problèmes.

Chapitre 5

Démonstration d'agilité avec APIA

Ce chapitre présente des exemples de MVs basés sur **APIA** dont un de fine granularité représentant un câble en interaction avec son environnement et un autre pour la cryochirurgie. Ils permettent de démontrer l'*agilité* apportée par cette architecture. De plus, la diversité des exemples de ce chapitre démontre qu'**APIA** est une architecture générique pouvant être employée pour plusieurs types d'applications.

5.1 Exemple d'interactions génériques

La figure 5.1 illustre un premier exemple qui consiste en un MV dans lequel un briquet peut mettre le feu à un arbre et où une hache peut couper un arbre.

Figure 5.1 : Exemple d'un MV permettant de couper un arbre et d'y mettre le feu.

Les acteurs de ce MV jouent plusieurs personnages. Premièrement, l'arbre est un combustible et peut se sectionner. De plus, lorsque qu'il prend feu, il devient un générateur de chaleur. Le briquet est un générateur de chaleur et la hache un générateur de force. Le modèle conceptuel servant à représenter cette scène doit être créé en plusieurs étapes.

La conception de ce MV avec **APIA** débute avec la définition des propriétés et des personnages, dont ceux requis pour mettre le feu tel qu'illustré à la figure 5.2.

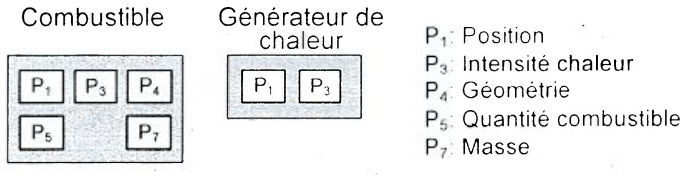

Figure 5.2 : Personnages impliqués dans une interaction de combustion.

La figure 5.3 montre les personnages nécessaires pour sectionner un acteur.

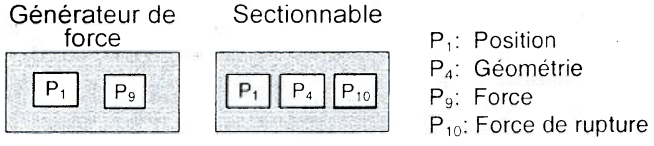

Figure 5.3 : Personnages requis pour sectionner un acteur.

La prochaine action consiste à créer des interactions pour ce MV : une pour mettre le feu et une autre pour sectionner. La figure 5.4 montre l'interaction qui permet au briquet de mettre le feu à l'arbre. Celle-ci se charge également d'entretenir la combustion jusqu'à ce que le combustible soit épuisé. Puisqu'aucune règle relationnelle n'est requise pour cet exemple, toutes les combinaisons d'acteurs jouant les personnages *Combustible* et *Générateur de chaleur* résulteront en une seule instance de cette interaction.

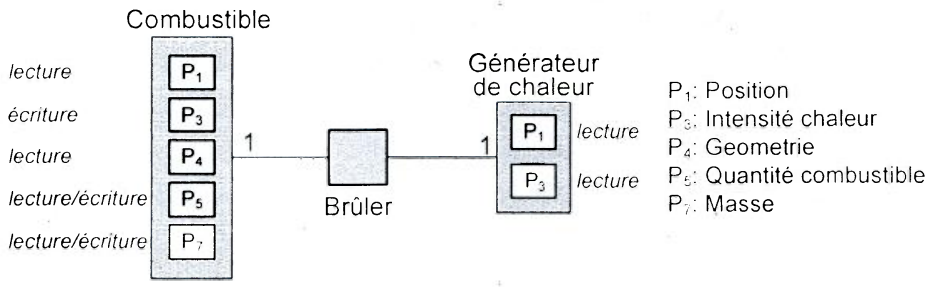

Figure 5.4 : Définition de l'interaction *Brûler* qui agit entre un personnage *Générateur de chaleur* et un personnage *Combustible*.

Le même résultat serait possible avec deux interactions. Une première aurait mis le feu et une autre aurait complétée la combustion. Si un personnage *En Feu* avait été assigné à l'acteur *Arbre* par la première interaction, le *gestionnaire de la cohérence présente* aurait instancié la seconde.

La figure 5.5 montre l'interaction qui sert à sectionner l'arbre. Cette interaction agit entre deux personnages *Sectionnable* et *Générateur de force*. Les propriétés incluses dans les personnages supportent une modélisation minimale pour les besoins de l'exemple.

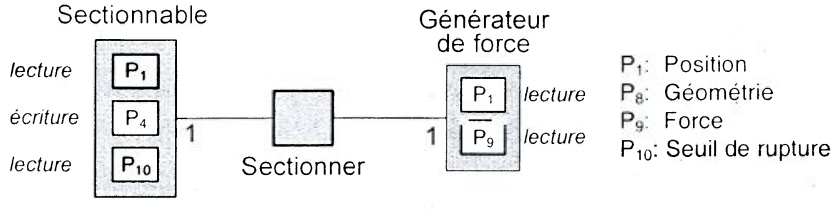

Figure 5.5 : Définition de l'interaction *Sectionner* qui agit entre un personnage *Sectionnable* et un personnage *Générateur de Force*.

Le MV prend ensuite forme avec la définition et l'instanciation des acteurs, l'instanciation des propriétés et leur assignation aux acteurs. Par la suite, les interactions sont ancrées et les personnages sont assignés aux acteurs. Les acteurs *Arbres* incarnent à la fois les personnages *Combustible* et *Sectionnable* alors que *Briquet* et *Hache* incarnent les personnages *Générateur de chaleur* et *Générateur de force* respectivement. La forêt inclut topologiquement quatre acteurs. Le résultat final est illustré par le diagramme APRI de la figure 5.6.

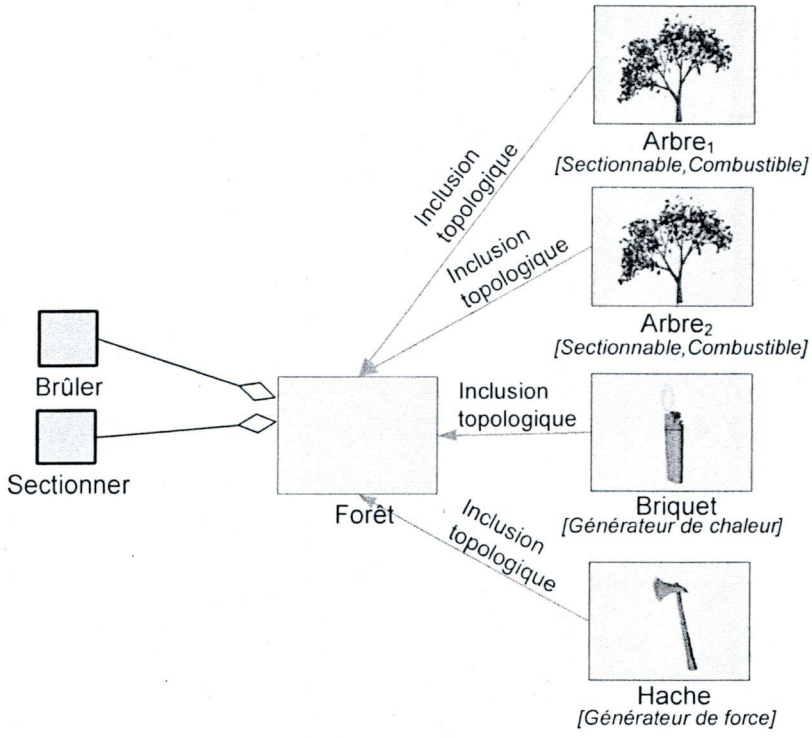

Figure 5.6 : Diagramme APRI pour le MV illustré à la figure 5.1.

Le *gestionnaire de la cohérence présente* intercepte les actions effectuées et vérifie si les règles d'interaction sont respectées. La figure 5.7 montre le résultat de l'instanciation des interactions. Les quatre acteurs interagissent à l'aide de quatre instances d'interactions.

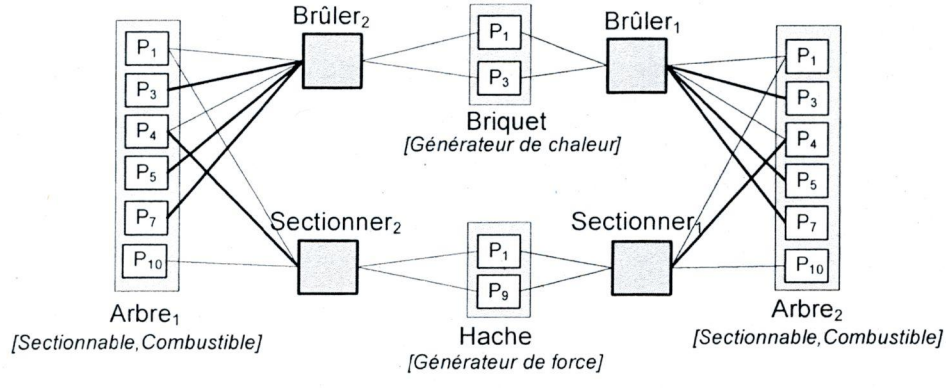

Figure 5.7 : Diagramme d'instance des éléments (sans les relations) montrant les multiples instances d'interactions et les accès aux propriétés.

L'interaction entre les acteurs est facilitée par l'utilisation d'une combinaison de personnages et d'interactions. Par exemple, un scénariste réutilise ce MV pour en concevoir un autre encore plus fantaisiste. Il peut substituer la hache et le briquet par une main qui aurait la capacité de couper les arbres ou d'y mettre le feu. La généralisation du comportement permet donc à un acteur d'incarner plusieurs personnages et à un personnage d'être incarné par plusieurs acteurs. Cette liberté d'action permet au scénariste d'assigner les personnages aux acteurs qu'il désire. Ce MV est donc *agile* car il permet l'extensibilité avec peu de modification des éléments existants.

5.2 Exemple de réutilisation et de liberté d'action avec un feu de camp

L'exemple du feu de camp de la figure 2.5 illustre un MV qui requiert une certaine capacité de réutilisation et de liberté d'action. Par exemple, un usager coupe une branche de l'arbre. Cette branche garde toutes les propriétés de l'arbre sauf la géométrie qui est modifiée. L'interaction de combustion de la section précédente est modifiée afin qu'elle ajoute, à un acteur qui prend feu, le personnage *Générateur de chaleur*. Il peut alors devenir à son tour un générateur de chaleur pour d'autres acteurs. Avec **APIA**, cette modification peut survenir en cours d'exécution. La figure 5.8 montre le deux instances d'interaction et deux acteurs pour un MV de feu de camp. Lorsqu'une branche prend feu sous l'action d'un briquet, elle devient elle-même un générateur de chaleur et peut mettre le feu à d'autres acteurs combustibles. Les branches brûleront si elles sont placées les unes sur les autres. Il sera également possible d'alimenter le feu en y ajoutant d'autres branches.

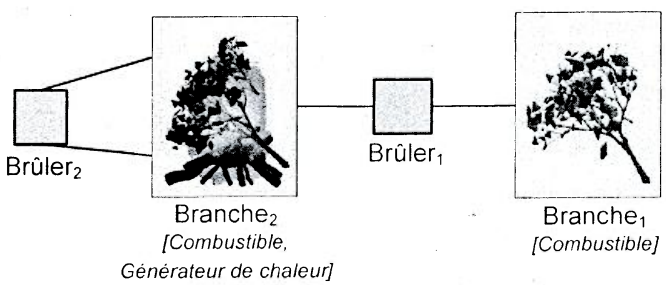

Figure 5.8 : Exemple d'effet en chaîne menant à un feu de camp et causé par un générateur de chaleur et un combustible.

La composition permet donc une plus grande liberté d'action et les règles d'interaction maintiennent le MV cohérents. Il est donc possible d'étendre un MV en cours d'exécution plus facilement parce que, si les bonnes propriétés existent, il ne suffit que d'ajouter des personnages et d'ancrer des interactions.

5.3 Exemple d'extensibilité avec l'ajout de la gravitation

L'exemple suivant démontre la capacité d'extension d'**APIA**. Les acteurs du MV précédent ne subissent pas la gravitation. Une extension en cours d'exécution corrigera cette situation. La figure 5.9 illustre l'interaction *Gravitation* avec les personnages desquels elle dépend.

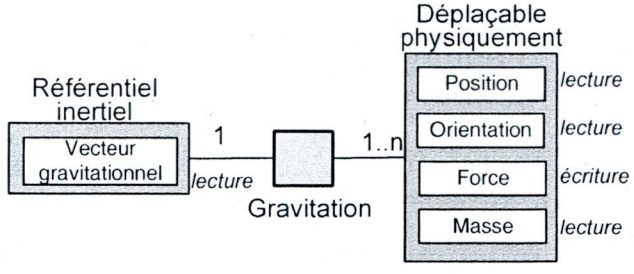

Figure 5.9 : Définition de l'interaction *Gravitation* faisant interagir deux personnages : *Référentiel inertiel* et *Déplaçable physiquement*.

Un acteur *Monde* est créé et ajouté à un MV initialement vide. Par la suite, le personnage *Référentiel inertiel* est assigné à l'acteur *Monde*. Finalement, l'interaction *Gravitation* y est ancrée. À chaque action posée, le *gestionnaire de la cohérence présente* vérifie les règles d'interaction. Pour la gravitation, il manque un personnage *Déplaçable physiquement*. La branche jouera ce personnage. L'assignation de ce personnage requiert les propriétés *Masse, Position et Force*. Cette dernière ne fait cependant pas partie de l'acteur *Branche*. Puisque cette propriété n'est utilisée qu'en écriture, le *gestionnaire de la cohérence présente* peut l'ajouter lors de l'assignation du personnage *Déplaçable physiquement* à l'acteur *Branche*. La figure 5.10 montre cette extension en genre et externe d'une entité.

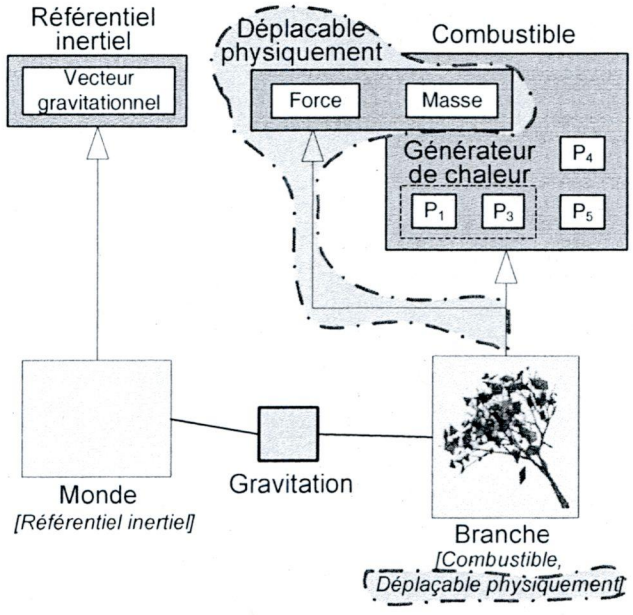

Figure 5.10 : Extension d'un MV dans le contexte **APIA** impliquant un nombre de modifications minimales pouvant être effectuées en cours d'exécution.

Cette extension est donc indépendante de la conception des acteurs de départ. L'arbre n'a pas été conçu afin qu'il soit coupé en branche. La branche n'a pas été conçue pour subir la gravitation. Lorsque les propriétés ne sont pas disponibles, elles peuvent être ajoutées ou calculées par les interactions. Ces extensions n'impliquent aucunement le scénariste original. Les autres architectures ou méta-modèles ne sont pas aussi *agiles*. En OO, une classe *Arbre* aurait été créée. Par la suite, la conception de la classe *Branche* aurait probablement impliqué la modification de la classe *Arbre* avec l'héritage ou l'agrégation. En effet, un arbre peut être défini comme composé de branches ou comme un type de branche. Ensuite, l'ajout de la gravitation aurait impliqué la modification des classes *Arbre* et *Branche*. Chaque ajout impliquant l'ensemble des classes, le tout aurait ensuite dû être recompilé. Il s'agit d'un processus plus complexe que celui requis avec **APIA**.

5.4 Exemple d'un MV agile basé sur un câble composé de cubes

Cet exemple montre l'évolution d'un MV par extension, composition et modification d'éléments et par réutilisation de comportements existants. Quoiqu'il existe de nombreuses façons de créer un tel MV, l'approche retenue a pour but de mettre en évidence les avantages d'**APIA**. Plusieurs éléments sont repris des exemples précédents.

Initialement, ce MV n'est constitué que d'un plancher, de murs et de quelques cubes en suspension dans l'air, tel qu'illustré à la figure 5.11 a). La gravitation sera ensuite définie et ajoutée, ce qui provoquera la chute des cubes dont le résultat est illustré en b). Un usager pourra intervenir dans ce MV avec une extension qui permettra de contrôler un cube afin d'en déplacer d'autres, tel qu'illustré en c). Ensuite, certains ajouts permettront la création d'un câble, tel qu'illustré en d), à partir des cubes existants afin de démontrer les possibilités de composition. Le câble sera ensuite plongé dans l'eau nouvellement ajoutée, tel qu'illustré en e), et de ses comportements, dont les effets sont visibles en f). Ensuite, le retrait de la liaison entre deux cubes consécutifs du câble résultera en la création de deux câbles distincts, tel qu'illustré en g). D'autres modifications d'intérêt, illustrées en h) auront également lieu. Les prochaines sections présentent chacune des étapes de l'évolution de ce MV, les conséquences des actions requises à son évolution ainsi que le rôle d'**APIA** pour chacune des étapes.

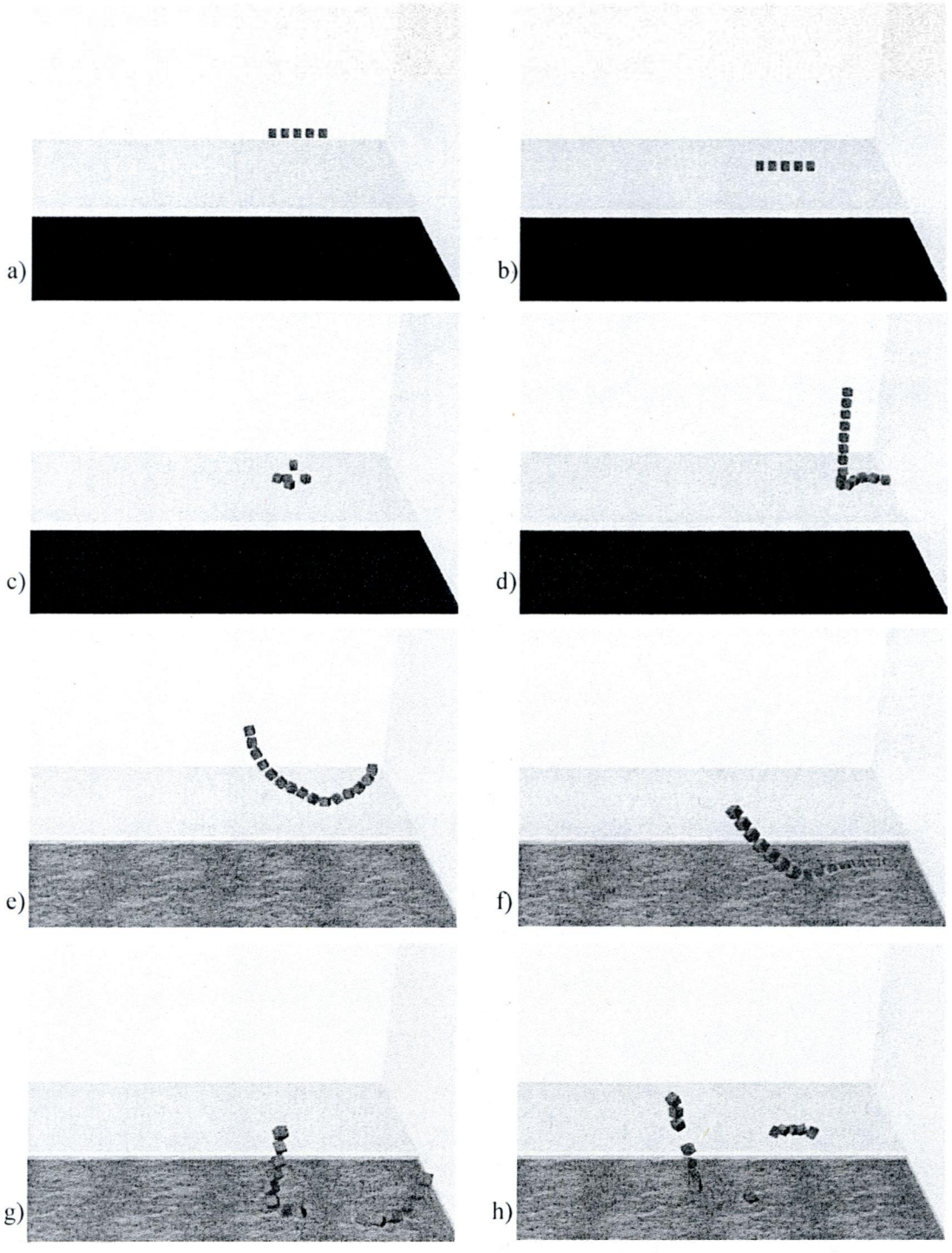

Figure 5.11 : Résultats de l'interaction de rendu après chaque série de modifications consécutives : a) MV initial b) ajout de la gravitation c) manipulation d'un cube d) création d'un câble e) ajout de l'eau f) de ses comportements g) sectionnement du câble en deux h) autres.

5.4.1 Définition des éléments de base

La première étape consiste à définir les éléments du MV initial. L'ensemble des propriétés utilisées dans cet exemple, illustré à la figure 5.12, est défini dès maintenant afin d'alléger le texte. Ces propriétés auraient pu être définies juste avant leur utilisation.

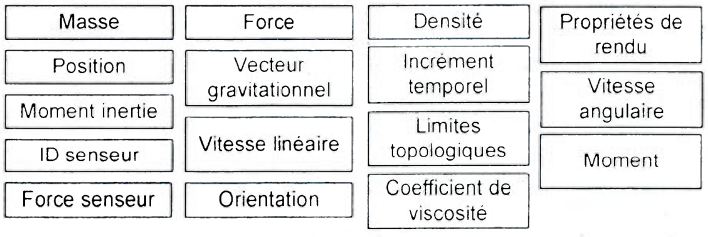

Figure 5.12 : Ensemble de propriétés pour le MV de l'exemple de câble.

Cet exemple repose sur l'utilisation des quatre relations illustrées à la figure 5.13. La relation *Inclusion topologique* représente l'inclusion d'un acteur dans un autre acteur tel un cube dans l'air ou dans l'eau. La relation *Observe* sert à relier un acteur visible à son observateur. La relation *Manipule* sert à relier un acteur manipulateur à l'acteur qu'il contrôle. Finalement, *Attaché à* relie deux acteurs en contact physique tels que deux maillons consécutifs d'un câble.

Figure 5.13 : Ensemble de relations pour le MV de l'exemple du câble.

Afin d'alléger le texte, la figure 5.14 illustre la définition simultanée des personnages *Déplaçable physiquement*, *Volumétrique*, *Déplaçable*, *Visible* et *Observateur* ainsi que des trois interactions qui se retrouveront dans le MV initial. Les multiplicités et les modes d'accès aux propriétés sont également définis. Ces interactions font interagir de un à trois personnages sous différentes multiplicités. *Vérifier inclusion topologique* sera expliqué dans la section 5.4.6.

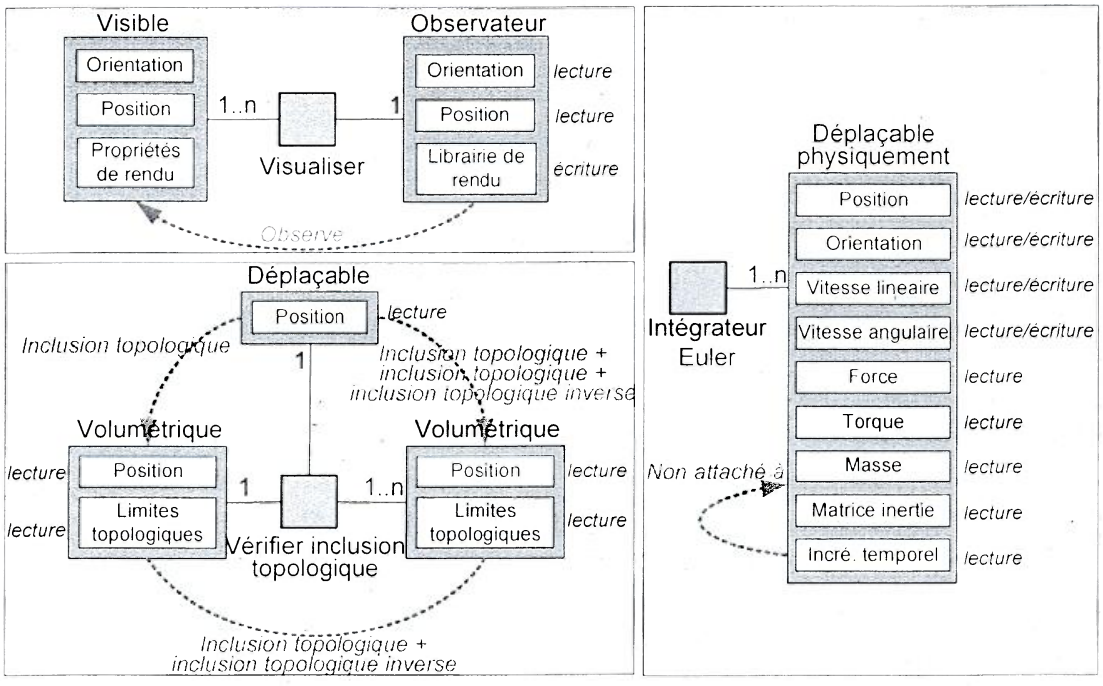

Figure 5.14 : Définition des interactions *Visualiser*, *Vérifier inclusion topologique* et *Intégrateur Euler* et des personnages du MV initial.

En résumé, **APIA** permet la définition d'éléments à fine granularité ce qui favorisera leur réutilisation. Leur définition peut avoir lieu en cours d'exécution. Ces actions de définition sont directement exécutées et ne passent pas par le *gestionnaire de la cohérence présente*. Quoique le MV soit maintenant plus riche de ces définitions, rien ne se déroule parce qu'il n'y a aucun acteur ni interaction d'ancrée.

5.4.2 Création d'un MV initial

Le MV initial, illustré à la figure 5.11 a), est construit à partir de la création d'acteurs, de l'assignation de personnages, de l'instanciation de relations entre ces acteurs et de l'ancrage d'interactions. Afin de simplifier la présentation de l'exemple, l'ensemble des propriétés requises est instancié et assigné dès le début. Les propriétés auraient néanmoins pu être ajoutées en cours d'exécution. La figure 5.15 illustre la résultante avec le diagramme APRI pour le MV initial.

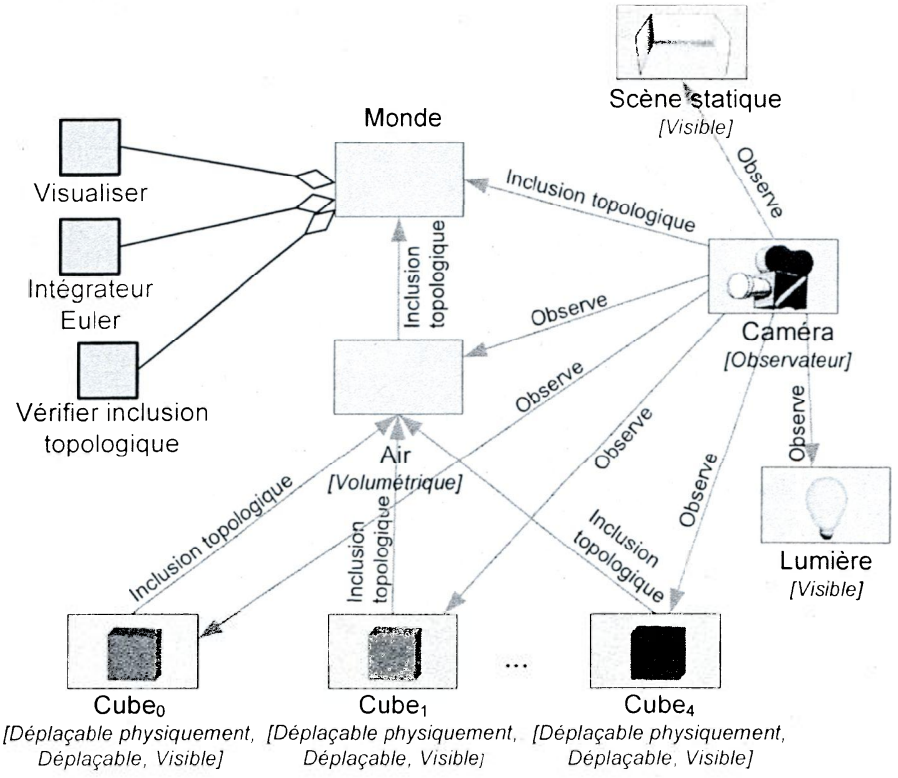

Figure 5.15 : Diagramme APRI du MV initial comprenant une scène statique, quatre cubes et trois interactions.

Lors de l'exécution, le *gestionnaire de la cohérence présente* traite une série d'actions qui résulte en un MV correspondant à ce diagramme. Lorsque les règles d'interaction sont rencontrées à la suite d'une action, le gestionnaire instancie cette interaction avec le bon ensemble d'instances de personnages. Par exemple, l'interaction *Visualiser* requiert qu'un acteur qui joue un personnage *Visible* soit lié à un acteur qui joue un personnage *Observateur*. Le respect des règles, dans ce cas-ci, amène la création de l'instance d'interaction $Visualiser_1$ tel qu'illustré à la figure 5.16 avec l'ensemble des instances de personnages *Cube* et *Caméra*. Si une autre instance de personnage *Visible* était instanciée et observée par la caméra, elle serait ajoutée à $Visualiser_1$. Également, la modification de la propriété *Position* d'un acteur par l'instance d'interaction $Intégrateur\ Euler_1$ causerait l'appel de l'instance d'interaction $Visualiser_1$.

