Université de Nice Sophia Antipolis

DEA Réseaux et Systèmes Distribués

Rapport de Stage

Application de la Programmation Réactive à la Modélisation en Physique

Présenté en juillet 2002

par Alexander Samarin

Responsable de stage : Frédéric Boussinot

Remerciements

J'aimerais remercier toute l'équipe du projet MIMOSA pour un excellent accueil.

Je tiens à remercier mes encadreurs, Frédéric Boussinot et Jean-Ferdi Susini, pour leur inspiration et leur aide durant ces derniers mois. Grâce à eux j'ai apris beaucoup de choses très intéressantes. Merci pour leur grande disponibilité et leur attention à mes questions, même aux plus naïves.

Je voudrais remercier également Jean-Paul Marmorat et Rémi Drai, les chercheurs automaticiens du Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris, pour leurs remarques extremement importantes sur le texte de mon rapport.

Merci beaucoup à Raul Acosta, Dimitri Petoukhov et Christian Brunette, pour leur soutien amical et pour l'ambiance chaleureuse :)

Je remercie Catherine Junker et Dominique Micollier pour leur soutien permanent pendant tout mon séjour en France.

Et enfin, je voulais dire un merci particulier à Nadia Maizi, la directrice du Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris, qui a rendu ce stage possible.

Alexander Samarin

21 juin 2002

Contents

1 Introduction

1.1 Description du sujet de stage

De nombreux systèmes de la Physique sont décrits par un jeu d'équations différentielles (équations de Newton, de Maxwell, de Boltzmann, etc). Lorsque leur résolution directe échoue, on fait appel à *la résolution numérique*. Cela revient à discrétiser le système d'équations, puis à en faire une intégration pas-à-pas. On emploie souvent le terme de *simulation continue*, ou *numérique*, pour cette technique. En fait, il s'agit bien de reproduire l'évolution dynamique du système - décrit dans ce cas par un modèle mathématique. Deux domaines principaux de simulation en physique *sont la simulation des systèmes dynamiques continus* et *le contrôle des systèmes continus. Les modèles hybrides* prennent aussi en compte les changement " instantanés " d'état du système continu.

Pourtant il peut être assez difficile de construire le modèle analytique initial. Plusieurs outils informatiques tels que Simulink, Dymola, ACSL, Modelica, et d'autres, dont le but principal est la simulation des systèmes continus, proposent différentes techniques de *construction graphiques* de modèles (*block diagram simulators*). Mais la simplicité de construction est liée souvent aux contraintes au niveau de la puissance expressive. Au contraire, la fiabilité de la description analytique du modèle (par exemple, sous forme des systèmes d'équations différentielles) peut rendre cette description beaucoup moins modulaire. Souvent les composants doivent être soumis à des règles assez strictes et très spécifiques à chaque situation particulière. Par exemple, dans la plupart des logiciels de simulation et de contrôle des systèmes continus, il n'est pas possible d'établir une connexion entre n'importe quel composant et n'importe quel d'autre. Ces contraintes peuvent être également un désavantage pour les composants qui au départ sont censés être réutilisables.

Dans ce texte on propose une nouvelle technique de simulation de systèmes dynamiques continus qui introduit un temps discret *explicite* dans le processus de simulation. Cette technique est basée sur *l'Approche Réactive* qui définit un modèle de programmation du parallélisme et dont la notion principale est celle d'*instant*. La notion d'*instant* permet de définir *un parallélisme synchrone* et un puissant mécanisme de communication entre composants parallèles – *la diffusion instantanée d'événements*. Ces deux mécanismes permettent de représenter un système physique complexe comme un ensemble d'entités interagissant et évoluant au cours du temps et d'effectuer ainsi *une décomposition du système.*

La simulation à base de *la programmation orienté-objets* associe à une entité *physique un objet logique* qui regroupe les données correspondant aux variables d'état de cette entité et les méthodes de traitement de ces données. Dans ce texte, il s'agira du langage *Java* qui possède un modèle objet simple et facile à appréhender et qui est utilisé de plus en plus souvent dans le domaine scientifique (en particulier, grâce à l'apparition recente de bibliothèques mathématiques développées). Le modèle *des objets réactifs* vise à intégrer les concepts de la programmation objet avec les principes de l 'Approche Réactive. Dans ce modèle, *un programme réactif* est lié à *un objet Java.* Le programme réactif a accès aux variables de l'objet lié et peut changer consécutivement, instant après instant, son vecteur d'état. Cela revient à dire que *l'état* d'une entité physique peut être encapsulé dans un objet Java, et son *comportement* l'est dans le programme réactif lié à cet objet. Ainsi, la dynamique d'un système physique complexe peut être simulée par l'exécution synchrone de plusieurs composants mis en parallèle et communicant par l'intermédiaire d'événements. Ce modèle permet d'implémenter de manière assez naturelle les deux aspects du modèle physique – discret et continu.

Les règles sémantiques de l'Approche Réactive définissent clairement les notions de présence, d'absence et de simultanéité d'événements. Le parallélisme logique permet ainsi d'implémenter le traitement synchrone de toutes les influences extérieures. Le comportement d'un composant peut être construit à partir de briques élémentaires simplement en mettant les traitements correspondants en parallèle. Dans ce texte, je propose un schéma général qui permet d'implémenter un traitement synchrone des influences extérieures de differents types tout en préservant la validité du modèle physique et la stabilité numérique des résultats. En résultat, on obtient un modèle simple et modulaire.

1.2 Structure du document

La première partie (Section 2) décrit les règles sémantiques principales de l'Approche Réactive et le modèle d'objets réactifs.

La partie suivante contient un analyse des modèles physiques classiques et, en particulier, de leur description en termes *des systèmes d'équations différentielles ordinaires.* Elle propose un apperçu des algorithmes numériques qui permettent de résoudre ce genre de systèmes. Cette partie se termine par un petit " état de l'art " sur les logiciels de simulation des systèmes continus.

La section 4 démontre comment on peut traduir les notions principales de la methodologie de construction des simulation en termes de la programmation réactive. En particulier, j'essaye d'établir les relations plus claires entre la discrétisation en modélisation et *les instants* en programmation réactive. Je décris le modèle de l'exécution synchrone des programmes réactifs des composants qui permet de décomposer un système physique complexe en plusieurs éléments indépendants mais parfaitement synchronisés du point de vue temporel. Je propose *le modèle des influences* et le schéma général de construction des programmes réactifs des composants qui permettent d'exprimer la continuété des interactions à travers les événements diffusés instantanément en gardant en même temps la forte modularité du modèle.

La section 5 est consacrée à l'implémentation du modèle proposé, ainsi qu'à la visualisation des résultats. J'ai implémenté quelques classes qui etendent le framework Icobjs servant pour la représentation graphique des simulations. Je decrit la classe Icobject et les éléments de contrôle des simulations que j'ai implémentés au cours de mon travail.
L'analyse faite dans les sections 3 et 4 m'a permis d'adopter et de faire une implémentation particulière des algorithmes numériques standards. Ils sont regroupés dans le package Numerics. L'imlémentation permet de changer facilement l'algorithme.

L'avant-dernière partie décrit quelques expérimentations virtuelles que j'ai faites et qui démontrent l'application du modèle proposé dans la section 4 ainsi que ses avantages. Plusieurs exemples tels que la simulation du problème de trois corps (" many-body problem "), les potentiels complexes, la modélisation des liquides simples, le traitement des forces dissipatives, les liens élastiques et les systèmes des réservoirs communicants sont construit à partir du même schéma de manière simple et efficace.

Enfin, dans la section 6 je souligne quelques résultats pricipaux de ce stage, ainsi que la problèmatique et les axes futurs de recherche.

2 Approche Réactive

L'Approche Réactive [1], qui dérive de *l'Approche Synchrone* [2], repose sur une découpe logique de l'exécution d'un programme en une succession d'*instants.* La notion d'*instant* permet de définir un parallélisme synchrone et un puissant mécanisme de communication entre composants parallèles - *la diffusion instantanée d'événements.* Les composants réagissent aux événements (messages simples ou *événements valués*) et maintiennent ainsi une interaction continue avec leur environnement et les autres composants.
Deux formalismes principaux qui implémentent les primitives de base de l'approche réactive en Java sont *Junior* [3] (noyau qui définit les règles sémantiques de l'exécution des instructions réactives implémentées en Java) et *SugarCubes* [4] (framework plus développé pour construire des systèmes réactifs dynamiques).

2.1 Notion d'instant

Typiquement l'exécution d'un programme réactif consiste à activer chaque branche parallèle du programme de manière cyclique. On considère que c'est l'environnement qui détermine le rythme des activations.

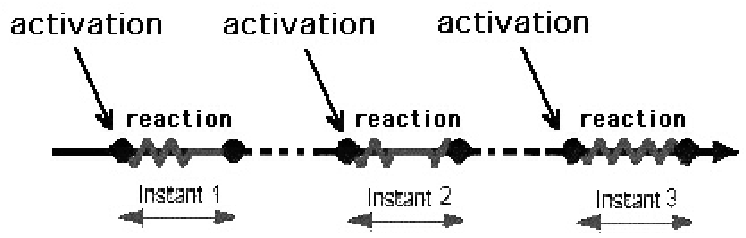

Figure 1: Un programme réactif

L'instant suivant est détecté lorsque toutes les branches parallèles ont stoppé leur exécution pour l'instant courant, c'est-à-dire qu'elles ont toutes réagi aux activations provenant de l'extérieur par l'intermédiaire d'événements.

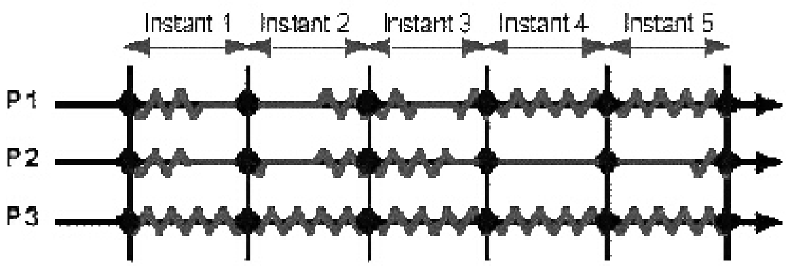

Figure 2: Exemple : trois programmes réactifs en parallèle

Aucune hypothèse n'est faite sur la durée de l'instant. Au niveau du langage, c'est l'exécution des instructions qui permet de savoir si un instant est fini ou non, c'est-à-dire on détermine la fin d'un instant si plus aucune instruction ne peut être exécutée dans l'instant courant et si on est sûr que plus aucun événement ne sera généré pendant cet instant.

2.2 Actions atomiques

Les opérations atomiques sont des procédures de traitement des données exécutées de façon atomique, c'est-à-dire qu'aucune exécution concurrente ne peut avoir lieu en parallèle pendant l'exécution d'une telle opération. Ainsi, il ne peut y avoir d'interférence durant l'exécution d'une action atomique qui risquerait de corrompre les données sur lesquelles le traitement opère. Ce fait est tres important dans les cas ou il est nécessaire d'implémenter un système gérant des activités concurrentes.

2.3 Communications entre les composants

L'*Approche Réactive* propose une notion *d'événements diffusés instantanément* à tous les composants parallèles d'un système réactif comme moyen de communication. Les événements sont des données de l'environnement d'exécution d'un système. Un programme réactif peut générer des événements ou utiliser ces événements pour paramétrer son exécution. La notion d'instant permet de définir clairement les notions de présence, d'absence ou de simultanéité d'événements auxquelles un programme peut réagir.
Les propriétés des événements sont suivantes :

- Les événements sont automatiquement réinitialisés au début de chaque instant. Ainsi, les événements ne sont pas des valeurs persistantes dans l'environnement, conservées entre les instants.
- Générer un événement qui a déjà été généré n'a aucun effet.
- Tous les composants parallèles dans un système réactif ont la même vision de chaque événement à chaque instant.
- Il n'est pas possible de décider si un événement est absent au cours d'un instant, avant que l'instant ne soit terminé. C'est en effet le seul moment où l'on est sûr que personne ne pourra plus le générer et viendrait ainsi contrarier cette hypothèse. De ce fait, la réaction à l'absence d'un événement est toujours reportée à l'instant suivant. Ceci constitue le principe de base de l'*Approche Réactive*.

Un programme réactif peut également manipuler des *expressions booléennes événementielles* basées sur la présence d'événements à un instant donné. Ces expressions booléennes sont appelées des *configurations événementielles*.

2.4 SugarCubes : primitives de base

Les SugarCubes [5] implémentent l'approche réactive en Java. Les SugarCubes se présentent sous forme d'un ensemble de classes Java qui se décompose en deux groupes :

- les instructions réactives, dont la sémantique réfère à la notion d'instant ;
- les machines réactives dont le rôle est d'exécuter les instructions réactives.

2.4.1 Instructions réactives

La classe `Instruction` implémente la notion d'instruction réactive. Une instruction réactive est activée par un appel de sa méthode `activ`. L'appel de `activ` renvoie une valeur décrivant l'état de l'instruction après exécution ; c'est l'une des trois valeurs suivantes :

TERM indique que l'instruction a définitivement terminé son exécution. Il ne reste rien à exécuter lors des prochaines activations.

STOP indique que l'instruction à terminé sa réaction pour l'instant courant. L'exécution reprendra en séquence à l'instant suivant.

SUSP indique que l'instruction a suspendu son exécution pour l'instant courant car elle n'a pas réussi à atteindre un état stable (TERM ou STOP) et doit être réactivée au cours du même instant. Cela permet par exemple à une branche du programme en attente d'un événement non encore émis de suspendre son exécution pour laisser la possibilité aux autres branches de s'exécuter et de pouvoir ainsi éventuellement générer cet événement dans l'instant courant.

On distingue trois types d'instructions :

- les instructions *atomiques* ;
- les instructions *unaires* qui opèrent sur un corps ;
- les instructions *binaires* qui comportent deux sous-instructions.

Le noyau des SugarCubes regroupe les instructions suivantes :

Nothing est une instruction atomique qui ne fait rien et termine instantanément.

Stop arrête l'exécution pour l'instant courant et passe en séquence à l'instant suivant.

Seq est une instruction binaire qui place deux instructions en séquence. Lorsque la première instruction termine, **Seq** passe immédiatement à l'exécution de la seconde.

Merge place deux instructions en parallèle. A chaque instant, elle active la première branche puis la seconde, au cours de la même réaction. **Merge** termine lorsque ses deux branches ont terminé.

Atom permet d'exécuter des opérations atomiques en Java.

Repeat et **Loop** sont des instructions unaires qui permettent de définir des boucles finies ou infinies.

Comme on l'avait déjà dit les SugarCubes permettent l'utilisation d'événements diffusés instantanément. De plus, il est possible de créer des *configurations événementielles* qui sont des expressions booléennes évaluées à partir de la présence ou de l'absence de différents événements. Les instructions événementielles sont les suivantes :

Generate génère un événement.

Await attend qu'une configuration d'événements soit satisfaite.

Until préempte son corps lorsqu'une une configuration d'événements est satisfaite (la préemption est faible, conséquence du rejet de la réaction instantanée à l'absence des événements).

Control asservit l'exécution de son corps à la présence d'un événement : le corps ne réagit qu'aux instants où un événement est présents et il est inactif aux autres instants.

When choisit entre ses deux branches suivant qu'une configuration événementielle est satisfaite ou non.

2.4.2 Machines réactives

Une machine réactive est une instruction unaire qui exécute son corps, appelé *programm*e, en lui fournissant un environnement d'exécution.

L'algorithme de détection des fins d'instants que la machine implémente obéit aux principes suivants :

- Le programme est activé tant que l'appel de sa méthode **activ** renvoie SUSP.
- Si l'exécution du programme renvoie SUSP, la machine s'assure que de nouveaux événements ont été générés au cours de cette activation. Si ce n'est pas le cas, aucune des branches du programme suspendues en attente d'un événement ne peut plus progresser. Aucun nouvel événement ne pourra donc survenir et la machine décrète alors la fin de l'instant en positionnant un drapeau pour signifier aux instructions suspendues que l'instant est terminé. Les événements sur lesquels les instructions du programme sont en attente sont donc considérés comme absents ; les instructions suspendues renvoient alors STOP, ce qui permet à la machine de terminer l'instant courant. La réaction à l'absence des événements est ainsi nécessairement reportée à l'instant suivant (prochaine activation de la machine).