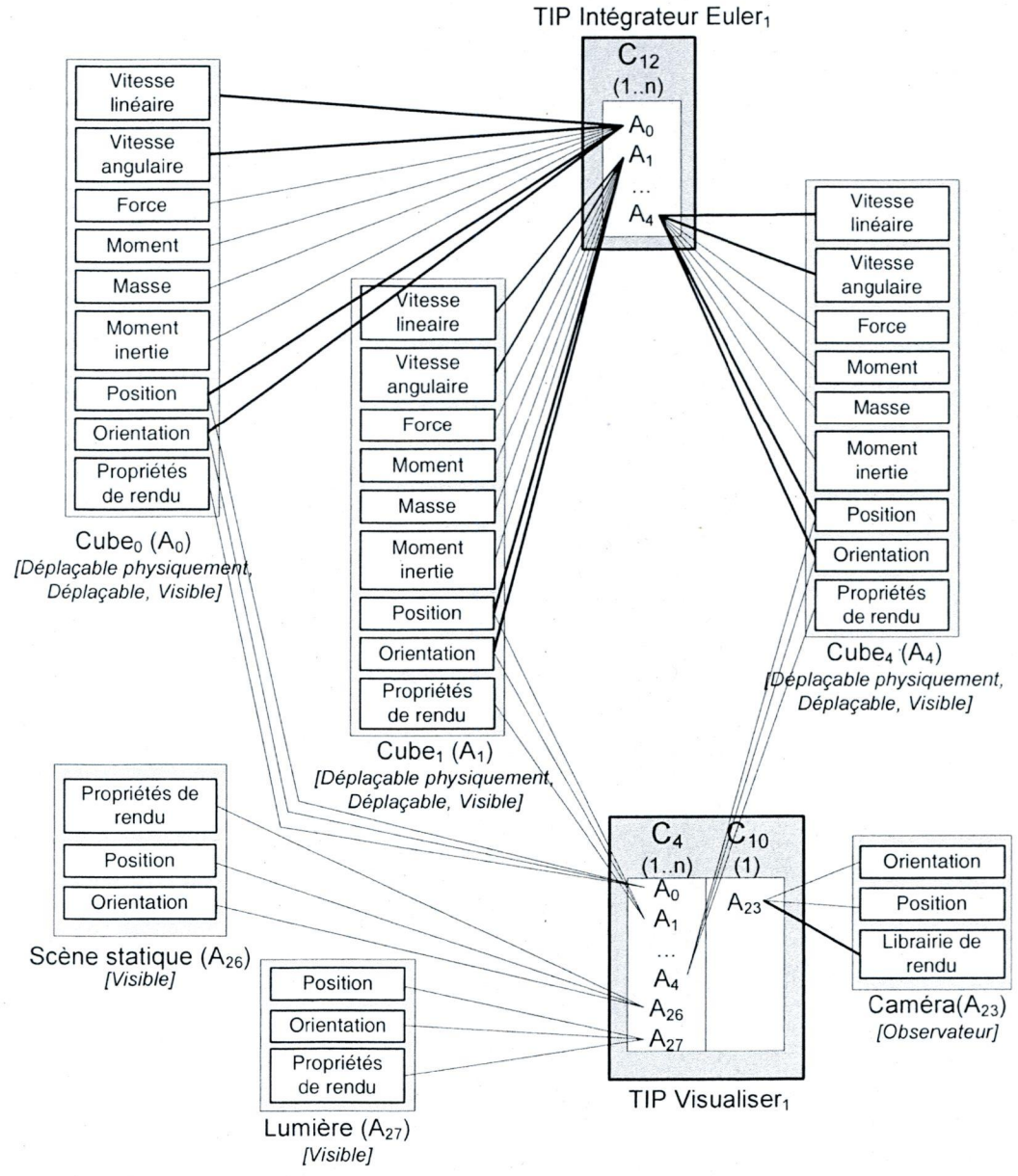

Figure 5.16 : Schéma d'instances d'interactions et d'acteurs montrant l'accès aux propriétés.

Cet appel est effectué en ajoutant cette instance d'interaction à la liste ordonnée des événements dépendant du temps. La séquence d'appel des instances d'interactions suit l'ordre établi par l'ordre de modification des instances de propriétés. La figure 5.17 illustre deux évolutions successives de cette liste ordonnée temporellement.

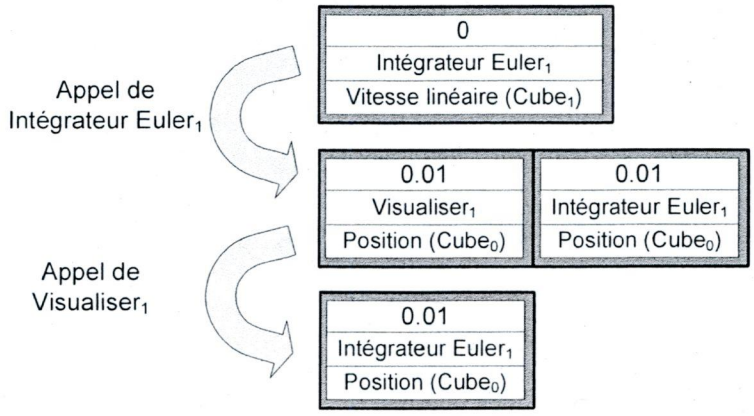

Figure 5.17 : Évolution de la liste des événements dépendants du temps lors de l'exécution.

L'appel des instances d'interactions est géré selon leur dépendance envers les instances de propriétés. La modification d'une seule instance de propriété peut donc causer l'appel de plusieurs instances d'interactions. À l'opposé, la modification de plusieurs instances de propriétés d'un même acteur par plusieurs instances d'interactions peut causer l'appel d'une seule instance d'interaction. Ce mécanisme est employé lorsqu'une instance d'interaction est ajoutée en cours d'exécution, permettant à l'*ordonnanceur* **APIA** de gérer les extensions du MV. Ses capacités sont cependant limitées. Par exemple, l'ordonnanceur ne peut détecter des boucles d'événements qui bloqueraient le MV à un temps donné. La représentation des dépendances entre propriétés et instances d'interactions étant déjà difficile à ce stade, la suite de cet exemple ne présentera pas de schémas d'accès aux propriétés

5.4.3 Ajout de la gravitation

Aucun mouvement n'est visible dans ce MV car aucune force n'est appliquée sur les acteurs. Les cubes restent donc immobiles dans l'air. La prochaine étape consiste à étendre ce MV avec la gravitation. La figure 5.18 illustre la définition du personnage *Référentiel inertiel* et de l'interaction *Gravitation*.

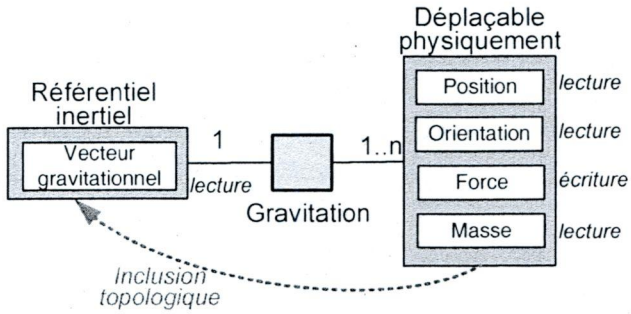

Figure 5.18 : Définition du personnage *Référentiel inertiel* et de l'interaction *Gravitation*.

La gravitation est ajoutée avec l'ancrage de l'interaction *Gravitation* et l'assignation du personnage *Référentiel inertiel* à l'acteur *Monde* tel qu'illustré à la figure 5.19.

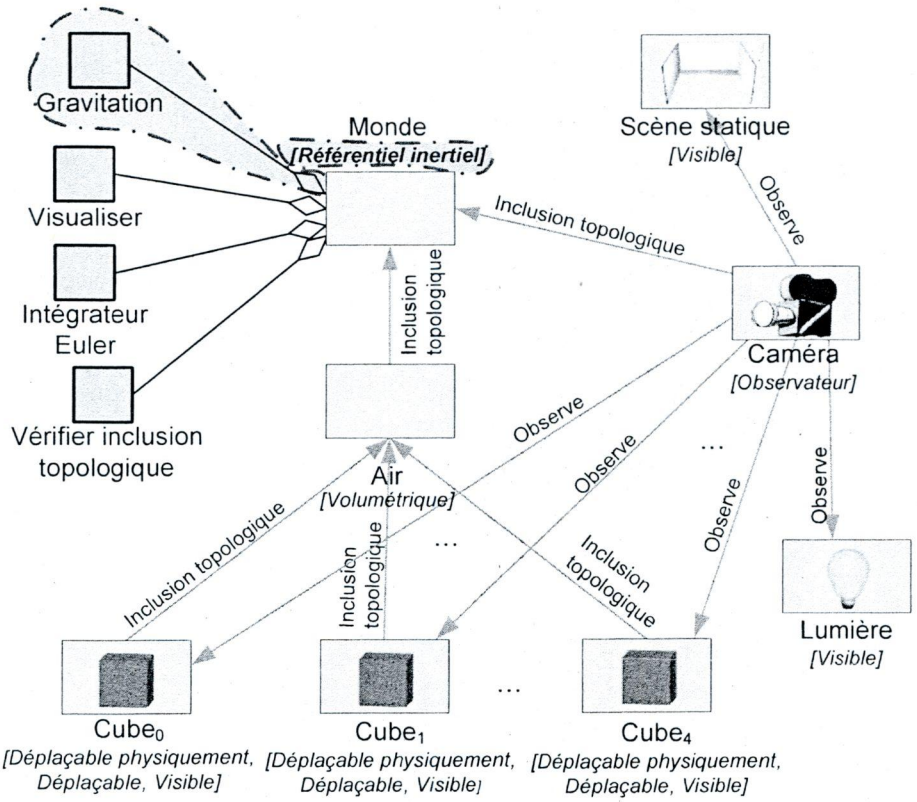

Figure 5.19 : Diagramme APRI de l'ajout de la gravitation avec l'assignation du personnage *Référentiel inertiel* à l'acteur *Monde* et l'ancrage de l'interaction *Gravitation* à l'acteur *Monde*.

Lorsque le personnage *Référentiel inertiel* est assigné à l'acteur *Monde*, le *gestionnaire de la cohérence présente* instancie l'interaction *Gravitation* et les cubes tombent au sol tel qu'illustré à la figure 5.11 b). Pour simplifier l'exemple, l'interaction *Intégrateur Euler* se charge de la détection de collisions. La figure 5.20 illustre le TIP de l'interaction *Gravitation* ainsi instanciée. Tout nouveau personnage *Déplaçable physiquement* inclus dans le référentiel inertiel serait ajouté à la colonne de gauche. L'instance d'interaction *Gravitation* ne sera appelée qu'une seule fois car la constante gravitationnelle ne sera pas modifiée dans cet exemple.

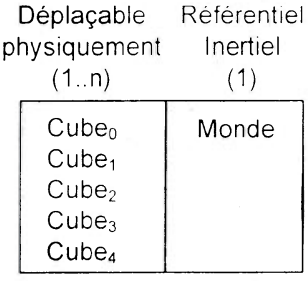

Figure 5.20 : TIP de l'instance d'interaction *Gravitation* faisant interagir n instances de personnages *Déplaçables physiquement* (colonne de gauche) avec une instance de personnage *Référentiel inertiel* (colonne de droite).

Ce MV a été étendu avec des ajouts minimes en cours d'exécution. La réutilisation de propriétés, de personnages et d'une relation existante a facilité sa mise en place. **APIA** permet donc de réutiliser un grand nombre d'éléments (voir section 2.3.2) lors d'extensions. Il en sera de même dans les autres exemples. L'ajout de la gravitation ne requiert donc pas une connaissance poussée du fonctionnement du MV existant et demande un effort modeste de la part du scénariste. En effet, l'interaction fournit de l'information de gestion que le *gestionnaire de la cohérence présente* se sert pour l'intégrer au MV existant.

5.4.4 Manipulation d'un cube

Les cubes se retrouvent rapidement au repos car la gravitation les fait tomber sur le sol. La prochaine extension ajoute l'interaction *Manipuler acteur* afin de permettre à un usager d'intervenir dans ce MV. La première étape, illustrée à la figure 5.21, consiste à définir les personnages *Manipulateur* et *Manipulable* ainsi que l'interaction *Manipuler acteur*.

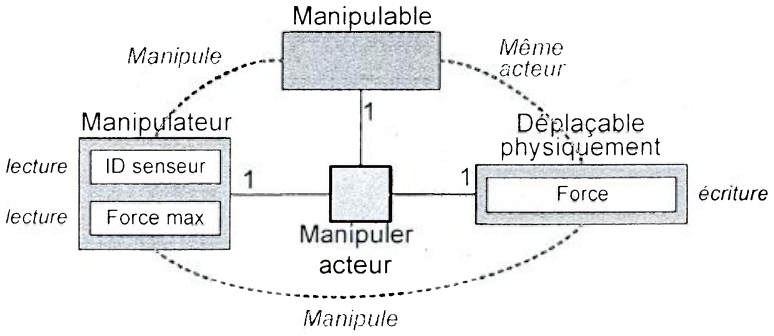

Figure 5.21 : Définition des personnages *Manipulateur* et *Manipulable* et de l'interaction *Manipuler acteur*.

Ensuite, la création de l'acteur *Manipulateur invisible 1*, l'ancrage de l'interaction *Manipuler acteur* et à l'ajout de la liaison *Attaché à* entre les acteurs $Cube_0$ *et Manipulateur invisible 1* permettent de manipuler $Cube_0$. Le diagramme APRI de la figure 5.22 illustre ces extensions.

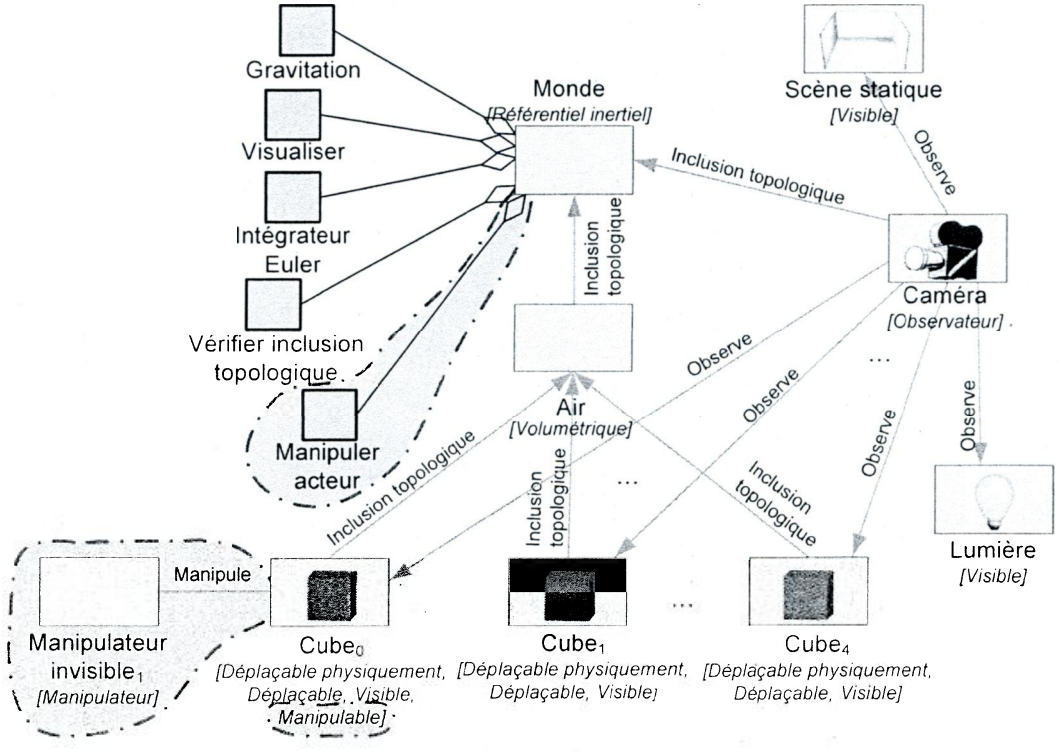

Figure 5.22 : Diagramme APRI de l'ajout de l'interaction *Manipuler acteur*, de l'acteur *Manipulateur invisible 1* et l'assignation du personnage *Manipulable* à l'acteur $Cube_0$ pour qu'un usager manipule les cubes.

L'interaction *Manipuler acteur* utilise un manche à balai pour contrôler un cube mais pourrait aussi bien faire intervenir d'autres périphériques tout en utilisant les mêmes propriétés et les mêmes personnages. La manipulation de l'acteur $Cube_0$ permet indirectement de déplacer d'autres cubes, tel qu'illustré à la figure 5.11 c), sans aucune autre modification. Cette possibilité repose sur un effet en chaîne des interactions qui n'est pas nécessairement prévu mais souhaitable. La composabilité d'un tel MV ne remet pas en question sa cohérence.

5.4.5 Création d'un câble par composition

Les quelques cubes déjà présents dans ce MV peuvent être assemblés en un câble. Pour qu'ils tiennent ensemble, il suffit d'y ajouter un comportement de câble qui sait comment reconnaître et gérer un tel assemblage. Il s'agit d'une réutilisation d'un modèle spécifique existant (voir section 2.3), d'une composition (voir section 2.2) de cubes déjà présents ainsi que d'une extension en genre et externe (voir section 2.5) avec des interactions et des relations, le tout pendant l'exécution du MV. L'interaction *Solutionneur de câble*, illustrée à la figure 5.23, contient le comportement du câble qui réutilise un modèle de dynamique des corps rigides, *Open Dynamic Engine* (Smith, 2006). Il s'agit d'un exemple de réutilisation de modèle spécifique supporté par **APIA** grâce aux règles d'interaction. L'interaction *Solutionneur de câble* permet le fonctionnement du modèle spécifique avec les acteurs et les propriétés requis à son exécution. Cette interaction utilise le même personnage que *Intégrateur Euler* avec une règle relationnelle différente. Dans ce cas, la relation *Attaché à* doit lier les cubes les uns aux autres.

Des cubes supplémentaires sont ajoutés afin de créer un câble au comportement réaliste. Ce câble est conçu avec l'ajout d'acteurs *Cube*, la définition de l'interaction *Solutionneur de câble,* la liaison des acteurs *Cube* avec les acteurs *Air* et *Caméra,* la modification des propriétés de tous les acteurs *Cube* pour les aligner, la liaison des acteurs *Cubes* en chaîne avec la relation *Attaché à* et, finalement, l'ancrage de l'interaction *Solutionneur de câble*. La création de l'acteur *Manipulateur invisible 2*, la liaison *Attachée à* avec l'acteur $Cube_{13}$ et l'assignation du personnage *Manipulable* permettront de manipuler l'autre bout du câble. La figure 5.24 illustre ces modifications, extensions et compositions.

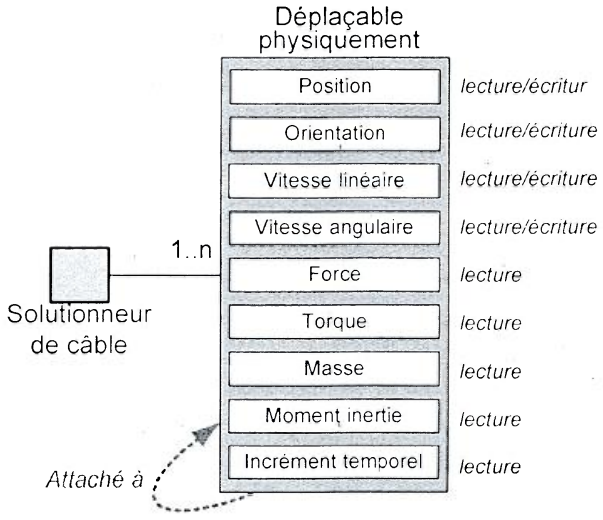

Figure 5.23 : Définition de l'interaction *Solutionneur de câble*.

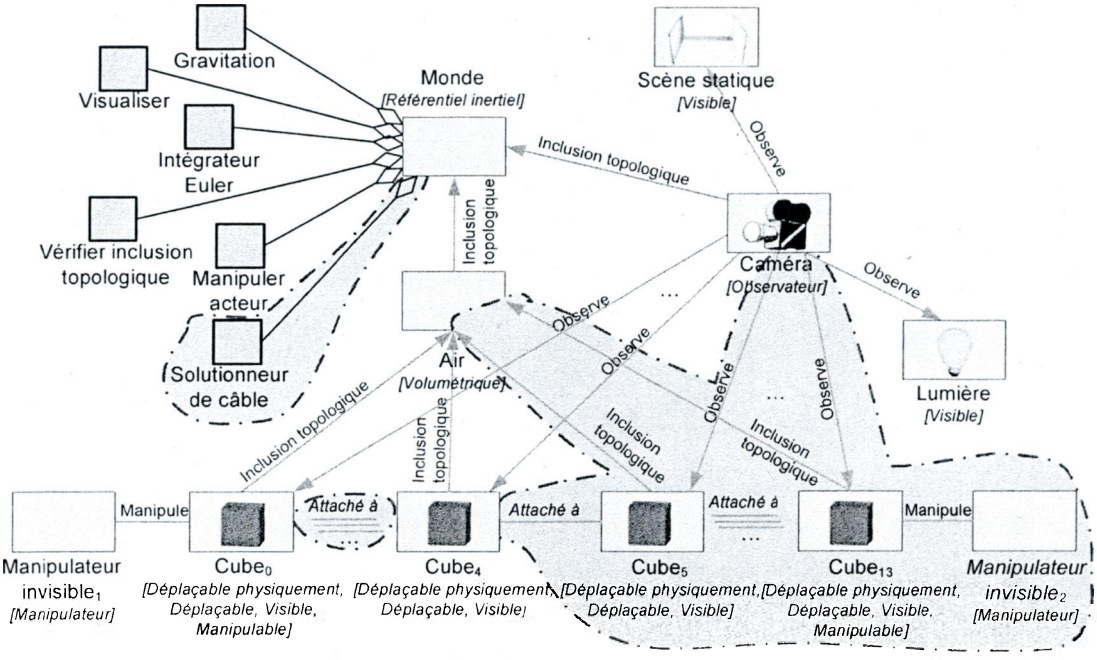

Figure 5.24 : Diagramme APRI de l'ajout des acteurs *Cube*, de relations et de l'interaction *Solutionneur de câble* afin de créer un câble.

Attachés les uns aux autres, les cubes deviennent soumis non plus à l'interaction *Intégrateur Euler* mais plutôt à *Solutionneur de câble*. Ils obéissent ainsi à un déplacement

caractéristique d'un câble, illustré à la figure 5.11 d). L'interaction *Intégrateur Euler* n'est en effet plus appropriée car elle gère les acteurs individuellement. **APIA**, à l'aide des règles d'interaction, effectue cette substitution. Puisque l'interaction *Solutionneur de câble* est ancrée avant l'ajout de relations, l'assignation des personnages et la liaison des cubes causent l'ajout d'instances de personnages à gérer par cette instance d'interaction. Les interactions, comme *Intégrateur Euler* et *Solutionneur de câble*, définissent cette capacité au moment de leur définition (voir section 4.1.5). Si la séquence inverse d'actions était posée, c'est-à-dire si l'interaction était ancrée après l'ajout des instances de relations, alors l'instance d'interaction aurait reçu, lors de son initialisation, l'ensemble des instances de personnages.

La création du câble démontre plusieurs avantages d'**APIA**. Premièrement, les cubes ne deviennent visibles qu'au moment de leur liaison par la relation *Observe* à l'acteur *Caméra* similairement à ce qui s'est passé lors de la création du MV initial (voir section 5.4.2). Les interactions continuent donc de se comporter adéquatement même après l'ajout de nombreux éléments. Il s'agit d'un avantage de l'approche par interaction qui groupe les éléments fortement couplés au sein d'un même élément. L'approche par interaction est moins sensible aux incohérences dues aux extensions et aux modifications futures que l'approche par entité. Ensuite, la modification de position des cubes est effectuée dans un mode scénariste (voir section 2.4.2) comparativement au mode usager employé avec l'interaction *Manipuler acteur*. **APIA** supporte donc ces deux modes. De plus, la manipulation de l'acteur $Cube_0$ ajouté précédemment continue non seulement de fonctionner mais permet aussi de manipuler le câble. Cette réutilisation intuitive d'un comportement générique simplifie donc la conception d'un câble en permettant sa manipulation indirectement. **APIA** supporte également une réutilisation similaire à celle illustrée à la figure 2.4 qui mise sur la réutilisation d'éléments déjà existants et ayant des comportements similaires. L'extension de cubes dans cet exemple consiste en une extension en nombre d'un même type d'entité alors que l'ajout de l'interaction *Solutionneur de câble* constitue une extension en genre, deux types d'extensions (voir section 2.5.1) supportés par **APIA**. Des extensions à partir d'éléments internes et externes (voir section 2.5.2) se retrouvent également dans cet exemple. L'ajout de la relation *Inclusion topologique* consiste en une extension à partir d'éléments internes alors que l'ajout du modèle de dynamique des corps rigides de *Open Dynamic Engine* consiste en une extension externe rendue possible par l'interfaçage à travers une interaction. Finalement, le câble pourrait être réutilisé en totalité, en

partie ou en pièces détachées telles les cubes. **APIA** permet donc une granularité variable dans la réutilisation (voir section 2.3.3). En effet, rien n'empêche de réutiliser simultanément un cube individuellement ou le câble en totalité. Un outil d'exportation faciliterait toutefois l'identification de l'ensemble des éléments requis à son exportation.

5.4.6 Ajout de l'eau

Un acteur *Eau,* possédant certaines propriétés illustrées à la figure 5.25, est ajouté au MV. Il est lié à l'acteur *Monde* avec une relation *Inclusion topologique*.

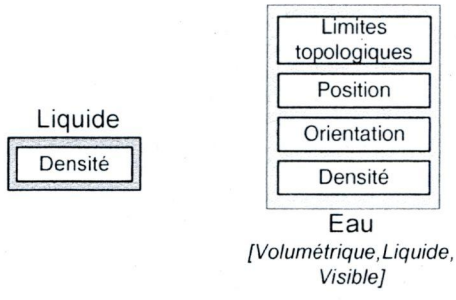

Figure 5.25 : Création du personnage *Liquide* et de l'acteur *Eau*.

Comme le montre la figure 5.11 e), l'eau est maintenant visible mais aucun cube ne peut interagir avec celle-ci car elle ne possède aucun comportement, situation qui sera corrigée dans la prochaine section. Par conséquent, dans une approche centrée sur l'interaction, la création d'un acteur qui joue peu de personnages n'étend pas significativement le MV.

L'ajout de l'eau permet d'expliquer le fonctionnement de l'interaction *Vérifier inclusion topologique* qui, comparativement aux autres interactions, change le modèle conceptuel en cours d'exécution. Pour ce faire, cette interaction ajoute et enlève des relations, une action du troisième groupe. Le rôle de l'interaction *Vérifier inclusion topologique* est de vérifier si une instance de personnage *Déplaçable* est encore incluse dans l'instance de personnage *Volumétrique*. Dans le cas contraire, l'interaction enlève la relation d'inclusion topologique et recherche dans quelle autre instance de personnage *Volumétrique* est incluse topologiquement l'instance de personnage *Déplaçable*. La figure 5.26 présente un MV pour cette explication.

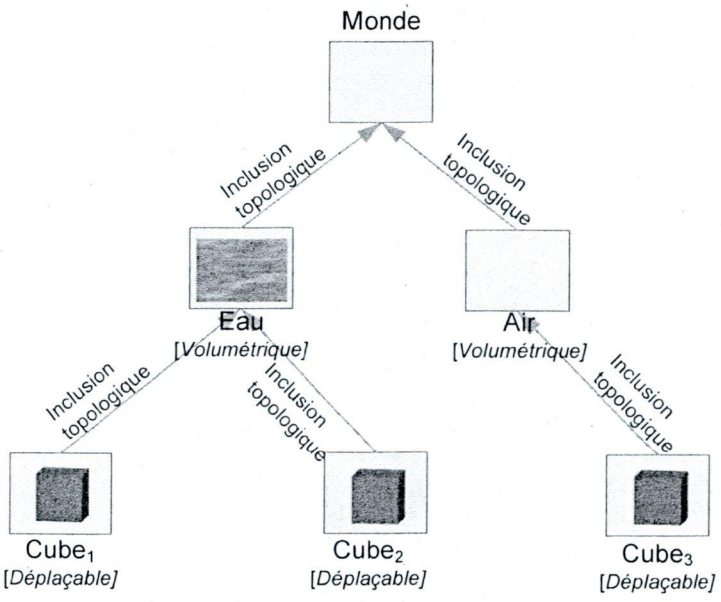

Figure 5.26 : Diagramme APR de cubes pouvant être inclus topologiquement dans l'eau ou dans l'air.

La figure 55.27 montre les TIPs correspondant au diagramme APR précédent. Chaque instance d'interaction *Vérifier inclusion topologique* est créée avec son TIP.

	Déplaçable (1)	Volumétrique (1)	Volumétrique (1..n)
TIP_1	$Cube_1$	Eau	Eau Air
TIP_2	$Cube_2$	Eau	Eau Air
TIP_3	$Cube_3$	Air	Eau Air

Figure 5.27 : TIPs pour l'interaction de vérification d'inclusion topologique du diagramme APR de la figure 5.26.

Les instances vérifient alors si les acteurs *Cube* sont inclus dans les acteurs *Eau* ou *Air*. Chaque acteur étant libre de se déplacer, lorsque qu'un acteur *Cube* change de milieu, l'instance d'interaction détecte ce changement par la valeur des positions et enlève l'instance de relation *Inclusion topologique* précédente, puis cherche, parmi les acteurs de la troisième

colonne, laquelle respecte la règle relationnelle *Inclusion topologique*. Dans cet exemple, une instance d'interaction détecte que le $Cube_3$ est passé de l'air à l'eau. La troisième instance d'interaction *Vérifier inclusion topologique* retire une instance de relation *Inclusion topologique* entre $Cube_3$ et *Air* pour en ajouter une entre $Cube_3$ et *Eau*. Ces deux actions génèrent deux événements indépendants du temps. Dès que l'événement en cours sera terminé, ils seront traités dans l'ordre d'arrivée. Si les actions étaient traitées dès que posées, les interactions ne pourraient pas terminer leur tâche. Dans ce cas, l'instance d'interaction *Vérification topologique* se serait terminée dès le retrait de la relation parce que les règles d'interaction n'auraient plus été respectées. Par conséquent, l'ajout de la relation n'aurait jamais été effectué.

Dans cet exemple, l'*ordonnanceur* consulte la liste des événements dès qu'il reprend le contrôle. Il y trouve la première action, c'est-à-dire le retrait d'une instance de relation. Le *gestionnaire de la cohérence présente* détruit alors l'instance d'interaction *Vérifier inclusion topologique* qui a justement posé cette action. Ensuite, l'action d'ajout d'une instance de relation cause l'instanciation de l'interaction *Vérifier inclusion topologique* avec *Eau* à la place de *Air* comme instance de personnage de multiplicité 1 (deuxième colonne). Le TIP correspondant est montré à la figure 5.28.

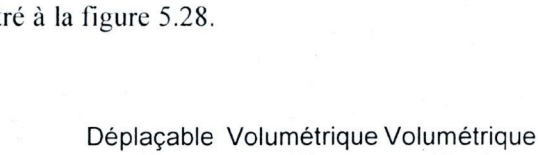

Figure 5.28 : TIP suite au changement de l'acteur $Cube_3$ de l'air à l'eau.

Ces changements sont transparents pour les instances de l'interaction. Ils sont pris en charge par les gestionnaires selon les règles définies lors de la conception de l'interaction. Cet exemple montre bien l'utilité des règles d'interaction. Les avantages sont nombreux.

- L'instance d'interaction obtient, en tout temps, une liste des instances de personnages qui interagissent. Le TIP et la LCR sont mis à jour automatiquement.
- L'instance d'interaction reçoit la liste des instances de personnages ajoutées ou enlevées. Le TIP et la LCR sont mis à jour automatiquement.

- Du temps de calcul est récupéré car le *gestionnaire de la cohérence présente* minimise, à travers les règles d'interaction, le nombre d'instances de personnages.