- La fin de l'instant n'est effective que lorsque toutes les instructions du programme, ont retourné STOP ou TERM : il ne reste plus d'instruction suspendue.

De nouvelles instructions peuvent être ajoutées dynamiquement en parallèle au programme d'une machine (entre deux réactions) par la méthode addProgram(). Ainsi la possibilité de créer de nouveaux composants parallèles au cours de l'exécution facilite la programmation, par exemple en n'obligeant pas à définir un nombre maximum de composants.

2.5 Objets réactifs

Le modèle *des objets ractifs* vise à intégrer les concepts de la programmation objet (ici il s'agira du language **Java** [6],[7]) avec les principes de l'Approche Réactive.

L'interface *Cube* dans **les SugarCubes** permet de lier un programme réactif à un objet Java. Un *Cube* peut être vu comme l'encapsulation dans une entité autonome :

- d'**un objet Java** constitué d'une structure de données et des méthodes de traitements de ces données;
- d'**un comportement réactif** propre à l'objet, qui ordonnance et contrôle dans le temps l'exécution des traitements de données.

À chaque instant, le comportement réactif d'un *Cube* exécute une réaction spécifique. Au cours de cette réaction, des opérations atomiques de traitement de données sont exécutées sous le contrôle du programme réactif. Ces traitements utilisent en général les méthodes de l'objet encapsulé par le *Cube*, lesquelles accèdent directement à la structure de données de l'objet. Ainsi, toutes les opérations atomiques définies dans le comportement d'un *Cube* partagent l'accès au même objet Java et donc aux données qu'il renferme.

```
Electron electron = new Electron(...);

Program program = SC.loop(
                      SC.seq(
                        SC.await(REACTION1,SC.action(new ChangeState(UP))),
                        SC.stop()));

machine.addProgram(SC.cube("electron",electron,program); // add the program
                                                          // in the machine
...

class ChangeState implements JavaAction {

   String flag;

   public ChangeState(String flag){this.flag=flag;}

   public void execute(Object self, LocalVariables vs, ReactiveEngine e) {

      Electron electron = (Electron)self;        // action ChangeState got
      State state = electron.getState();         // acces to the variables
                                                 // of linked object ;
      state.transition(flag);                    // then it can control it
   }
}
```

Les *Cubes* communiquent entre eux sur des canaux de diffusion identifiés par des événements sans jamais avoir besoin de la moindre connaissance des autres objets présents dans le système (en particulier, il n'est pas nécessaire d'obtenir de référence sur les objets avec lesquels on communique). Lorsqu'un *Cube* génère un événement, si quelqu'un est intéressé par cet événement il réagira à l'occurrence de celui-ci; si personne n'est intéressé, aucune réaction n'aura lieu. On peut donc concevoir le comportement d'un *Cube* de façon totalement indépendante de tous les autres éléments du système. Seule l'utilisation des événements diffusés permet de communiquer avec d'autres objets.

La classe statique SC implémente les primitives principales des SugarCubes sous forme de méthodes statiques.

Exemple : le code suivant associe un programme réactif à un objet Java. Ce programme consiste à générer à chaque instant l'événement contenant la valeur de la variable **spin** de l'objet Electron. C'est la méthode evaluate() de la classe GetSpin qui retourne à runtime la valeur actuelle du spin.

```
class Electron {

   public int spin;
   ...                    // class description
}

class PhotoEmission {

   ...

   Electron electron = new Electron(...);        // initialization

   Program p = SC.cube("electron",electron,      // linkage program-object
               SC.loop(
                 SC.seq(
                   SC.generate(MYSPIN, new GetSpin()),
                   SC.stop()
               )));

   machine.addProgram(p);

   ...

}
```

Nous allons voir plus en détails l'application des primitives de L'approche Réactive dans les deux dernières parties de ce texte.

3 Modélisation de la dynamique de systèmes physiques

Dans cette section on va établir quelques notions principales qu'on va utiliser dans ce texte et qui sont indispensables dans le domaine de la modélisation et de la simulations de systèmes physiques. On va donner un aperçu des outils mathématiques qui sont utilisés pour décrire la dynamique des phénomènes physiques, et considérer plus en détail l'un d'entre eux – *les équations différentielles ordinaires*, ainsi que les méthodes numériques qui permettent de résoudre ce type d'équations [9],[10],[11].

3.1 Modèles comportementaux classiques. Etats d'un système. Boîte noire.

En général la dynamique d'un phénomène physique peut être représentée comme l'évolution d'un ensemble de variables au cours du temps. Lorsqu'on utilise les ordinateurs pour simuler un phénomène physique, ou pour résoudre numériquement un modèle mathématique décrivant ce phénomène, il est nécessaire d'effectuer le passage au *temps discret*. Pour cela on découpe l'axe temporel en intervalles équirepartis (ou non-équirepartis) en associant ainsi aux fonctions continues les fonctions discrètes :

$$t \to t_i, \quad f(t) \to f(t_i), \quad t_{i+1} - t_i = \Delta t \tag{1}$$

L'ensemble de valeurs X_i décrit la dynamique du système avec une certaine précision. Alors on peut introduire *la notion d'état du système au moment* t_i. Le but de *la modélisation* c'est de décrire les règles, plus ou moins abstraites, selon lesquelles le système évolue au cours du temps (*description analytique*). Dans ce texte il ne s'agira pas de donner les règles de transition de n'importe quel état à n'importe quel état, mais seulement décrire les *passages de l'état précédent à l'état suivant*, c'est-à-dire pouvoir calculer l'état X_{i+1} à partir de l'état X_i. L'objectif est de représenter le comportement dynamique " entrée - sortie " d'un système.
Les logiciels de *simulation* visent à implémenter ces règles et effectuer le calcul d'états pas-à-pas. Ces logiciels utilisent des valeurs réelles (voir expérimentales) pour les paramètres du système dans le but de pouvoir prédire le comportement du système en réalité ou dans des conditions très proches. La partie du code qui implémente le mécanisme du calcul consécutif d'états d'un système est appelée souvent *la boîte noire*.
Le schéma du fonctionnement d'un logiciel de simulation représente donc l'activation cyclique de la boîte noire avec les nouveaux paramètres d'entrée. *La boîte noire* encapsule ainsi *le comportement du système*.

3.2 Modèle analytique. Equations différentielles.

Très souvent la dynamique d'un phénomène physique peut être décrite de manière analytique à l'aide des équations différentielles. Dans ce cas on parle d'un modèle dit " *de connaissance* ".
Nous allons considérer en particulier *les équations différentielles ordinaires du premier ordre* :

$$\dot{x} \equiv \frac{dx}{dt} = f(x), x \in R^n \tag{2}$$

ou encore *les systèmes d'équations différentielles ordinaires* :

$$\begin{cases} \dot{x}_0 = f_0(x_0, x_1, ..., x_n) \\ \dot{x}_1 = f_1(x_0, x_1, ..., x_n) \\ \cdots \\ \dot{x}_n = f_n(x_0, x_1, ..., x_n) \end{cases} \tag{3}$$

Dans certains cas, on peut trouver une solution de (3) analytiquement (modélisation mathématique). Pourtant, dans les systèmes physiques on rencontre des relations non linéaires qui sont responsables du type de comportement rencontré dans la réalité. La prise en charge de ces relations ne permettra plus, en principe, une solution analytique du modèle. En outre, quand le modèle d'un système est linéaire, la complexité de celui-ci, liée à sa dimension élevée, peut rendre la solution analytique inopérante.
C'est dans ces cas-là qu'on peut essayer d'obtenir *solution approchée* de (3) avec les méthodes numériques.

3.3 Discrétisation du temps et différences finis

Après avoir fait le passage au temps discret (1) nous pouvons remplacer la dérivée d'une fonction par l'approximation de ce type : (" forward finite difference ")

$$\frac{dx}{dt} = f(x,t) \approx \frac{x_{i+1} - x_i}{\Delta t}$$
$$x_{i+1} = x_i + f(x_i, t_i)\Delta t + O((\Delta t)^2) \tag{4}$$

3.4 Algorithmes numériques

Considérons l'équation (2). Le but est de trouver la solution $X_i = X(t_i)$, i=1,... sachant les conditions initialles X_0.
Si nous négligeons l'erreur $O((\Delta t)^2)$ dans (4), nous pouvons trouver consécutivement (pas à pas) tous les X_i comme (algorithme *itératif*):

$$x_{i+1} = x_i + f(x_i, t_i)\Delta t$$

Cet algorithme s'appelle *l'algorithme d'Euler.* Dans le cas d'*un système d'équations différentielles ordinaires* on peut écrire

$$\overrightarrow{x}_{i+1} = \overrightarrow{x}_i + \overrightarrow{f}(\overrightarrow{x}_i)\Delta t \tag{5}$$

Prenons comme un exemple *l'équation de Newton du mouvement* :

$$\frac{d^2x}{dt^2} = \frac{F}{m} \tag{6}$$

Le système d'équations correspondant est :

$$\begin{cases} \dot{x} = V \\ \dot{V} = a = \frac{F}{m} \end{cases} \tag{7}$$

Alors deux formes de l'algorithme d'Euler sont possibles (*Euler " original "* et *Euler-Cromer* respectivement) :

$$\begin{array}{ll} V_{i+1} = V_i + a_i\Delta t \qquad & V_{i+1} = V_i + a_i\Delta t \\ x_{i+1} = x_i + V_i\Delta t \qquad & x_{i+1} = x_i + V_{i+1}\Delta t \end{array}$$

où $a_i = \frac{F(t_i)}{m}$ et $F = F(x(t), V(t), t)$.

Considérons toujours l'équation de Newton (6). On peut améliorer les résultats en faisant l'estimation dans un état intermédiaire (*l'algorithme d'Euler-Richardson*) :

$$\begin{array}{l} V_{\text{mid}} = V_i + \frac{1}{2}a_i\Delta t \\ x_{\text{mid}} = x_i + \frac{1}{2}V_i\Delta t \\ a_{\text{mid}} = \frac{1}{m}F(x_{mid}, V_{mid}, t_i + \frac{1}{2}\Delta t) \end{array}$$

Alors l'état suivant se calcul comme :

$$\begin{array}{l} V_{i+1} = V_i + a_{mid}\Delta t \\ x_{i+1} = x_i + V_{mid}\Delta t \end{array}$$

Une autre technique consiste à utiliser l'expression explicite des séries de Taylor. L'algorithme le plus simple et le plus connu est celui de Verlet. Son application à (7) donne (sa " self-starting " forme qui s'appelle " *Velocity Verlet* "):

$$x_{i+1} = x_i + V_i\Delta t + \tfrac{1}{2}a_i(\Delta t)^2$$
$$V_{i+1} = V_i + \tfrac{1}{2}(a_{i+1} + a_i)\Delta t$$

Enfin la méthode la plus connue et qui semble être la plus sûre et la plus précise parmi les algorithmes à pas constant est *l'algorithme de Runge-Kutta d'ordre 4*. *L'algorithme de Runge-Kutta d'ordre 2* est équivalent à celui d'Euler-Richardson mentionné plus haut.

L'algorithme de Ringe-Kutta d'ordre 4 effectue quatre estimations au cours de chaque pas.
Considérons l'équation (2) sous la forme :

$$\frac{dx}{dt} = f(x,t)$$

La forme générale de l'algorithme de Runge-Kutta d'ordre 4 s'écrit comme :

$$k_1 = f(x_i, t_i)\Delta t$$
$$k_2 = f(x_i + \tfrac{k_1}{2}, t_i + \tfrac{\Delta t}{2})\Delta t$$
$$k_3 = f(x_i + \tfrac{k_2}{2}, t_i + \tfrac{\Delta t}{2})\Delta t$$
$$k_4 = f(x_i + k_3, t_i + \Delta t)\Delta t$$

$$x_{i+1} = x_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

La dérivé est calculée d'abord au début de l'intervalle, puis de deux façons différentes au milieu et enfin à la fin de l'intervalle.

Les réferences sur quelques excellents ouvrages concernant les algorithmes numériques sont données dans la bibliographie ([9], [10], [11]). Sur le sujet d'integration de l'équation de Newton on peut adresser à [13], [9].

3.5 Choix d'algorithme

La solution numérique n'est qu'une approximation. Donc il faut tenir compte des erreurs possibles.
Il faut bien sur préciser que le choix d'algorithme peut dépendre de plusieurs paramètres tels que :

- Le niveau de conformité aux lois principales de la physique (conservation d'énergie, par exemple). Ainsi un algorithme qui n'est pas *stable*, c'est à dire, qui accumule les erreurs au cours du calcul, va “ rajouter ” à chaque pas dans le système une quantité ΔE d'énergie, ce qui rend les résultats non-valables du point de vue physique. Pourtant certains algorithmes contournent cette difficulté et diminuent “ automatiquement ” les erreurs en prenant en compte les dérivés de plus grand ordre et en faisant des estimations intermédiaires.

- La précision nécessaire (pour atteindre un niveau de précision donné on peut, dans la plupart de cas, diminuer le pas Δt, pourtant une réduction importante de Δt peut provoquer l'augmentation d'erreurs d'arrondi)

- Utilisation des algorithmes dits “ self-starting ” - (si l'algorithme ne l'est pas, alors on peut calculer quelques premiers pas avec un algorithme “ self-starting ”)

- Certains cas supposent *un pas de calcul de longueur variable* (pour, par exemple, augmenter la précision autour des points “ difficiles ”)

- les ressources disponibles

3.6 Exemple : l'équation du pendule simple

L'équation de mouvement du pendule simple est un exemple d'équation non-linéaire pour laquelle une solution analytique explicite n'existe pas.

$$\ddot{\theta} + \frac{g}{L} sin\theta = 0$$

On peut trouver une solution numérique, c'est-à-dire utiliser une méthode numérique pour trouver la fonction $\theta(t)$ comme tous les θ_i. Le système correspondant ici est :

$$\begin{cases} \dot{\theta} = \omega & \text{l'équation pour l'angle} \\ \dot{\omega} = -\frac{g}{L} sin\theta & \text{l'équation pour la vitesse angulaire} \end{cases}$$

Dans ce cas particulier on doit utiliser un algorithme précis au deuxième ordre.

3.7 Outils analytiques

Il existe plusieurs approches analytiques différentes dans les domaines de modélisation et du contrôle des sytèmes continus.
On va mentionner deux d'entre eux qui semblent être très proches de ce travail.

Fonctions de transfert / Transformées de Laplace – une technique qui permet de résoudre les équations differentielles *linéaires* d'ordre quelconque et dont la notion de base est la notion d'*état du système à l'instant* t_i. C'est un outil très developpé qui permet de représenter le comportement d'un système dynamique sous forme d'*équation d'état*. Un exemple d'application de cette technique dans le cadre d'expérimentations autours de la programmation réactive a été implémenté au Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris. Dans certains cas cette implémentation produit quelques petites instabilités numériques; pourtant l'application de cet outil semble être très prometteuse et nécessite une étude complémentaire.

Diagrammes de Forrester [14] – technique de structuration graphique des sytèmes dynamiques complexes, permettant de représenter un ensemble d'entités dynamiques interagissantes. Un diagramme de Forester se construit à partir de très peu d'éléments. Ces éléments sont :

- le niveau, il correspond aux variables d'état du système
- le flux, il représente la transmission d'une grandeur physique
- la variable auxiliaire
- le canal d'information, il représente un flux d'information
- le nuage, il représente une source ou un puits

3.8 Outils informatiques de simulation

L'application de l'approche orientée - objets en modélisation permet d'associer à une entité physique (*objet physique*) un ensemble de données et de méthodes pour les manipuler (*objet logique*). Le choix de ces entités peut varier selon le niveau et l'échelle nécessaires – microscopique ou macroscopique.

Hormis des systèmes continus on distingue aussi *les sytèmes aux événements discrets* qui visent à modéliser des changements "instantanés" dans le système, et *les systèmes hybrides* qui prennent en compte les deux composants - discret et continus (par exemple, un modèle des circuits électriques avec les événements "switch"). Dans ce cas-la, on introduit un modèle *d'événements d'état* dans le processus de simulation.

Parmi les outils qui proposent différentes tecniques de simulation et d'analyse des modèles continus on peut mentionner (très brièvement) :

Langages de simulation

ACSL (Advanced Continuous Simulation Language) [15] : Description et spécification d'équations différentielles. Temps discret et continu. Pré-compilation en Fortran. Compatibilité avec des modules en Fortran, C et Ada.

Desire [17] : langage interprété. Simulation des structures variées. Réutilisation des modules.

Langages de simulation orientés – objets

Dymola (Dynamic Modeling Laboratory) [18], [19] : Description de modèles en termes d'objets. Equations différentielles de grand ordre. Héritage multiple au niveau des composants. Plug-ins pour la construction graphique (Dynasim). Compilation en ACSL, Desire et SIMULINK.

Modelica [20], [21] : language puissant pour modélisation des systèmes physiques complexes (plus plusieurs instruments suplémentaires (graphiques, compilation, etc))

VHDL-A [22] : L'un des standards reconnus (IEEE). Description orienté-objet (mais pas d'héritage). Réutilisabilité et modularité fortes. Application principale – modélisation des circuits électriques.

“ Block Diagram Simulators ” - le but est de rendre la simulation plus modulaire

SIMULINK [25], [26] : Extension de MATLAB [24]. Construction graphique “ point-and-click ” de systèmes continus. Simulation du temps continu (équations differentielles) et du temps discret (équations en différences). Compatible avec le language MATLAB.

SystemBuild [27] : Extension de Xmath. Construction graphique. Modèles sous forme d'ensemble d'équations différentielles algébriques. Specification développée des événements d'état.

Pour plus d'information sur le sujet des logiciels de simulation on peut adresser à [28], ou [23]. La description des outils de simulation existant à ce moment-là sur le marché est donnée dans [23].
Remarque : Tous ces systèmes semblent utiliser implicitement un temps commun pour tous les composants. Pourtant, la spécification faible de l'aspet temporel souvent rends le traitement des événements d'état assez compliqué (en particulier dans les sytèmes hybrides). Seul le langage ACSL [16] propose un support donnant aux événements une caractéristique temporelle [23].
Dans ce texte on va discuter *un modèle synchrone* qui introduit de façon explicite un axe temporel dans les simulations et qui permet d'exprimer de manière assez naturelle les deux aspects - discret et continu.

4 Application de la programmation réactive à la modélisation

Lorsque l'on considère un système dynamique, on postule qu'il existe une frontière séparant ce système de l'environnement dans lequel il vit. Un modèle, en tant que représentation abstraite d'un système réel, est composé de :

- un ensemble de définitions permettant d'identifier les éléments qui constituent le modèle
- un ensemble de relations spécifiant les interactions entre les éléments qui apparaissent dans le modèle
- un ensemble de relations unidirectionnelles entre le système et l'environnement.

L'un des buts principaux de ce stage était de démontrer sur quelques exemples simples comment on peut traduire ces éléments en termes de la programmation réactive.

4.1 Evolution de l'état d'un objet au cours du temps

Comme on l'a vu dans la section précédente, on peut modéliser l'évolution d'un système physique continu en calculant consécutivement ces états avec l'un des algorithmes mentionnés plus haut.

L'association du pas de discrétisation à *un instant* introduit de façon naturelle *un temps discret* et permet de décrire le comportement d'un objet physique, ce comportement étant en fait les règles selon lesquelles son état change d'instant en instant. Cela correspond très bien au modèle *des objets réactifs*, décrit dans la section 2. Rappelons que dans ce modèle un programme réactif, qui est liée à un objet Java, est exécuté par la machine réactive en parallèle avec les autres programmes réactifs. Ce programme peut changer instant après instant les variables de l'objet lié. On peut donc voir que le comportement d'un objet physique peut être encapsulé en un programme réactif et son état - dans l'objet correspondant.

Prenons un exemple très simple - mouvement d'une particule en absence de forces extérieures. Son état $\{\mathbf{x},\mathbf{V}\}$ évolue selon la loi d'inertie (*la première loi de Newton*) : à l'absence de forces extérieures la vitesse reste constante. Le vecteur d'état au moment t_{i+1} peut être calculé comme :

$$\begin{aligned} &a_{i+1} = 0 \\ &V_{i+1} = V_i \\ &x_{i+1} = x_i + V_{i+1} \end{aligned}$$

La classe Particle encapsule les variables d'état de cet objet ainsi que son programme réactif :

```
class Particle {

   Position position;              //  state_1
   Speed speed;                    //  state_2
   Acceleration acceleration;      //  acceleration
   Value mass;
   Program program;

   public Particle(Position position, Speed speed, double m){

      this.position = position;
      this.speed = speed;
      this.mass = new Value(m);
      program = SC.loop(
                  SC.seq(
                    SC.action(new Inertia()),
```

```
                        SC.stop()));
   }
}
```

Si nous négligeons les forces d'interaction, le comportement d'une Particule consiste à maintenir sa vitesse.

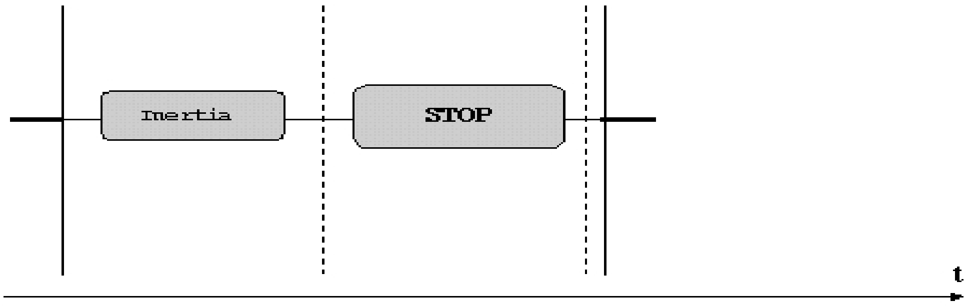

Figure 3: Programme réactif d'une Particule

L'action atomique **Inertia** implémente cette loi :

```
class Inertia implements JavaAction {

   Particle particle;
   Position position;

   public void execute(Object self, LocalVariables vs, ReactiveEngine e){

      particle = (Particle)self;
      position = particle.position;
      position.x += particle.speed.x;
      position.y += particle.speed.y;
   }
}
```

4.2 Mise en parallèle. Exécution synchrone.

Comme on l'avait déjà dit, nous pouvons à tout moment rajouter de nouveaux programmes réactifs dans la machine d'exécution avec sa méthode addProgram(). Ces programmes sont exécutés en parallèle de manière synchrone, c'est-à-dire, à chaque instant toutes les branches de l'arbre d'exécution sont activées, et la fin d'un instant ne survient que lorsque toutes les branches ont stoppé leur exécution pour l'instant courant (voir aussi le fonctionnement de l'opérateur **Merge** dans la Section 2).

Reprenons l'exemple précédent. Considérons **N** objets de type Particle, chacun ayant le même comportement **Inertia** décrit ci-dessus. Le code suivant rajoute leurs programmes dans une machine réactive et l'active en boucle infini :

```
class IdealGas {

   Particle particle;
```

```
    Machine machine;
    ...

    public void main(String args[]) {

       ...

       machine = SC.machine();
       for (int i=0;i<N;i++) {
          particle = new Particle(...); // initialization
          machine.addProgram(SC.cube(particle+i,particle,particle.program));
       }
       while (true) {machine.react();}
    }
 }
```

Remarquons qu'on ajoute dans la machine réactive un programme réactif tout en associant à ce programme un objet Java particulier (la méthode statique SC.cube() relie le programme d'une Particule à l'objet qui encasule les variables d'état de cette Particule). Ainsi l'action **Inertia** pourra accéder aux variables de la Particule par l'objet self qui lui va être transmis à runtime.

Le résultat de l'exécution de ce programme sera le suivant : à chaque instant *un pas* de la boucle **Loop** de *tout* programme réactif sera exécuté. Cela veut dire qu'à la fin de l'instant courant la variable **Position** de *tout* objet Particle (= l'état de toute particule) va être changée par l'action atomique **Inertia** associée à cet objet.

L'exécution synchrone définit ainsi *un temps commun* pour tous les objets. Du point de vue de la modélisation physique, cela signifie que *le changement de l'état du système constitué de N Particules après une intervalle dt est équivalent aux N changements particuliers (correspondant à chaque Particule) exprimés en termes d'actions atomiques, mises en parallèle et exécutés par la machine réactive.* Ce raisonnement permet ainsi de traiter un système physique de N particules comme un ensemble de N particules " indépendantes ". En généralisant cette idée on peut dire que, dans certains cas, l'exécution synchrone permet de décomposer un système physique en plusieurs éléments indépendants plus ou moins complexes (chacun possédant sa propre dynamique), mais absolument cohérents du point de vue temporel. Les avantages de cette technique sont discutés plus bas, mais on peut déjà remarquer sa transparence et sa forte modularité.

4.3 Interactions et influences extérieures

Le modèle d'*objets réactifs* enrichit la notion d'objet en y introduisant un certain axe temporel, commun pour tous les composants. Cela à son tour permet de considérer un système physique complexe comme un ensemble d'entités indépendantes. Les objets logiques correspondants sont donc plus facilement réutilisables. Un modèle physique construit à partir de tels composants élémentaires reste, de manière naturelle, cohérent et parfaitement synchronisé, ce qui n'est pas toujours facile à implémenter même avec une décomposition logique en *objets* en programmation orienté - objets.
La décomposition de modèles physiques selon les relations " *entité - environnement - influences* " ressemble beaucoup au modèle de la programmation réactive, où chaque composant réagit aux événements diffusés et maintient ainsi une interaction continue avec son environnement et les autres composants. De l'autre côté, la première chose qui peut caractériser une action externe qui influence une entité physique c'est la différence entre le comportement de cette entité en absence de l'influence et celui en présence de celle-ci. Cette différence fondamentale caractérise également la réponse aux événements dans *la programmation événementielle* dont la programmation réactive fait partie. Il est donc naturel d'essayer d'utiliser la diffusion instantanée des événements pour décrire les interactions entre des entités physiques.

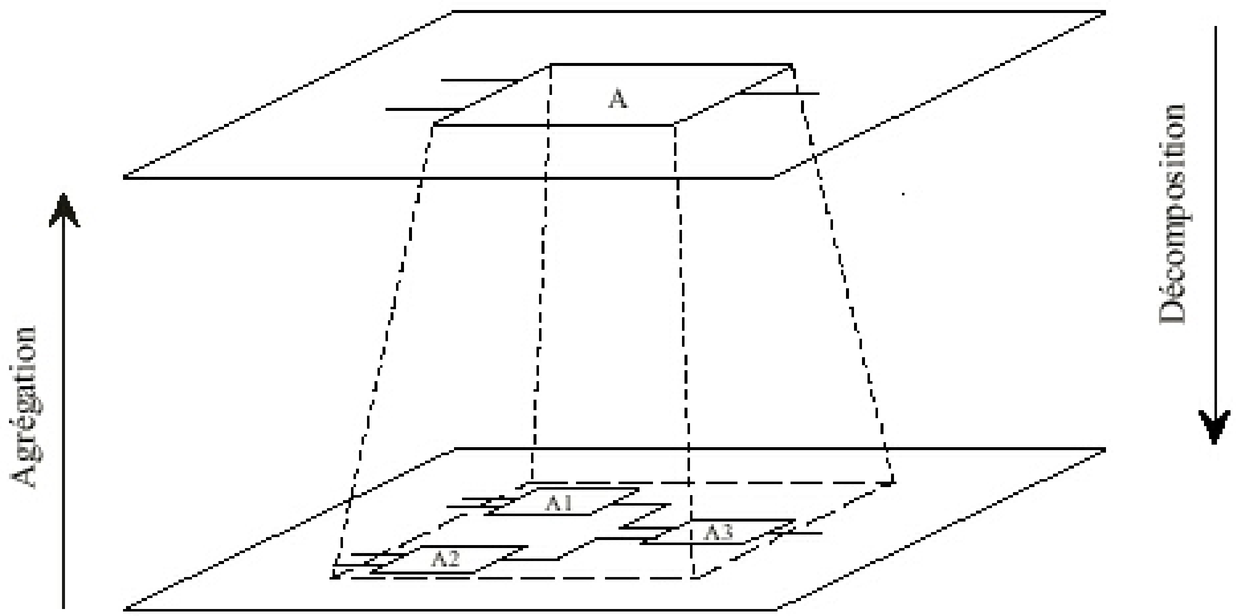

Figure 4: Décomposition d'un modèle physique

L'exemple suivant démontre comment (de façon très intuitive) on peut modéliser une influence d'une entité physique sur une autre à l'aide des événements. Imaginons un objet " Satellite " dont le comportement consiste à maintenir sa vitesse, c'est à dire, qu'à chaque instant une action atomique (**Inertia**) calcule une mise à jour des coordonnées du satellite en fonction de la vitesse. Supposons maintenant que le comportement de cet objet soit capable en parallèle de répondre à un événement " gravité ", c'est-à-dire de calculer la force d'attraction et de modifier ainsi la vitesse en fonction de l'accélération radiale reçu. On remarque que l'événement " gravité " doit contenir quelques valeurs nécessaires pour calculer cette accélération. *La mise en parallèle de ces deux comportements nous donne un satellite qui essaye de maintenir à chaque instant sa vitesse tout en s'exposant aux forces d'attraction qui modifient à chaque instant le vecteur de la vitesse.*

En rajoutant dans la machine réactive en parallèle un objet " Planète " qui émet à chaque instant l'événement " gravité " avec les valeurs appropriées, nous obtenons un modèle dynamique simplifié " Planète-Satellite ". Si plusieurs planètes sont ajoutées en parallèle dans le système, chaque planète génère à chaque instant une occurrence particulière de l'événement " gravité ". La primitive **Callback**, décrite ci-dessous, permet de répondre à chaque occurrence différente de l'événement " gravité " en appliquant systématiquement le traitement atomique simulant l'interaction gravitationnelle. On obtient ainsi naturellement la superposition physique des effets. Enfin, si plusieurs satellites sont présents dans le système, le principe de diffusion instantanée des événements de l'approche réactive garantit naturellement que chaque Satellite simule les mêmes interactions résultantes des différentes occurrences de l'événement " gravité " à chaque instant. [5]

Remarquons que l'objet " Satellite " est capable de fonctionner correctement en l'absence d'une " Planète " ou en sa présence. Ceci est aussi valable pour l'objet " Planète ". Cela veut dire que, malgré le fait que ces objets sont capables d'interagir, ils sont autonomes et indépendants et, donc, plus facilement réutilisables.

Cet exemple a été implémenté dans le cadre des expérimentations menées autour de la programmation réactive. Pourtant il est nécessaire de modifier ce modèle afin de rendre les résultats valables du point de vue physique. Dans les parties qui suivent, on verra sur quelques exemples simples, comment on peut généraliser un schéma permettant d'implémenter les interactions différentes et le traitement des influences de différents types (provenant de plusieurs objets differents), tout en préservant la validité du modèle physique et la stabilité numérique des résultats.

4.4 Influences extérieures comme les événements valués

Dans la version 3 des **SugarCubes**, les événements peuvent véhiculer des valeurs qui sont diffusées à l'ensemble des comportements en parallèle dans le système (voir aussi la Section 2). La primitive **Callback/Listener** est destinée au traitement des événements valués. La classe Callback permet de réagir à un événement émis plusieurs fois au cours du même instant avec des valeurs différentes. Chaque nouvelle occurrence d'un événement au cours d'un même instant provoque une réaction de cette primitive et déclenche l'exécution d'une action atomique Java chargée de traiter ses valeurs associées. Cette primitive est implementée en sorte qu'elle ne se termine qu'à la fin de l'instant, afin d'être sûr qu'aucun événement destiné à ce **Callback** particulier ne sera plus généré, c'est-à-dire, que toutes les occurrences de cet événement au cours de l'instant courant seront traitées [5].