5.4.7 Ajout des comportements de l'eau

Ce MV peut être étendu afin d'ajouter des comportements de viscosité et de flottaison (Archimède) à l'eau. La définition des personnages et des interactions est illustrée aux figures 5.29 et 5.30. Ces interactions posent des règles d'interaction plus complexes que les précédentes. L'explication de l'interaction *Viscosité* permettra de démontrer l'*agilité* résultant d'**APIA**.

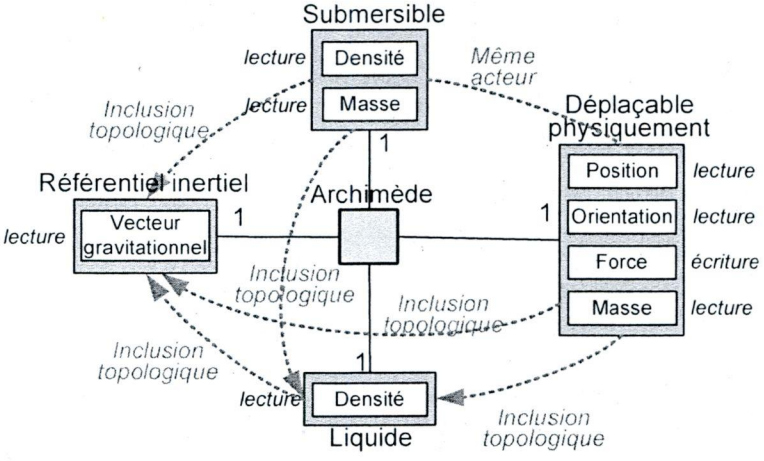

Figure 5.29 : Définition du personnage *Submersible* et de l'interaction *Archimède*.

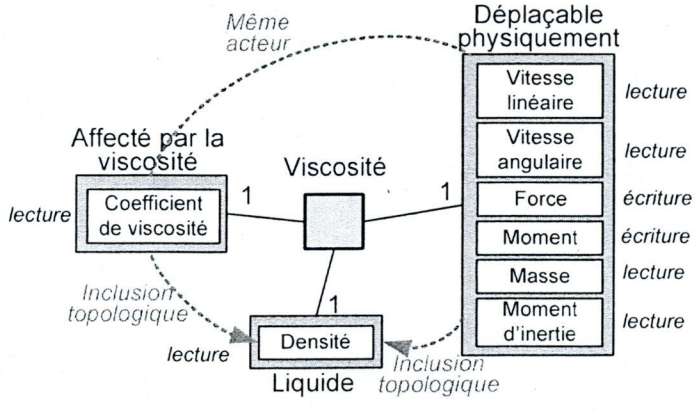

Figure 5.30 : Définition du personnage *Affecté par la viscosité* et de l'interaction *Viscosité*.

L'interaction *Viscosité* fait interagir trois types de personnages : *Affecté par la viscosité*, *Déplaçable* et *Liquide*. Ils sont tous de multiplicité 1. Les règles relationnelles inter-personnage imposent que le personnage *Affecté par la viscosité* et le personnage *Déplaçable* soient joués par le même acteur. De plus, les règles imposent que le personnage *Liquide* soit inclus topologiquement dans les deux autres personnages. La figure 5.31, qui illustre le diagramme APR de deux cubes placés dans l'eau, vient supporter l'explication de la raison de ces règles.

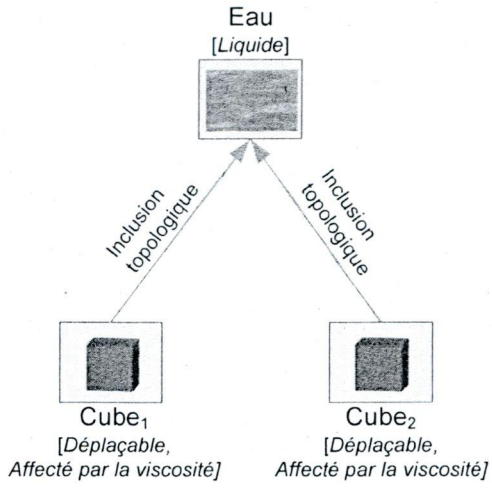

Figure 5.31 : Diagramme APR pour deux cubes plongés dans l'eau.

L'interprétation des règles fournies avec l'interaction résultera en la création de deux instances d'interactions. La figure 5.32 en montre les TIPs.

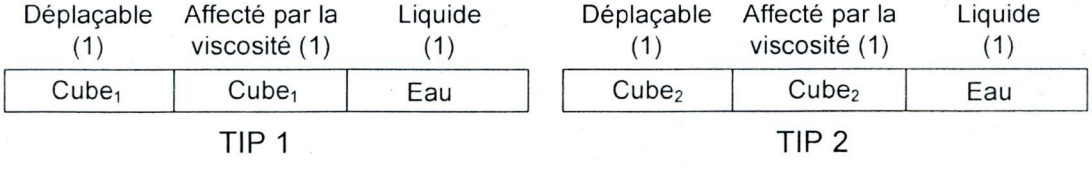

Figure 5.32 : TIPs pour chacune des instances de l'interaction *Viscosité*.

Le *gestionnaire de cohérence présente* créera donc deux TIPs, un premier avec l'ensemble d'instances de personnage $\{Cube_1, Cube_2, Eau\}$ et un second avec l'ensemble $\{Cube_2, Cube_2, Eau\}$. Sans la règle *Même acteur* entre les personnages *Déplaçable* et *Affecté par la viscosité*, le gestionnaire aurait crée des instances de l'interaction *Viscosité* avec les TIPs

contenant les ensembles $\{Cube_1, Cube_2, Eau\}$ et $\{Cube_2, Cube_1, Eau\}$. Ces ensembles n'ont aucun sens car, à partir de ces TIPs, les instances de l'interaction *Viscosité* auraient pris le coefficient de viscosité d'un acteur, $Cube_1$ par exemple, pour appliquer la force sur un autre acteur, $Cube_2$.

Les autres interactions ne seront pas documentées en détail mais reprennent les mêmes concepts. Il est à noter que certaines interactions comme *Archimède* et *Viscosité* sont définies avec des multiplicités de 1. Il existera donc une instance d'interaction par combinaison de personnages qui respecte les règles inter-personnage. Certaines interactions comme la *Gravitation* ont une multiplicité de 1..n. Le *gestionnaire de la cohérence présente* instanciera une seule interaction car il n'existe qu'un seul référentiel inertiel. D'autres choix de multiplicité sont également possibles. Les interactions *Viscosité* ou *Archimède* fonctionneraient également avec une multiplicité de 1..n, ce qui montre l'*agilité* qu'apporte **APIA**.

Lorsque l'interaction s'applique toujours sur les mêmes instances de personnages, la performance n'est pas le principal critère pour choisir la multiplicité. Par exemple, l'exécution de l'interaction *Gravitation* scénarisée avec une multiplicité de 1..n ne sera plus rapide qu'une version avec multiplicité 1 que si le nombre d'instances de personnage est grand et si le temps d'exécution de l'interaction est court. En effet, avec une multiplicité de 1..n, la boucle d'itération qui applique la force à chaque instance de personnage se situe à l'intérieur de l'instance d'interaction. À l'opposé, avec une multiplicité de 1, le *gestionnaire de la cohérence présente* appelle chacune des instances d'interaction avec chacune des instances de personnages. La boucle d'itération se situe alors dans le gestionnaire. Comme sa boucle d'exécution est déterministe et rapide (voir Annexe A), le choix entre une seule instance de l'interaction avec multiplicité 1..n et plusieurs instances de cette même interaction avec multiplicité 1 n'est pas motivé par des critères de performance d'exécution. Cependant, dans le cas où une interaction est instanciée et détruite constamment, le choix entre une multiplicité 1 et 1..n peut être motivé par des critères de performance. Il s'avère en effet plus rapide d'ajouter ou d'enlever une instance de personnage à une instance d'interaction que de créer ou de détruire une instance d'interaction.

L'extension avec les comportements de l'eau se déroule en cinq phases : la définition du personnage *Submersible* et de l'interaction *Archimède*, l'ancrage de l'interaction *Archimède*

à l'acteur *Monde*, la définition du personnage *Affecté par la viscosité* et de l'interaction *Viscosité*, l'ancrage de l'interaction *Viscosité* à l'acteur *Monde* et l'assignation des personnages ***Submersible*** et *Affecté par la viscosité* aux acteurs *Cube*. La figure 5.33 illustre les éléments ajoutés au MV pour obtenir une interaction entre les cubes et l'eau. La caméra, la scène statique et la lumière ont été retirées afin d'alléger la figure. Dans ce MV, le *gestionnaire de la cohérence présente* instancie une interaction *Viscosité* et une interaction *Archimède* par acteur *Cube*.

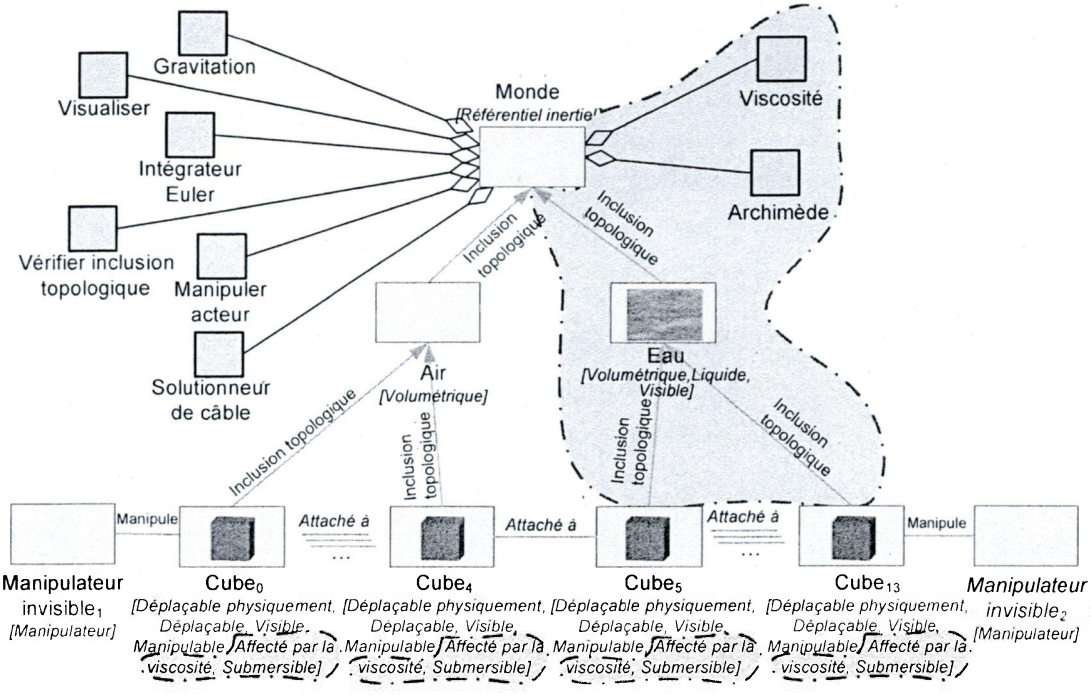

Figure 5.33 : Diagramme APRI de l'assignation des personnages *Affecté par la viscosité* et *Submersible* aux acteurs *Cube*, de l'ajout de l'acteur *Eau* et de l'ancrage des interactions *Viscosité* et *Archimède* afin que les acteurs interagissent avec l'eau.

Par suite de ces extensions, les cubes flottent dans l'eau et sont ralentis proportionnellement à leur vitesse. Ce comportement des cubes se transmet à la chaîne de façon transparente. La conception d'un tel MV est facilitée car les acteurs peuvent interagir individuellement avec l'eau et collectivement dans un contexte de câble, ce qui évite la conception spécifique d'un câble qui interagit avec l'eau. Ici encore, la composabilité ne cause

pas de problème. À ce stade, les cubes alternent entre l'air et l'eau car l'interaction *Gravitation* s'applique seule dans l'air et l'interaction *Archimède* repousse le cube hors de l'eau. L'interaction *Vérifier inclusion topologique* se charge d'inclure topologiquement les cubes dans l'un ou l'autre selon leur position. Cette modification des relations en tant qu'usager (voir section 2.4.2) des liens entre les éléments (voir section 2.4.1) est donc supportée par **APIA**.

Le *gestionnaire de la cohérence présente* vérifie continuellement les règles et, lorsque l'interaction *Vérifier inclusion topologique* ajoute ou retire une relation, il instancie ou détruit l'instance d'interaction. Ceci peut survenir plusieurs centaines de fois par seconde pour l'ensemble du MV, ce que supporte **APIA**. Le résultat global est l'observation du comportement émergent de flottaison à la surface qui est le résultat de la combinaison de trois interactions et non d'une interaction spécifique de flottaison de surface. En dernière observation, le câble peut présenter un profil typique d'un câble dans l'eau dont la partie submergée est plus droite que celle dans l'air qui est courbée par la force gravitationnelle, tel qu'illustré à la figure 5.11 f). Ce phénomène résulte de la composition et de l'extension du MV et non d'une interaction spécifique.

5.4.8 Sectionnement du câble en deux

Puisque qu'un câble est défini par des cubes reliés entre eux avec une relation, le retrait d'une relation entre deux cubes créera deux chaînes. Par exemple, en enlevant la relation *Attaché à* entre l'acteur $Cube_6$ et $Cube_7$, le *gestionnaire de la cohérence présente* se rend compte que l'interaction *Solutionneur de câble* en dépend. Suite à la vérification de la LCR et du TIP, il divise l'instance d'interaction en deux. Le MV se retrouve donc avec deux petits câbles gérés chacun par une instance d'interaction tel qu'illustré à la figure 5.11 g). **APIA** gère cette modification en cours d'exécution en appliquant les règles d'interaction inscrites dans le modèle conceptuel. Le *gestionnaire de la cohérence présente* ne fera interagir que des acteurs attachés les uns aux autres. L'ordonnancement des deux instances est pris en charge automatiquement par **APIA**. Chaque instance sera appelée à son tour.

Cette modification d'un câble en deux segments constitue un bon exemple de modification du MV basée sur la modification des liens entre entités (voir section 2.4.1). La modification actuelle est de type scénariste (voir section 2.4.2). Une modification du type

usager aurait pu être possible avec une interaction qui vient sectionner le câble comme des ciseaux.

5.4.9 Autres actions

Il est possible d'enlever une relation qui lie le dernier cube d'un câble et le cube suivant afin de l'isoler. Le cube, maintenant libéré du câble, est retiré de l'interaction *Solutionneur de câble*. L'interaction *Intégrateur Euler* s'applique à nouveau. Le cube tombe alors dans l'eau à cause de la gravitation et flotte à la surface à cause de la force d'Archimède, tel qu'illustré à la figure 5.11 h). Il s'agit d'un exemple de composabilité similaire à celui de l'exemple de la figure 2.4.

La dernière action consiste au retrait de la propriété de rendu de l'acteur $Cube_9$. Cette action a des conséquences car le personnage *Visible* requiert cette propriété. Le gestionnaire retire donc ce personnage de l'acteur $Cube_9$. L'interaction *Visualiser* requérant ce personnage, le gestionnaire enverra un message à l'instance d'interaction visée pour l'avertir de cette modification. L'acteur $Cube_9$ disparaît alors visuellement de la scène mais continue d'interagir à d'autres niveaux. Par exemple, le câble reste fonctionnel tel qu'illustré à la figure 5.11 h). La fine granularité des acteurs et des actions implique une fine granularité de leur gestion par les gestionnaires.

5.4.10 Synthèse et réflexions sur l'exemple de câble

Cet exemple a démontré qu'**APIA** permet la conception de MVs *agiles* et que cette *agilité* est gérée en cours d'exécution. Ce même exemple aurait pu être construit à partir du MV de la figure 5.1, et ce, même en cours d'exécution. Les décisions de conception dans un tel MV ne sont pas faciles. Par exemple, si les cubes avaient été nommés maillons, ceci les aurait désignés comme futurs maillons dès leur conception et aurait limité leur utilisation future en tant qu'acteurs individuels. Un changement ultérieur pour un nom plus générique comme cube aurait compensé cette lacune. Toutefois, de telles corrections *a posteriori* doivent être évitées afin de faciliter les extensions futures. Peu de problèmes de ce type, appartenant aux groupes d'actions 1 et 2 (voir section 4.2), ont été rencontrés lors de la conception de cet exemple. Pour les actions du groupe 3, seuls des problèmes de performance ont parfois été observés.

L'utilisation du méta-modèle conceptuel et du processus **APIA** ne sont donc pas des conditions suffisantes pour obtenir un MV *agile* mais facilitent son implantation. Chacune des extensions de cet exemple n'a pas remis la version précédente en question, comme il est souvent le cas en OO (voir section 3.2.5). La figure 5.34 reprend le diagramme des méta-niveaux de la figure 3.6 avec **APIA** comme méta-niveaux et l'exemple de câble de la section 5.4 pour les deux autres niveaux. Les images à l'intérieur des boites proviennent des chapitres précédents. Le lecteur peut s'y référer pour plus de détails.

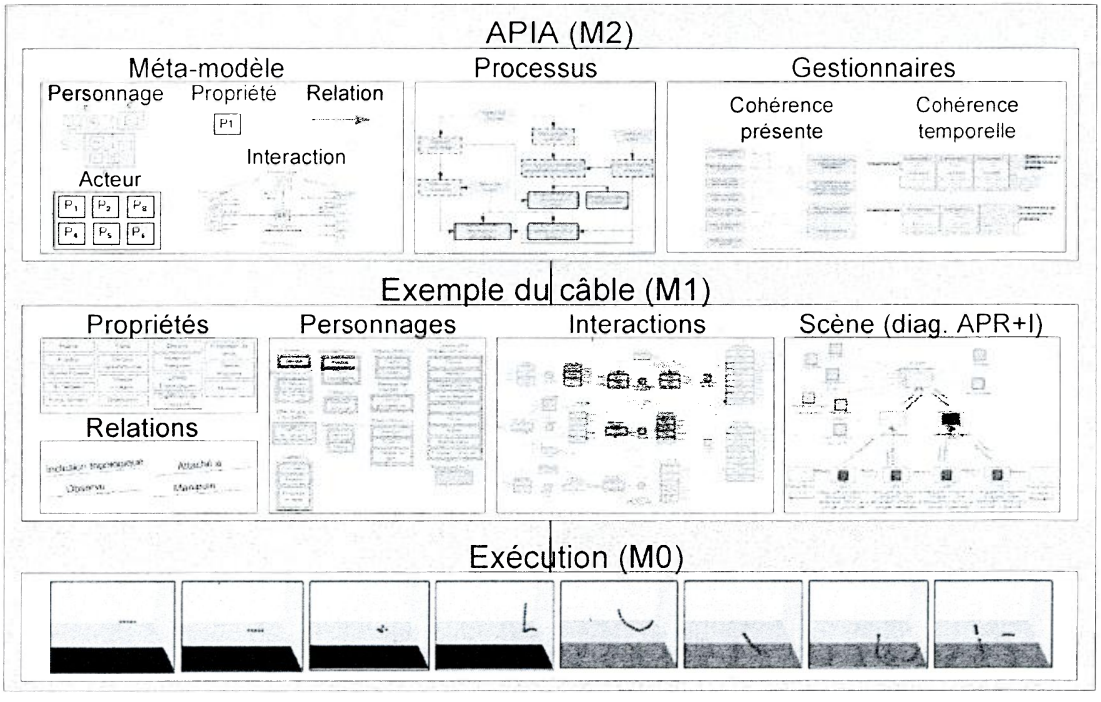

Figure 5.34 : Représentation des méta-niveaux du MV avec le câble dans lequel l'exécution est dictée par ce qui est décrit dans le modèle conceptuel du MV, lui-même conforme au méta-modèle conceptuel **APIA**.

5.5 Exemple de MV pour la cryochirurgie

La cryochirurgie assistée par IRM pour le traitement de nombreux cancers est un exemple d'intervention chirurgicale pouvant bénéficier de l'utilisation d'un MV pour l'entraînement, la planification et l'assistance aux interventions. Les besoins en modélisation et en implantation diffèrent légèrement pour chacun de ces modes. Toutefois, il existe

suffisamment de similitude entre eux pour les développer et les exécuter à partir d'une même base puis exploiter l'*agilité* d'**APIA** pour réutiliser, composer et étendre au maximum les éléments communs. Cette section démontre l'*agilité* résultante d'**APIA** dans la conception des différents modes d'utilisation des MVs pour la cryochirurgie.

Cette application s'inscrit dans le cadre du projet Virtual Environments, from 3D Representations to Task planning and EXecution, ou VERTEX (Poussart *et al.*, 2000). Ce projet vise le développement d'une architecture générique de simulation et des outils nécessaires pour réaliser l'acquisition in situ des données dans le but de réaliser la planification, l'entraînement et l'assistance à l'opération de façon optimale en utilisant des MVs qui recréent les comportements physiques pour une application donnée.

La planification, première étape de l'intervention, sert à établir les tâches et à optimiser celles-ci selon plusieurs critères (coût, temps, performance, etc.). Elle évalue donc les différents scénarios possibles pour une intervention donnée. Ensuite, l'entraînement vise à reproduire certaines conditions rencontrées durant l'intervention réelle afin d'améliorer les habilités et l'efficacité de l'humain qui effectue la tâche. Le but consiste à se familiariser avec une tâche donnée par la répétition et l'enseignement. L'assistance est l'étape ultime du processus des interventions. Par une simulation parallèle à l'intervention en cours, ce mode permet d'assister la ou les personne(s) qui effectue(nt) l'intervention. Ce mode implique que les paramètres des modèles proviennent du monde réel et soient mis à jour en temps réel. Ces données sont intégrées au MV afin de construire la modélisation qui sert de comparaison avec le monde réel.

La cryochirurgie (Harrison, 2003) est une technique qui consiste à traiter certaines pathologies par le froid. Une sonde cryogénique est insérée dans les tissus de façon minimalement invasive. La boule de glace formée vient alors détruire les tissus malades. Cette technique permet d'éviter l'hospitalisation prolongée. Afin de guider l'opération, l'utilisation d'un système d'IRM d'intervention (IRMi) permet de voir la position de la sonde, de la tumeur et de la boule de glace en temps réel. La figure 5.35 montre un chirurgien effectuant une opération de cryochirurgie dans l'appareil d'IRM.

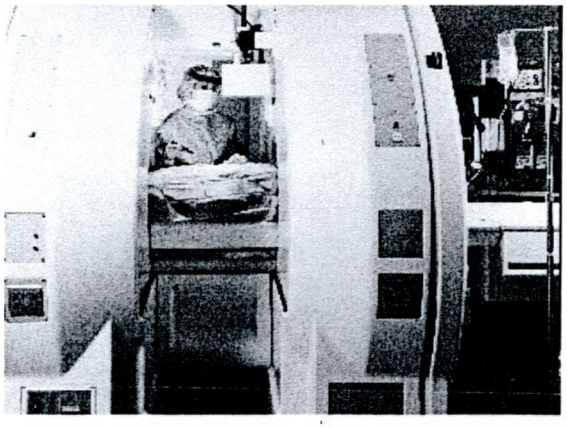

Figure 5.35 : Cryochirurgie assistée par images à résonance magnétique.

La destruction des tissus cancéreux est effective lorsque la température atteint -40°C. Parce que seulement la limite à 0°C n'est visible à l'IRM, le chirurgien doit évaluer, selon son expérience, la zone à l'intérieur de la boule de glace qui se situe à -40°C. La figure 5.36 montre un exemple de boule de glace imagée par un appareil d'IRM.

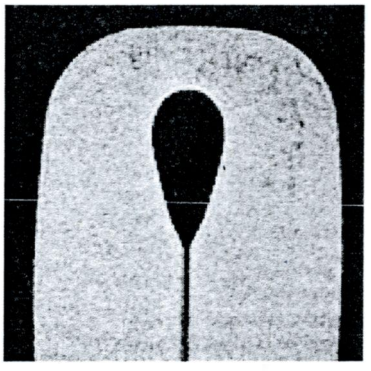

Figure 5.36 : Image de résonance magnétique (RN) montrant en noir les tissus gelés découlant de l'absence de résonance des protons dans les zones gelées.

Les prochaines sections montrent l'*agilité* résultant de l'utilisation **APIA** pour les modes d'entraînement, de planification et d'assistance à l'intervention de cryochirurgie. Quoique cette application a utilisé les données provenant des vrais équipements, elle n'a jamais impliqué de vrais opérateurs ou d'équipements reliés en temps réel avec le MV.

5.5.1 MV pour l'entraînement de la cryochirurgie

Premièrement, **APIA** a servi à implanter un MV d'entraînement d'interventions de cryochirurgie (Harrison, 2003). La figure 5.37 a) illustre une vue globale d'un MV avec un appareil d'IRMi et une sonde cryogénique insérée dans un patient, tel qu'illustré à la figure 5.37 b). Ensuite, la modélisation physique du processus de gel en représentation isotherme 3D, illustrée à la figure 5.37 c), permet au chirurgien d'observer et de mieux comprendre l'évolution de la boule de glace dans l'espace et, par conséquent, la distribution de température, dont la limite à -40°C de destruction des tissus.

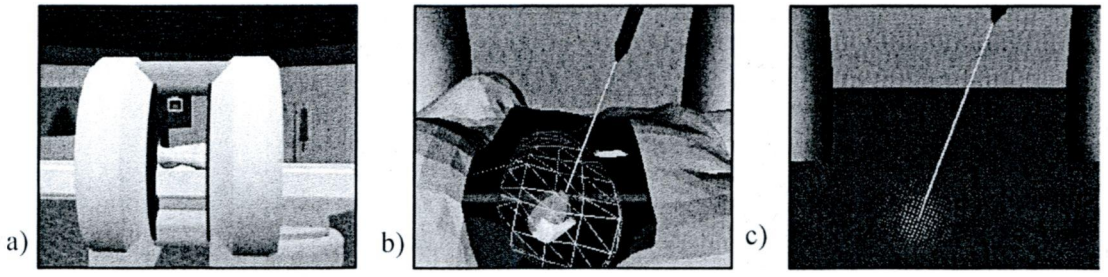

Figure 5.37 : MV développé avec **APIA** permettant l'entraînement des tâches de cryochirurgie : a) environnement d'intervention b) sonde cryogénique pénétrant à l'intérieur de la tumeur c) processus de gel modifiant la distribution spatiale de température.

La figure 5.38 montre l'ensemble des interactions, personnages et propriétés utilisés pour les trois modes incluant un mode d'entraînement. Cet exemple réutilise l'interaction *Visualiser* et de nombreux éléments des exemples précédents. L'interaction *Gel* réutilise un réseau de neurones (Basile-Bellavance & Harrison, 2000), ce qui constitue un autre exemple de modèle spécifique supporté par **APIA**. Ce modèle prédit la distribution de température dans la boule de glace à partir de la distribution précédente. Ce réseau de neurones ne supporte qu'une seule sonde cryogénique, ce qui est représenté par une multiplicité de 1 pour le personnage *Source de froid*. Une amélioration du réseau de neurones afin de supporter plus d'une sonde cryogénique n'impliquerait qu'une modification de multiplicité de 1 à 1..n pour le personnage *Source de froid*. Cet exemple montre la facilité avec laquelle **APIA** permet les modifications et les extensions. L'interaction *Simulateur d'IRM* n'a pas été implantée mais montre qu'elle permettrait d'étendre le MV d'entraînement avec la création, de façon réaliste, d'une image RM produite à partir de l'état du MV comme si l'intervention était réelle.

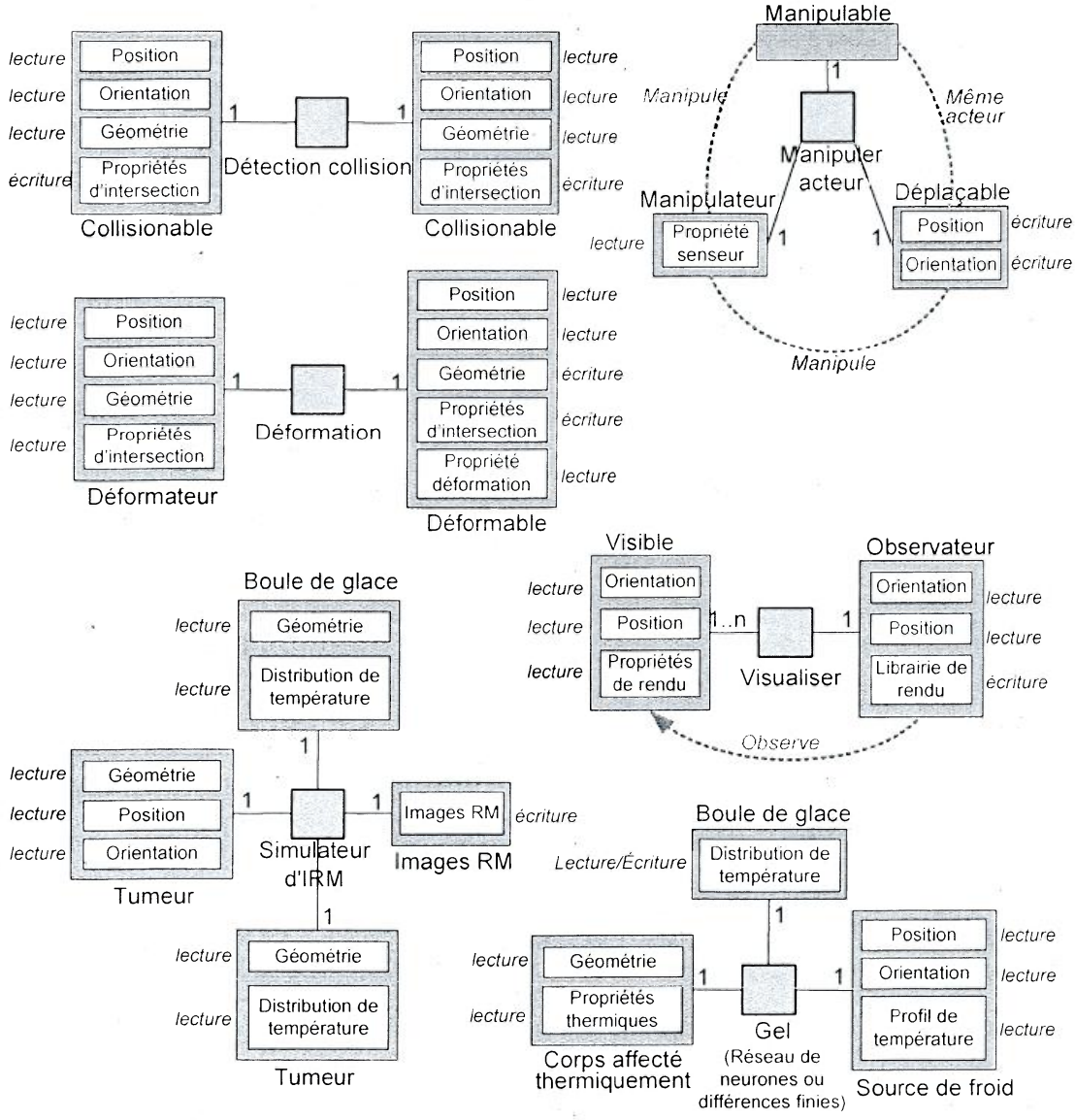

Figure 5.38 : Ensemble des interactions utilisées dans les modes de planification, d'entraînement, et d'assistance à la cryochirurgie.

Les interactions *Détecter collisions* et *Déformation* implantent la déformation du foie sous la force de la sonde cryogénique. Bien que l'objectif initial visait la déformation avec un modèle d'éléments finis (Schwartz, 2003), la lourdeur de cette approche pour une application d'entraînement a empêché son utilisation en mode temps réel. L'interaction *Déformation* implante plutôt un modèle plus simple. De plus, la déformation et le processus de gel agissent séquentiellement car, autrement, il faudrait tenir compte de leur interaction. Plusieurs

interactions ne définissent pas de règles relationnelles. **APIA** gère donc ce cas en instanciant une interaction pour chaque combinaison de multiplicité. Étant donné qu'il existe une seule instance par personnage pour la plupart de ces interactions, il en résulte une seule instance par interaction.

La figure 5.39 illustre le diagramme APRI ainsi que les communications avec le monde réel, soit senseurs et effecteurs servant à l'entraînement de la cryochirurgie. L'usager manipule un capteur à six degrés de liberté Stylus ^{MD} Isotrack II^{MD} de la compagnie Polhemus qui envoie les données de position et d'orientation à la sonde cryogénique virtuelle avec les mécanismes de senseurs. La scène virtuelle est affichée à l'usager par l'interaction Visualiser qui montre la pièce, l'appareil d'IRM, la sonde cryogénique, le patient, le foie à l'intérieur duquel se trouve une tumeur, une représentation isotherme 3D de la distribution de température ainsi que les images RM découlant de la situation courante telles qu'elles seraient calculées par l'interaction Simulateur d'IRM.

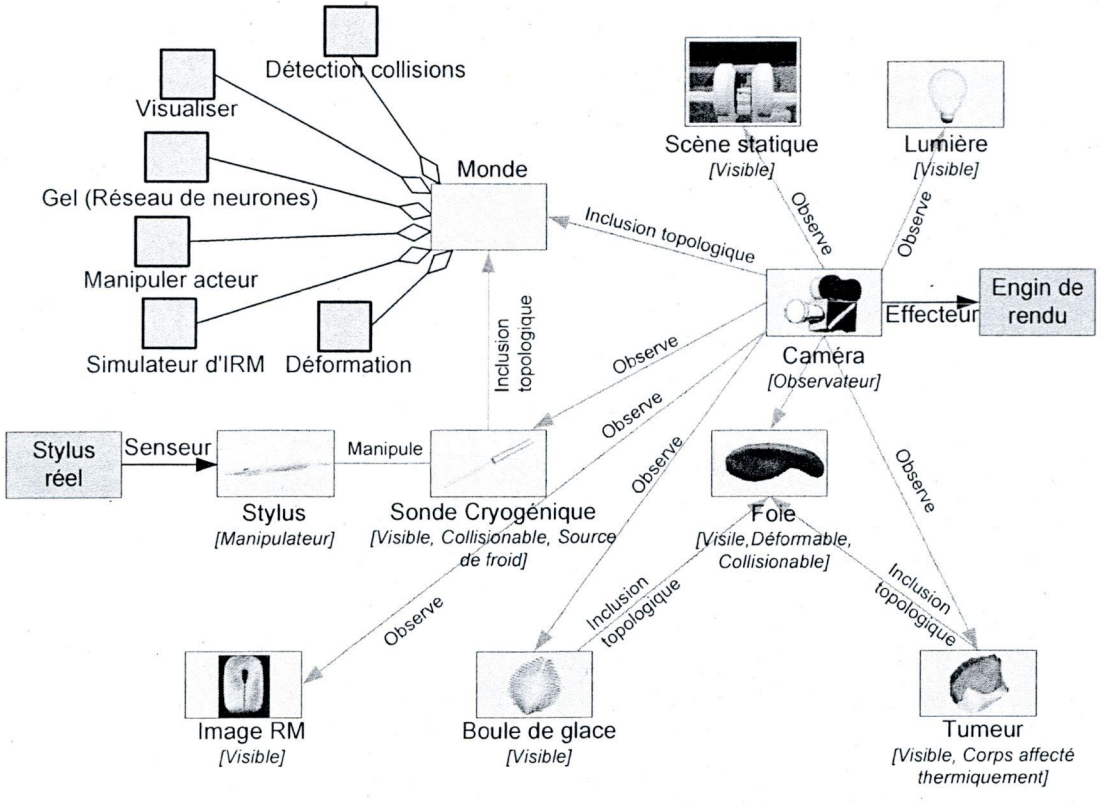

Figure 5.39 : Diagramme APRI pour l'entraînement d'interventions cryochirurgicales.

5.5.2 MV pour la planification de la cryochirurgie

Les tumeurs adoptent parfois des formes géométriques simples comme des sphères. Une boule de glace en forme de goutte d'eau créée par une cryosonde suffit alors à les éliminer. Cependant, avec des tumeurs de formes plus étendues, le chirurgien doit combiner plusieurs cryosondes tout en minimisant la destruction des tissus sains environnants. Lorsque colocalisées, ces cryosondes créent des géométries de tissus gelés relativement complexes. Le résultat de la combinaison des sondes est difficilement prévisible par un chirurgien. Un outil de planification devient alors nécessaire.

Un tel outil nécessite une représentation géométrique de la ou des tumeurs ainsi qu'un modèle de propagation thermique du front de froid dans les tissus mous. Les tumeurs étant visibles dans l'IRM, il est possible d'en reconstruire un modèle 3D et de l'ajouter à un MV. Avec la modélisation du processus de gel et de la déformation des tissus, un MV tel qu'illustré à la figure 5.37 b) permet de planifier le positionnement optimal des sondes et le temps de gel requis pour accomplir l'opération chirurgicale. L'outil de planification permettra d'établir la meilleure façon de positionner les sondes cryogéniques dans la tumeur afin de maximiser sa destruction tout en minimisant la destruction des tissus sains.

La première étape consiste à l'acquisition d'images RM d'un patient afin d'en segmenter la ou les tumeur(s) à l'aide d'algorithmes de traitement d'images (Arnold, 2000). La ou les tumeur(s) est(sont) placée(s) ensuite dans le MV où un usager vient y insérer une ou plusieurs sonde(s) cryogénique(s) et démarrer le processus de gel. La meilleure configuration spatiale pour la ou les sonde(s) cryogénique(s) peut être explorée. L'*agilité* résultant d'**APIA** permet de réutiliser un maximum d'éléments développés lors du MV d'entraînement dans celui de planification. Une interaction, illustrée à la figure 5.40, doit d'abord être ajoutée.

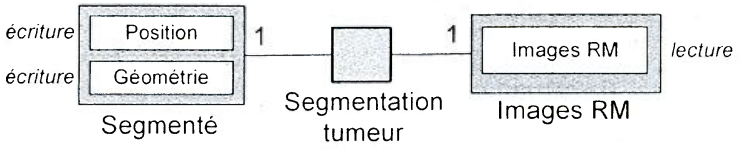

Figure 5.40 : Définition de l'interaction *Segmentation tumeur* de la tumeur des images RM.

La figure 5.41 illustre la modification du MV précédent avec les ajouts et les retraits, cer derniers étant représentés avec par des régions estompées entourées d'un trait pointillé.

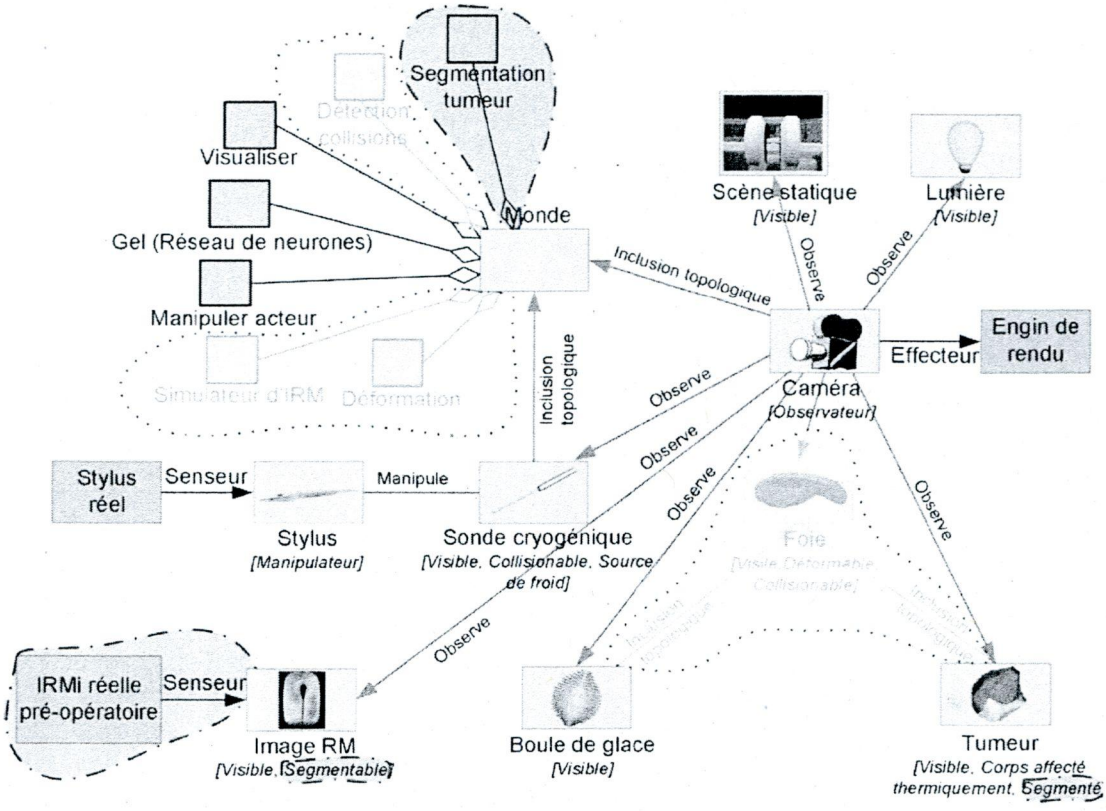

Figure 5.41 : Diagramme APRI des modifications requises pour transformer le MV d'entraînement en MV de planification d'interventions de cryochirurgie.

La déformation et la détection de collisions ne sont plus nécessaires dans le mode de planification dont le but est de trouver le positionnement optimal de la sonde.

5.5.3 MV pour l'assistance à l'intervention cryochirurgicale

Un MV peut calculer la distribution de température entre la sonde et la zone de changement de phase, tous les deux disponibles avec l'IRM. La représentation de ce calcul, superposée à l'écran de l'appareil d'IRM qui montre les images RM tel qu'illustré à la figure 5.42, permet au chirurgien de connaître avec précision la région traitée à -40°C. Cette présentation d'information en provenance du MV dans le monde réel se nomme réalité

augmentée (Milgram & Kishino, 1994). Elle permet à l'usager de ne jamais perdre contact avec le monde réel tout communiquant avec des éléments d'information du MV.

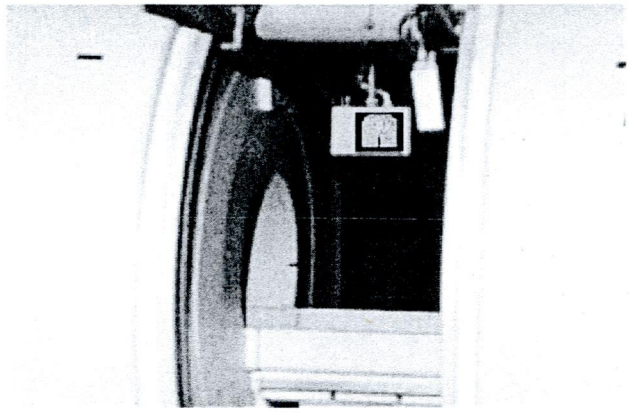

Figure 5.42 : Utilisation de la réalité augmentée qui consiste à superposer la température calculée dans le MV sur l'écran de l'appareil d'IRMi.

APIA peut être employé pour implanter un tel MV d'intervention de cryochirurgie. Dans ce mode, les images RM proviennent du monde réel. La boule de glace et la sonde cryogénique sont segmentées à chaque mise à jour des images RM. Un modèle de différences finies (Harrison *et al.*, 2000 ; Harrison, 2003), valide pour une seule sonde, remplace le précédent modèle de réseau de neurones. Il s'agit d'un autre exemple de modèle spécifique supporté par le méta-modèle conceptuel d'**APIA**. Ce modèle prédit la distribution de température par une interpolation entre les conditions limites extraites à l'aide d'une segmentation continue de la sonde et de la boule de glace à partir des images RM. La prédiction de température est retournée à l'appareil d'IRMi réel. L'assistance à l'intervention réutilise également plusieurs éléments en provenance de d'autres modes. La figure 5.43 illustre le diagramme APRI avec les extensions, mais sans les retraits, par rapport au MV d'entraînement ainsi que le senseur et l'effecteur des images RM qui sont lues et sur lesquelles sont superposés les profils de température calculés.

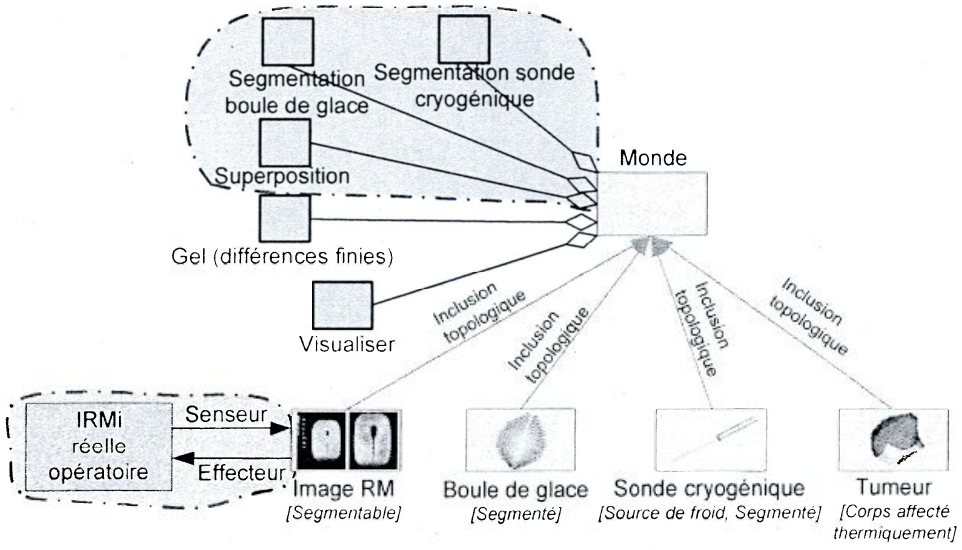

Figure 5.43 : Diagramme APRI pour le mode d'assistance aux interventions de cryochirurgie.

5.6 Exemple de MV pour l'inspection de barrages

APIA a également servi au développement de MVs d'entraînement et d'assistance à l'inspection de barrages. Pour cette dernière application, un MV a été interconnecté avec un sous-marin dans les locaux de l'Institut de recherche d'Hydro-Québec (IREQ). Cette expérience la capacité à modéliser différents domaines et à augmenter la réutilisabilité, l'extensibilité et la composabilité. On y retrouve un autre exemple de modèle spécifique de câble sous-marin, développé et validé à l'Université de Victoria (Buckham *et al.* 2004), qui a été intégré au sein des modèles conceptuels d'APIA afin de concevoir le MV d'entraînement à l'inspection de barrages. Bien que les applications de sous-marin et cryochirurgie soient différentes, l'agilité intrinsèque à APIA en permet l'implantation sans poser de problèmes conceptuels majeurs. Puisque cet exemple est similaire à celui de la cryochirurgie, il est disponible à l'annexe B.

5.7 Autres applications

Plusieurs autres MVs ont été conçus à partir d'**APIA**. Monnet (Drouin *et al.*, 2004), Crisis (Dinel, 2005) et Urban Zone sont illustrés à la figure 5.44. Martel (2005) s'est également servi des concepts à la base d'**APIA**.

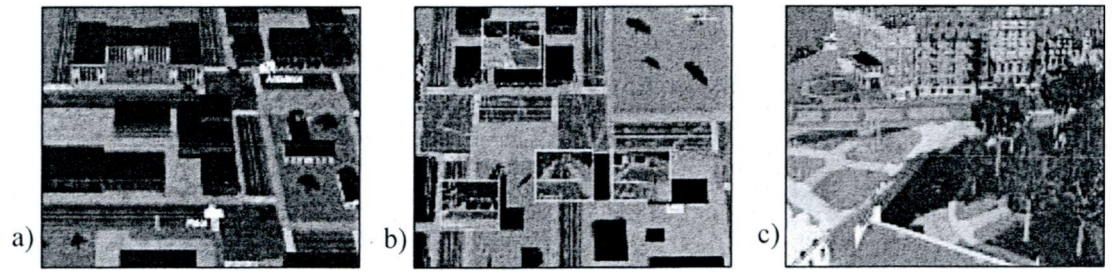

Figure 5.44 : Autres MVs développés avec **APIA** : a) Monet pour le suivit de piétons à travers un réseau de caméras b) Crisis pour un environnement collaboratif c) Urban Zone intégrant des bases données externes géo référencées.

Plusieurs constats émergent de l'utilisation d'**APIA**. D'abord, la fine granularité d'APIA et l'utilisation de règles d'interactions ainsi que des personnages permettent à une même architecture de modéliser six MVs dissemblables. Il est donc apparent qu'*APIA* permet une *agilité* suffisante pour implanter de nombreux types de MVs. Plusieurs améliorations ont été identifiées suite au développement de ces MVs. Premièrement, les règles d'interaction devraient permettre des formulations plus sophistiquées : conditions sur les propriétés, formulation avec langages scriptés, interactions instantanées. De plus, une réimplantation de l'architecture au niveau des interfaces et du codage devrait rendre les éléments plus facilement accessibles à travers le noyau pour permettre une meilleure modifiabilité et extensibilité. Une interface de scénarisation pourrait aussi faciliter le travail de conception. Des problèmes de rapidité ont également été observés. Ceux-ci étaient principalement dus à l'utilisation d'une règle relationnelle désignée *All*, qui permet n'importe quel chemin relationnel entre deux acteurs, ce qui augmentait considérablement la taille de la LCR. Son retrait a éliminé le problème sans nuire à l'*agilité*. Finalement, une gestion du temps plus automatisée supportant l'*agilité* permise par **APIA** faciliterait le développement d'applications.

5.8 Mesures de rapidité d'exécution d'APIA

La rapidité d'exécution n'a jamais été l'objectif principal. Toutefois, **APIA** ne doit pas ajouter une charge supplémentaire trop importante par rapport aux autres approches. Tel que mentionné au chapitre 4, il existe deux sortes d'événements : ceux dépendants du temps, c'est-à-dire qui provoquent l'appel des instances d'interactions par la modification d'instances de propriétés, et ceux indépendants du temps.

Pour la première catégorie, il faut vérifier les dépendances des interactions envers les propriétés. Cette tâche n'ajoute qu'une faible surcharge par rapport à une approche d'appels sans vérification. À l'exception du démarrage pour instancier les éléments dans le MV, les applications de cryochirurgie et d'inspection de barrages ne contiennent que ce type d'événements. Par conséquent, ces MVs s'exécutent de façon déterministe. Pour le MV d'entraînement de cryochirurgie, les cinq instances d'interactions sont appelées aux 10 ms, ce qui représente un taux de 100 Hz ou 500 appels d'instances d'interactions par seconde. Les données du Stylus^{MD} arrivent à toutes les 10 ms et les images sont envoyées à toutes les 30 ms à l'affichage. L'application d'inspection de barrages s'exécute également à 100 Hz. Par conséquent, à chaque 10 ms, l'ensemble des instances d'interactions, incluant les détections de collisions réparties sur les noeuds de calcul, sont appelées. Dans le MV d'entraînement de manipulation de sous-marins, **APIA** effectue 16 appels pour les interactions, un pour l'affichage et deux pour les senseurs, ce qui représente un total de 19 appels à chaque 10 ms. Le tout représente environ 2000 appels par seconde. L'exemple du MV avec un câble du chapitre 5 est exécuté de la même manière. Le noyau appelle les instances d'interactions selon les événements qu'elles génèrent.

Les événements indépendants du temps amènent une plus grande variabilité à l'exécution du MV. L'exemple du câble de la section 5.4 contient une bonne part de ces événements. La liaison de deux acteurs, l'assignation d'un personnage à un acteur et l'ancrage d'une interaction à un acteur exigent plus de temps car le *gestionnaire de la cohérence présente* vérifie les règles d'interaction pour l'ensemble des interactions qui en dépendent, et ce, pour l'ensemble du MV. Le temps de gestion de ces trois événements peut alors s'avérer plus important. Pour le MV avec un câble, le gestionnaire doit vérifier les règles d'interaction à chaque ajout de relation *Inclusion topologique* par l'interaction *Vérifier Inclusion topologique*.

Dans cet exemple, le gestionnaire vérifiera les règles d'interaction pour *Gravitation*, *Viscosité* et *Archimède* car elles dépendent de cette relation. Pour un câble à 14 cubes totalement immergé, il y aura 14 x 2 (*Viscosité* et *Archimède*) + 1 (*Gravitation*) instances d'interactions à vérifier à chaque 10 ms. Le temps requis par le gestionnaire dépend donc de la complexité du MV et de la complexité des règles. Lorsque l'action *Lier acteur* est posée, le gestionnaire doit vérifier si des interactions doivent être instanciées. Le temps pris pour cette vérification est proportionnel à la complexité des règles relationnelles et de multiplicité. Tel qu'illustré à la figure 5.45, l'interaction *Solutionneur de câble* monopolise la majeure partie du temps de vérification des règles lors de l'ajout de liaisons. Des optimisations du *gestionnaire de la cohérence présente* pour ce type de règles d'interaction pourraient ramener la vérification à un temps comparable aux autres interactions.

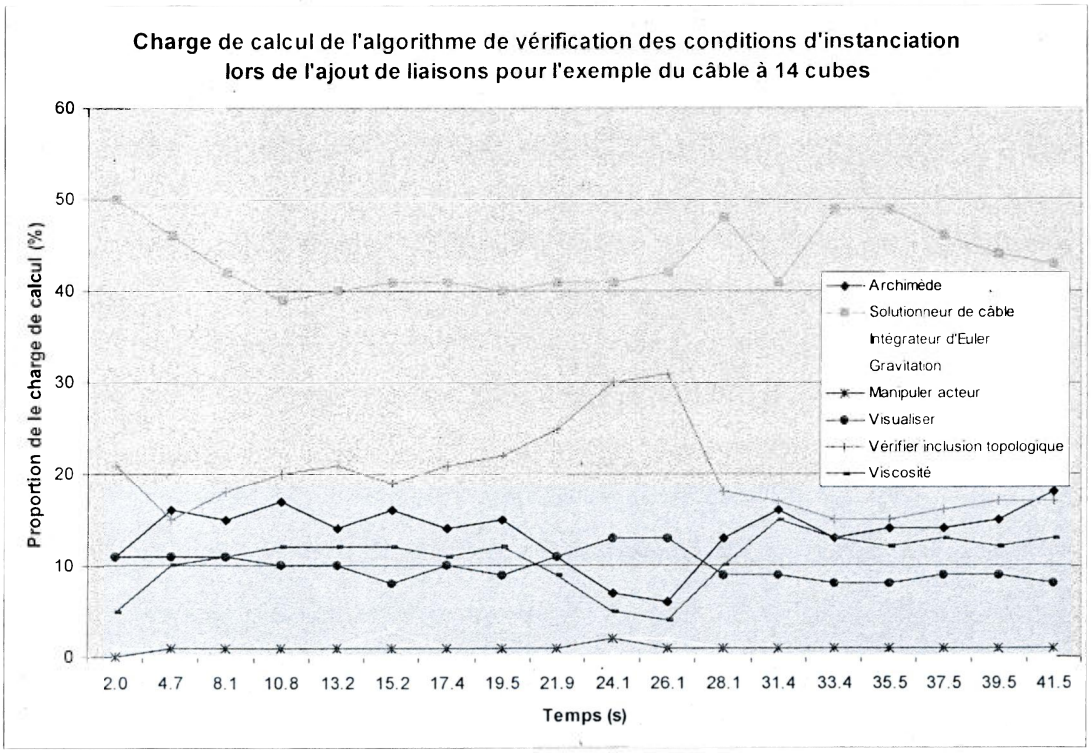

Figure 5.45 : Charge de calcul relative entre les interactions de la vérification des règles d'interaction lors de l'ajout de relations pour l'exemple du câble. (sur Pentium M 2Ghz)

Tel que mentionné précédemment, le noyau répartit son temps entre l'appel des

instances d'interactions et la gestion du MV. La figure 5.46 illustre la proportion du temps processeur pour chacun des événements gérés par le gestionnaire lorsque le câble de la section 5.4 est à la surface de l'eau, ce qui correspond au pire cas. La gestion de l'action d'ajout d'une relation entre deux acteurs et l'appel de l'instance d'interaction *Solutionneur de câble* utilisent la majeure partie du temps de calcul. Dans ce cas, des relations sont ajoutées et enlevées jusqu'à plusieurs centaines de fois par seconde, causant ainsi chaque fois l'instanciation et la destruction des instances d'interactions *Archimède* et *Viscosité*.

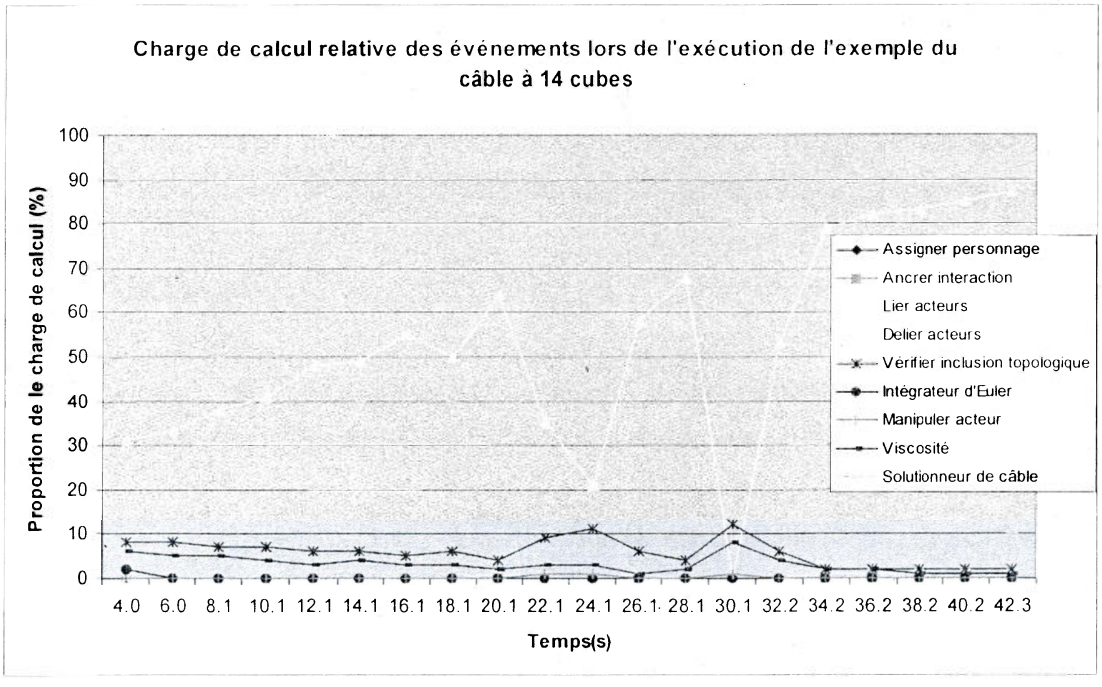

Figure 5.46 : Charge de calcul relative du traitement des événements générés lors de l'exécution de l'exemple du câble.

Il existe de nombreuses optimisations possibles dans **APIA** pour limiter le temps processeur assigné à la gestion du MV. Par exemple, lors de l'instanciation de certaines interactions du type de *Solutionneur de câble,* il serait préférable de ne pas construire la LCR, car celle-ci requiert trop de temps processeur. Le retrait d'une relation prendrait plus de temps mais beaucoup moins que ce qui est requis pour construire la LCR à chaque ajout de relation.

5.9 Conclusion sur la démonstration d'agilité avec APIA

Ce chapitre a démontré qu'**APIA** permet le développement de MVs avec des efforts modestes de la part du scénariste, facilite la réutilisation d'éléments et la modification à travers les différents modes et applications. Les quelques exemples ont présenté des MVs plus *agiles* et faciles à créer. Plusieurs types de compositions, d'extensions, de réutilisations, de modifications et une plus grande liberté d'action sont supportés. Ces exemples ont également démontré la capacité des modèles conceptuels à supporter un grand nombre de modèles spécifiques ainsi que la versatilité d'**APIA** à concevoir des MVs dissemblables.

Chapitre 6

Analyse d'APIA

Le chapitre 5 et, dans une moindre mesure le chapitre 4, ont présenté des exemples d'*agilité* résultant d'**APIA**. Ce chapitre reprend et discute séparément des stratégies utilisées dans **APIA** et mentionnées précédemment qui permettent d'augmenter l'*agilité*.

6.1 Centrer la modélisation sur l'interaction

L'interaction permet de représenter et de contenir une large gamme de modèles, que ceux-ci interagissent de façon symétrique ou asymétrique. L'interaction facilite également l'interopérabilité. Alors que les entités doivent définir un ensemble de protocoles pour interagir, ces protocoles sont désormais majoritairement inclus dans l'interaction. Par conséquent, si deux acteurs n'interopèrent pas "thermiquement" ou "optiquement", il est plus facile d'ajouter une interaction de ce type entre eux qu'en modifiant deux entités. Les acteurs participent quand même à cette interopérabilité, c'est-à-dire qu'elles doivent jouer les personnages requis, donc posséder les propriétés requises.

Les entités sont généralement interdépendantes dans une approche centrée sur l'entité, alors que les interactions le sont moins dans une approche centrée sur l'interaction. Par conséquent, un scénariste peut inclure plus facilement un grand nombre d'interactions dans un MV car il n'a pas à constamment modifier les entités. Cette présence de plusieurs interactions dans un même MV peut favoriser l'apparition de comportements émergents tel que décrit dans la section 5.4.7. De tels comportements n'émergent pas d'une interaction spécifique mais plutôt de la combinaison de plusieurs interactions sur de nombreux acteurs.

Les interactions facilitent également les différents types de modifications. La modification d'un comportement par un changement de valeur d'une propriété est prise en charge automatiquement par l'interaction. La section 2.4.1 en présente un exemple facilement modélisable avec une interaction. La modification fonctionnelle est également possible avec l'utilisation d'interactions et de règles. Dans ce même exemple, l'interaction dépendrait de la tension dans la corde pour s'instancier et lancer des objets. La modification d'un MV par les relations est également possible dans **APIA**. L'interaction *Vérifier inclusion topologique* du MV avec un câble de la section 5.4, qui change la relation *Inclusion topologique* entre les acteurs *Cube* et les acteurs *Air* ou *Eau*, en est un exemple.