Reprenons l'exemple précédent du système de N particules. On va créer un objet **BigMass** qui nous servira comme un centre de gravité. Supposons que l'influence de cet objet aux objets de type **Particle** est exprimé en termes des événements valués, c'est-à-dire qu'à chaque instant il va émettre l'événement " gravité " contenant les informations sur son positionnement actuel ainsi que le paramètre correspondant à sa masse.

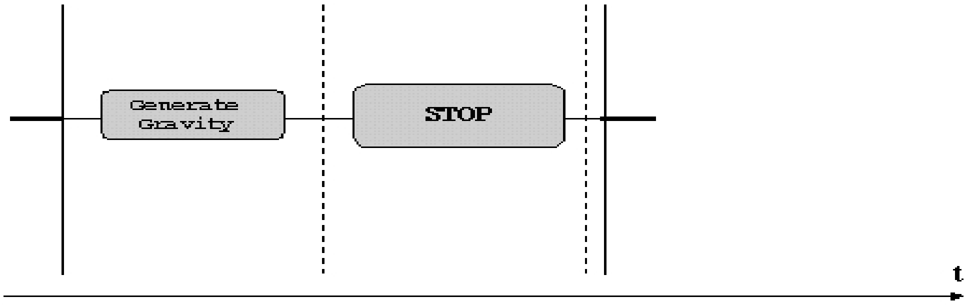

Figure 5: Programme réactif d'une Planète

```
class BigMass extends Particle{

   public BigMass (Position position, double m) {
      super(position, new Speed(0,0),m);
      program = SC.loop(
                  SC.seq(
                    SC.generate(GRAVITY,new  GenGravity()),
                    SC.stop()));
   }
}
```

Ici **GRAVITY** est le nom de l 'événement (" gravité "), et la classe GenGravity est une classe qui implémente l'interface JavaArgumentExpression et qui va véhiculer les informations sur les paramètres de l'influence (elle implémente la méthode evaluate() qui retourne à runtime l'évaluation des paramètres nécessaires – de sa masse et du positionnement actuel).
Le code suivant crée une planète et l'ajoute dans le système :

```
BigMass planete = new BigMass(...);
machine.addProgram(SC.cube(planete,planete,planete.program);
```

Maintenant modifions le programme associé à la classe Particle afin de permettre à la Particule de réagir aux événements “ gravité ”. On va rajouter également deux variables **potential** et **force** correspondant au potentiel d'interaction et à la force résultante reçue.

```
Potential potential = new GPotential();
Force force = new Force(0,0);
...
program =
   SC.loop(
     SC.seq(
       SC.callback(GRAVITY, new Collect(potential)),
       SC.seq(
         SC.action(new Compute()),
         SC.action(new Inertia()))
));
```

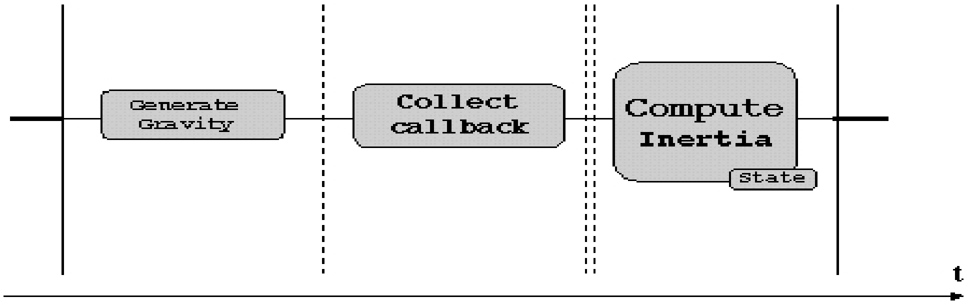

Figure 6: Programme réactif d'une Particule

A chaque occurrence de l'événement “ gravité ”, la méthode execute() de l'action atomique **Collect** est appelée. Cette action rajoute une valeur à la variable **force** selon le potentiel choisi. Cette variable est alors utilisée par l'action atomique **Compute** qui calcule le vecteur de la vitesse comme :

$$a_i = \frac{F_{recieved}}{m} \quad et \quad V_{i+1} = V_i + a_i \tag{8}$$

L'action atomique **Inertia** calcule alors la position suivante :

$$x_{i+1} = x_i + V_{i+1}$$

Ce qui est en fait l'expression “ intuitive ” de l'algorithme d'Euler-Cromer décrit dans la Section 3.
Pourtant l'algorithme d'Euler-Cromer n'est pas stable du point de vue énergétique. Il faut donc avoir un autre algorithme, plus efficace, pour obtenir des résultats stables. Dans le cas des particules interagissantes nous devons utiliser l'une des méthodes d'au moins deuxième ordre de précision afin de maintenir la conservation de l'énergie. La démonstration **Fields** dans la section 5 démontre l'application de l'algorithme de Verlet à un problème similaire.

4.5 Traitement des influences de différents types

Si nous mettons plusieurs **Callbacks** en parallèle nous obtenons le modèle suivant.
Imaginons, que chaque **Callback** réponds à une influence particulière (forces d'attraction, forces dissipatives de friction, liens élastiques, etc). A la fin de l'instant courant, tout Callback aura répondu à toutes les occurrences

d'événement qu'il " écoute ". Chacun d'entre eux aura donc rajouté une valeur à une variable commune (qui dans le cas des particules est la variable **force**) selon la nature de l'influence traitée.

Exemple : les influences dans la mécanique sont des forces.

$F_e = -kx$	action d'un lien elástique
$F_f = -CV (F_f = -CV^2)$	forces de frottement
$F_g \approx mg$	force d'attraction de la Terre proche de la surface
$F_a = G\frac{m_1 m_2}{{r_{12}}^2}$	forces d'interaction gravitationelle
$F_p = -\frac{dU}{dr}$	forces d'interaction dans d'autres champs potentiels
...	

Cette variable peut alors être utilisée pour calculer la vitesse et la position suivantes, tout comme dans (8).

Jusqu'à présent on négligeait les forces d'interaction. Pourtant, chaque particule possède sa propre masse et, donc, exerce son influence sur les autres particules. Pour implémenter les interactions entre les particules il faut rajouter à son programme réactif une action atomique qui va émettre l'événement " gravité " correspondant à la Particule. Cet événement va contenir les informations sur la position et la masse actuelles de la particule.

En réunissant tout, on obtient un schéma général de programmes réactifs des composants :

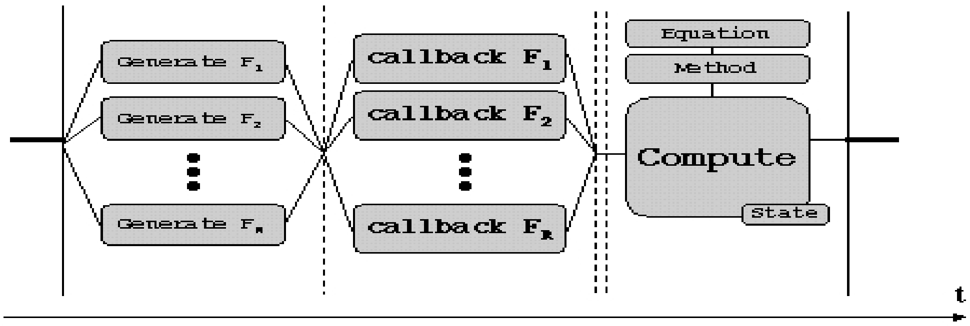

Figure 7: Programme réactif d'un composant : schéma général

Ainsi, le programme réactif d'un composant représente donc (en séquence) :

- Emission des influences F_k
- R **callbacks** mis en parallèle, chacun rajoute son apport (sa contribution) à une variable commune en fonction de son type.
- Calcul de l'état suivant avec l'un des algorithmes en prenant en compte les influences reçues.

Le schéma (Figure 7) permet d'établir des relations entre les entités physiques. En même temps toutes les informations reçues correspondent à l'état **actuel** de l'objet-émetteur, c'est-à-dire tout composant aura reçu tous les messages qui lui était destinés à la fin du même instant.

On voit, que ce schéma nous permet de traiter toutes les influences extérieures "simultanement". Remarquons que ce schéma possède une très grande modularité, tout en restant stable. Dans le contexte d'*expérimentations* *virtuelles* cela signifie que l'utilisateur peut exprimer ses idées de manière très simple et naturelle, en rajoutant ou en supprimant de la machine réactive des objets réactifs différents correspondant à telle ou telle entité physique et en rajoutant en parallèle à son programme réactif les proprietés nécessaires. Puisque les objets sont autonomes et ne sont pas censés avoir l'accès direct aux objets influençants, toute disparition de n'importe quelle influence sera traitée comme l'absence des événements du type correspondant, ce qui va correspondre à *une contribution nulle* du **callback** destiné à traiter ce type d'influence. Par exemple, dans le cas des liens élastiques, ca veut dire qu'on peut " couper " à tout moment n'importe quel lien (voir démonstration **Links** dans la Section 5). Du point de vue des outils de simulation, cela veut dire que ce schéma permet de simuler le modèle *vide.*

Quelques exemples d'application du schéma 7, qui ont été implémentés au cours de ce stage, sont décrits dans la section suivante.

5 Implémentation

5.1 Framework Icobj

Le framework **Icobj** est développé au sein du projet MIMOSA. Actuellement il existe plusieurs implémentations des Icobjs. L'implémentation utilisée pour la représentation graphique des simulations est celle en **Java** à base des **SugarCubes V3** [5].

En général *un Icobj* encapsule *un objet graphique (une apparence)*, et *un programme réactif.* L'utilisateur peut créer, activer et détruire chaque Icobj.

Les Icobjs existent et vivent dans **un workspace**, qui gère l'exécution des programmes réactifs dans la machine réactive, ainsi que, à chaque instant, les apparences des Icobjs dans un contexte graphique (celui de l'applet Java dans le cas présent (AWTWorkspace)).
Ajouter un icobj dans **le workspace** signifie ajouter son programme réactif dans la machine réactive en parallèle avec les programmes des autres icobjs. L'ajout d'un Icobj dans le workspace enregistre l'apparence graphique de l'Icobj dans la liste des images à montrer.

L'icobj **Constructor** permet de construire de nouveaux Icobj en copiant et en mettant en séquence ou en parallèle les comportements réactifs des autres Icobjs. **Le constructeur** représente donc *un moyen de programmation graphique.* La construction de programmes réactifs à partir des comportements élémentaires (“ go right ”, “ go left ”, “ bounce ”, “ attract ”, etc) permet de créer des systèmes assez complexes de manière intuitive.

Pour plus d'information sur les Icobjs voir [1].

5.2 Expérimentations virtuelles

Le framework Icobj peut servir comme une base de construction des expérimentations virtuelles dans le contexte d'*enseignement à distance.* L'utilisateur, par exemple, peut arranger toutes les icônes selon ses besoins, il peut déplacer n'importe quel objet. La construction graphique permet de créer des objets différents avec des propriétés et des comportements nécessaires. Ainsi, par exemple, tout programme considéré dans ce texte peut être construit à partir des comportements réactifs élémentaires, ce qui en fait permet de considérer ce système comme un “ block-diagram simulator ” de très bas niveau. Dans le cas le plus simple ces comportement élémentaires sont :

- la génération d'un événement
- les traitements des événements de différents types, les traitements *paramétrés* (par exemple, avec des potentiels différents)
- l'action atomique **Compute** correspondant à tel ou tel algorithme
- les comportements de base : action **Inertia**, action **Collide**, action **Bounce**
- etc

Bien sûr, ce package doit être etendu afin de rendre l'utilisation des icobjs plus commode et plus efficace pour des expérimentations virtuelles. Les tous premiers pas (comme, par exemple, les moyens de paramétrisation fine des composants) ont été réalisés au cours du travail sur l'implémentation du modèle, décrit dans ce texte. L'implémentation d'un package **Physobj** semble être un sujet très intéressant et prometteur.

5.3 Packages

La classe **Icobject** étends la classe **AWTIcobj** standard et encapsule quatre éléments principaux :

- Quelques variables communes comme, par exemple, le période d'échantillonnage
- Trois programmes réactif prédéfinis :
  - **initProgram** – programme d'initialisation de l'apparence graphique et des variables d'état (par défault SC.action(new InitImage(MyAWTImage image)))
  - **basicProgram** - le comportement " de travail " lui-même (SC.nothing() par défaut)
  - **controlProgram** - contrôle en parallèle (génération de l'événement " control me " lorsqu'on appuie sur un Icobject; déclenchement ou arrêt de l'Icobject lorsqu'il reçoit l'événement "start" ou "stop").
- La méthode driven() qui permet de combiner ces programmes en un comportement final
- Quelques actions atomiques basiques (les classes internes) – **InitImage**, **GeneratePosition**, **MyInertia**, **BounceOnBorders**, **Collide**

Ainsi n'importe quel objet dans une simulation est une extension particulière de la classe **Icobject**, typiquement avec un **basicProgram** particulier et une apparence (**MyAWTImage**) particulière.

5.3.1 Eléments de contrôle

Le Package **Control** inclue trois éléments de contrôle d'expérimentation.

Panneau de contrôle Les boutons "start" et "stop" servent à déclencher et à arrêter la simulation. L'appui sur le bouton produit une génération d'un événement dans la machine réactive. Puisque chaque Icobject est " enveloppé " par un programme de contrôle à base d'un **Until** (voir la section 2), la génération de l'événement correspondant provoque le déclenchement ou l'arrêt de tout programme parallèle à la fin du même instant.

Ainsi, après avoir ajouté un nouvel Icobject en appuyant sur un icobj-créateur correspondant, il faut *activer son programme de travail* avec le bouton " start ". Ceci est valable pour tout Icobject ajouté.
Le bouton "one step" ajoute dans la machine réactive le programme suivant :

```
SC.seq(
  SC.seq(
    SC.generate("start"),
    SC.stop()),
  SC.generate("stop"));
```

Ce qui provoque l'exécution d'un instant de chaque programme.

Enfin le champ textuel sert à modifier les paramètres de l'Icobject qui se trouve en focus du **Controller** (voir au-dessous).
Remarquons, que toute action de l'utilisateur ne rajoute le programme correspondant qu'à la fin de l'instant courant, lorsque tous les composants soient dans un état stable, ce qui garantit une certaine cohérence et sécurité d'interaction " utilisateur – ordinateur " au cours de la simulation.