L'approche centrée sur l'interaction facilite également la conception d'un MV car l'interaction est peu affectée par le nombre d'acteurs. Ces derniers ne doivent que posséder les propriétés et jouer des personnages sans se soucier des autres acteurs. Dans une approche centrée sur l'entité, la conception de chaque nouvelle entité implique la considération des interactions avec les n autres entités du MV et *vice-versa*. Le gestionnaire instancie autant d'instances d'interaction que de combinaisons d'acteurs répondants aux règles d'interaction, même si des acteurs sont ajoutés en cours d'exécution. Dans le cas où des entités doivent interagir d'une nouvelle façon, une approche centrée sur l'entité requiert une modification de toutes les entités concernées. Ceci implique un probable réusinage, une tâche qui diffère d'une entité à l'autre et qui s'avère difficile en cours d'exécution. Dans l'approche centrée sur l'interaction, l'ancrage d'une nouvelle interaction et l'ajout de personnages aux acteurs concernés déclenchera, par le gestionnaire, l'instanciation d'interaction selon les règles, peu importe le nombre d'acteurs. La taille du MV influence donc moins sur le niveau d'effort requis lors d'extensions.

L'approche centrée sur l'interaction implique la considération des dépendances et des conflits entre les interactions. L'exemple de cryochirurgie avec le gel et la déformation qui agissent séquentiellement en est un exemple de solution. **APIA** ne possède toutefois pas de mécanismes particuliers pour gérer ces cas. Il s'agit également d'un problème de l'approche centrée sur l'entité.

Cette approche implique deux autres concepts importants : les règles d'interaction qui permettent de gérer la cohérence et les personnages qui définissent la capacité d'agir de façon générique.

6.1.1 Règles d'interaction

Les règles d'interaction sont des conditions d'application, donc d'instanciation de nouvelles interactions ou de modification de celles existantes. Elles sont établies par rapport aux éléments d'APIA. Les règles permettent :

- la gestion de la cohérence présente avec la définition des conditions d'instanciation de l'interaction selon des règles relationnelles et de multiplicités ;
- la gestion de la cohérence temporelle et son automatisation avec la définition des conditions d'appel d'une interaction ou d'une instance d'interaction ;
- l'optimisation de l'application des interactions ; et
- l'encapsulation de nombreux types de modèles spécifiques avec les règles de multiplicités et relationnelles.

Premièrement, les conditions d'instanciation de l'interaction, en collaboration avec les gestionnaires, permettent de gérer la cohérence présente du MV. De plus, elles permettent l'interaction de plusieurs acteurs sans que ceux-ci définissent eux-mêmes leurs conditions d'interaction. Si ces acteurs jouent les personnages requis et s'ils sont liés les uns aux autres d'une certaine façon, alors les instances d'interaction se chargent de les faire interagir adéquatement. Puisque l'interaction vient avec ses règles, l'ajout d'une interaction non connue à l'avance est simplifié. Le contexte dans lequel agit l'interaction ressemble à l'héritage en OO mais sans les désavantages mentionnés à la section 3.2.5. Les conditions d'instanciation de l'interaction permettent également une grande réutilisation des comportements dans d'autres MVs dont la base (relations, propriétés et personnages) est la même. Ces règles assurent ainsi l'intégration de parties de MV, facilitant ainsi les caractéristiques de l'*agilité* en cours d'exécution.

Les règles d'appel des interactions en fonction de la modification des propriétés permettent de gérer la cohérence temporelle en facilitant l'ordonnancement des instances d'interactions. La réutilisation, la composition et l'extension en sont facilitées parce que l'ancrage d'une interaction, donc l'extension du MV, ne requiert pas une connaissance des mécanismes de son exécution. L'exemple de la figure 4.33 le démontre. Dans ce cas, l'*ordonnanceur* permet la gestion des appels des instances d'interactions sans l'intervention d'un humain. Il s'adapte à un MV modifié en cours d'exécution, ce qui est représentatif de

l'*agilité*.

Les règles permettent également d'optimiser l'application des interactions dans le MV. Un mécanisme similaire existe avec les DDM de HLA. Toutefois, il s'implante plus difficilement car il implique une coordination entre tous les fédérés et les entités afin de supporter l'aiguillage des attributs. Dans **APIA**, l'optimisation d'une interaction entre les acteurs n'implique aucune implication de leur part. Par conséquent, lors de son ancrage, la vérification de la règle donnera lieu à l'instanciation de l'interaction entre les bons acteurs. De plus, similairement à HLA et contrairement aux autres approches comme MASSIVE-3 ou VEOS, l'optimisation de l'application n'est pas limitée à la position spatiale. L'interaction de deux entités pourrait dépendre de d'autres propriétés comme la fréquence.

Finalement, les règles facilitent la création et l'implantation de l'interaction. Elles supportent différent personnages selon diverses multiplicités et relations. Ainsi, l'interaction peut encapsuler de nombreux types de modèles spécifiques, tant symétriques qu'asymétriques. Lors de l'implantation, l'adaptation d'un comportement existant et la conception d'un nouveau comportement en seront facilités.

6.1.2 Personnage comme abstraction de la capacité d'interagir

Les personnages amènent une généricité dans la capacité d'interagir des acteurs. Ils permettent ainsi aux acteurs de jouer de plusieurs rôles et d'interagir avec de nombreux autres acteurs. Par conséquent, en collaboration avec les interactions, ils augmentent l'*agilité* en facilitant l'ajout et la réutilisation de comportements.

Les personnages ne supportent pas des interactions à différents niveaux de détails ou de fidélité. En effet, il n'existe aucune règle ni type de personnages permettant aux gestionnaires de substituer des modèles de différents niveaux de fidélité. Cette tâche reviendrait à un *ordonnanceur* plus sophistiqué. Celui-ci choisirait le degré de fidélité et de détails, donc l'interaction appropriée, selon différents critères comme le niveau de ressources disponibles.

6.2 Gestionnaires et algorithmes

Les gestionnaires prennent en charge de nombreuses tâches du fonctionnent du MV. Ils facilitent le travail du scénariste car ils appliquent, tout au long de l'exécution, les directives fournies par d'autres scénaristes sous forme de règles d'interaction et des algorithmes prédéfinis, s'assurant ainsi du bon fonctionnement du MV même lorsque des modifications ont lieu. Par exemple, une majorité de MVs laisse aux scénaristes la tâche d'optimiser l'application des comportements. Dans **APIA**, cette optimisation est fournie avec l'interaction et lorsqu'un scénariste réutilise celle-ci, le *gestionnaire de la cohérence présente* instancie les interactions entre les bons acteurs. Le gestionnaire s'assure également de la conformité du déroulement du MV avec le modèle conceptuel.

Les gestionnaires, principalement celui de la cohérence présente, comprennent suffisamment les éléments du modèle conceptuel pour assurer la cohérence du MV, même lorsque celui-ci est modifié en cours d'exécution. Il faut évidement que le MV soit décrit de façon cohérente. Si le MV est cohérent pour toutes les actions du groupe 1 et 2, alors toute action du groupe 3 devrait le maintenir cohérent au fil des modifications effectuées. L'exemple du MV avec un câble démontre ce principe.

La gestion du temps d'**APIA** ne supporte l'*agilité* que dans une certaine mesure. En effet, les scénaristes doivent s'assurer qu'il n'existe aucune boucle qui empêcherait l'avancement du temps.

6.3 Processus d'actions à différents niveaux de conséquences

Tel que mentionné précédemment, l'*agilité* repose sur la capacité d'effectuer des actions qui la concrétisent. Toutefois, ces actions influencent l'*agilité* future. **APIA** propose une grande diversité d'actions rendant le MV *agile*. Selon le groupe auquel appartient l'action, elle aura un effet plus ou moins important sur l'*agilité* future. Pour cette raison, **APIA** répartit les actions selon différents groupes. Pour certaines actions, le scénariste doit être conscient que ses choix peuvent compromettre de façon importante l'*agilité* future. Ces actions se retrouvent dans le groupe 1. Certaines autres actions, du groupe 2, auront des conséquences limitées sur l'*agilité*. Le scénariste doit être conscientisé à la réutilisation, la composition, l'extension et à

la modification mais il peut tout de même supposer que ces caractéristiques ont déjà été considérées par d'autres dans le premier groupe d'actions et qu'il possède une certaine liberté d'action. Finalement, le groupe 3 contient un grand nombre d'actions qui augmentent l'*agilité* présente, facilite la gestion de la cohérence du MV et influence peu l'*agilité* future. Ceci constitue une amélioration par rapport aux autres architectures. Cette classification en trois niveaux facilite la collaboration entre les scénaristes. Pour un domaine donné, des spécialistes pourraient définir un ensemble d'éléments de référence utilisés par des scénaristes posant une majorité d'actions de niveau 2 ou 3.

Dans HLA, les actions que peut effectuer le scénariste se retrouvent dans deux catégories : création du FOM et création des fédérés. Puisque la première catégorie est définie avant l'exécution, l'extension en cours d'exécution est limitée. Les seuls types d'extension sont l'ajout d'entités d'un type déjà défini (extension en nombre), la création d'un attribut déjà défini et l'envoi d'une interaction HLA déjà connue. De plus, comparé aux autres architectures telles que HLA, OO, Bamboo, VEOS et DEVA 3, le nombre d'actions que peut effectuer le scénariste en cours d'exécution est plus important dans **APIA**.

6.4 Méta-modèle conceptuel

Le méta-modèle conceptuel permet de contraindre le scénariste à utiliser certains éléments et d'une manière qui facilite la modifiabilité, l'extensibilité, la réutilisabilité, la composabilité et la liberté d'action. Puisque tous les MVs sont conformes au méta-modèle conceptuel, les gestionnaires savent comment les gérer car ils en connaissent leur contenu éventuel. Par exemple, un gestionnaire peut interpréter toutes les règles de multiplicité des interactions car il sait qu'elles seront définies et en connait le sens. Il en est de même des autres éléments. Il existe un compromis pour ce méta-modèle conceptuel. D'un côté, un méta-modèle conceptuel sophistiqué faciliterait la gestion mais contraindrait la conception. D'un autre côté, un méta-modèle conceptuel simpliste laisserait toute la liberté au scénariste mais ne permettrait pas de gérer efficacement la cohérence du MV. **APIA** ne correspond probablement pas à un compromis optimal mais encadre suffisamment la conception des MVs pour obtenir et gérer un plus grand nombre de caractéristiques de l'*agilité* que les autres approches tout en étant flexible pour concevoir des applications fort différentes.

6.5 Relation et personnages permettant des MVs sémantiques

Les relations et les personnages permettent une grande diversité de modèles conceptuels. Contrairement à de nombreuses approches de conception de MVs comme le VRML, où seule la relation de référentiel existe, les relations peuvent être de n'importe quelle nature. En généralisant les structures présentées dans l'état de l'art, les relations et les personnages permettent de créer des MVs avec une sémantique, en facilitant ainsi la gestion. Elles servent à décrire les règles d'interaction entre acteurs, similairement à l'approche utilisée dans VEOS. De plus, les relations permettent d'optimiser l'application des interactions entre les acteurs et ainsi sauver du temps de calcul. Elles peuvent même être ajoutées au MV en cours d'exécution. Leur conception s'avère délicate car elles appartiennent au premier groupe d'actions. Les relations ne supportent pas d'attributs, ce qui pourrait être une limitation pour certaines modélisations. Par exemple, la valeur de la force agissant dans un maillage ne pourrait être assignée à la relation entre ses noeuds.

6.6 Architecture à trois composantes

Alors que les autres architectures reposent sur des composantes graphiques, de scénarisation et de distribution, le noyau d'**APIA** définit des composantes jugées plus essentielles dans l'atteinte d'une grande *agilité* : un méta-modèle conceptuel, un processus et des algorithmes de gestion. Même si celles-ci améliorent l'*agilité* lorsque prises séparément, elle l'est davantage lorsque les trois composantes sont réunies.

6.7 Fine granularité et variété des éléments

Avec **APIA**, le scénariste peut définir plusieurs types d'éléments (propriété, personnage, relation, acteur et interaction) individuellement et avec une granularité fine. Cette approche facilite la composition ainsi que l'extension, la modification et la réutilisation de ces éléments. Comparativement à **APIA**, HLA ne permet, en général, que la réutilisation des fédérés en entiers. Bien qu'il soit possible de concevoir une simulation HLA qui aurait la granularité d'**APIA**, il n'existe aucune règle qui oblige une telle granularité. Quant aux autres architectures, comme l'OO, VEOS, DEVA 3, MAVERIK, MASSIVE ou VRML, elles

n'offrent pas un tel niveau de granularité. Certaines permettent l'ajout de comportements, d'autres l'ajout d'entités mais aucune ne permet tous ces éléments simultanément. Cependant, la granularité a son revers. Il peut être difficile de concevoir et de gérer un MV avec plusieurs éléments. Des outils de scénarisation, des algorithmes optimisés et une implantation misant sur la rapidité peuvent atténuer ces problèmes. Une brève investigation des outils de scénarisation, présentée à l'annexe A, a permis de mettre au point des scripts de scénarisation pour **APIA**, mais le développement d'outils et d'une méthode en vue de leur utilisation représente un effort de recherche qui dépasse les objectifs de cette thèse. Les algorithmes présentés à la section 4.3.1 permettent à **APIA**, malgré sa fine granularité, de fonctionner en conformité avec le temps réel lorsque qu'implanté tel que décrit à l'annexe A.

6.8 Conclusion sur l'analyse d'APIA

Ce chapitre a revue les différentes stratégies employées par **APIA** afin d'améliorer l'*agilité*. Il est possible d'en dégager des principes généraux dans l'atteinte de l'*agilité* :

- centrer la modélisation sur l'interaction et utiliser un mécanisme générique de la capacité d'interagir tel que personnage ;
- forcer la définition des règles de gestion de la cohérence par les scénaristes ;
- définir des algorithmes qui maintiennent la cohérence ;
- définir un processus composé de nombreuses actions avec des conséquences progressives sur l'*agilité* future ;
- baser le noyau de l'architecture sur un méta-modèle, un processus et des algorithmes de gestion ;
- définir un méta-modèle ;
- permettre des MVs avec une sémantique et l'exploiter ; et
- définir une variété d'éléments de fine granularité.

Tel que démontré précédemment par plusieurs exemples, cette *agilité* vient faciliter la conception et l'évolution de MVs ainsi qu'offrir une expérience riche et réaliste aux usagers.

Chapitre 7

Conclusion

7.1 Rappel de la problématique et des objectifs

Les domaines de la réalité virtuelle et de la simulation peuvent bénéficier de MVs moins coûteux et plus rapides à concevoir, à modifier ou à étendre. De plus, la qualité de l'expérience virtuelle souffre parfois de nombreuses lacunes due à une trop grande rigidité. Tel que présenté dans le chapitre 3, les architectures existantes ne permettent pas de concevoir des MVs *agiles*, c'est-à-dire qui favorisent la réutilisabilité, l'extensibilité, la modifiabilité et la composabilité en cours d'exécution ainsi que la liberté d'action.

L'objectif de cette thèse était de définir une architecture, désignée **APIA**, comprenant un méta-modèle conceptuel, un processus et des gestionnaires permettant d'augmenter l'*agilité* des MVs. Afin de démontrer l'*agilité* et la qualité générique de l'architecture proposée, cette thèse a présenté plusieurs exemples et plusieurs modes d'utilisation de MVs.

Les contributions de cette thèse se situent à deux niveaux. D'abord, la majeure partie des contributions sont de nature conceptuelle. Ensuite, le développement d'applications d'interventions de nature critique dans plusieurs modes constitue également une contribution notable. Les deux prochaines sections en effectuent la récapitulation. Ensuite, un rappel des difficultés et des limitations est effectué. Finalement les axes de recherche future sont présentés.

7.2 Contributions conceptuelles

La thèse amène les contributions conceptuelles suivantes :

- approche centrée sur l'interaction avec utilisation de règles ;
- algorithmes de gestion de la cohérence ;
- processus d'actions a différents niveaux de conséquences ;
- utilisation et définition d'un méta-modèle ;
- relation et personnages permettant des MVs sémantiques ;
- architecture basée sur un méta-modèle conceptuel, un processus et des gestionnaires ;
- fine granularité des éléments ; et
- caractéristiques de l'*agilité*.

Chacun des paragraphes suivants effectue un rappel.

La principale contribution de cette thèse est l'approche centrée sur l'interaction (voir sections 4.1.5 et 6.1) qui réunit les composantes fortement couplées d'un MV. Dans une approche centrée sur l'entité, ces dernières doivent définir un ensemble de protocoles pour interagir et chaque entité accomplit une partie de l'interaction, ce qui les rend fortement dépendantes les unes des autres. L'approche proposée regroupe ces protocoles autour de l'interaction et des personnages. Les acteurs et les interactions sont donc moins dépendants les uns des autres. Par conséquent, l'utilisation d'un grand nombre d'interactions et d'acteurs en est facilitée. Ces nombreuses interactions et acteurs peuvent favoriser l'apparition de comportements émergents (voir section 5.4). Ensuite, l'interaction peut contenir une large gamme de modèles, tant de types symétriques qu'asymétriques pour toute combinaison de multiplicité. De plus, les personnages (voir sections 4.1.3 et 6.1.2) empêchent la dépendance des interactions envers les acteurs, facilitant ainsi leur réutilisation. Finalement, les règles d'interaction (voir sections 4.1.5 et 6.1.1) permettent :

- la gestion de la cohérence présente avec la définition des conditions d'instanciation de l'interaction selon des règles relationnelles et de multiplicités ;
- la gestion de la cohérence temporelle et son automatisation avec la définition des conditions d'appel d'une interaction ou d'une instance d'interaction ;
- l'optimisation de l'application des interactions ; et

- l'encapsulation de nombreux types de modèles spécifiques avec les règles de multiplicités et relationnelles.

Les algorithmes des *gestionnaires de la cohérence présente* et *temporelle* (voir sections 4.3 et 6.2) s'assurent que le déroulement du MV est conforme aux modèles conceptuels et que le MV est cohérent, des qualités nécessaires dans un MV en perpétuelle modification. Ils agissent lorsqu'une action du processus est posée. Les gestionnaires relèvent les scénaristes de cette tâche.

La définition d'un processus (voir section 4.2) contraint la création et la modification des MVs selon des règles strictes. Effectuer des actions constitue en soi l'*agilité* (modifiabilité, extensibilité, composabilité, réutilisabilité, liberté d'action) mais influence également le niveau d'*agilité* future. Dans son processus, **APIA** propose donc des groupes d'actions dont les conséquences sur l'*agilité* future est progressive. Le processus agit comme intermédiaire entre les éléments du méta-modèle conceptuel et les algorithmes de gestion.

Une contribution importante de cette thèse repose sur l'utilisation d'un méta-modèle conceptuel (voir sections 4.1 et 6.4) pour encadrer la conception de MVs afin de garantir une certaine *agilité*. Il décrit le MV dans une structure précise, permettant aux gestionnaires de gérer les conséquences des actions qui supportent l'*agilité*. Cette approche pourrait être appliquée pour d'autres objectifs. La méta-solution est une contribution au même titre que la solution elle-même.

Les relations et les personnages permettent de donner une sémantique aux MVs (voir sections 4.2.3 et 6.5), facilitant ainsi la gestion des règles d'interaction. Il s'agit d'une généralisation des structures des MVs existants (voir section 3.2).

De plus, **APIA** se distingue des autres architectures grâce à l'intégration de trois composantes : un méta-modèle conceptuel, un processus et des algorithmes de gestion. Même si ces composantes individuelles améliorent l'*agilité*, elle est davantage améliorée lorsque les trois composantes sont réunies.

La fine granularité d'**APIA** permet la définition et l'instanciation des éléments de façon individuelle en cours d'exécution, comparativement aux autres architectures plus monolithiques et contraignantes (voir section 6.7).

Finalement, une autre contribution a trait à la classification de MVs *agiles* (voir chapitre 2) selon plusieurs caractéristiques : la composabilité, la réutilisabilité, la modifiabilité, l'extensibilité et la liberté d'action. Des sous-catégories ont été définies pour chacune de ces caractéristiques. Il s'agit d'une classification suffisante pour mieux cerner des caractéristiques souvent négligées des MVs, définir un vocabulaire et analyser l'*agilité* des architectures existantes.

7.3 Contribution applicatives

Cette thèse présente deux applications principales ayant servi à tester **APIA**. Elles constituent des contributions en elles-mêmes.

D'abord, **APIA** a permis d'investiguer l'utilisation de MVs, basés sur une architecture commune, pour les modes de planification, d'entraînement et d'assistance aux interventions de cryochirurgie. Ces MVs n'ont jamais servis dans un contexte réel parce que de nombreuses considérations éthiques restent à régler. Ils ont cependant été utilisés dans un contexte expérimental.

APIA a permis une investigation similaire pour l'application d'entraînement et d'assistance aux inspections de barrages à l'aide de sous-marins téléopérés. Le MV d'assistance a pu être testé dans un environnement réel et s'est montré pleinement opérationnel.

7.4 Difficultés et limitations

Il reste beaucoup à accomplir pour accroître l'*agilité*. APIA amène des pistes de solution mais de nombreux problèmes persistent. D'abord, **APIA** dépend de la bonne volonté des scénaristes qui doivent créer les éléments en considérant les utilisations futures, surtout au niveau des propriétés et des relations. Cette dépendance laisse place aux erreurs et à la négligence qui peuvent limiter l'*agilité* future. Ensuite, la fine granularité des éléments de base peut être difficile à gérer. En effet, en utilisant une approche plus granulaire que l'OO qui mise sur l'encapsulation, **APIA** ne bénéficie plus de certains avantages de l'OO qui cache la complexité en exposant une simple interface de méthodes. La conception dans **APIA** requiert de la minutie, une erreur dans le nom d'un personnage ou d'une propriété peut résulter en un MV différent de celui souhaité. Un outil de scénarisation graphique supportant le méta-modèle

conceptuel d'**APIA** faciliterait la gestion de la complexité apportée par la granularité. De plus, il pourrait être combiné à un processus de scénarisation plus complet afin d'imposer des contraintes de conception qui augmenteraient davantage l'*agilité*.

Une autre limitation d'**APIA** a trait à la formulation limitée des règles d'interaction. Le développement de plusieurs applications a montré que les règles pourraient être plus sophistiquées : conditions sur les propriétés, formulation avec langages scriptés, interactions instantanées. Il faudrait toutefois s'assurer que ces formulations ne réduisent pas trop la vitesse d'exécution.

Bien que l'*ordonnanceur* présenté dans cette thèse gère des MVs *agiles*, il a été conçu avec des fonctionnalités minimales. Il est peu adaptatif relativement à ce qui serait nécessaire pour ordonnancer des interactions capables de modifier les propriétés dans le futur. Actuellement, tous les changements de propriétés ont lieu à l'instant présent sauf pour quelques interactions prédéterminées. L'*ordonnanceur* reconnaît ces interactions et génère des événements avec un incrément de temps défini dans le modèle conceptuel. De plus, l'exécution parallèle et répartie de plusieurs instances d'interactions permettrait d'accroître la puissance de calcul et, ainsi, la taille des MVs. Des techniques récentes d'ordonnancement, telles que le *time warp* et la simulation optimiste (Fujimoto, 2000), pourraient être utilisées. Des mécanismes de retour en arrière pour tout le MV seraient alors requis.

Finalement, le paradigme de conception avec **APIA** diffère des autres approches, principalement dû à son approche centrée sur l'interaction et sa fine granularité. Ce changement de paradigme pourrait nuire à son acceptation. Le développement d'outils de scénarisation pourrait toutefois en minimiser l'impact.

7.5 Axes de recherche future

Ces travaux de doctorat ont permis d'identifier plusieurs pistes de recherche future. Premièrement, plusieurs améliorations pourraient être apportées à **APIA**. Les règles d'interaction pourraient être plus évoluées pour introduire des formulations plus complexes. L'*ordonnanceur* pourrait gérer les MVs *agiles* sans connaissance *a priori* et permettre l'exécution parallèle en mode optimiste dans une architecture non centralisée. Il en résulterait une plus grande *agilité* et une meilleure extensibilité de la puissance de calcul. Ce travail

impliquerait une modification du méta-modèle conceptuel afin de permettre l'ajout d'information sur l'exécution de l'interaction.

Un axe de recherche intéressant à explorer consisterait à définir un ensemble d'éléments comme les propriétés, les relations et les personnages pour un domaine d'application donné. Des règles de composition pourraient aussi être prédéfinies. L'ontologie résultante permettrait d'accroître l'*agilité* pour un domaine spécifique.

Les composantes de l'architecture constituent une approche générique à de nombreux défis des MVs et logiciels. Leur développement afin de répondre à un objectif différent de l'*agilité* serait une piste à investiguer.

Finalement, cette thèse a présenté une ébauche de processus de scénarisation avec, en annexe, quelques tentatives pour développer des outils de scénarisation. Il reste de nombreux de travaux à effectuer dans ce domaine, du point de vue méthodologique, architectural et sur le plan de l'interface graphique. Les générateurs automatiques de code à partir d'outils de scénarisation et de modélisation pourraient également améliorer la conception de MVs *agiles*.

Outre ces recherches potentielles, il reste de nombreuses voies à explorer dans le domaine des MVs. La simulation, la réalité virtuelle, le génie logiciel et autres domaines continueront d'y contribuer.

Bibliographie

1278 IEEE Standard. (1993). *IEEE Standard for information technology - protocoles for distributed simulation applications: Entity information and interaction.* IEEE Standard 1278-1993. New York: IEEE Computer Society.

Activeworlds. (2006). [En ligne] Disponible à http://www.activeworlds.com

Arnold, C. (2000). *A Approach for Semi-automated Liver Tumor Segmentation in MR Images.* Mémoire de maîtrise, Université Laval.

Arnold, K., Gosling, J., & Holmes, D. (2000). *The Java progamming language.* Reading, MA: Addison-Wesley.

Atkinson, C., & Kühne, T. (2001). The Essence of Multilevel Metamodeling. *Comptes rendus de la 4^{e} conférence internationale sur le Unified Modeling Language, Modeling Languages, Concepts, and Tools,* 9-33.

Basile-Bellavance, Y., & Harrison, N. (2000). *Conception et entraînement d'un réseau de neurones pour la prédiction de température autour d'une sonde cryogénique.* Rapport de stage en génie informatique, Université Laval.

Baudrillard, J. (1981). *Simulacre et simulation.* Éditions Gallimard.

Bernier, F., & Poussart, D. (2000). Architecture of a flexible virtual environment targeting physical simulation. Affiche, *Comptes rendus de la 10^{e} conférence annuelle IRIS/PRECARN.*

Bernier, F., Poussart, D., Laurendeau, D., & Simoneau-Drolet, M. (2002). Interaction-Centric Modelling for Interactive Virtual Worlds: the APIA Approach, *Comptes rendus de la 16^{e} conférence ICPR,* 1007-1010.

Bernier, F., Simoneau, M., & Poussart, D. (1999). Physical simulation engine for augmented reality applications. Affiche, *Comptes rendus de la 9e conférence annuelle IRIS/PRECARN.*

Bernstein, B., Hall, D. A., & Trent, H. M. (1958). On the Dynamics of a Bull Whip, *The Journal of the Acoustical Society of America,* 30 (12), 1112-1115.

Blaszczak, M. (1997). *Professional Mfc With Visual C++5.* Wrox Press. 1061 p.

Blau, B., Hugues, C.E., Moshell, J.M., & Lisle, C. (1992). Networked Virtual Environments. *Comptes rendus de la conférence ACM SIGGRAPH.* 157-160.

Blaxxun. (2006). [En ligne] Disponible à http://www.blaxxun.com

Boivin.E. (2002). *Environnement de développement logiciel destiné à la communication interactive de données virtuelles au monde réel.* Mémoire de maîtrise, Université Laval.

Borgeat, L. (2000). *Un outil de modélisation et de scénarisation pour les environnements de realite virtuelle.* Proposition de thèse de doctorat, Université Laval.

Bray, T., Paoli, J., & Aperberg-McQueenm C. M. (2006). [En ligne]. *Extensible Markup Language (XML). W3C Proposed Recommendation 1997.* Disponible à http://www.w3.org/TR/PR-wml-971208

Bricken, W., & Coco, G. (1994). The VEOS Project. *Presence: Teleoperators, and Virtual Environments*, 3(2), 111-129.

Buckham, B. Driscoll, F.R., & Nahon. M. (2004). Development of a Finite Element Cable Model for Use in Low-Tension Dynamics Simulation. *Journal of Applied Mechanics*. 71 (4), 476-485.

Burns, D., & Osfield, R. (2006). [En ligne]. *OpenSceneGraph.* Disponible à http://openscenegraph.sourceforge.net/index.html

Buschmann, F., Meunier, R., Rohnert, P., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System Of Patterns.* West Sussex, England: John Wiley & Sons Ltd.

Capps, M., Stotts, D., Duff, J., Purtilo, J. (1996). Distributed interoperable virtual environments. *Comptes rendus de la 3e conférence international Configurable Distributed Systems*, 202-209.

Capps, M., Watsen, K., & Zyda, M. (1999). M. Cyberspace and Mock Apple Pie: A Vision of the Future of Graphics and Virtual Environments, *IEEE Computer Graphics & Applications,* 19 (6), 8-11.

Card, O.S. (1985). *Ender's Game*. Tor Science Fiction.

Carey, R., Bell, G., & Marrin, C. (1997). The Virtual Reality Modeling Language ISO/IEC DIS 14772-1.

Carlsson, C., & Hagsand, O. (1993). DIVE- A Multi-User Virtual Reality System. *Comptes rendus du IEEE Virtual Reality Annual International Symposium,* 394-400.

Codella , C. F., Jalili, R., Koved, L., & Lewis, J.B. (1993). A Toolkit for Developing Multi-User, Distributed Virtual Environments. *Comptes rendus de la conférence VRAIS '93*, 401-407.

Das, T. K., Singh, G., Mitchell, A., Kumar, P.S., & McGee, K. (1997). NetEffect: A network architecture for large-scale multi-user virtual worlds. *Comptes rendus de la conference ACM Symposium on Virtual Reality Software and Technology (VRST 1997)*, 157-163.

Daubrenet, S., Pettifer, S., & West, A. (2000). Behaviours and Relationships: Providing Structure and Behaviour for Shared Virtual Environments. *Comptes rendus de la conférence UKVRSIG*, 117-126.

Davis, P.K., & Anderson, R.H. (2004). Improving the Composability of Department of Defense Models and Simulations. *Journal of Defense Modeling and Simulation*, 1 (1), 5-17.

DigitalSpace Traveler. (2006). [En ligne]. Disponible à http://www.digitalspace.com/traveler/

Dinel, F. (2005). *Framework logiciel pour le travail collaboratif en réalité augmentée: Application à la gestion de crises urbaines.* Mémoire de maîtrise, Université Laval.

DOOM. (2006). [En ligne]. Disponible à http://www.idsoftware.com/games/doom/doom3/

Drouin, R., Laurendeau, D., & Branzan-Albu, A. (2004). MONNET : Simulation of the inter-node communication strategy for a network of loosely-coupled nodes in a surveillance application, *Comptes rendus de la 14th Canadian Conference on Intelligent Systems*. 23.

Eddon, G., & Eddon, H. (1998). *Inside Distributed COM.* Redmond, WA: MicrosoftPress.

Everquest. (2006). [En ligne]. Disponible à http://everquest.station.sony.com/

Fabre, Y., Pitel, G., Soubrevilla, L., Marchand, E., Géraud T., & Demaille A. (2000). A Framework to Dynamically Manage Distributed Virtual Environments. *Comptes rendus de la deuxième International Conference on Virtual Worlds (VW'2000)*, 54-64.