Icobj Controller Cet Icobj est rajouté dans le workspace dés le début. Son programme réactif consiste à écouter l'événement “ control me ”, qui se produit lorsqu'on appuie sur un Icobject. Le **Controller** montre l'image symbolique (ou l'apparence graphique actuelle) de l'Icobject sous contrôle, et ajoute un petit trait rouge dans son apparence (Figure 8).
Nous pouvons à tout moment modifier les variables de l'Icobject sous contrôle (pour l'instant ce n'est valable que pour les variables de type simple **Value** contenant une seule variable de type **double**).En cliquant sur les noms correspondants, qui se trouvent au-dessous de la zone graphique du **Controller**, on peut choisir les variables à modifier et effectuer ainsi une paramétrisation “ fine ” de chaque élément du système.
Dans le cas où l'icobject sous contrôle possède un champ correspondant à la valeur de sa vitesse, le **Controller** permet de changer le vecteur de la vitesse simplement en cliquant dans sa zone graphique (Figure 8):

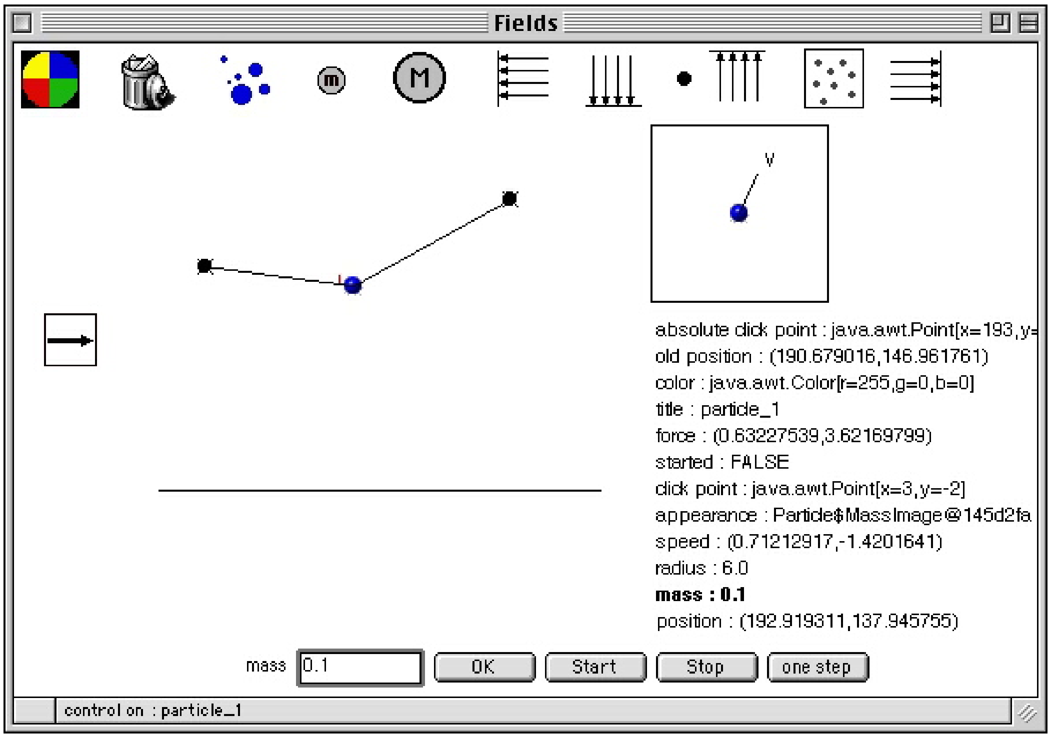

Figure 8: Contrôle de simulation

Icobject Connector Certains Icobjects peuvent être connectés les uns avec les autres. Le **Connector** permet de connecter de manière unidirectionnelle deux Icobjects. Dans ce cas-là, le premier objet ajoute dans son programme la génération de l'événement “ *action_name to object_name* ”, où *object_name* est le nom du deuxième objet. Le nom *action_name* correspond dans le programme du deuxième objet à un traitement particulier (i.e. à un **callback** particulier qui écoute ce canal événementiel). L'apparence graphique de l'icobject **Connector** est une fleche dans un carré (Figure 8).

5.3.2 Implémentation des algorithmes numériques

Quatre algorithmes qui ont été implémentés au cours de ce stage sont regroupés dans le package **Numerics** : Euler, Euler-Richardson, Velocity Verlet et Runge-Kutta 4. L'implémentation permet de changer facilement l'algorithme, ainsi que l'équation correspondante.
L'interface Method définit deux méthodes :

```
public interface Method {
   public void initialize(double stepSize);
   public boolean step();
}
```

L'interface Equation permet de définir un système d'équations différentielles du premier ordre sous forme de deux tableaux : state[] et rate[], où state[] contient le vecteur d'état courant et rate[] définie la vitesse de grandissement de chaque variable d'état, i.e. l'expression pour sa dérivé. Deux méthodes – getState() et getRate(double[] state, double[] returnRate) – servent à extraire ces deux vecteurs. L'objet **Equation** est transmis comme le paramètre lors de la création de l'objet implémentant un algorithme numérique.

Schématiquement un pas de calcul de l'algorithme d'Euler consiste alors à effectuer les opérations suivantes (forme vectorielle simplifié) :

```
// instant i
equation.state = equation.getState() + equation.getRate(state) * dt
return true
```

En appelant systématiquement la méthode **step()** on résout progressivement l'équation qui est transmise à l'algorithme lors de son initialisation.

Pourtant dans certains algorithmes l'état suivant se calcule en prenant aussi en compte les dérivés calculés dans des points intermédiaires. Le calcul d'un pas ne se termine qu'après un certain nombre d'estimations. On va supposer qu'une estimation occupe *un instant*. Alors le calcul d'un pas avec l'algorithme d'Euler occupe un instant, avec les algorithmes d'Euler-Richardson et de Verlet - deux instants, et enfin avec la méthode de Runge-Kutta - quatre instants.

Exemple : méthode générique **step()** de l'algorithme d'Euler-Richardson :

```
step++

// instant i

if step=1 {
state=equation.getState()
oldstate=arraycopy(state) //  save state Xi
state = oldstate + equation.getRate() * dt / 2 // calculate X_mid
// go receive influences at X_mid
return false
}

// instant i+1

if step=2 {
state=equation.getState();
state=oldstate + equation.getRate() * dt // finally compute Xi+1
                                         //with estimated rates at X_mid
step=0
```

```
return true // step is finished
}
```

Le même schéma est valable pour les autres algorithmes. Par exemple l'algorithme de Verlet se décompose en deux étapes suivantes :

```
// Equation must be in form that
// positions are odd state[] elements
// velocities are even state[] elements

step++

// instant i

if step=1 {
state=equation.getState()
// calculate positions using old velocities
state_odd = state_odd + equation.getRate() * dt +
            dt * dt * equation.getRate()
// partialy update velocities
state_even = state_even + equation.getRate() * dt / 2

return false                   // go receive influences at X_i+1
}

// instant i+1

if step=2 {
state=equation.getState()
// update velocities with rates at X_i+1
state_even=state_even + equation.getRate() * dt / 2
step=0
return true // step is finished
}
```

En général, on peut dire qu'un pas se compose d'un certain nombre de *micro-pas*. Dans cette implémentation *un micro-pas* d'algorithme correspond à *un instant*. La méthode **step()** change la fois suivante l'état du composant et retourne *true* si l'algorithme a terminé le calcul du pas courant entier et *false* si ce n'est pas encore le cas. Cette variable peut être utilisée, par exemple, pour l'affichage de l'icobject à la fin du pas de calcul.
L'activation consécutive de l'action atomique **Compute** produit à chaque fois l'exécution de la méthode **step()** de l'algorithme et fait avancer la résolution de l'équation. La méthode **step()** est une méthode générique qui effectue des estimations différentes en fonction du micro-pas courrant.

Exemple :

```
class Compute implements JavaAction {

   Equation newton = new Newton();
   Method method = new RK4(newton);
```

```
        boolean finished;
        ...

        public void execute(...) {
            ...
            finished = method.step();
            position.x=newton.state[1];
            position.y=newton.state[3];
            speed.x=newton.state[2];
            speed.y=newton.state[4];
            ...
        }
    }
```

En même temps, la méthode getRate() lors de son appel par l'algorithme retournera la valeur accumulée dans la variable qui corresponds à l'influence résultante, reçue lors de l'instant précédent.
Les valeurs trouvées lors de l'instant courant sont utilisées après pour l'émission des influences. Ainsi, tous les composants ont absolument le même " rythme " de calcul, i.e. ils effectuent tous les mêmes estimations au cours de chaque pas et s'échange des messages contenant les information correspondant aux résultats obtenus lors du même instant.

Bien évidemment tous les composants doivent utiliser le même algorithme et être synchronisés au niveau des étapes intermédiaires (par exemple, chaque composant peut attendre d'abord l'occurence de l'événement " step " avant de déclencher son **basicProgram**).

Un sujet de recherche très intéressant serait d'intégrer les algorithmes numériques directement dans le noyau réactif. La propriété " d'atomicité " en quelque sorte des pas d'algorithmes nécessite l'introduction de la notion de *macro-étape*, qui ressemble à celle *d'un instant* dans la programmation réactive et ne suppose pas d'intervention au cours du calcul de l'état courant. Ainsi, par exemple, l'ajout dans la machine réactive du programme d'un composant ne doit se faire qu'à la fin de la *macro-étape* courante. Des nouvelles primitives telles que **State**, **StateAction**, **Algorithme** permettraient de simplifier la construction des modèles de simulation et, en même temps, augmenter considérablement l'efficacité du simulateur.

5.4 Démonstrations

Toutes les démonstrations permettent de changer dynamiquement tous les paramètres du système ainsi que le positionnement de tout objet. Bien sûr, dans certains cas cette intervention de l'utilisateur ne se conjugue pas très bien avec le fonctionnement des algorithmes numériques, ce qui peut constituer le sujet d'étude d'intégration des formalismes des modèles de contrôle des systèmes continus et des modèles hybrides dans les expérimentations virtuelles à base d'icobjs.

5.4.1 Fields

L'un des exemples d'application du schéma (7) proposé dans la section précédente est la modélisation numérique du fameux problème " many body ", dont la solution analytique n'existe pas pour le moment actuel.

Mouvement dans un champ central. Modèle analytique. Dans le cas des champs centraux, l'énergie potentielle d'interaction entre deux particules ne dépends que de la distance entre les particules :

$$U_{12} = U(r_{12})$$

De même, la force d'interaction entre deux particules est une fonction de r_{12}

$$F_{int} = -\frac{dU(r_{12})}{r_{12}} \tag{9}$$

L'interaction se manifeste sous forme de l'action de deux forces : F_{12} – force éprouvée par la particule 1, et F_{21} – force éprouvée par la particule 2 :

$$\overrightarrow{F_{12}} = -\overrightarrow{F_{21}}$$

$$|F_{12}| = |F_{21}| = |F_{int}|$$

Dans le cas de plusieurs particules, l'énergie totale d'interaction pour une particule i est (cas simplifié):

$$U_i(r) = U(r_{i1}) + U(r_{i2}) + U(r_{i3}) + \dots$$

où $U(r_{ij})$ sont les membres correspondant aux interactions deux à deux de toutes les particules. La force totale peut alors être trouvée comme dans (9) [12].

L'équation du mouvement pour chaque particule s'écrit comme :

$$\frac{d^2\overrightarrow{x}}{dt^2} = \frac{\overrightarrow{F}}{m}$$

On peut appliquer donc l'une des méthodes numériques décrites dans la section 3. Le système correspondant d'équations est (cas 2-dimensionnel):

$$\begin{cases} \dot{x_0} = x_1 & \text{equation for position x-component} \\ \dot{x_1} = \frac{F_x}{m} & \text{equation for velocity x-component} \\ \dot{x_2} = x_3 & \text{equation for position y-component} \\ \dot{x_3} = \frac{F_y}{m} & \text{equation for velocity y-component} \end{cases}$$

La classe Newton est une classe interne de l'Icobject Particle et implémente l'inerface Equation :

```
public class Newton implements Equations {

   Value2 f;
   Value m;
   double[] state = new double[4];

   public double[] getState() {
      state[0] = getPosition().x;
      state[1] = getSpeed().x;
      state[2] = getPosition().y;
      state[3] = getSpeed().y;
      return state;
   }
   public void getRate(double[] state,double[] rate) {
      f = getValueOfField(FORCE); // accumulated influences
      m = getValueOfField(MASS);  // mass

      rate[0] = state[1];
      rate[1] = f.x/m.value;
      rate[2] = state[3];
```

```
        rate[3] = f.y/m.value;
      }
    }
```

Remarque : la classe Newton étant une classe interne de l'Icobject Particle a l'accès direct à ces variables.

Système " Planète - Satellites " On considère deux types d'icobjets : Planète et Satellite avec les programmes décrits dans la section 4 (Figure 5 et Figure 6 respectivement). Le Satellite a un seul callback qui répond aux événements " gravite " et utilise la classe GPotential (potentiel du type $\frac{1}{r}$) pour calculer la force résultante.

$$U_G = \frac{kMm}{r}; \quad \overrightarrow{F_G} = -\frac{kMm}{r^3}\overrightarrow{r}$$

Le calcul dunouvel état est effectué à l'aide de l'algorithme d'Euler :

```
    Newton newton = new Newton();
    Method method = new Euler(newton);
```

En cliquant sur les boutons correspondants, on crée et ajoute dans la machine réactive ces Iobjets. Puis, on initialise la vitesse du satellite et on démarre l'expérimentation avec le bouton "Start". Le résultat est une animation qui montre le mouvement du satellite et son orbite elliptique.

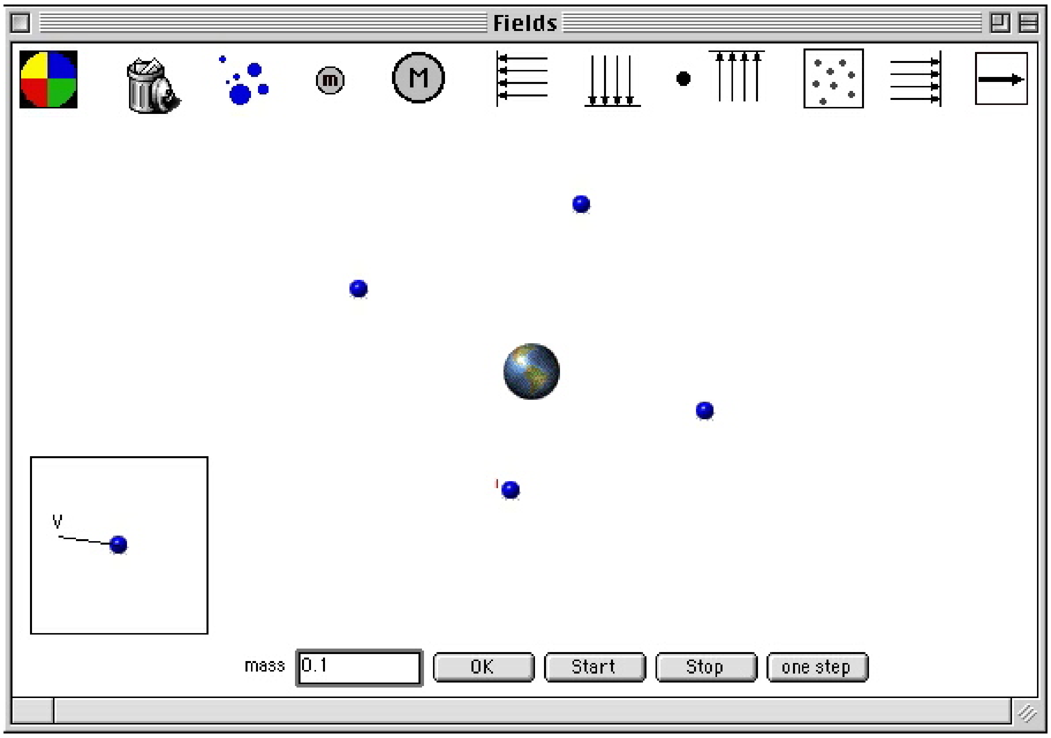

Figure 9: Système "Planète-Satellites"

(La suppression de la Planète, par exemple, enlève l'influence gravitationnelle, donc à partir de ce moment-là le satellite va maintenir la vitesse qu'il avait au moment de la disparition de la Planète et, donc, effectuer un mouvement rectiligne uniforme)

instant	*area*	*instant*	*area*
10	50.00000789877577	210	49.99997049150361
20	50.000010348925095	220	49.99997128694003
30	50.000014607761344	230	49.999976881051055
40	50.000020192776894	240	49.9999706371461
50	50.000015910328706	250	49.99998694496779
60	50.00000102696996	260	49.99995980418291
70	50.00001393549763	270	49.99998208097069
80	49.99998585175197	280	49.999993721433384
90	49.99999372372784	290	50.00001253174417
100	50.00000941583178	300	50.00001968530512
110	50.00003399684186	310	50.00001196730014
120	50.00003688330152	320	50.000004289159584
130	50.000018737281785	330	49.99998979047747
140	49.99999224711348	340	49.99999128771907
150	49.99998186131791	350	49.99999367949761
160	49.999968235676796	360	49.99999974934812
170	49.999969321617144	370	49.99998698839946
180	49.99997726088585	380	49.9999911049798
190	49.99998370931858	390	49.99999211690654
200	49.999981054621685	400	49.99999683052841

Figure 10: Vérification par la seconde loi de Kepler

Remarquons que l'application de l'algorithme d'Euler ou Euler-Cromer rend le modèle instable, ce qui s'exprime en précession lente de l'orbite du satellite. L'application de l'algorithme de Verlet enlève cette instabilité et on peut voir que le satellite revient à sa position initiale après avoir fait une spire.

```
Method method = new VerletV(newton);
```

Pour vérifier le bon fonctionnement de l'algorithme on peut se baser, par exemple, sur la deuxième loi de Kepler (le Satellite doit balayer des aires égales par un intervalle de temps donné) et verifier la conservation du produit vectoriel $\frac{1}{2}dt[\overrightarrow{r} \times \overrightarrow{V}] = \frac{1}{2}dt(r_xV_y - r_yV_x)$, où r est la distance entre la Planète et le Satellite et V est la vitesse du Satellite. Les résultats de cette vérification pour le système "Une Planète - Un Satellite" avec le pas de calcul égal à 0.05 sont donnés dans le tableau (Figure 10). Chaques dix instants on mésure l'aire balayée. Le tableau montre les resultats au cours de deux spires (environs 400 instants). On voie que cette valeur reste constante au cinquième chiffre près.

On peut également rajouter de nouveaux satellites et même de nouvelles planètes (Figure 9). Pourtant il faut choisir des valeurs appropriées pour les masses des satellites afin de diminuer l'interaction entre eux et laisser donc essentiellement une seule interaction – la réponse à l'influence de la Planète.

En paramétrisant les valeurs de masse des satellites, on peut construire un modèle de système solaire, par exemple, un modèle " Satellite – Lune - Terre " ou bien un modèle simple d'atome de l'helium.

Potentiels complexes En rajoutant en parallèle plusieurs callbacks qui traitent les événements " gravité ", mais avec des potentiels différents, on obtient un traitement des potentiels complexes.

$$U_{12} = U_1(r_{12}) + U_2(r_{12}) + \dots$$

Par exemple, on peut modéliser les forces répulsives qui ont leur effet aux distances proches en rajoutant au **callback(GPotential)** en parallèle un **callback(MPotential)** simulant le comportement de la particule dans *le potentiel de Morse* [12] (Figure 11):

$$U_M(r) = e^{-2r} - 2e^{-r}$$

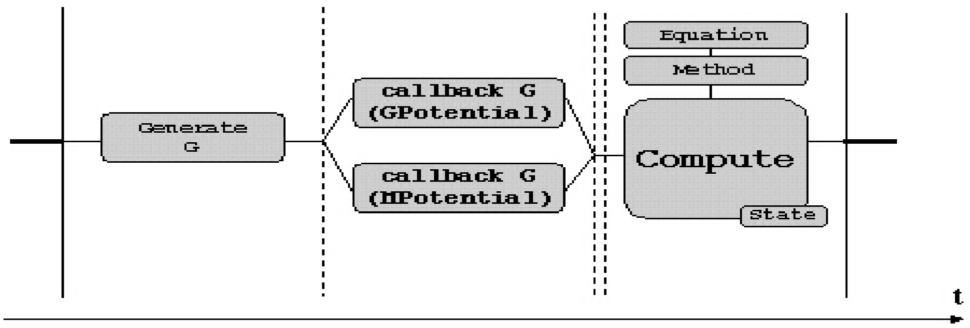

Figure 11: Ajout du traitement d'encore un potentiel

Ainsi deux particules vont s'attirer aux longues distances et se repousser aux courtes distances et, donc, avoir des mouvements d'oscillation.

Molécules simples Gaz, liquides et solides sont des exemples de systèmes contenant plusieurs particules mutuellement interagissant. Sachant les lois physiques au niveau microscopique, on peut essayer d'expliquer le comportement d'un tel système (ou de systèmes plus complexes, tels que les polymères et les protéines) au niveau macroscopique.

La méthode de *dynamique moléculaire* vise à modéliser ce genre de système en effectuant le calcul des forces intermoléculaires et des trajectoires de chaque particule. Les trajectoires et les valeurs moyennes le long de ces trajectoires sont alors utilisées par *la mécanique statistique* pour expliquer le comportement du système au niveau macroscopique.

Supposons que les molécules sont sphériques et chimiquement inertes. Dans le cas d'un liquide simple, on peut utiliser *le potentiel de Lennard-Jones* pour la modélisation des interactions intermoléculaires [12]:

$$U_{LJ}(r) = 4(\frac{1}{r^{12}} - \frac{1}{r^6})$$

La classe *LJPotential* implémente le calcul de la force résultante. Maintenant ajoutons dans le programme d'une Particle un seul callback avec *LJPotential* comme le paramètre de potentiel.
L'ajout dans la machine réactive d'un ensemble de N objets Particle permet de simuler ainsi des liquides simples (Figure 12). (Bien évidement dans des cas réels la simulation suppose la présence d'un très grand nombre de particules (à partir de 10^7))

Forces dissipatives L'icobjet **Air** simule la présence d'un environnement qui engendre une force dissipative lorsque la particule bouge dans cet environnement. Cette force est proportionnelle à la vitesse de la particule et dépends aussi de sa taille et de la densité de l'environnement. Cette dernière est émise à chaque instant par l'icobject **Air** (Figure 13).
Modifions le programme de la Particule et ajoutons encore un callback qui va traiter les événements provenant des objets **Air** (Figure 14).
Reprenons l'expérimentation "Planète-Satellites". En rajoutant un ou plusieurs icobjects **Air** dans la machine par sa création dans le workspace (en appuyant sur l'icobj de création correspondant), on peut observer un phénomène bien connu – la chute graduelle des satellites dans l'atmosphère de la Terre (Figure 15).

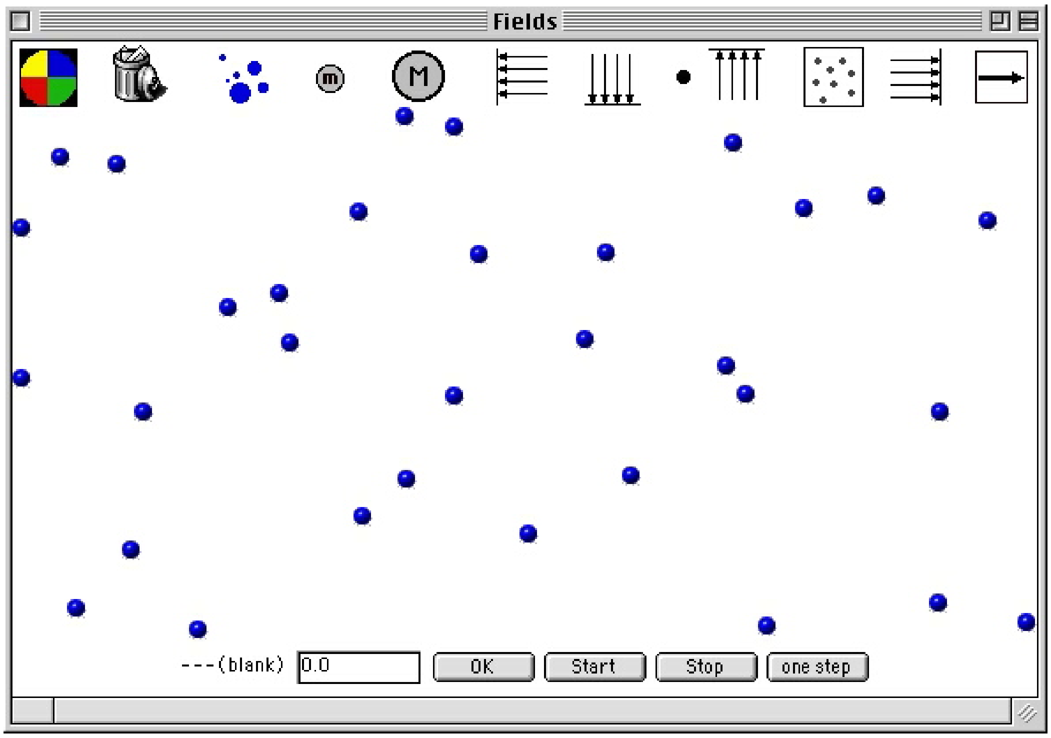

Figure 12: Dynamique moléculaire. Potentiel de Leonard-Jones

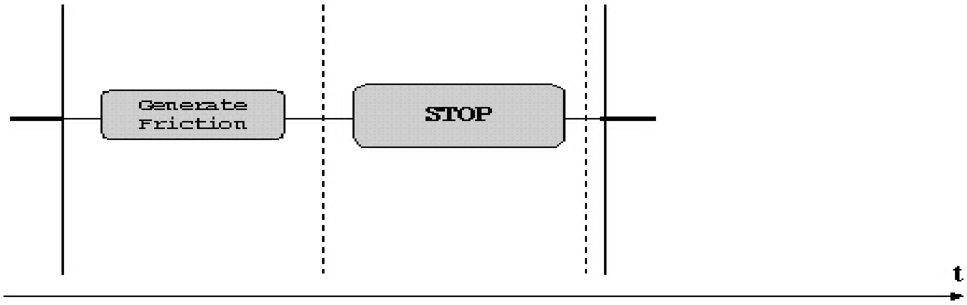

Figure 13: Programme réactif de l'icobject Air

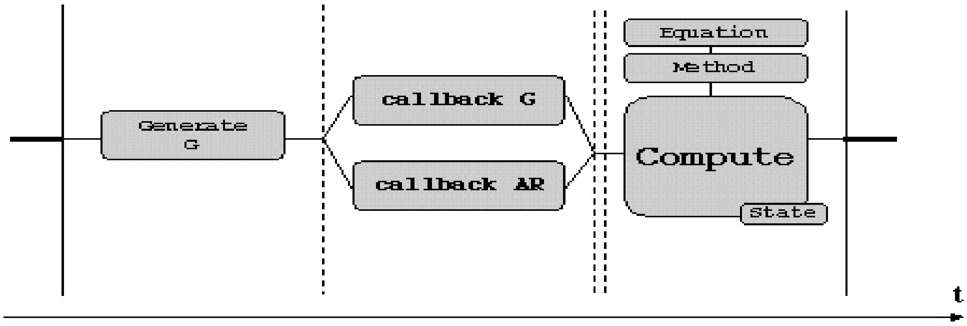

Figure 14: Programme réactif d'une Particule

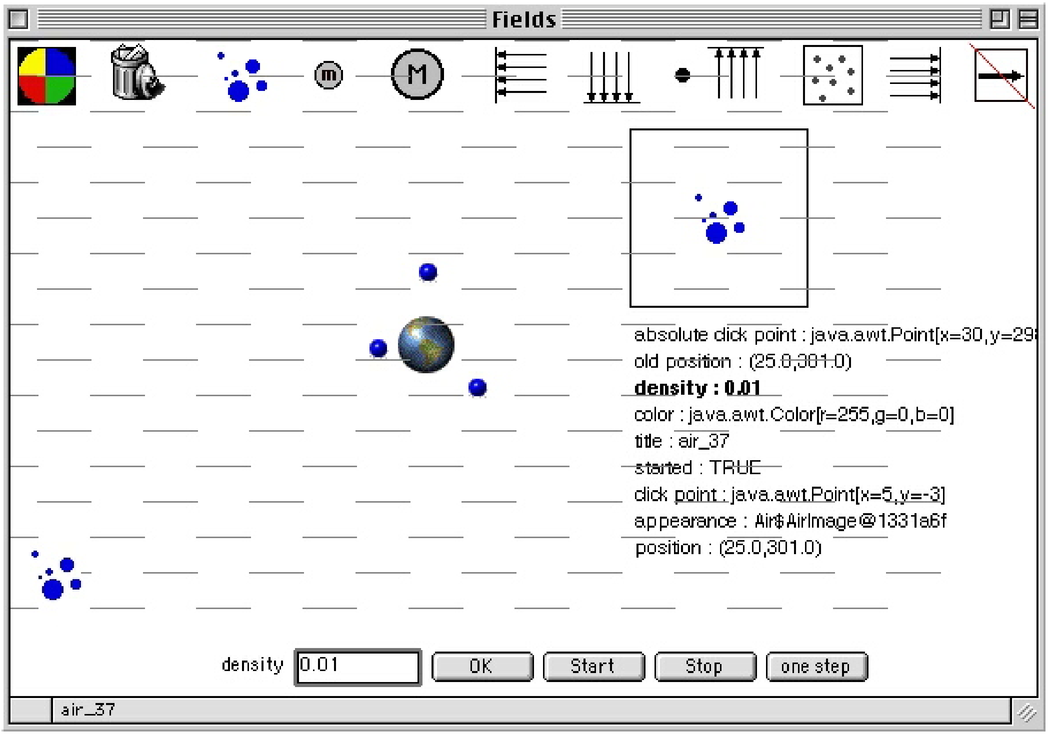

Figure 15: Chute des satellites

5.4.2 Links

Ajoutons au programme de la Particule en parallèle un callback qui va traiter les influences des liens élastiques. Ce callback va " écouter " les occurrences de l'événement " *elastic_to_object_myname* " (Figure 16).

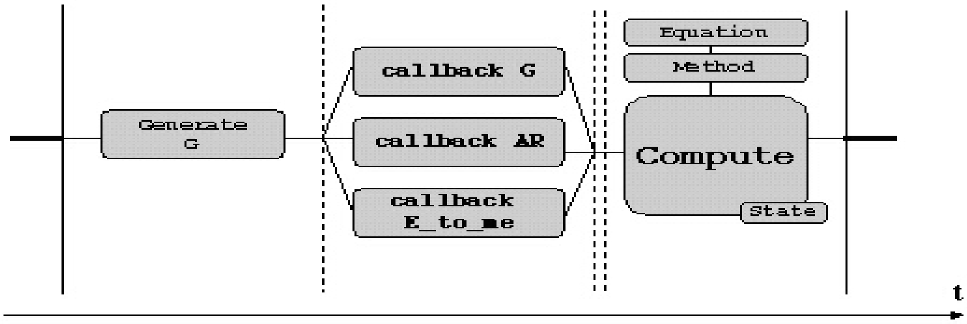

Figure 16: Programme réactif d'une Particule

Le callback " *elastic_to_object_myname* " va rajouter à la variable **force** sa contribution ($F_e = -kx$).
L'icobject **Elastic** implémente le comportement d'un point fixe avec un lien élastique. Son programme réactif consiste à emèttre à chaque instant les événements destinés aux particules attachées au point. L'icobject **Connector** permet d'y rajouter de nouvelles particules en créant de nouveaux canaux de communication entre les particules et le point (en ajoutant la génération d'un événement " *elastic_to_objet_name* " dans le programme **d'Elastic**) (Figure 17). Cet événement contient les information sur le positionnement du point fixe ainsi que la valeur de paramètre k.

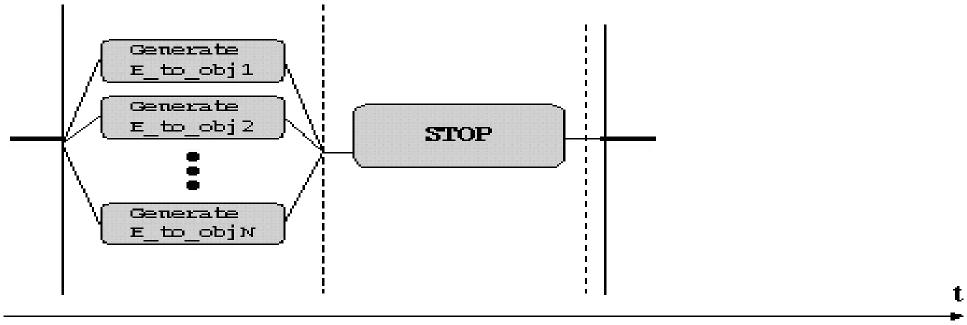

Figure 17: Programme réactif de l'icobject Elastic

Les particules effectuent des mouvements périodiques. On peut trouver un équilibre statique en rajoutant l'action d'un ou de plusieurs icobjects **Air** (Figure 18).