Fayad, M.E., Johnson, R.E, & Schmidt, D.C. (1999). *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley & Sons.

Fishwick, P.A. (1995). *Simulation Model Design & Execution: Building Digital Worlds.* Upper Saddle River, NJ: Prentice-Hall.

Fishwick, P.A. (1996). Extending Object-Oriented Design for Physical Modeling. *ACM Transactions on Modeling and Computer Simulation.*

Flight Simulator. (2006). [En ligne]. Disponible à http://www.microsoft.com/games/flightsimulator/

Fricke, E., Schulz, A., Wenzel, S., Negele, H. (2000). Design for Changeability of Integrated Systems within a Hyper-Competitive Environment. *Comptes rendus de la conférence INCOSE Colorado 2000.*

Fujimoto, R.M. (2000). *Parallel and Distributed Simulation Systems.* New York: Wiley-Interscience.

Gagnon, M. (1999). *Agent de Copie*. Rapport de stage en génie informatique, Université Laval.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gibson, W. (1982). *Burning Chrome*. Ace Books.

Gibson, W. (1984). *Neuromancer*. Ace Books.

Green, M. (1994). *Object Modeling Language (OML)*, Version 1.1. Département d'informatique, Université de l'Alberta.

Greenhalgh, C.M., & Benford, S. (1995). MASSIVE: A distributed virtual reality system incorporating spatial trading. *Comptes rendus de la 15e International Conference on Distributed Computing Systems (DCS'95)*, IEEE Computer Society Press, 27-34.

Greenhalgh, C.M., & Benford, S.D. (1997) Boundaries, Awareness and Interaction in Collaborative Virtual Environments, *Comptes rendus du 6ième International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*.

Greenhalgh, C.M., Purbrick, J., & Snowdown, D. (2000). Inside MASSIVE-3: Flexible support for data consistency and world structuring. *Comptes rendus de la conférence Collaborative Virtual Environments 2000*, 119-127.

Halo. (2006). [En ligne]. Disponible à http://www.microsoft.com/games/halo/default.asp.

Harrison, N. (2003). *Simulateur de Cryochirurgie: Prédiction de la distribution de température*. Mémoire de maîtrise, Université Laval.

Harrison, N., Gilbert, B., Jeffrey, A., Lestage, R., Lauzon, M., & Morin, A. (2005) KARMA: Materializing the Soul of Technologies into Models. *Comptes rendus de la conférence I/ITSEC 2005*.

Harrison, N., Larose, F., Laurendeau, D., & Moisan, C. (2000). Thermal Mapping with a Neural Network Approach for the Planning and Conduct of MR Guided Cryosurgeries, *Comptes rendus de la 8e réunion scientifique sur la ISMRM*, 1351.

Heeter, C. (1992). Being there: The Subjective Experience. *Presence: Teleoperators, and Virtual Environments,* 1(2), 262–271.

Henning M., & Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.

HLA. (2006). High Level Architecture. [En ligne]. Disponible à https://www.dmso.mil/public/transition/hla/.

Hubbold, R., Cook, J., Keates, M., Gibson, S., Howard, T., Murta, A., West, A., & Pettifer, S. (2001). GNU/MAVERIK: A micro-kernel for large-scale virtual environments. *Presence: Teleopérators: and virtual environments.* 10(1), 22-34.

Jaramillo, J.L., Vangheluwe, H., & Moreno, M.A. (2003). Using meta-modelling and graph grammars to create modelling environments. Bulletin of the EATCS 81, 180-194.

Johnson, R., & Foote, B. (1988). Designing Reusable Classes. *Object-Oriented Programming*, 1, 22-35.

King Quest VIII. (2006). [En ligne]. Disponible à http://en.wikipedia.org/wiki/King's_Quest_VIII:_The_Mask_of_Eternity.

Levine, L. (2005). Reflections on Software Agility and Agile Methods: Challenges, Dilemmas and the Way Ahead. *Comptes rendus de la International Working Conference, Business Agility and Information Technology Diffusion.* 353-365.

Loo, P.L. (1991). *The Starship Manual (Version 2.0)*, ISS TR#91-54-0. Institute of Systems Science, National University of Singapore, Kent Ridge, Singapore 0511.

Macedonia, M. R., Zyda, M. J., Pratt, D.R., Barham, P.T., Zeswitz, S. (1994). Npsnet: A Network Software Architecture For Large Scale Virtual Environments. *Presence: Teleoperators, and Virtual Environments*, 3(4), 265-287.

Martel, H. (2005). *Simulation et environnements virtuels : la détection de collisions entre objets à géométrie déformable.* Thèse de doctorat, Université Laval.

Mikhajlov, L., & Sekerinski, E. (1998). Study of The Fragile Base Class Problem. *Comptes rendus de 12th Conference on Object-Oriented Programming*. 355-382.

Milgram, P., & Kishino, F. (1994). A taxonomy of mixed reality visual display. *IEICE Transactions on Information and Systems*, E77-D(12):1321-1329.

Miller, D, Pope, C., Arthur, C., & Waters, R. M. (1989). Long-Haul Networking of Simulators. *Comptes rendus de 12th Interservice/Industry Training Systems Conference.* 2.

Object Management Group. (1998). *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed.

Opdyke, W. F., & Johnson, R. J. (1990). Refactoring: An Aid in Designing Application Frameworks. *Comptes rendus du symposium on Object-Oriented Programming Emphasizing Practical Applications*. 145-160.

Open Community. (2006). [En ligne]. Disponible à http://www.merl.com/projects/opencom/WWW/.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley. 480 p.

Paul, R.J., & Taylor, S.J.E. (2002). What Use is Model Reuse: Is There a Crook at the End of the Rainbow? *Comptes rendus de 2002 Winter Simulation Conference*, 648-652.

Pettifer, S., Cook, J., Marsh, J., & West, A. (2000). DEVA3: Architecture for a large-scale distributed virtual reality system. *Comptes rendus de ACM Conference on Virtual Reality Software and Technology (VRST 2000).* 33-40.

Petty, M.D., & Weisel, E.W. (2003). A Composability Lexicon. *Comptes rendus du* Spring 2003 Simulation Interoperability Workshop. 181-187.

Pidd, M. (2002). Simulation Software and Model Reuse: A Polemic. *Comptes rendus de la 2002 Winter Simulation Conference*. 772-775.

Poussart, D. (2006). *Complexity: An Overview of its Nature and Manifestations, and of S&T Convergence, with Comments on their Relevance to Canadian Defence*. Rapport à DRDC, Contrat W7701-014245.

Poussart, D., Laurendeau, D.,Bernier, F., Simoneau-Drolet, M, Harrison, N, Ouellet, D., & Moisan, C. (2000). Designing Virtual Environments for Critical Transactions and Collaborative Interventions: the Vertex Framework for Networked, Physics-Compliant Objects, *Comptes rendus de la conférence Advances in Infrastructure for Electronic Business, Science, and Education on the Internet.*

Pratt, D. R., Ragusa, L. C., & von der Lippe, S. (1999). Composability as an Architecture Driver. *Comptes rendus de la conférence Interservice/Industry Training, Simulation and Education Conference 1999.*

Reese, R., & Wyatt, D. L. (1987), Software reuse and simulation. *Comptes rendus de la 19e conférence* Winter Simulation. 185-192.

Renderware. (2006). [En ligne]. Disponible à http://www.renderware.com/.

Rohlf, J., & Helman, J. (1994). IRIS Performer: A HignPerformance Multiprocessing Toolkit for Real-Time 3D Graphics. *Comptes rendus de la conférence SIGGRAPH'94.*

Rymaszewski, M., James, A. W., Wallace, M., Winters, C., Ondrejka, C., Batstone-Cunningham, B., Rosedale, P. (2006). *Second Life : The officiel Guide.* Sybex, 342 p.

Saleh, J. H., Hastings, D. E., & Newman, D. J. (2001). Extracting The Essence Of Flexibility In System Design. *Comptes rendus de la conférence 3e NASA/DoD Workshop on Evolvable Hardware*. 59-73.

Schmidt, D.C., Gokhale, A., Harrison, T.H., & Parulkar, G. (1997a). A High-performance Endsystem Architecture for Real-time CORBA , *IEEE Communications Magazine*, 12 (2).

Schmidt, D.C., & Hudson, S.D. (2001). *C++ Network Programming, Vol. 1: Mastering Complexity with ACE and Patterns,* Addison-Wesley Professional. 336 p.

Schmidt, D.C., Levine, D., & Harrison, T. (1997b). The Design and Performance of a Real-time CORBA Object Event Service. *Comptes rendus de la conférence OOPSLA '97.*

Schmidt, D.C., Wang, N., & Vinoski, S. (1999). Collocation Optimizations for CORBA. *C++ Report*, 11 (9).

Schroeder, W.J., Martin, K.M., & Lorensen, W.E. (1996). The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. *Comptes rendus de la conférence IEEE Visualization '96.*

Schwartz, J.M. (2003). *Calcul rapide de forces et de déformations mécaniques non-linéaires et visco-élastiques pour la simulation de chirurgie*. Thèse de doctorat, Université Laval.

Shaw, C., & Green, M. (1993). The MR Toolkit Peers Package and Experiment. *Comptes rendus de la conférence IEEE Virtual Reality Annual International Symposium.* 18-22.

Singh, G., Serra, L., Png, W., Wong, A., & Ng, H. (1995). BrickNet: Sharing Object Behaviors on the Net. *Comptes rendus de la conférence IEEE Virtual Reality Annual International Symposium (VRAIS'95).* 19-25.

Smith, R. (2006). [En ligne]. Disponible à http://www.ode.org/.

Soto, M., & Allongue, S. (2002). Modeling Methods for Reusable and Interoperable Virtual Entities in Multimedia Virtual Worlds. *Multimedia Tools Appl.*, 16(1-2), 161-177.

Stefik, M., & Bobrow, D.G. (1985). Object-Oriented Programming: Themes and Variations. *AI Magazine.* 6(4): 40-62.

SUN. (2006). *Java Remote Method Invocation (Jave RMI).* Disponible à http://java.sun.com/products/jdk/rmi/.

Szyperski, S. (2002). *Component Software: Beyond Object Oriented Programming.* Addison-Wesley Professional.

Tardif, A. (1999). *CORBAGen.* Rapport de stage en génie informatique, Université Laval.

Ultima Online. (2006). [En Ligne] Disponible à http://www.uo.com/.

US DoD. (1995a). *A Glossary of Modeling and Simulation Terms for Distributed Interactive Simulation (DIS).* Washington.

US DoD. (1995b). *Modeling and Simulation Master Plan.*

Vaghi, I, Greenhalgh, Benford, S. (1999). Coping with inconsistency due to network delays in collaborative virtual environments. *Comptes rendus de la conférence Virtual Reality Software and Technology.* 42-49.

Viskit. (2000). [En ligne] Disponible à http://www.viskit.com.

W3C. (2006). [En ligne] *XML Schema.* Disponible à http://www.w3.org/XML/Schema.

Wang, Q., Green, M., & Shaw, C. (1995) EM - An Environment Manager For Building Networked Virtual Environments. *Comptes rendus de la conférence VRAIS 95*, 11-18.

Warcraft III. (2006). [En ligne]. Disponible à http://www.blizzard.com/war3/.

Waters, R., Anderson, D., Barrus, J., Brogan, D., Casey, M., McKeown, S., Nitta, T., Sterns, I., & Yerazunis, W. (1996). *Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability.* Rapport technique TR-96-02a. MERL.

Watsen, K., & Zyda, M. (1998). Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments, *Comptes rendus de la conference Virtual Reality Annual International Symposium (VRAIS'98)*, 139-146.

Wemecke. J. (1994). *The Inventor Mentor*. Addison-Wesley.

WIKI MMOG. (2006). [En ligne]. Disponible à http://en.wikipedia.org/wiki/Massively_multiplayer_online_game.

Winston, M. E., Chaffin, R., & Herrmann, D. (1987). A taxonomy of part-whole relations, *Cognitive Science*, 11, 417-444.

XIOP. (2006). [En ligne] Disponible à http://xml.coverpages.org/xiop.html.

Zeigler, B., Praehofer, H., & Kim, T.G. (2000). *Theory of Modeling and Simulation.* Second edition. London: Academic Press.

Zyda, M., & Darken, R.P. (1998). *Projects in VR: the Naval Postgraduate School's Moves curriculum*, IEEE Computer Graphics and Applications, 18 (3), 8 -11.

Annexe A

Implantation d'APIA

Les prochaines sections présentent l'implantation d'**APIA** dans un *framework* réparti. Le *framework* global est d'abord introduit. Les sections subséquences présentent ses modules, soit le contrôleur, les senseurs, les effecteurs, le noyau et les noeuds de calcul.

A.1 *Framework* global

Un *framework* se définit (Fayad *et al.*, 1999) comme une conception réutilisable d'un système qui décrit comment ce système est décomposé en un ensemble d'objets interagissant. Il consiste en un ensemble de classes coopérant entre elles pour produire un cadre réutilisable permettant de résoudre une gamme de problèmes similaires. Le *framework* définit l'architecture de l'application : sa structure générale, sa répartition en classes et en objets, les responsabilités de chacune des classes ainsi que la boucle principale d'exécution. Le *framework* prend les décisions de conception et l'usager doit le paramétriser ou l'étendre pour concevoir une application. La conception d'un MV selon une architecture comprenant un méta-modèle conceptuel et un processus est possible avec un *framework* car il permet d'encadrer le scénariste. L'encadrement peut prendre la forme d'héritage, de gestion d'événements bien spécifiques ou d'envoi de messages, incluant les appels de méthodes à un ou à des éléments prédéfinis.

Puisque les *frameworks* visent l'économie de temps et d'argent, les décisions de conception d'un *framework* héritent des qualités fondamentales du génie logiciel comme la réutilisation et la modularité tout en fournissant une solution générique à une gamme de problèmes. Ceci explique pourquoi les *frameworks* permettent généralement d'augmenter la

modularité, la réutilisabilité et l'extensibilité. Les *frameworks* utilisent souvent une approche par composantes pour atteindre ce but et doivent donc gérer l'intégration de composantes dans une même application. Les *Microsoft Foundation Classes*, ou MFC (Blaszczak, 1997), par exemple, définissent comme standard d'intégration la liste des événements qui peuvent être utilisés par les vues, cadres, gadgets logiciels (*widgets*) ou documents ainsi que les classes de base desquelles il faut dériver. Plusieurs programmeurs peuvent alors concevoir différentes vues qui seront ensuite intégrées et pourront interagir ensemble, sans même que celles-ci n'aient à s'entendre au préalable entre elles. Par exemple, le glisser déposer (*drag & drop*) définit un standard d'interaction entre deux éléments graphiques.

L'approche par *framework* a été retenue pour implanter **APIA**. Le processus s'est déroulé en deux itérations. La première implante tous les éléments à l'exception des personnages. Elle a servi à concevoir une version initiale des applications de cryochirurgie et d'inspection de barrages. Le *gestionnaire de la cohérence présente* limitait les modifications en cours d'exécution. La seconde version poursuit l'implantation de l'architecture mais délaisse la répartition d'autres fonctionalités afin de faciliter la démonstration de l'*agilité*. Elle a servi à l'implantation de l'exemple du MV avec un câble, de la seconde version des applications de cryochirurgie et d'inspection de barrages ainsi que des autres applications détaillées dans le chapitre 5. La figure A.1 illustre ce *framework*.

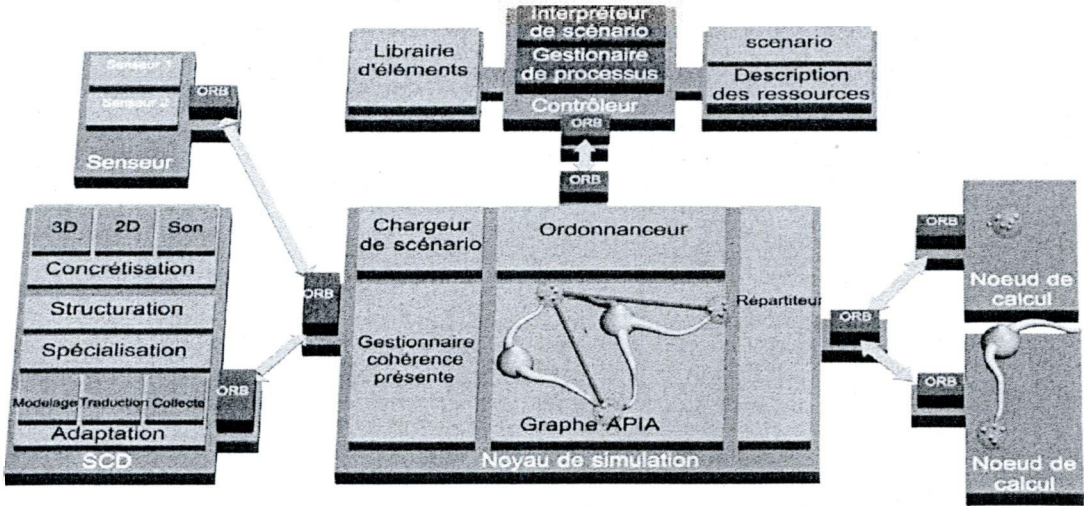

Figure A.1 : *Framework* **APIA** comprenant un noyau de simulation, un système de communication de données (SCD), des senseurs, un contrôleur et des noeuds de calcul.

La figure A.2 illustre le diagramme de paquetages (*packages*) de premier niveau d'**APIA**. Il est déjà possible de voir que le *framework* est basé sur la modularité.

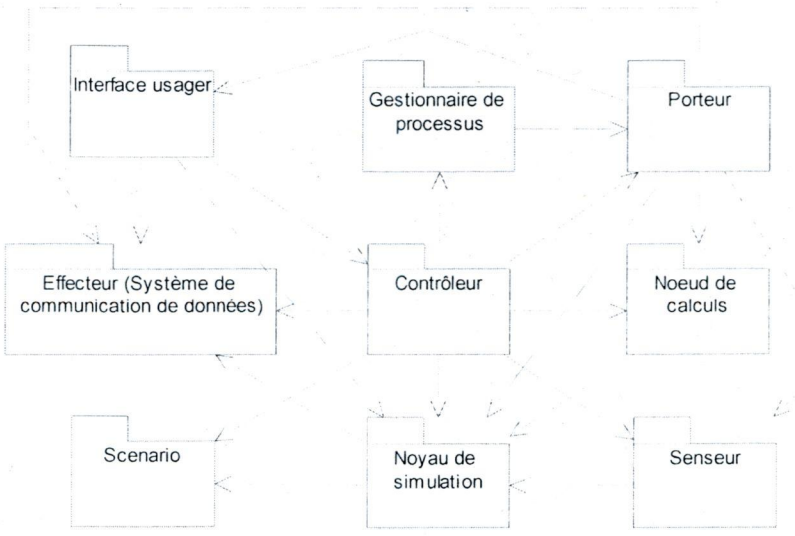

Figure A.2 : Diagramme UML montrant les paquetages du *framework* **APIA**.

Le *framework* est constitué de 20 librairies dynamiques et chaque scénario repose sur une dizaine de librairies dynamiques en moyenne. Cette modularité facilite la répartition et permet de composer un MV sur mesure à chaque application. La modularité s'établit à deux niveaux. Le premier niveau, présenté dans la section A.2, comprend les modules d'**APIA** : noeud de calcul, effecteur, senseur et noyau de simulation et contrôleur. Elle constitue le macro *framework*. Par le choix du nombre et du type de modules, **APIA** permet de supporter une diversité d'applications. Le deuxième niveau, présenté à la section A.5, constitue le micro *framework*, c'est-à-dire le noyau qui contient les gestionnaires et les éléments. Sa modularité permet de supporter l'*agilité* d'**APIA**.

Le développement du *framework* a suivi les bonnes pratiques du génie logiciel basées sur le standard UML et la génération automatique de code. Il est programmé en langage C++ parce qu'il permet une plus grande rapidité d'exécution ainsi qu'une plus grande flexibilité de programmation et parce que de nombreux modèles physiques et librairies du domaine de la réalité virtuelle sont écrits dans ce langage. La première version du *framework* repose sur 300 classes alors que la deuxième version en utilise 100.

Le *framework* repose sur l'Adaptive Communication Environment, ou ACE (Schmidt & Hudson, 2001), une architecture OO en C++ qui encapsule les principales fonctionnalités du système d'exploitation à travers un ensemble d'interfaces et de patrons de conception orientés communications et exécution. ACE fournit une couche d'abstraction pour la gestion de la concurrence, les synchronisations, les communications inter-processus, les mémoires partagées, les mécanismes de démultiplexage d'événements, l'édition de liens dynamique et les mécanismes d'accès aux fichiers. ACE fonctionne sur plus de vingt systèmes d'exploitation, ce qui permet à **APIA** de fonctionner sur Linux, Windows NT, 2000, XP32 et XP64.

Pour les communications réseau, le framework repose sur The ACE ORB (TAO) (Schmidt *et al.*, 1997a). TAO consiste en une implantation temps réel gratuite et en sources ouvertes de la spécification CORBA, développée à l'Université de Washington à Saint-Louis. CORBA est un protocole de communication retenu pour concevoir l'architecture. Il existe depuis 1991 et il est l'un des nombreux intergiciels sur le marché. Tout comme le *Distributed Common Object Model*, ou DCOM (Eddon & Eddon, 1998) et Java RMI (SUN, 2006), CORBA définit son propre protocole de communication, une définition d'interface quelque peu différente du standard Interface Definition Language (IDL) ainsi qu'un ensemble de services.

Les besoins en performance et en temps réel de certaines applications devant être implantées dans cette thèse nécessitent une implantation CORBA performante et déterministe. Le choix de CORBA comme intergiciel au lieu de DCOM et de JAVA RMI a été principalement influencé par TAO car il est rapide et déterministe. De plus, sa disponibilité en sources ouvertes permet l'ajout ou la modification de certaines parties en collaboration avec l'équipe de développement. TAO offre de nombreux avantages par rapport aux autres implantations CORBA. Par exemple, il supporte une fine granularité des serveurs, ce qui permet de les implanter dans une librairie dynamique similairement à l'approche DCOM. De plus, TAO optimise les appels intra-processus ce qui permet de ramener le temps d'appel entre objets CORBA à une vitesse comparable au C++ lorsque ceux-ci sont situés dans le même processus.

Les prochaines sections présentent les modules d'**APIA** avec les décisions d'implantation qui supportent l'*agilité* recherchée.

A.2 Contrôleur, gestionnaire de processus et porteur

Le contrôleur consiste en un point de contrôle unique de la simulation. Le contrôleur accomplit deux fonctions : il réparti les modules et les éléments en plus d'interpréter le scénario. La figure A.3 décrit les relations entre les diverses composantes dans une représentation UML.

Le contrôleur obéit au principe de contrôle-modèle-vue (Buschmann *et al.*, 1996). Par conséquent, son interface graphique est séparée. Cette interface est multiple, illustré par le lien A à la figure A.3, répartie et transitoire, permettant un contrôle et une visualisation des éléments logiciels de la simulation par plusieurs individus. Le contrôleur est persistant et prend la forme d'une librairie dynamique pouvant être instanciée dans un porteur (*surrogate*), dans un service Windows (ou *deamon* sur Unix) ou avec l'interface de contrôle. Le contrôleur est contenu dans un processus persistant qui communique avec les gestionnaires de processus, les porteurs et les modules instanciés par ceux-ci. Un porteur est un patron de conception employé, entre autres, dans la technologie DCOM afin de contenir une DLL dans un processus. *Dllhost.exe* joue ce rôle pour plusieurs applications dans Windows. Il peut exister au maximum un contrôleur (lien B) et un gestionnaire de processus (lien C) par ordinateur. Le contrôleur déploie les modules (lien F), à travers les gestionnaires de processus (lien D) et les porteurs (lien E) sur plusieurs ordinateurs sur le réseau pour en exploiter la puissance ou pour envoyer l'information aux usagers de ces ordinateurs.

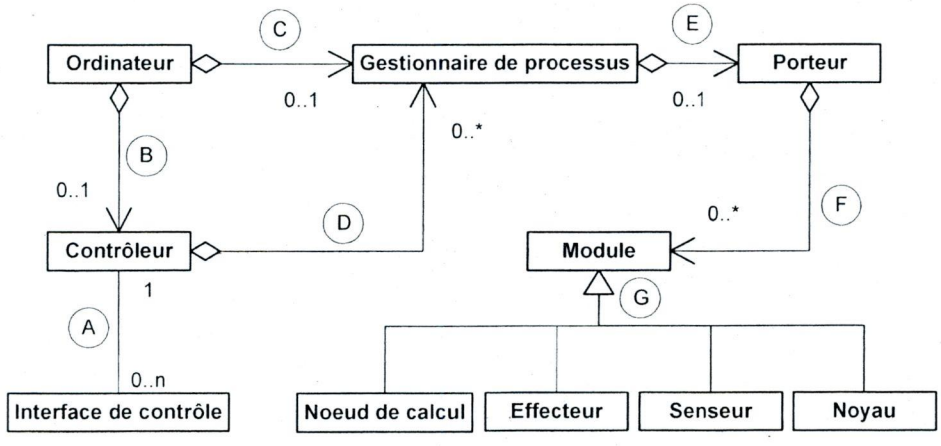

Figure A.3 : Diagramme UML des relations entre le contrôleur et les différentes composantes avec lesquelles il interagit.

Le MV démarre lorsqu'un usager, à partir de l'interface usager du contrôleur, envoie la commande au contrôleur de charger un scénario, tel qu'illustré à l'étape 1 de la figure A.4. Le contrôleur analyse alors le scénario qui demande la création de modules sur certains ordinateurs. Le contrôleur envoie une commande de création (étape 2) de processus au gestionnaire qui lance un porteur (étape 3). Ce dernier s'enregistre alors au contrôleur (étape 4). À partir de ce moment, un gardien (*watchdog*) s'assure en permanence que le porteur est fonctionnel grâce à des appels répétitifs (étape 5). Le contrôleur peut maintenant demander l'instanciation du module par un appel au porteur (étape 6). Le module s'enregistre alors au contrôleur (étape 7).

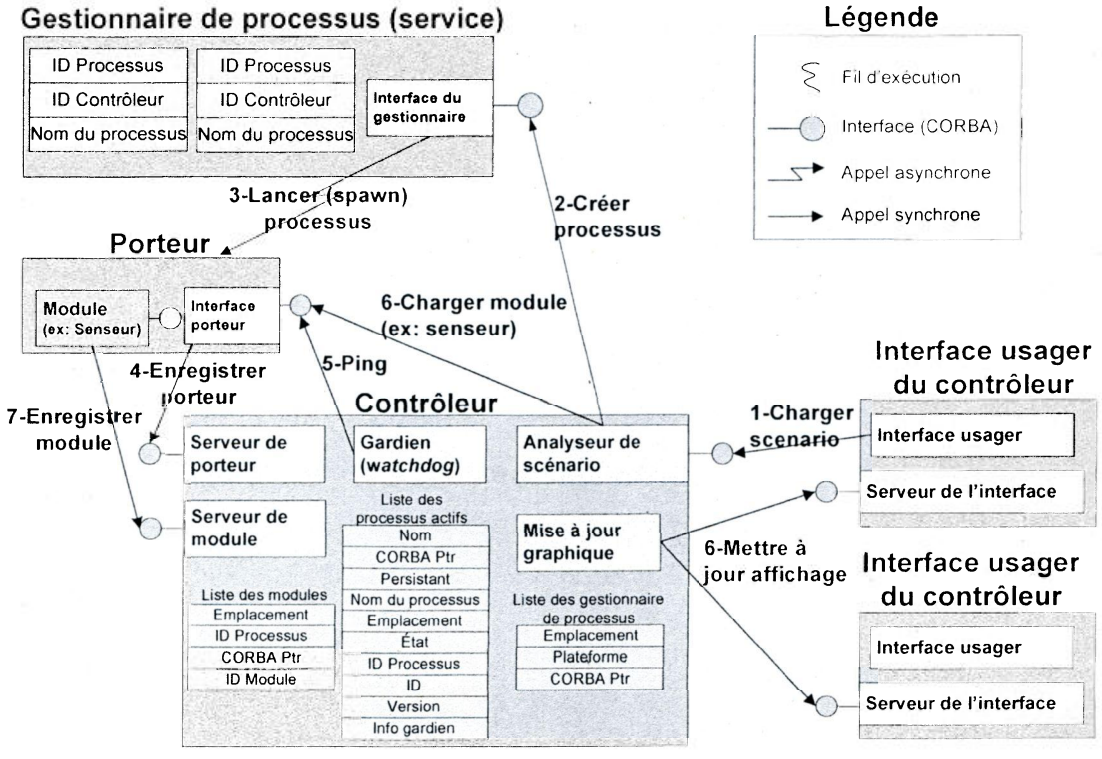

Figure A.4 : Processus de création d'un module à partir d'une commande de l'interface.

Bien que le processus semble complexe, il permet aux modules sous format de librairies partagées ou dynamiques d'être instanciés sur plusieurs ordinateurs et dans plusieurs processus. L'ensemble des modules requis pour un MV peuvent être instanciés dans un même processus afin de simplifier l'exécution, diminuer les latences et faciliter le déverminage. À l'opposé, les modules peuvent tous résider sur un ordinateur différent afin d'accroître la

puissance de calcul et de rejoindre les usagers. Les applications implantées avec ce *framework* et présentées au chapitre 7 exploitent cette capacité d'**APIA**. Finalement, un fichier de configuration définit le nombre et l'emplacement de chacun des modules.

A.3 Scénario et données génériques

Un *framework* de MV repose, entre autres, sur un langage de scénarisation, un environnement de scénarisation et un analyseur de scénario. Le langage de scénarisation développé pour ce *framework* est basé sur le méta-modèle conceptuel avec des ajouts liés à l'implantation et à la répartition. Les scénarios peuvent être décrits dans une série de fichiers XML (Bray *et al.*, 2006) ou en langage de programmation C++ et compilés dans une librairie dynamique. La figure A.5 illustre ces deux formats de scénario à chaque extrémité. L'approche C++ peut être utilisée seule alors que celle XML doit utiliser une interface C++, qui réside dans le noyau, afin d'interpréter le scénario.

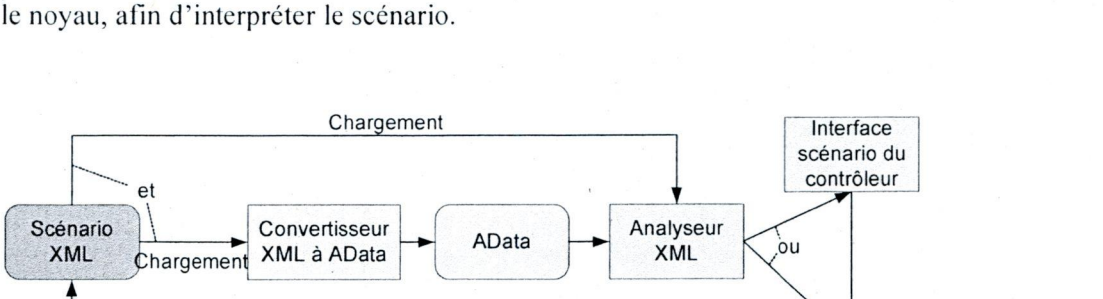

Figure A.5 : Diagramme de flot de données du chargement et de la sauvegarde en format XML ou dans une classe contenue dans une librairie dynamique.