Considérons la configuration suivante (Figure 18):

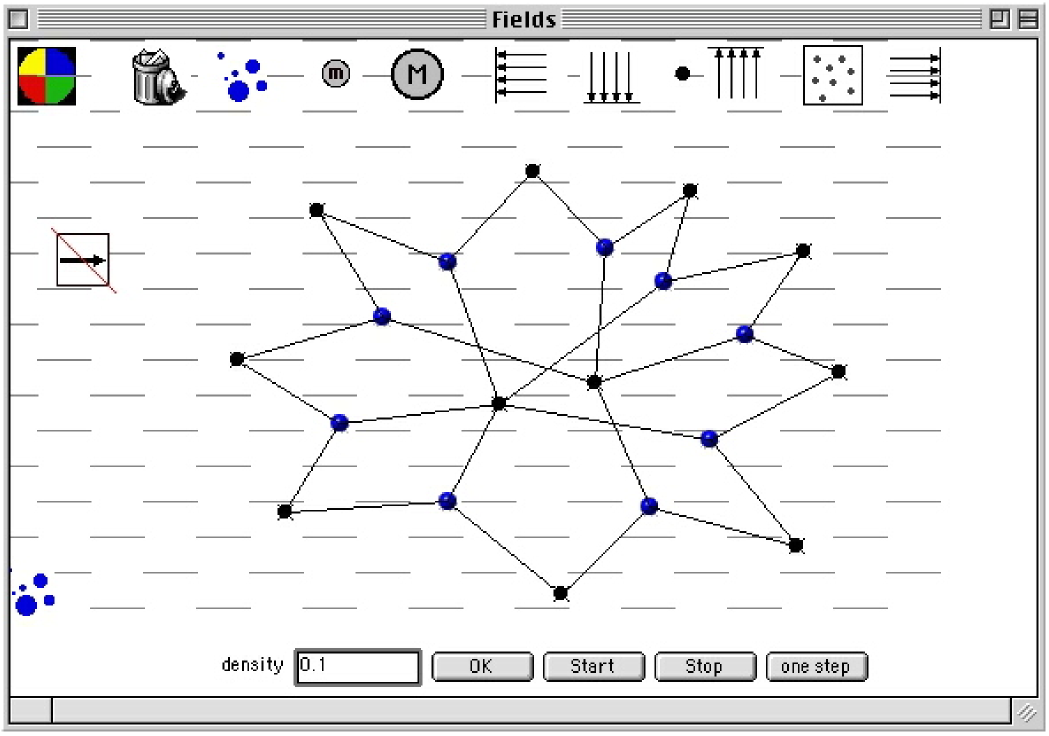

Figure 18: Equilibre statique

Rappelons, que la suppression d'un point fixe signifie une disparition d'influence correspondante. Très vite un nouvel équilibre statique s'établit (Figure 19).

5.4.3 Réservoirs

Un autre exemple d'application du schéma général (7) décrit dans la section 4 est le modèle des réservoirs. C'est un modèle simplifié, qui néglige certains détails, mais qui permet tout de même de démontrer la technique de décomposition.

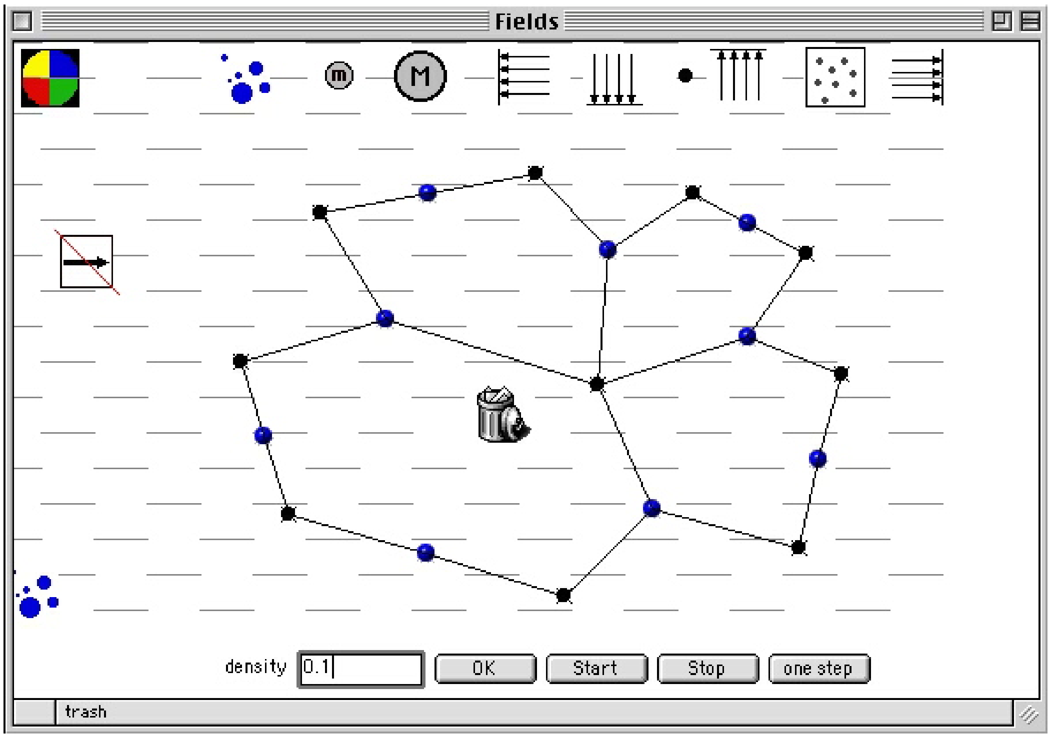

Figure 19: Un point fixe central est coupé

Modèle analytique Un **Reservoir** représente un bac de stockage avec un tuyau de sortie infiniment fin au-dessous (Figure 20):

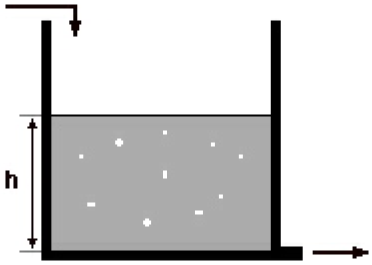

Figure 20: Un Reservoir

Le processus d'écoulement du liquide est décrit par le modèle suivant :

$$S\frac{dh}{dt} = q_{in} - q_{out}$$

$$q_{out} = C_\nu\sqrt{\frac{\rho g}{g_c}h}$$

Le débit de sortie q_{out} est exprimé par la loi de Bernouilli :

$$q_{out} = C_\nu\sqrt{P - P_a}$$

où C_ν est un coefficient dépendant de la vanne, P est la pression au fond du bac et P_a est la pression ambiante à l'extérieur du bac. En supposant que $P = P_a + \frac{\rho g h}{g_c}$, où ρ est la masse volumique du liquide, g est l'accélération de la gravité et g_c est un coefficient, on obtient ainsi l'expression pour le débit de sortie q_{out}.

$$q_{out} = C_\nu \sqrt{\frac{\rho g}{g_c} h} \tag{10}$$

Ce modèle est non-linéaire même dans le cas où q_{in} est nul. Or si q_{in} est non nul et s'il possède sa propre dynamique $q_{in} = q_{in}(t)$, la complexité du modèle augmente très vite. Tel est le cas, par exemple, du modèle de plusieurs réservoirs communicants.

Démonstration L'icobject **Reservoir** a le comportement réactif suivant (Figure 21):

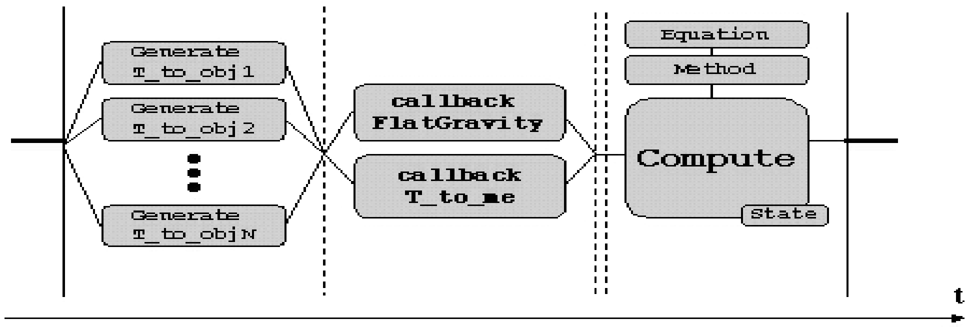

Figure 21: Programme réactif de l'icobject Reservoir

Ainsi, le débit de sortie q_{out} dont la valeur est calculée comme (10), est divisé par le nombre de Reservoirs-récepteurs et dirigé vers eux sous forme d'événements.

L'Icobject **Perturbator** encapsule le comportement d'une pompe. Il génère un débit constant ou non-constant (**régime constant** ou **régime sinusoïdal**) et le dirige vers les **Reservoirs** connectés (Figure 22):

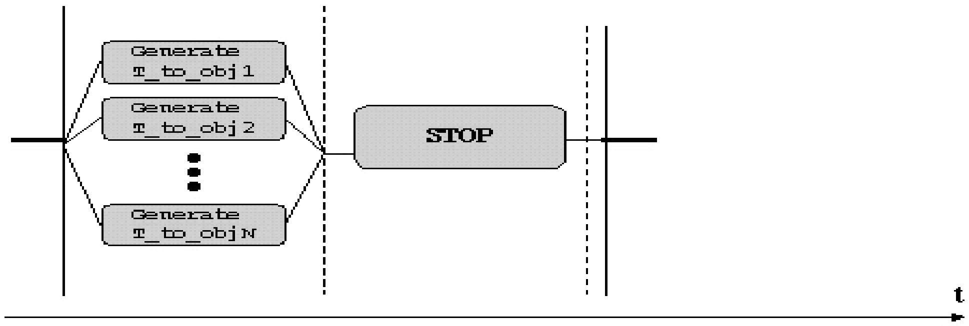

Figure 22: Programme réactif de l'icobject Perturbator

La classe Throughput encapsule l'équation de débit du Reservoir :

```
public class Throughput implements Equations {

   double[] state = new double[2];
   public double[] getState() {
```

```
        state[0]=h.value; // height of liquide, state[1] is a time counter
        return state;
    }
    public void getRate(double[] state,double[] rate) {

        rate[0]=(q_in.value-q_out.value)/S.value;
        rate[1]=1;
    }
}
```

Remarque : la classe Throughput étant une classe interne de l'Icobject Reservoir a l'accès direct aux variables q_{in} et q_{out} de l'icobject **Reservoir**.
En appliquant l'une des méthodes numériques on résout à chaque pas du temps l'équation de débit.

```
Througput throughput = new Throughput();
Method method = new EulerRichardson(throughput);

boolean finished;
...

class Compute implements JavaAction {

    public void execute ( ... ) {
        finished=method.step();
        h.value = throughput.state[0];
        ...
    }
}
```

La démonstration montre comment on peut décomposer un système de réservoirs en plusieurs objets interagissant et évoluant au cours du temps. La simulation permet de créer un système, de paramétrer les débits des pompes (constant ou sinusoïdal), de paramétrer les caractéristiques des réservoirs et de connecter les composants (Figure 23):
Un seul Reservoir en présence de la gravité se vide graduellement après un certain temps t (Figure 24).
Naturellement, la connexion d'un Reservoir à lui-même ne produira rien.
L'expérimentation la plus simple consiste à établir deux connexions unidirectionnelles : " Reservoir1-Reservoir2 " et " Reservoir2-Reservoir1 ". Supposons que le modèle néglige le travail nécessaire pour monter une quantité de liquide en hauteur h (tuyau infiniment fin). Dans ce cas l'ajout d'une quantité V de liquide dans un Reservoir provoquera un établissement graduel d'équilibre dans le système après un certain temps (Figure 25). (Bien sur, la destruction d'un d'entre eux provoquera le vidage graduel de celui qui reste.) Le même phénomène se démontre dans un système composé de trois Reservoirs.

Considérons maintenant un système composé d'un Reservoir et d'une pompe avec un débit constant. On observe que le Reservoir commence à se remplir graduellement et au bout d'un certain temps il trouve un point d'équilibre.

Ce n'est pas le cas si on change le régime de la pompe (régime sinusoïdal). La hauteur du liquide commence à osciler. Pourtant il suffit de connecter encore un réservoir pour observer la diminution des oscillations du débit sur la sortie du premier Reservoir (Figure 26).

Cela veut dire que l'ajout d'un réservoir intermédiaire permet de stabiliser considérablement un débit instable (le débit d'une pompe avec un régime $q(t)$).

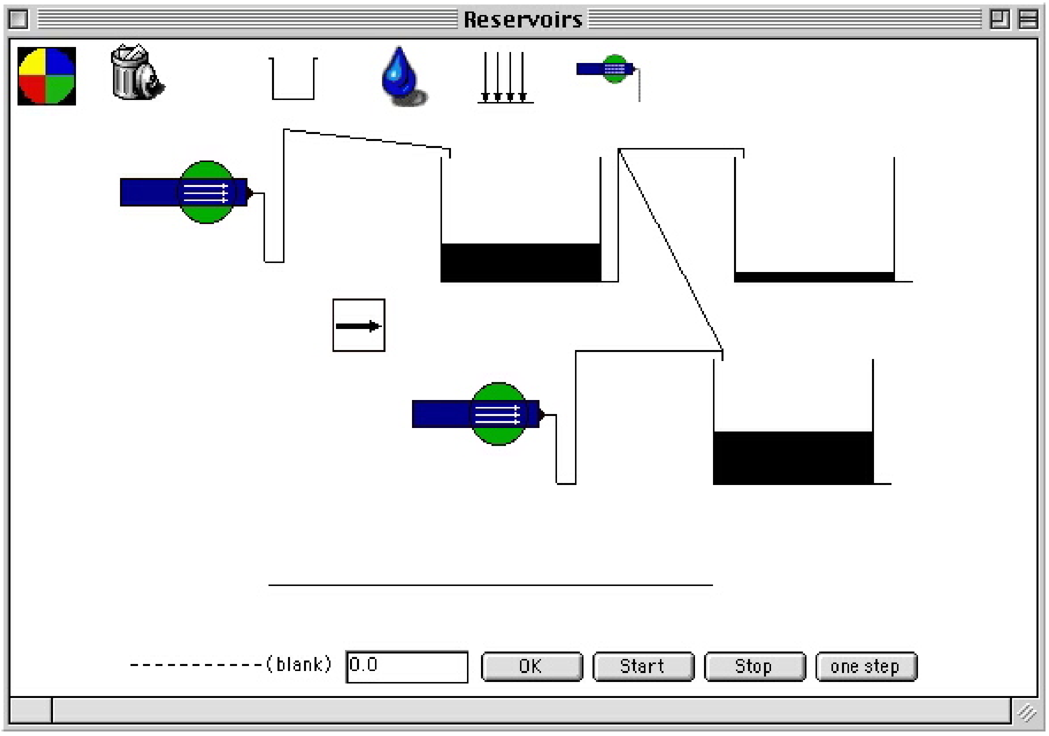

Figure 23: Connexion des composants

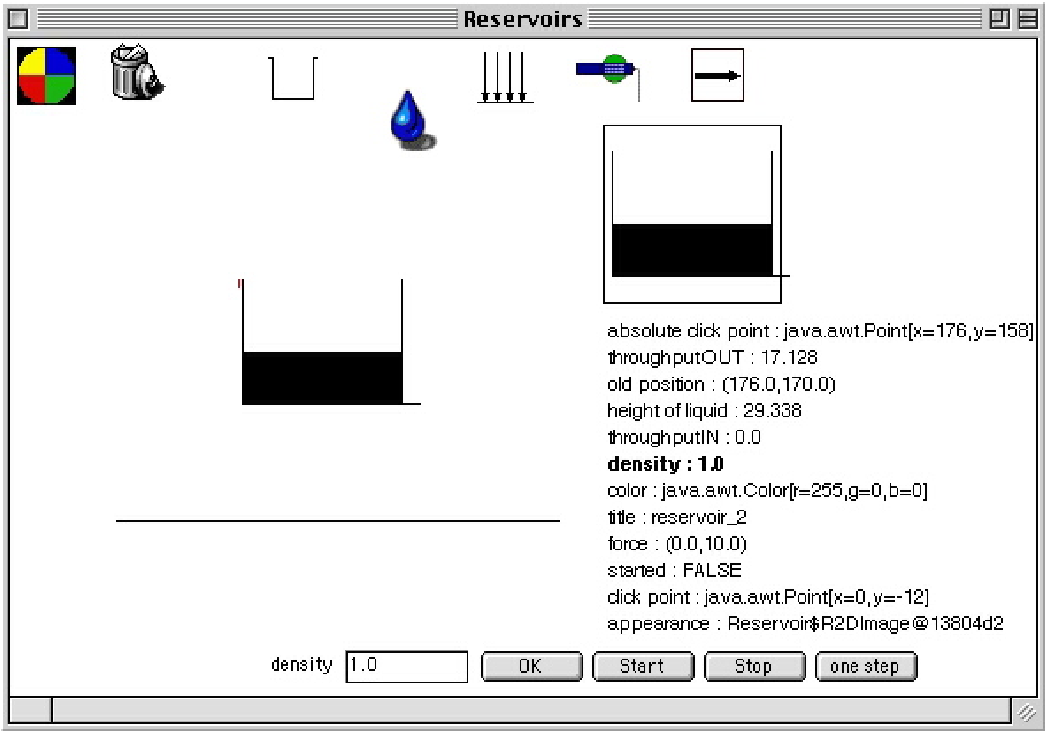

Figure 24: Ecoulement du liquide

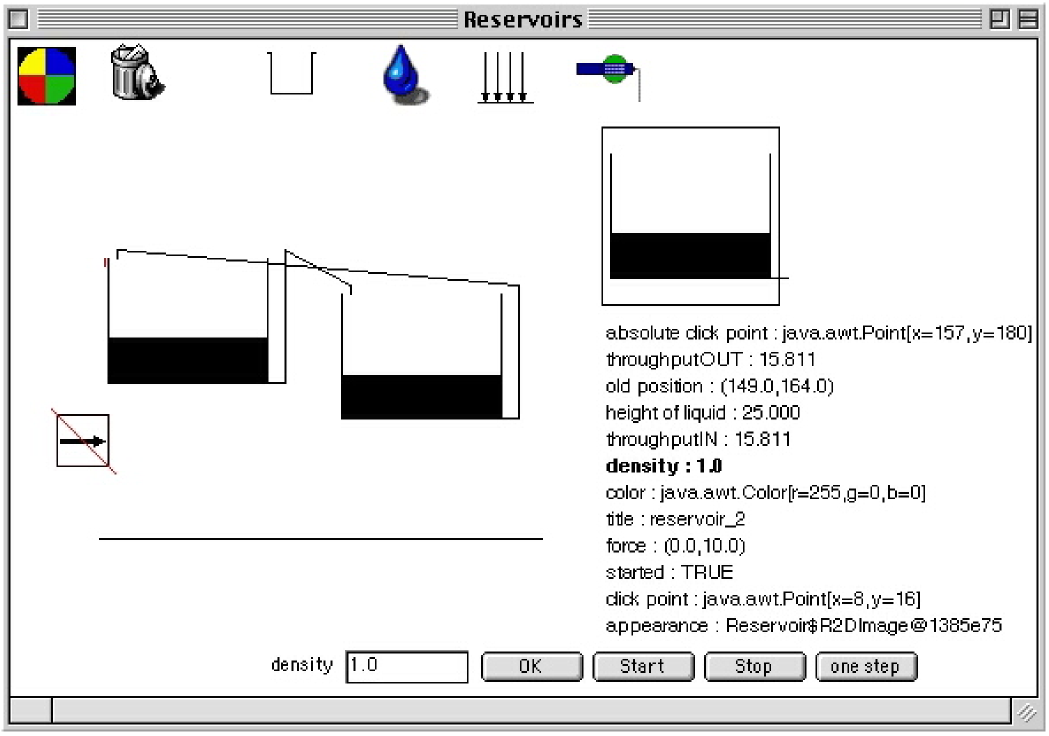

Figure 25: Deux Reservoirs : équilibre statique

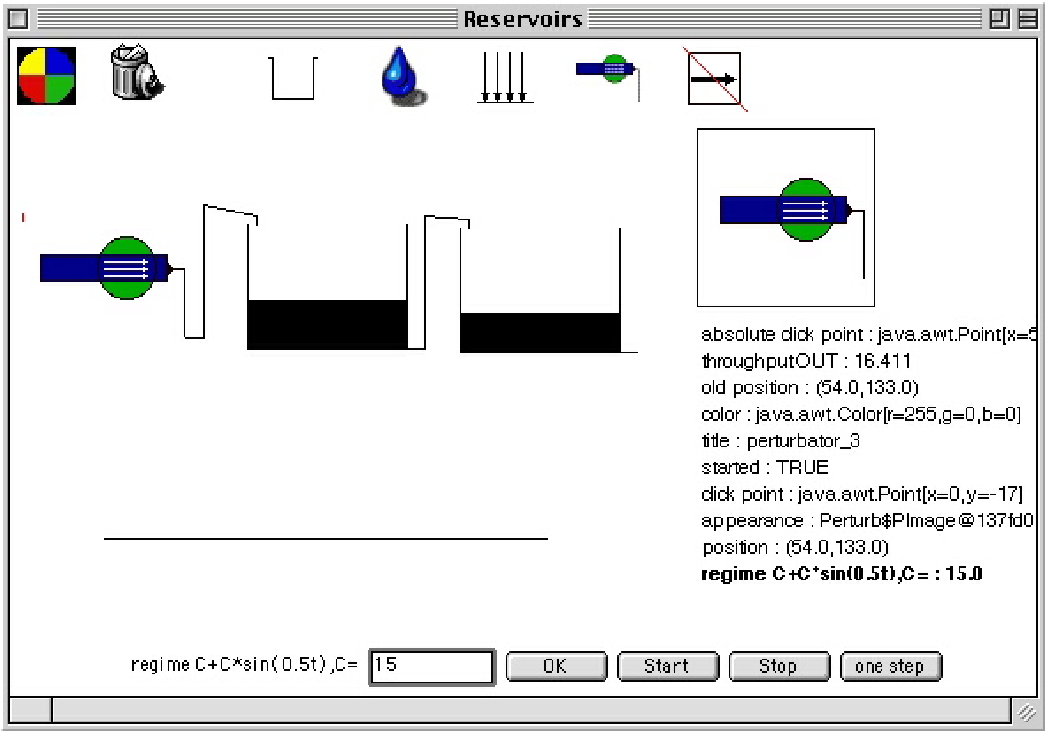

Figure 26: Diminution des oscillations de débit

6 Conclusion

Ce travail a démontré la possibilité d'application de la programmation réactive à la modélisation des phénomènes physiques. On peut mentionner quelques résultats principaux :

6.1 Résultats du travail

L'approche réactive offre les moyens souples et efficaces pour la gestion de composants parallèles. Le temps discret explicite (*les instants*), commun pour tous les composants, permet de traiter les éléments du système physique indépendament les uns des autres, ce qui rend les modèles plus modulaires et réutilisables. Par exemple, on peut à tout moment rajouter de nouveaux composants dans la simulation aussi bien que retirer des objets et leurs influences du système.

L'introduction d'une horloge explicite dans les objets rend les composants plus modulaires. On peut dire que la décomposition en objets en programmation orienté-objets s'applique sur l'état (les données) des objets et la programmation réactive s'applique sur leurs comportements (on a la modularité au niveau des comportements et non seulement au niveau des variables d'état).

L'approche réactive vue comme un moteur de calcul de *point fixe* (au cours de l'instant) semble s'appliquer de façon naturelle à l'implémentation des interactions entre objets physiques. On utilise la diffusion instantanée d'événements pour établir les connexions entre les composants. Ainsi, par exemple, une connexion bidirectionnelle peut être implémentée comme deux connexions unidirectionnelles. Le modèle événementiel de l'approche réactive garantit naturellement la synchronisation d'une telle interaction réciproque. (L'un des exemples les plus démonstratifs est l'implémentation absolument naturelle de la troisième loi de Newton – action est égale à la réaction).

Le schéma (7)proposé dans la section 4 permet d'implémenter les relations entre les objets, ainsi que de " construire " le comportement d'un objet physique à partir de comportements élémentaires en les mettant en parallèle. Ce modèle peut servir comme une base pour un " block diagram simulator " de très bas niveau (modularité au niveau de la construction des comportements).

Les entrées et les sorties des modules réactifs ainsi créées sont synchronisés de manière naturelle, ce qui permet d'appliquer les algorithmes itératifs " classiques " pour le calcul d'états de chaque composant et d'avoir donc des résultats numériques très précis. Il faut mentionner que l'application de ce schéma ne se limite pas à l'utilisation des algorithmes que j'ai implémentés au cours de ce travail; ceci est valable pour tous les algorithmes numériques standards.

Les démonstrations à base du framework Icobjs étendu et des SugarCubes V3 implémentent les principes mentionnés au-dessus. Le même schéma permet de simuler plusieurs phénomènes physiques différents qui sont décrits par les équations différentielles ordinaires du premier ordre en paramétrisant leurs conditions initiales avec les moyens de contrôle des simulations.

6.2 Problèmes à resoudre et solutions possibles

Cette étude sur l'application des principes de l'approche réactive en modélisation n'etait que le premier pas et nécessite donc une recherche complémentaire. Parmi les problèmes qui restent à resoudre on peut mentionner :

La nécessité d'un syntaxe plus simple pour l'écriture des programmes des composants. Le language *Rejo* [1] avec des règles syntaxiques simplifiés qui est implémenté actuellement au sein du projet MIMOSA peut être très utile dans ce cadre.

La construction graphiques des comportements des composants. L'icobj Constructor permet de construire graphiquement un comportement complexe à partir des comportements simples en les mettant en séquence ou en parallèle. Pourtant, les programmes de chaque composant doivent tous avoir une structure simple prédefinie afin d'être cohérents avec le schéma de fonctionnement des algorithmes numériques. Il faut donc avoir un constructeur capable de rajouter des comportements en parallèle seulement à certains endroits du programme du composant ainsi que capable de copier non seulement les programmes mais aussi les variables d'état.

L'aspect interaction " homme – machine " dans les expérimentations virtuelles. L'intervention " trop forte " de l'utilisateur au cours du pas de calcul peut provoquer des instabilités numériques. L'une des solutions possibles est la mise en œuvre d'une version particulière du framework Icobjs (Physobjs) qui soit capable de traiter corrctement les commandes de l'utilisateur et en même temps d'offrir un ensemble de comportements basiques. Ce nouveau framework permettra, par exemple, de spécifier de manière plus directe les notions d'apparence graphique du composant et de son état.

6.3 Axes futurs de recherche

Simulateur réactif. La technique de décomposition à base de l'approche réactive permet d'implémenter un simulateur de systèmes physiques complexes, continus et hybrides. Les deux caractéristiques principales de ce simulateur sont : simplicité de construction de modèles et forte modularité des composants.

Machines numériques. Ce travail a démontré la possibilité d'application des algorithmes numériques classiques dans le cadre des simulations parallèles. Ainsi, n'importe quel algorithme d'integration peut être implémenté avec quelques petits changements. Le sujet extremement intéressant serait d'implanter les techniques numériques modernes directement dans le noyau réactif. Il est possible de suppléer les primitives de base de l'approche réactive par des nouvelles primitives telles que **State**, **StateAction**, **Algorithme**, par exemple. Cette approche permettra d'augmenter significativement l'efficacité du simulateur ainsi que de se débarasser de la plupart des problèmes mentionnés au-dessus.

Application aux problèmes de la physique. De nombreux systèmes de physique se décomposent naturellement en sous-systèmes en interaction. La technique proposée est particulierement adaptée à la résolution de ce type de problème, en permettant une construction modulaire et progressive d'un modèle global. Parmi les problèmes qui semblent être intéressants de résoudre par ces techniques on peut noter les problèmes de mécanique rationnelle à un grand nombre de corps (" many-body problem ") et les interactions physico-chimiques, soit à l'echelle sub-atomique (par exemple, collision de deux noyaux atomiques), soit à l'echelle intermoléculaire.

Interfaçage avec d'autres simulateurs tels que Matlab et son extension Simulink. Le language Simulink permet de créer des modules " ouverts " et interfacer une simulation avec d'autres logiciels. Ce fait permettra d'implémenter des blocs particuliers de Simulink qui seront à "l'écoute" de l'horloge du simulateur réactif. Cela enrichira les moyens de simulation de chaque technique :

- en utilisant la puissance mathématique et graphique de Matlab/Simulink pour la résolution des modèles
- en facilitant la construction des modèles à travers l'exécution synchrone des composants réactifs. L'introduction de la communication par diffusion augmentera l'expressivité des modules Simulink, qui ne permettent pour l'instant qu'une communication poit à point.

Enseignement à distance et laboratoire virtuel pour l'école supérieure. Un outil d'enseignement à distance à base des machines numériques représente un très grand intérêt grâce à la simplicité de constructions des phénomènes physique même les plus complexes. La mise en parallèle de comportements élémentaires permet de reflèter les propriétés des objets physiques et l'ajout dynamique des composants facilite beaucoup le processus d'expérimentation virtuelle.

References

[1] Reactive Programming Site, *http://www.inria.fr/mimosa/rp*, server at INRIA Sophia-Antipolis

[2] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Pub., 1993

[3] F. Boussinot, L. Hazard, J.-F. Susini, *Junior Reactive Kernel*, Rapport de recherche INRIA 3732, 1999

[4] F. Boussinot, J.-F. Susini, *The SugarCubes Toolbox*, Software Practice and Experiences, 1998

[5] J.-F. Susini, *L'Approche Réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior*, Thèse de doctorat de l'Ecole des Mines de Paris, 2001

[6] P. Naughton, *The Java Handbook*, Osborne McGraw-Hill, 1996

[7] Sun Microsystems Inc., Java Technology *http://java.sun.com/*, official site

[8] F. Boussinot, *Les objets réactifs en Java*, Presses Politechniques et Universitaires Romandes CTFT FTR&D, 2000

[9] W.R. Bennet, *Scientific and Engineering Problem-Solving with the Computer*, Prentice Hall, 1976

[10] F.S. Acton, *Numerical Methods That Work*, corrected edition, Mathematical Association of America, 1990

[11] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B. P. Flannery *Numerical Recipes*, second edition, Cambridge University Press, 1992

[12] A.D. Suhanov, *Fundamental Course of Physics. Volume 1 - Corpuscular Physics*, Agar, Moscow, 1996

[13] R.P. Feinman, R.B. Leighton, M. Sands, *The Feynman Lectures on Physics*, Vol. 1, Addison-Wesley, 1963

[14] J.J. Dolado and F.J. Torrealdea, *Formal Manipulation of Forrester Diagrams by Graph Grammars*, IEEE Trans. on Syst., Man Cyb. V18, N6, Nov/Dec 1988, pp. 981-996

[15] Mitchell and Gauthier Assoc., *ACSL : Advanced Continous Simulation Language - Reference Manual*, 10th edition, Mitchell & Gauthier Assoc, Concord, MA, 1991

[16] ACSL web site, *http://www.acslsim.com/*, products and release notes

[17] Desire 2000, *http://hometown.aol.com/gatmkorn/*, overview

[18] H Elmqvist, *Dymola - User's Manual*, Dynasim AB, Research Park Ideon, Lund, Sweeden, 1995

[19] Dymola and Dynasim site, *http://www.dynasim.se/*

[20] M. Tiller, *Introduction to Physical Modeling with Modelica*, Kluwer Academic Pub., 2001

[21] Modelica site, *http://www.modelica.org/*

[22] VHDL-A guide, *http://www.acc-eda.com/vhdlref/*, guide and technical overview

[23] M. Otter, F. E. Cellier, Section 1.7.2 "Software for Modeling and Simulating Control Systems" in *The Control Handbook*, edition W.S. Levine, CRC Press, Boca Raton, FL, 1996

[24] Matlab, *http://www.mathworks.com/products/matlab/*, Matlab : introduction, download, overview

[25] Mathworks Inc., *SIMULINK - User's Manual*, South Natick, MA, 1992

[26] Simulink on Mathworks site, *http://www.mathworks.com/products/matlab/*

[27] Xmath, Xmath modules, Systembuild, *http://www.mathworks.com/support/matrixx/knowbase/doc7/*, Xmath and Systembuild documentation and download

[28] P.J. Mosterman, *An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages*, 1999