A.3.1 Approche C++

L'écriture du scénario suit l'ordre du processus décrit dans la section 4.2. La version C++ implante, par les constructeurs de classes ou de méthodes, l'ensemble des actions du processus de la section 4.2. Par exemple, la classe acteur possède une méthode "int Actor::AnchorInteraction (Interaction interactiontoAnchor)". L'approche C++ permet de répartir le scénario dans une ou plusieurs librairies dynamiques pouvant êtres chargées en cours d'exécution. Cette interface est offerte dans un ensemble de classes qui repose sur le patron de

conception des pointeurs intelligents (*smart pointers*). Les méthodes et classes sont accessibles dans tout le *framework*, mais elles sont généralement employées à partir de la classe *Scenario* et de la classe *Interaction* (voir section A.5). L'interaction *Vérifier inclusion topologique*, décrite dans le section 5.4.1, est un exemple d'interaction qui utilise l'interface C++ pour modifier le MV en cours d'exécution en ajoutant et en retirant des relations entre deux instances de personnage *Volumétrique* jusqu'à une centaine de fois par seconde sur un processu Pentium M 2Ghz..

L'approche C++ s'avère rapide et facile d'utilisation pour de petits scénarios mais devient rapidement difficile à gérer pour un grand nombre d'éléments. À titre indicatif, l'exemple complet du MV avec un câble de la section 5.4 contient 350 lignes de code pour l'ensemble du scénario et 500 lignes de code pour l'implantation des interactions. Un environnement de scénarisation serait utile même à cette taille de MV. La version C++ ne permet toutefois pas de sauvegarder le scénario à un instant précis. L'approche XML corrige cette lacune.

A.3.2 Approche XML

Une autre approche, basée sur le XML, permet la définition des éléments dans plusieurs fichiers schéma (W3C, 2006) et XML. Les fichiers d'acteurs contiennent les instances de propriétés et l'assignation de personnages. Un autre fichier XML, nommé scénario, instancie les acteurs et effectue l'ancrage des interactions. L'ensemble des fichiers définit le scénario mais le MV est démarré par le chargement de ce dernier qui, ensuite, charge les autres fichiers. Il représente donc le scénario du point de vue de l'usager. Ce fichier XML est lu par l'analyseur de scénario XML qui utilise le contrôleur ou l'interface C++ de scénarisation du noyau de simulation. Lors du chargement du scénario, l'analyseur consulte les fichiers de définition des acteurs instanciés. Ceux-ci contiennent la valeur de chaque instance de propriété. À ce moment, les fichiers XML de définition d'acteurs sont envoyés au convertisseur XML à AData car les valeurs des instances de propriétés sont définies à partir des AData.

Les AData définissent un ensemble de classes qui recréent les types de données simples en C++ (entier, flottant, booléen, etc.) et des types complexes (structures et séquences), le tout interfacé avec des pointeurs intelligents. Ces données permettent l'ajout et la définition de

types de données personnalisés sans recompiler ou redémarrer le MV. Les AData permettent également au *framework* de gérer les instances de propriétés. Par exemple, lorsqu'une instance d'interaction change la valeur d'une instance de propriété, une variable "mise à jour" du AData est changée, facilitant le suivi des modifications des instances de propriétés et, conséquemment, l'appel des instances d'interactions.

Les données AData sont convertibles en format XML (voir AData à XML dans la figure A.5) ou en format *Any*. Le format XML des AData permet de sauvegarder ou de charger les instances de propriétés. Il est inspiré du XIOP (2006), aujourd'hui disparu. Le format *Any* permet de les transmettre par réseau avec CORBA ou de les sauvegarder en binaire avec les *Common Data Representation* (Henning & Vinoski, 1999). Ce formatm *Any* de conversion de paramètres en format binaire pour la transmission de données sur le réseau. Ce même format sert à la sauvegarde dans des fichiers. Le concept de données en XML pour la partie fichier et en AData pour la partie mémoire a été conçu pour être employé dans des applications autres qu'**APIA**. Il a pu ainsi être réutilisé dans l'architecture de simulation KARMA (Harrison *et al.*, 2005).

La figure A.6 illustre les différents fichiers de scénario et la relation entre les définitions XML des éléments. Les fichiers schéma contiennent la définition des types de données, des propriétés et des personnages en faisant référence les uns aux autres. Les vérifications au niveau des normes XML à l'aide de schémas servent à valider les références des AData dans les propriétés ainsi que les valeurs des personnages, des propriétés et des AData. Une propriété ne pourrait pas, par exemple, utiliser un type de données non défini. Le vérificateur de la librairie XML valide les fichiers lors du chargement. Par exemple, un fichier schéma personnage serait rejeté par le vérificateur de la librairie XML s'il y avait une erreur dans l'utilisation d'une propriété ou si celle-ci n'était pas définie dans un autre fichier schéma. À l'exception des propriétés, les éléments se retrouvent dans des fichiers XML. L'analyseur XML d'**APIA** doit alors se charger de la validation.

Le MV se sauvegarde à n'importe quel moment de son exécution en langage XML, ce qui a été tenté avec succès avec des scénarios de test. Le lecteur aura sans doute remarqué qu'il n'existe pas de fichiers relation dans l'approche XML. Son ajout au *framework* serait néanmoins simple.

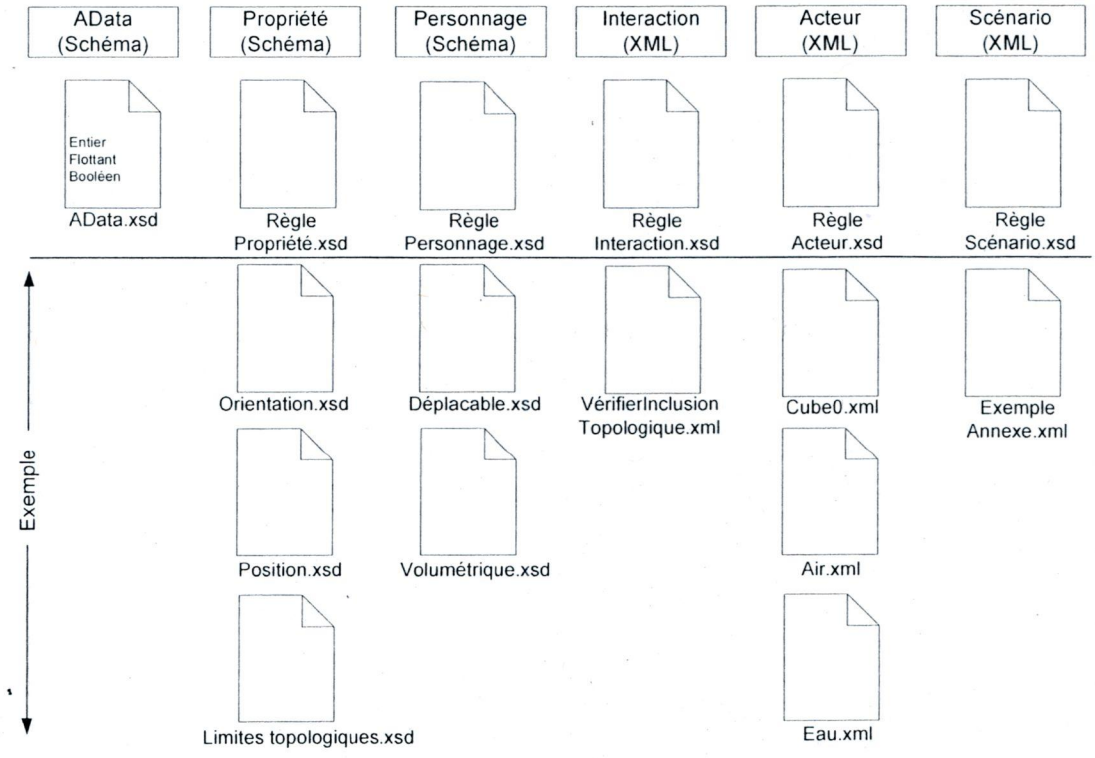

Figure A.6 : Définition des types de données, des propriétés et des personnages dans des fichiers schéma XML et des interactions dans des fichiers XML.

A.3.3 Environnement de scénarisation

Il n'existe pas d'environnement de scénarisation propre à **APIA**. En effet, le développement d'un tel environnement est problématique car les modifications fréquentes au méta-modèle conceptuel dues au contexte de recherche impliquent des changements au format du scénario et, conséquemment, à l'environnement de scénarisation. Il doit donc s'adapter à l'évolution de deux parties dont il dépend. Sa maintenance s'avère alors difficile. Les scénarios **APIA** s'écrivent donc à partir de simples éditeurs de fichiers XML et C++. L'écriture de plusieurs scénarios et des prototypes d'environnement de scénarisation en Visio, en C++ pour la génération des éléments (Tardif, 1999), en 3D Studio Max et en UML (Borgeat, 2000) ont toutefois révélé son importance.

La granularité et le grand nombre de types d'éléments dans **APIA** accroissent la difficulté et les sources d'erreurs lors de l'écriture des scénarios. De plus, ces erreurs sont

difficilement détectables dans un éditeur de texte. Il n'est pas surprenant de rencontrer ces problèmes que résout l'approche OO. En tentant d'éviter le manque d'*agilité* résultant de l'OO, **APIA** perd également certains avantages de celui-ci. Les *Visual Builders* (Fayad *et al.*, 1999), intégrés à un environnement de scénarisation, permettraient de combler cette lacune en développant des applications à l'aide d'outils graphiques qui facilitent le travail. L'expérience acquise durant cette thèse a révélé que la capacité de réutilisation des éléments et la facilité de conception d'un MV reposent probablement autant sur un bon environnement de scénarisation que sur une architecture favorisant l'*agilité*.

A.4 Senseur et effecteur

Les senseurs et effecteurs échangent de l'information entre les mondes réel et virtuel. Dans **APIA**, les senseurs communiquent l'information du monde réel vers le MV. Ils font la lecture d'un senseur physique (souris, manche à balai, etc.) afin d'en transmettre la valeur à travers le canal d'événements TAO sous format *Any* jusqu'à une propriété d'où elle sera utilisée par une instance d'interaction. Les senseurs jouent le rôle des *Suppliers* et les instances de propriétés *Senseur* jouent le rôle des *Consumers* de la figure A.7. Les mécanismes de filtrage et d'aiguillage sont utilisés pour transmettre les données entre les senseurs et les acteurs.

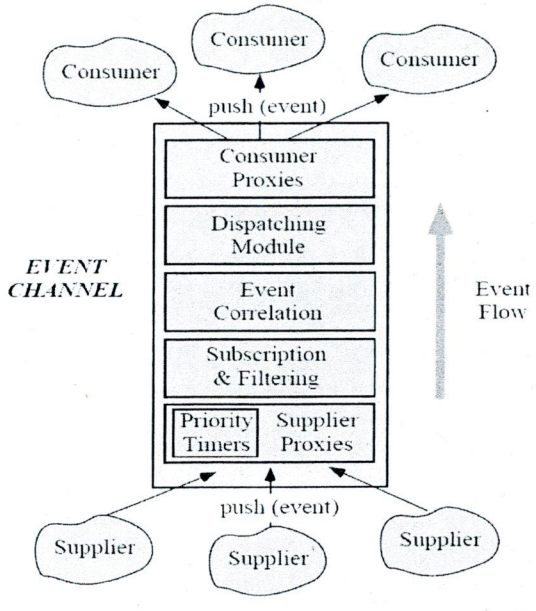

Figure A.7 : Architecture du canal d'événements temps réel de TAO (Schmidt et al.,1997b).

Par conséquent, un grand nombre de types de données peut être envoyé vers le MV à condition que l'interaction connaisse le type en question. Une interaction conçue après ce senseur n'aura donc pas ce problème, ce qui concorde avec le processus **APIA**. La création d'un senseur repose sur la dérivation d'une classe *Senseur* en implantant les méthodes virtuelles pures. Cette classe fournit un fil d'exécution et le concepteur du senseur est responsable d'appeler la méthode *SendEvent* périodiquement. Cette méthode acheminera, de façon transparente et selon les instructions du scénario, l'information dans la bonne instance de propriété, et donc au bon acteur, dans le noyau. Les senseurs peuvent être assignés de plusieurs façons dans le noyau. Dans la première, le senseur est assigné à un objet quelconque, comme un manche à balai est assigné à un sous-marin afin de l'animer. Dans la seconde, un senseur peut être assigné à son équivalent virtuel, comme le senseur d'image de l'appareil d'IRM est assigné à un générateur d'IRM dans le MV. Dans la troisième façon, le senseur est assigné à son inverse, comme un micro réel est assigné à un haut-parleur virtuel. Les différentes alternatives d'assignation montrent bien que le *framework*, quoique flexible et générique, pourrait être amélioré. Pour ce faire, l'amélioration de l'architecture reposerait sur au moins deux principes : mieux intégrer les senseurs et effecteurs au modèle conceptuel pour en tenir compte dans le méta-modèle conceptuel d'**APIA** et permettre à un usager du MV d'interagir avec le monde réel de façon transparente.

Chaque envoi est marqué dans le temps afin d'assurer le bon ordre à l'arrivée. Grâce au mécanisme des porteurs, les senseurs peuvent être répartis géographiquement. Ils peuvent être localisés dans un processus séparé ou avec d'autres modules tels que l'interface graphique et/ou le noyau. Le Spacepuck$^{MD\ 1}$, les capteurs Fastrack dont le Stylus$^{MD\ 2}$ pour la cryosonde, les capteurs de position d'un sous-marin, le simulateur d'images RM, le manche à balai, la souris et le clavier sont autant d'exemples de senseurs déjà implantés et supportés dans *le framework*.

Les effecteurs effectuent l'opération inverse des senseurs. Ils prennent l'information du MV, donc la valeur des instances de propriétés, pour la transmettre vers le monde réel. Deux effecteurs ont été développés dans **APIA**. Le premier est un effecteur qui permet de contrôler

1. Le Spacepuck^{MD} est une sorte de souris à six degrés de liberté.
2. Le Stylus^{MD} est un capteur à six degrés de liberté de la forme d'un crayon qui utilise la mesure du champ magnétique en courant alternatif. Il est fabriqué par Polhemus.

un sous-marin à partir du MV (voir section 5.6). Le deuxième rend visuellement ou de façon sonore les données du MV. Tel que mentionné dans le chapitre 3, de nombreux MVs sont basés sur les graphes de scène avec tous les problèmes qui en découlent. **APIA** sépare les données et la gestion du MV de l'engin de rendu. Cette approche est également employée dans le domaine de la simulation. Dans HLA, par exemple, le rendu est accompli par un fédéré. L'engin d'affichage (Bernier *et al.*, 1999 ; Poussart *et al.*, 2000) et l'agent de copie (Gagnon, 1999) jouaient ce rôle à l'origine, mais manquaient de flexibilité et n'étaient pas suffisamment génériques. Le système de communication de données, ou SCD (Boivin, 2002) permet de combler les lacunes de l'engin de rendu. Le SCD, illustré à la figure A.8, rend donc l'engin de rendu désuet. Ce SCD est suffisamment générique et flexible pour supporter l'*agilité* des MVs basés sur **APIA**.

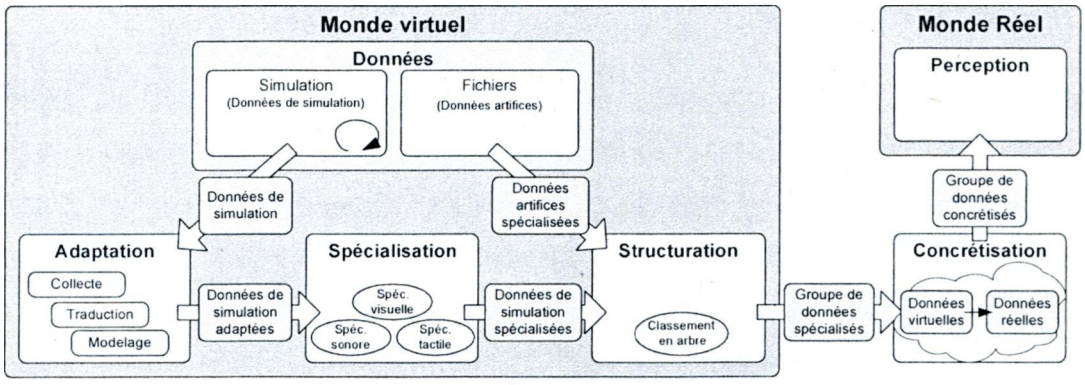

Figure A.8 : SCD transmettant à l'usager de façon sensorielle les données du MV selon des règles flexibles.

Le SCD prend les données du MV, les traite et les assigne à des éléments visuels, sonores ou autres. Le SCD développé dans le cadre du projet VERTEX est générique ; il peut communiquer un grand nombre de types de données de plusieurs façons (texture 3D, géométrie 3D, graphique 2D, sonore, etc.) pouvant servir dans de nombreuses applications (inspection de barrages, cryochirurgie, etc.) avec un minimum de reconfiguration et de paramétrage. La flexibilité du SCD est essentielle car des MVs comme ceux de l'exemple de la section 5.4 requièrent une grande capacité d'adaptation de l'affichage en cours d'exécution. En effet, la librairie de rendu ne peut savoir à l'avance ce qui devra être rendu dans des MVs *agiles*.

Le SCD repose sur un processus de quatre étapes qui transmet les données du MV à un

usager sous une forme qui peut être perçue. Les données passeront donc par une adaptation, une spécialisation, une structuration et une concrétisation. Même si trois des quatre étapes appartiennent au domaine du MV, elles ne sont pas localisées avec le noyau de simulation mais plutôt avec l'interface graphique pour des raisons d'efficacité. L'adaptation consiste à prendre une donnée fournie dans le format du MV pour l'amener au SCD, à la traduire dans un format compréhensible au SCD et à la modeler. Le SCD utilise les AData comme format de données. Ensuite, grâce à la spécialisation, une donnée sera assignée à un type visuel, sonore ou autre. La donnée est ensuite structurée. Dans le cas du 3D, la structuration peut prendre la forme d'un graphe de scène. Finalement, les données sont concrétisées, par exemple, en appelant le rendu d'une caméra 3D qui produit un résultat visuel 3D sur un écran.

Le SCD a considérablement facilité le développement d'interfaces graphiques et du rendu 3D des applications conçues avec **APIA**. Tout au long de son évolution et des projets dans lesquels il a été utilisé, le SCD a supporté les librairies de rendu WorldToolKit, Viskit (2000), VisualizationToolkit (Schroeder *et al.*, 1996), Renderware (2006), Performer (Rohlf & Helman, 1994) et OpenSceneGraph (Burns & Osfield, 2006). Plusieurs causes expliquent l'utilisation de nombreuses librairies de rendu tout au long du développement du framework : les compagnies ou les produits tels WorldToolKit et Viskit disparaissent, les produits deviennent désuets par rapport aux compétiteurs ou encore, une librairie peut être adéquate pour une application mais non pour une autre. Toutes ces causes viennent renforcer la nécessité d'un SCD générique. La section A.7 montre des exemples de sorties du SCD.

A.5 Noyau

Le noyau, illustré à la figure A.9, implante en C++ les éléments et les gestionnaires tels que décrits au chapitre 4. Cette représentation du noyau montre simultanément les deux versions du *framework*. Dans la première version, le *gestionnaire de la cohérence présente* est minimal, les personnages n'existent pas, les instances de propriétés et d'interactions sont implantées en objets CORBA pour permettre leur répartition et le répartiteur peut appeler simultanément plusieurs instances d'interactions. La seconde version accroît l'*agilité* par des améliorations au *gestionnaire de la cohérence présente* et l'ajout des personnages, mais retire la capacité de répartition ainsi que l'exécution parallèle des instances d'interactions.

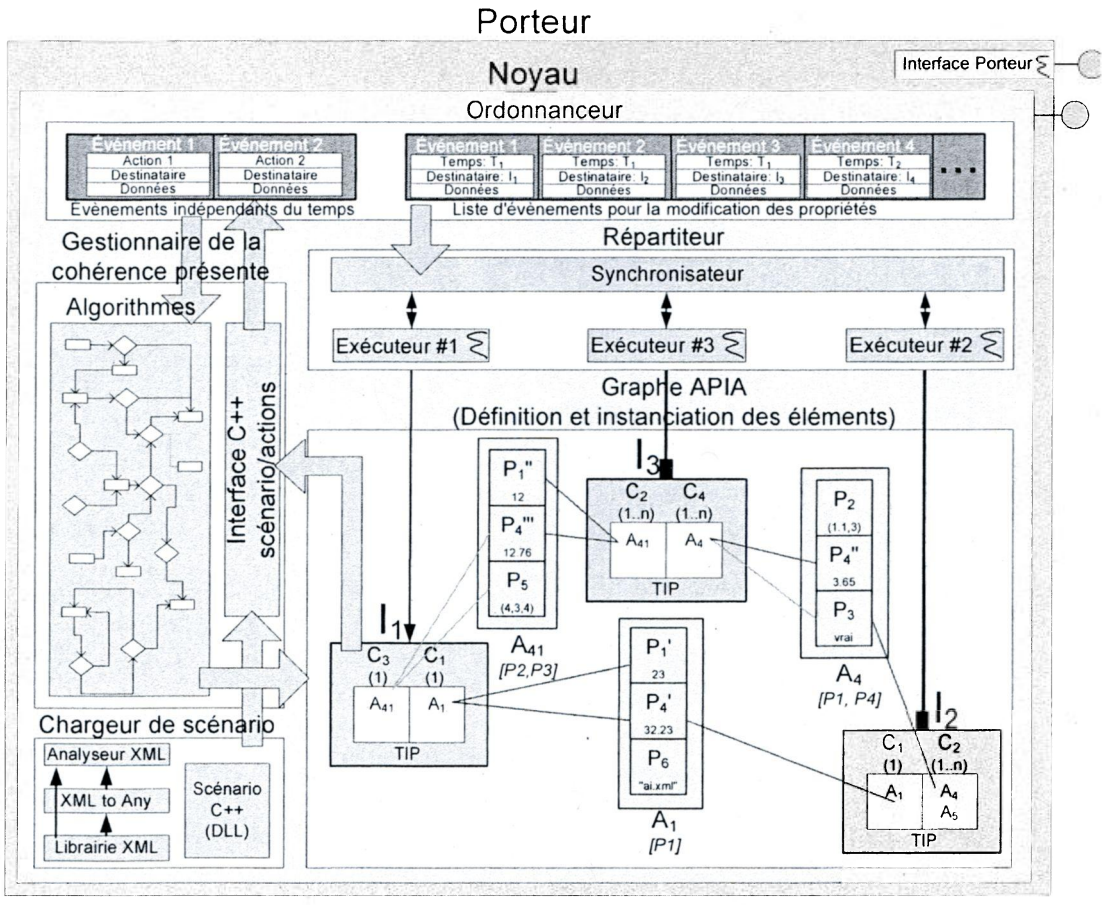

Figure A.9 : Noyau d'**APIA** avec ses trois gestionnaires, les éléments de base et un chargeur optionnel de scénario.

Le *gestionnaire de la cohérence temporelle* est implanté par l'*ordonnanceur*, celui de la cohérence présente par le gestionnaire du même nom ainsi qu'une partie de l'*ordonnanceur* et le gestionnaire de répartition par le **répartiteur**. Le reste du noyau implante la définition et l'instanciation des éléments dans un graphe unique dont les noeuds sont des éléments et des instances d'éléments. Ce graphe utilise plusieurs mécanismes, listés ci-dessous, afin de supporter l'approche granulaire et dynamique d'**APIA**.

- Tous les éléments et leur instance sont implantés dans des classes séparées. Les éléments peuvent être définis et instanciés séparément en cours d'exécution.
- Les interactions sont implantées dans des librairies dynamiques permettant leur chargement en cours d'exécution.

- Les instances de propriétés utilisent les AData, et les objets CORBA placés dans des librairies dynamiques dans la version répartie, afin de permettre leur définition en cours d'exécution et leur modification par les instances d'interactions.

Les instances de propriétés reposent sur les AData qui sont instanciés par le chargeur de scénario C++ ou les interactions. L'implantation d'une interaction repose sur l'héritage de la classe *Interaction* et sur la surcharge de certaines méthodes. Le tout est compilé dans une librairie dynamique pour être chargé en cours d'exécution. Chaque librairie contient une fabrique d'objets (*object factory*) (Gamma *et al.*, 1995) qui, à la demande du noyau, instancie l'interaction spécifique héritant de la classe *Interaction*. Grâce au polymorphisme et au *late-binbing*, une interaction peut être créée, compilée et appelée sans recompilation ni redémarrage du noyau. Le *late-binding* est un mécanisme de liaison par lequel le C++ implante le comportement polymorphique des objets qui ne sont pas encore chargés en mémoire. Les porteurs utilisent la même approche avec les modules. La figure A.10 illustre un exemple d'implantation d'une interaction et d'utilisation des instances de propriétés.

Le noyau offre l'ensemble des actions du processus et éléments décrits dans la section 4.2 par le biais des classes et des méthodes. Ces dernières sont incluses dans l'interface C++ scénario/actions du *gestionnaire de la cohérence présente* de la figure A.9. L'interaction et le chargeur de scénario accèdent à **APIA** à travers une interface de pointeurs intelligents qui rend disponible, de façon sécuritaire, l'ensemble des actions du processus. Chaque action est analysée avec un ou des algorithmes décrits dans la section 4.3 qui modifie le graphe **APIA** en conséquence.

Le noyau constitue un *framework* en lui-même et peut être employé sans les autres modules. Dans ce cas, l'inversion de contrôle disparaît et il devient une librairie. La section A.6 traite exclusivement de la répartition du MV à travers les noeuds de calcul.

```
class TopologicalInclusionDetection : public APIA::Kernel::InteractionInstance
{
public:
  virtual long CharacterInstanceInitialization ();
  virtual long CharacterInstanceTermination ();
  virtual long PropertyModified (std::vector<APIA::Kernel::PropertyInstance>
                                  propInstanceList);

  Data::Map<Data::String,Data::Vector3> topologicalLimitProperty;
  Data::Vector3 position;
  Data::Vector3 coinA;
  Data::Vector3 coinB;

  APIA::Kernel::CharacterInstance movable;
  APIA::Kernel::CharacterInstance includedVolumetricBody;
  std::vector<APIA::Kernel::CharacterInstance> allVolumetricBody;
};

long TopologicalInclusionDetection::CharacterInstanceInitialization()
{
  allVolumetricBody = GetCharacterInstanceList(2);
  includedVolumetricBody = GetCharacterInstance(1);
  movable = GetCharacterInstance(0);
  topologicalLimitProperty = (Data::Map<Data::String,Data::Vector3>)
  includedVolumetricBody.GetData("TopologicalLimitsProperty");
  coinA = (Data::Vector3) topologicalLimitProperty[Data::String("coinA")];
  coinB = (Data::Vector3) topologicalLimitProperty[Data::String("coinB")];
  Data::Vector3 position = (Data::Vector3) movable.GetData("PositionProperty");
  return 0;
}

long TopologicalInclusionDetection::CharacterInstanceTermination() { return 0; }

long TopologicalInclusionDetection::CharacterInstanceAddedRemoved (std::vector
<CharacterInstance> removedCharacterInstanceList, std::vector<CharacterInstance>
addedCharacterInstanceList)
{
  allVolumetricBody = GetCharacterInstanceList(2);
  return 0;
}

long TopologicalInclusionDetection::PropertyModified (std::vector<PropertyInstance>
                                                       propertyInstanceList)
{
  if ((position[0] < coinA[0]) || (position[0] > coinB[0])
    || (position[1] < coinA[1]) || (position[1] > coinB[1])
    || (position[2] < coinA[2]) || (position[2] > coinB[2]))
  {
    Actor movableActor = movable.GetActor();
    for (unsigned int i=0; i< allVolumetricBody.size(); ++i)
    {
      topologicalLimitProperty = (Data::Map<Data::String,Data::Vector3>)
                          allVolumetricBody[i].GetData("TopologicalLimitsProperty");
      coinA = (Data::Vector3) topologicalLimitProperty[Data::String("coinA")];
      coinB = (Data::Vector3) topologicalLimitProperty[Data::String("coinB")];
      if ((position[0] > coinA[0]) && (position[0] < coinB[0])
        || (position[1] > coinA[1]) && (position[1] < coinB[1])
        || (position[2] > coinA[2]) && (position[2] < coinB[2]))
      {
        movableActor.LinktoActor(topologicalInclusion,allVolumetricBody[i].GetActor());
      }
    }
    movableActor.UnLinkfromActor(topologicalInclusion,includedVolumetricBody.GetActor())
  }
  return 0;
}
```

Figure A.10 : Implantation en C++ de l'interaction *Vérifier inclusion topologique* présentée dans la section 5.4.1.

A.6 Noeud de calcul

Lors de l'exécution d'un MV, le *gestionnaire de la cohérence présente* et les instances d'interactions utilisent la majeure partie du temps de calcul du processeur. En conséquence, pour accroître la puissance de calcul, les instances d'interactions doivent s'exécuter sur d'autres processeurs. La méthode de répartition employée laisse une copie presque intégrale du MV sur un processeur central. Pour les interactions instanciées sur d'autres processeurs, il existe une version locale de l'instance de l'interaction désignée souche (*stub*). Elle joue le rôle de substitut chargé de transmettre l'appel ou le message à la vraie version, l'instance d'interaction, sur un ordinateur distant. L'*ordonnanceur* appelle le répartiteur qui, à son tour, appelle la souche. Cette souche transmet l'appel à l'instance d'interaction répartie. L'appel d'une instance d'interaction est toujours effectué de l'*ordonnanceur* à partir du processeur central. Ce parallélisme dans l'appel des instances d'interactions est rendu possible avec le répartiteur. L'*ordonnanceur* maintient une liste d'événements pour les modifications de propriétés. Tous les événements étiquetés avec un temps identique sont appelés simultanément, tâche dont se charge le répartiteur. La figure A.11 illustre une liste de quatre événements, dont trois sont étiquetés avec un temps T_1 et un autre avec un temps T_2.

Ces événements sont destinés à des instances d'interactions différentes. Par conséquent, l'*ordonnanceur* demandera au répartiteur d'effectuer l'appel des instances d'interactions un à trois simultanément. Le répartiteur appellera les souches qui, à leur tour, appelleront les instances d'interactions. Le rôle du répartiteur pour cette requête de l'*ordonnanceur* sera terminé lorsque toutes les instances d'interactions auront complété leur tâche. Le répartiteur sert donc à synchroniser l'appel des instances d'interactions.

Idéalement, **APIA** utiliserait une approche décentralisée pour répartir le MV. Elle permettrait une plus grande stabilité car la disparition d'un noeud ne causerait pas d'arrêt. De plus, la puissance de calcul serait plus extensible. L'implantation décentralisée est toutefois complexe, surtout pour des MVs *agiles*. Une approche alternative, la répartition centralisée, a toutefois été retenue afin d'exploiter la puissance de calcul de plusieurs ordinateurs pour les applications de cryochirurgie et d'inspection de barrages.

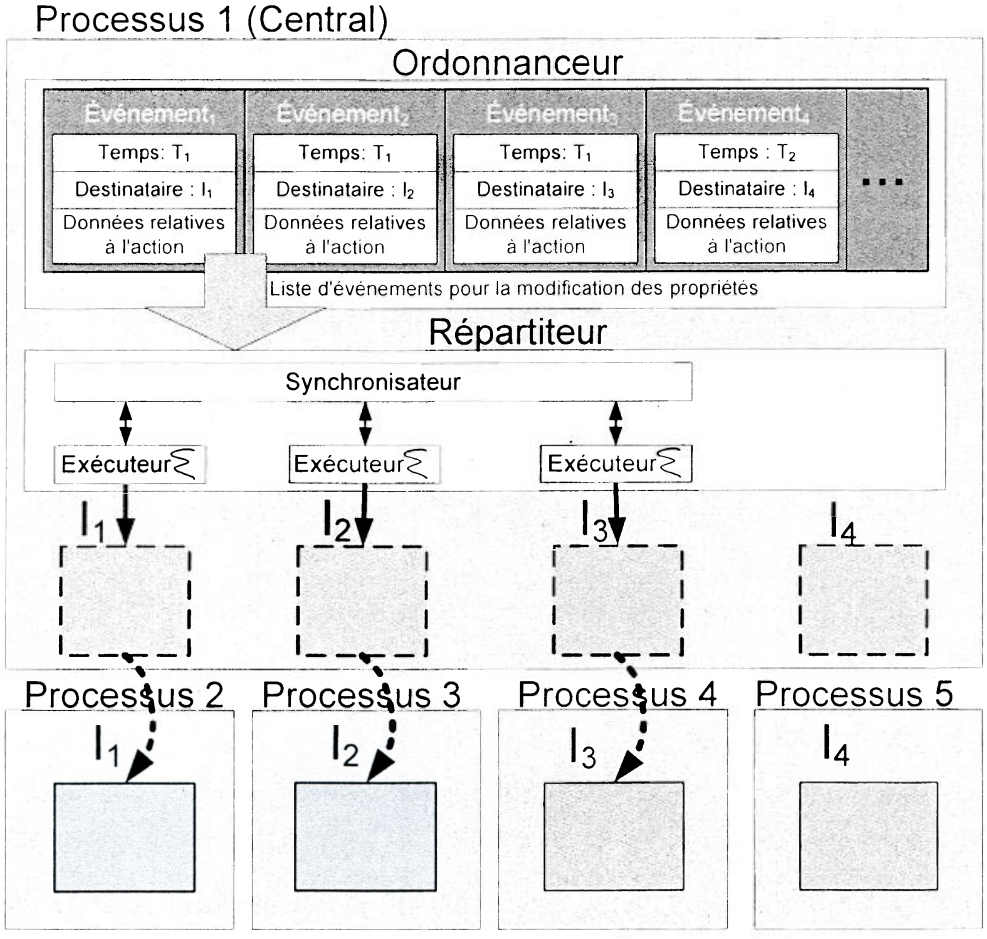

Figure A.11 : Exemple de l'exécution parallèle de trois interactions.

Ces événements sont destinés à des instances d'interactions différentes. Par conséquent, l'*ordonnanceur* demandera au répartiteur d'effectuer les appels simultanés des instances d'interactions un à trois. Le répartiteur appellera les souches qui, à leur tour, appelleront les instances d'interactions. Le répartiteur synchronisera ces appels avec un patron *Barrier*.

La figure A.12 présente l'implantation de l'approche de répartition basée sur la spécification CORBA. Le noyau contient un *ordonnanceur* responsable d'appeler le répartiteur qui, à son tour, assigne la tâche d'appeler une instance d'interaction à chacun de ses exécuteurs. Chaque exécuteur possède son propre fil d'exécution qui lui permet d'appeler séparément une instance d'interaction implantée en objet CORBA (appels 1 et 2 dans la figure A.12). Chaque

appel est bloquant. Dans la théorie des communications, un appel est bloquant lorsqu'il est *Connection Oriented Network Protocol* (CONP) tel que le *Transmition Control Protocol / Internet Protocol* (TCP/IP) en opposition au mode datagramme (*connectionless*) tel que le *User Datagram Protocol* (UDP). La méthode retourne lorsque la réception et le traitement du message sont confirmés. L'utilisation de plusieurs fils d'exécution permet d'appeler simultanément plusieurs instances d'interactions même si chaque appel est bloquant. L'exécuteur est l'appelant et l'instance d'interaction est l'appelée. L'instance d'interaction peut alors à son tour appeler les instances de propriétés, illustré par les appels 3, 4, 5 et 6 à la figure A.12, également implantées en objet CORBA. L'instance d'interaction devient ensuite l'appelant et l'instance de propriété, l'appelée. L'instance d'interaction joue ce double rôle. Ces appels s'effectuent à travers les objets répartis CORBA à l'aide de la souche (*stub)* et du squelette (*skeleton)*. La souche reproduit l'interface de l'objet afin d'empaqueter la requête pour, ensuite, la transmettre au *Object Request Broker* (ORB) qui transmet la requête au squelette. Ce dernier peut être localisé dans le même processus, dans un autre processus sur le même ordinateur ou sur un autre ordinateur. Les scénarios contiennent un champ permettant de définir l'emplacement des instances d'interactions et de propriétés par un numéro de noeud de calcul. Pour les souches et squelettes co-localisés, des optimisations spéciales de TAO rendent ces appels presque aussi rapides qu'en C++ (Schmidt *et al.*, 1999), ce qui est loin d'être le cas à travers le réseau. Le choix de ces emplacements a des répercussions sur la performance d'exécution du MV dont il faudra tenir compte.

Chaque interaction, implantée sous forme d'objets CORBA, se retrouve dans une librairie dynamique permettant son chargement en cours d'exécution. Toutes les instances de propriétés utilisent le même objet CORBA avec des Any convertibles en AData comme paramètre des méthodes *SetValue* et *GetValue*.

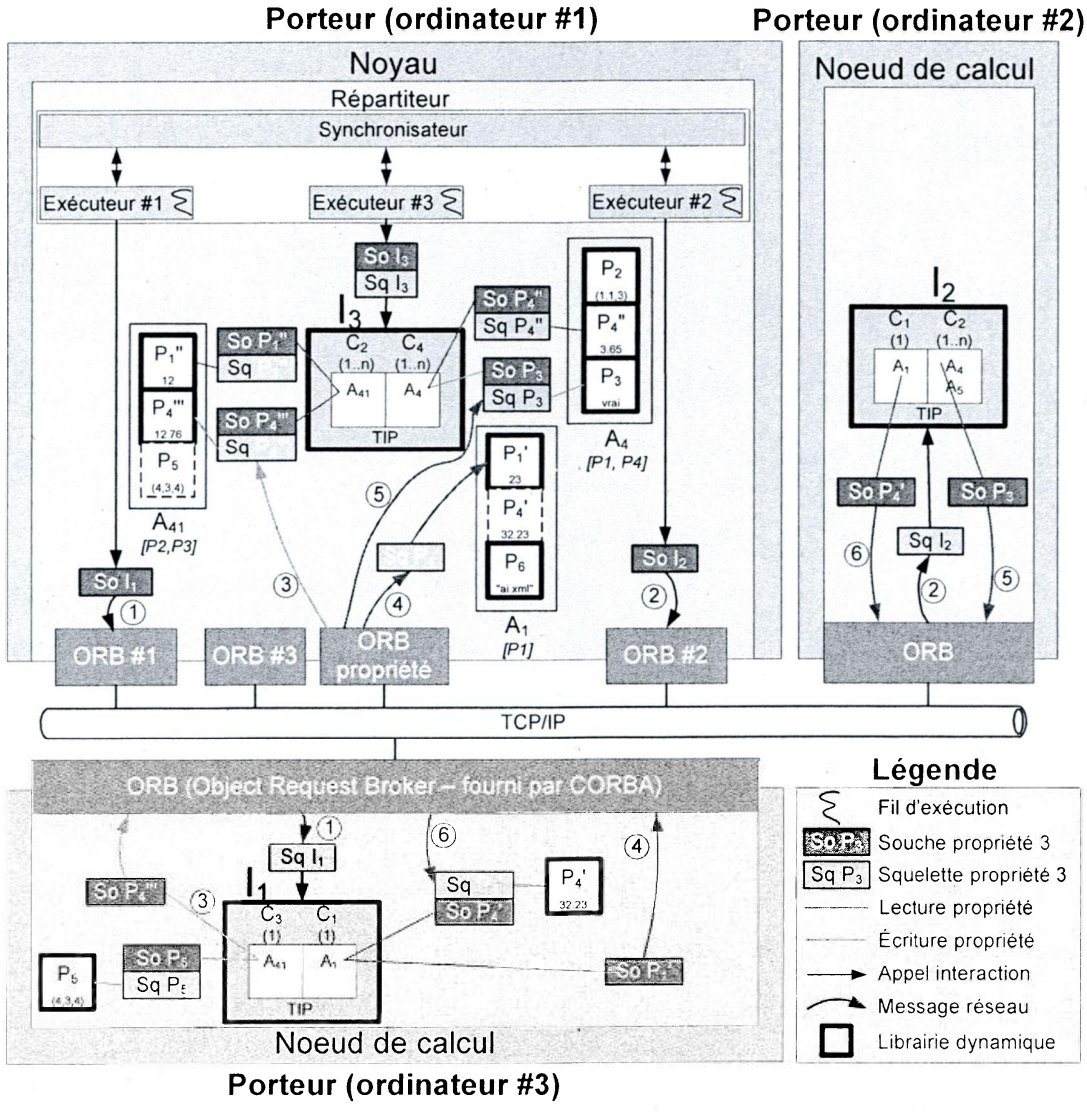

Figure A.12 : Implantation de la répartition de la charge de calcul avec le noyau et les noeuds de calcul.

A.7 Interface

La première version de l'interface **APIA**, illustrée à la figure A.13, contient l'interface du contrôleur, la sortie du SCD, certains senseurs et une fenêtre de déverminage.

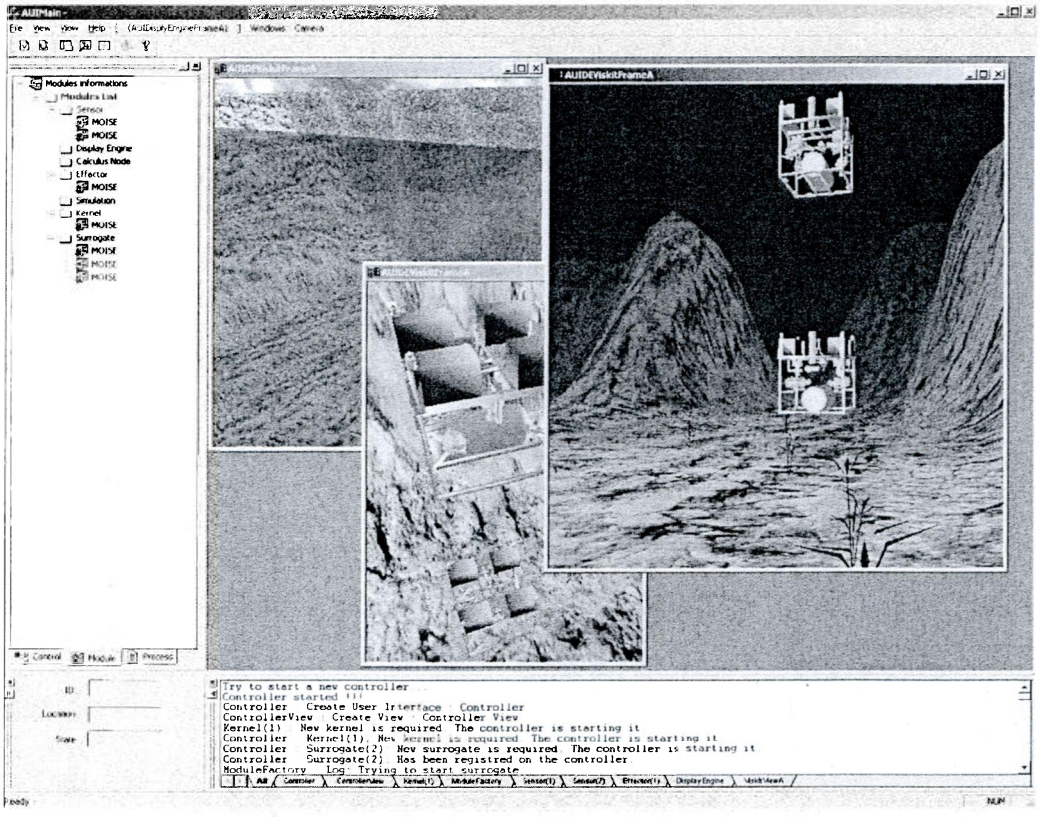

Figure A.13 : Première version de l'interface graphique d'**APIA** comprenant une interface au contrôleur, une sortie visuelle du SDC, certains senseurs et une fenêtre de déverminage.

Cette interface combine les travaux effectués dans le cadre du SCD et de l'interface de contrôle réalisée dans le cadre de cette thèse. La seconde interface, illustrée à la figure A.14, met l'accent sur la visualisation des éléments du noyau de simulation et leur instanciation, des événements et de la sortie graphique. Elle offre également des fonctionnalités de contrôle d'intervention en cours d'exécution, dont ceux présentés dans la section 5.4.

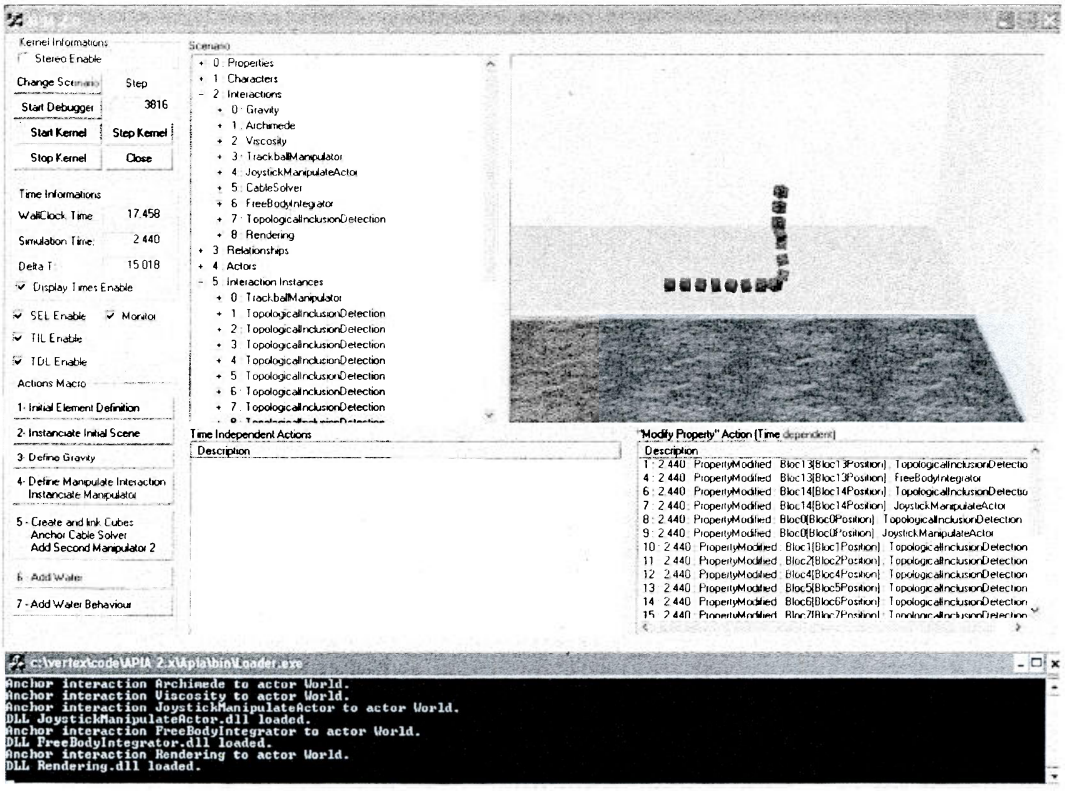

Figure A.14 : Seconde version de l'interface graphique d'**APIA** mettant l'accent sur le contrôle et la visualisation détaillée d'éléments du noyau.

A.8 Conclusion sur l'implantation

L'implantation dans un *framework* met l'accent sur l'aspect générique afin de permettre le développement d'un grand nombre de MVs. Le noyau suit de près l'architecture conceptuelle d'**APIA** et ne limite donc pas l'*agilité*, sauf pour des considérations de puissance de calcul requises par les gestionnaires et les instances d'interactions. Par exemple, des règles d'interaction trop complexes ou un grand nombre d'acteurs imposeraient une charge algorithmique trop lourde au *gestionnaire de la cohérence présente*.

Au cours de son développement, l'implantation est passée par de nombreuses évolutions pendant lesquelles des problèmes de maintenance sont apparus. Un *framework* de cette taille contient plusieurs niveaux de dépendances qui causent une cascade de modifications qui complexifient le développement. Par exemple, une modification dans les AData causée par une amélioration dans l'analyseur de scénario demandera une modification du noyau et des

interactions car elles en dépendent. Plusieurs cycles d'évolution ont également fait ressortir plusieurs alternatives d'implantation pour **APIA**. L'approche des graphes actuellement utilisée est difficile à déverminer et à optimiser. Une approche en tableaux, similairement à ce qui est utilisé dans les librairies spécialisées dans les graphes, serait à considérer.

De nombreuses autres itérations permettraient d'améliorer un *framework* encore à ses débuts. Ce mécanisme d'itération est fondamental à son évolution mais cause plusieurs problèmes. L'entretien du *framework*, la migration des applications vers les versions ultérieures et le support aux usagers des versions précédentes sont autant de problèmes à gérer. Le développement de nombreuses applications expose également le développeur du *framework* au piège d'y inclure les fonctionnalités communes à plusieurs applications. Il s'agit probablement du piège le plus dangereux car il devient facile de perdre le contrôle et de s'éloigner de l'objectif principal. Puisqu'un *framework* représente une solution élégante et bien intégrée, le travail d'abstraction sous-jacent demandera un perpétuel réusinage, retardant ainsi l'atteinte de l'objectif principal. Par exemple, avec **APIA**, les senseurs, effecteurs et noeuds de calcul sont difficiles à concilier avec les éléments et les gestionnaires.

Le développement de la première version, quoique générique et répartie, s'est arrêtée parce qu'elle devenait trop difficile à maintenir et à comprendre, une situation similaire à NPSNET. La deuxième version a plutôt misé sur l'accroissement de l'*agilité* au détriment des fonctionnalités, tout en conservant les modules génériques du premier *framework*. La programmation C++ de l'ensemble a également révélée qu'un langage ou un intergiciel plus dynamique faciliterait l'implantation d'**APIA**.

Un autre piège rencontré lors de l'élaboration du *framework* consiste à vouloir représenter l'architecture dans le paradigme de l'outil principal sur lequel il repose, CORBA dans ce cas-ci. Les progrès les plus importants ont été obtenus lorsque l'utilisation de CORBA a été abandonnée. Par analogie, lorsqu'on n'a qu'un marteau dans sa trousse à outils, tous les problèmes finissent par ressembler des clous.

Finalement, l'expérience acquise durant ces travaux laisse croire que la capacité de réutilisation des éléments et la facilité de conception d'un MV reposent autant sur un bon environnement de scénarisation que sur une approche favorisant l'*agilité*. De plus, les senseurs et les effecteurs pourraient être mieux intégrés aux modèles conceptuels découlant d'**APIA**.

Annexe B

Exemple de MV pour l'inspection de barrages

B.1 Introduction

Pour évaluer le niveau de dégradation de ses barrages, Hydro-Québec a développé un sous-marin téléopéré équipé de divers outils de mesure. Ce sous-marin mesure la dégradation des barrages, telles que des fissures. Ces informations sont enregistrées dans une base de données pouvant être analysée ultérieurement. Similairement à la cryochirurgie, l'utilisation des MVs peut améliorer la planification, l'entraînement et l'assistance aux inspections de barrages. Cette application s'inscrit également dans le cadre du projet VERTEX.

Un MV permet d'entraîner l'opérateur à effectuer des tâches critiques. Par exemple, un opérateur doit éviter le plus possible que le câble ne s'enroule autour du sous-marin. Par conséquent, le sous-marin et le câble doivent être modélisés avec un niveau élevé de fidélité. L'entraînement et l'assistance aux opérateurs sont les deux modes développés à partir d'**APIA**. La figure B.1 a) illustre le sous-marin de l'IREQ à l'intérieur d'un bassin dans le laboratoire de Varenne. À partir de la modélisation de cet environnement réel dans un MV basé sur **APIA**, il est possible d'assister l'opérateur en opérant un sous-marin virtuel en conformité avec un sous-marin réel tel qu'illustré à la figure B.1 b). Cette approche permet de pallier aux problèmes de visibilité des caméras embarquées dans le sous-marin à cause des mauvaises conditions de visibilité sous l'eau. L'entraînement de l'opérateur est également possible avec **APIA** en reproduisant les comportements physiques d'un sous-marin dans un MV, tel qu'illustré à la figure B.1 c).

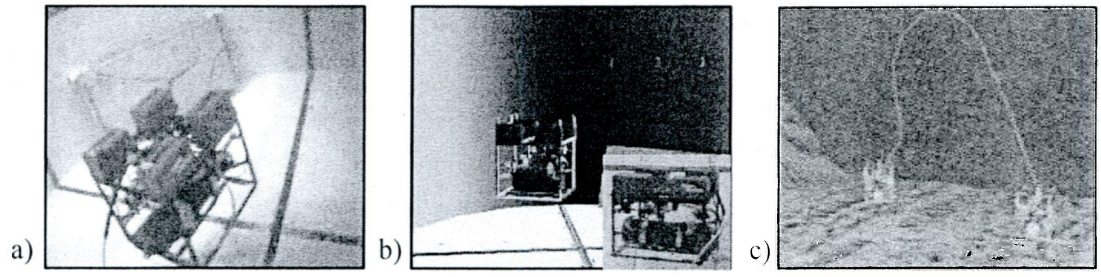

Figure B.1 : Utilisation d'APIA pour l'application d'inspection de barrages : a) sous-marin réel de l'IREQ b) virtualité augmentée pour le contrôle facilité du sous-marin c) réalité virtuelle pour l'entraînement.

B.2 MV pour l'entraînement d'inspection de barrages

L'entraînement à l'opération du sous-marin a pour but de se familiariser avec la téléopération du sous-marin dans des conditions parfois difficiles, par exemple lorsque les courants rendent les déplacements laborieux. Une majorité d'éléments utilisés dans ce MV d'entraînement proviennent de l'exemple du MV avec un câble de la section 5.4 ce qui vient démontrer la réutilisabilité. La principale différence se situe dans le modèle de câble qui, au lieu d'être basé sur *Open Dynamic Engine*, a été développé et validé par l'Université de Victoria (Buckham *et al.* 2004) afin d'accroître le degré de fidélité. D'autres éléments comme les acteurs *Sous-marins*, *Câble* et *Scène statique* ont été crées. L'interaction *Détecter collisions* a également été définie.

La figure B.2 illustre le diagramme APRI de l'application du sous-marin incluant les communications avec le monde réel. Les sous-marins, reliés entre eux par un câble à des fins de démonstration sont contrôlés à partir de manches à balai. La figure B.1 c) illustre ce MV.

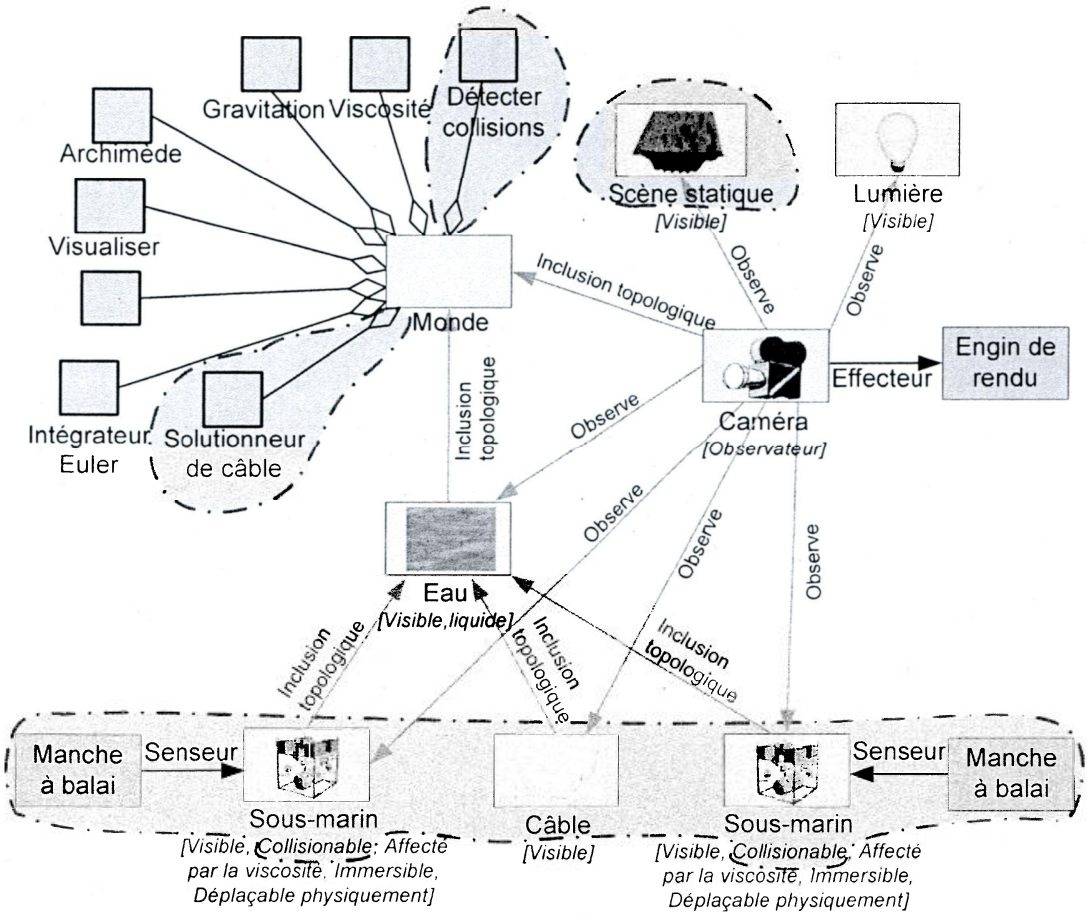

Figure B.2 : Diagramme APRI pour l'application d'entraînement à l'inspection de barrages montrant les ajouts requis à partir de l'exemple du câble.

B.3 MV pour l'assistance à l'intervention d'inspection de barrage

Quoique le système de positionnement du sous-marin soit précis, il s'avère très difficile de diriger le sous-marin avec la lecture de capteurs de position car les chiffres affichés ne donnent pas une information très conviviale pour un humain. De plus, même si le sous-marin possède une caméra embarquée, les conditions de visibilité à partir d'une certaine profondeur sont réduites à quelques dizaines de centimètres, tel qu'illustré à la figure B.3, et ne permettent pas de diriger le sous-marin avec efficacité.

Figure B.3 : Exemple d'image prise avec la caméra attachée au sous-marin.

Un MV peut reproduire le sous-marin et afficher les fissures en 3D. Si ce MV représente la position et contrôle le sous-marin réel, l'opérateur peut alors y accomplir ses tâches. Dans cette approche, désignée virtualité augmentée (Milgram & Kishino, 1994), un usager évolue dans le MV sans se préoccuper de l'existence réelle ou non des éléments avec lequel il interagit. La transposition de ses actions dans le monde réel est effectuée automatiquement. De plus, l'opérateur pourrait inspecter le barrage virtuel avec les fissures numérisées du barrage réel.

L'assistance à l'intervention consiste donc à opérer le sous-marin dans le MV, où les conditions d'opération sont optimales, et à transposer les commandes ou les déplacements dans le monde réel. Ce mode requiert une pré-modélisation de l'environnement réel ou une acquisition en cours d'exécution de l'environnement. L'application d'assistance à l'opération développée avec **APIA** est basée sur une modélisation avant l'exécution, représentée par la scène statique à la figure B.4. L'opérateur peut alors manipuler le sous-marin réel à travers celui virtuel, tel qu'illustré à la figure B.1 b). Cette application a été testée avec succès à l'IREQ. APIA a pu contrôler le sous-marin

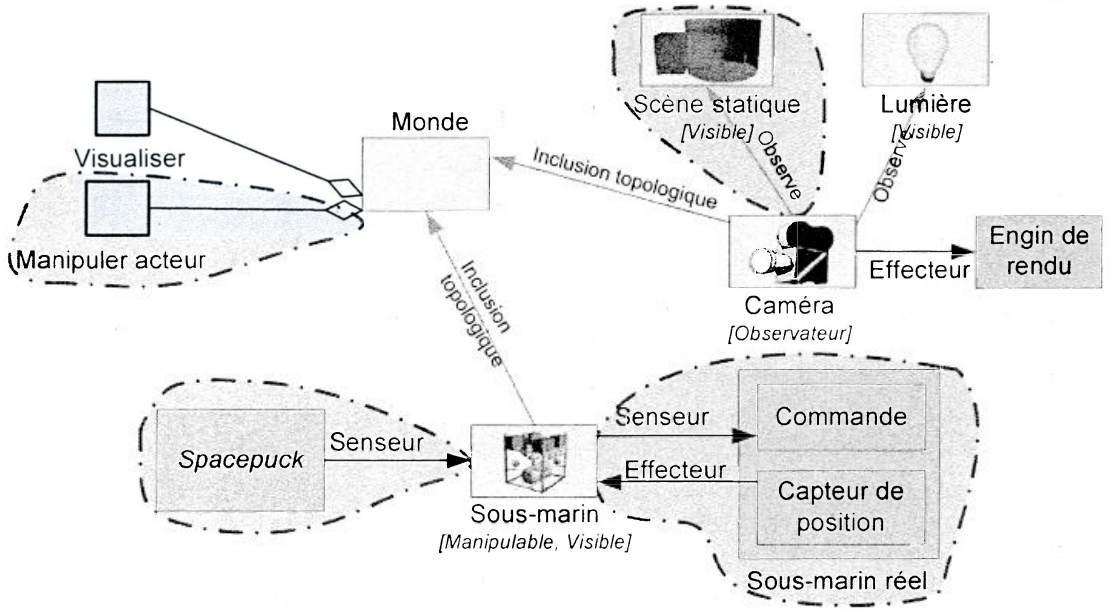

Figure B.4 : Diagramme APRI pour l'assistance à l'intervention d'inspections de barrages à l'aide d'un sous-marin téléopéré montrant les ajouts requis par rapport à l'application d'entraînement.

Le sous-marin se contrôle à l'aide d'un Spacepuck^{MD}. La position du sous-marin virtuel est mise à jour avec un capteur de position sonore et inertiel embarqué. La commande de force est alors envoyée au sous-marin réel asservi en forces. Bien que le développement de ce MV ait requis près de deux semaines, son interfaçage avec le vrai sous-marin s'est effectué en une demi-journée sur le site de l'IREQ. La manipulation du sous-marin réel à travers l'application basée sur **APIA** s'est révélée très simple, en partie dû aux habiletés acquises lors de l'utilisation du MV d'entraînement décrit précédemment.