La Programmation par Objets Réactifs (POR)

Guillaume Doumenc
Soft Mountain
Ophira 2, route des Dolines
Sophia-Antipolis
06565 Valbonne, France
email: gdo@crios.inria.fr

Frédéric Boussinot
Ecole des Mines de Paris
Centre de Mathématiques Appliquées
06904 Sophia-Antipolis, France
email: fb@cma.cma.fr

Résumé. Nous présentons dans ce document, un modèle de programmation fondé sur l'approche objet, l'approche réactive et la communication par diffusion. Ce nouveau modèle permet de décrire naturellement les interactions pouvant apparaître entre les objets. Nous présentons enfin brièvement une implémentation réalisée avec les langages **C++** et **RC**.

Mots clés. Programmation orientée objet. Concurrence. Programmation réactive. Evénements. Communication par diffusion.

Abstract. We introduce a new programming model based on the object oriented approach, the reactive approach and the broadcast communication. In this model, interactions between objects are described in a natural way. At last, we present an implementation realized with the **C++** and **RC** languages.

Key Words. Concurrent object-oriented programming. Reactive programming. Events and broadcast communication.

Introduction

La *programmation orientée objet*, dont les concepts sont apparus avec le langage SIMULA [DMN68], connaît à l'heure actuelle un essor considérable et trouve des applications dans de nombreux domaines. Elle se caractérise par une programmation qui regroupe (*encapsule*) les données et les opérations sur celles-ci en une seule entité appelée *objet*. Cette vision localisée des informations et l'indépendance entre les objets engendrent un besoin naturel de distribution et de parallélisme.

La plupart des langages orientés objets actuels soit sont séquentiels (C++ [Stou86], CLOS [X3J13]), soit lorsqu'ils intègrent la concurrence et le parallélisme, possèdent des primitives proches d'un système d'exploitation qui sont lourdes à utiliser (SMALLTALK [GoRo83], ACTORS [Ag86]). Les modèles de communication entre objets sont généralement fondés sur la notion de message (un objet envoie un message à un autre objet) et sont implémentés de façon asynchrone (mise en attente et traitement dans l'ordre d'arrivée des messages). Le comportement global des systèmes devient alors non déterministe et difficilement compréhensible quand le nombre d'objets est important.

Nous nous proposons dans ce papier, de définir un modèle de programmation fondé sur la notion *d'objets réactifs* réagissant de façon *synchrone* à des *événements*. La communication entre les objets se fait par *diffusion* de ces événements.

Dans le modèle *réactif* les programmes comportent des points d'arrêt à travers lesquels l'exécution se déplace au cours des activations [BoDo91]. En découpant l'exécution des programmes en portions, on introduit explicitement une notion d'*instant* qui permet de donner une portée temporelle à l'existence des événements. Un événement est dit *présent* dans un instant, lorsqu'il est engendré au cours de cet instant, et *absent* sinon.

Dans le cadre réactif, le langage synchrone ESTEREL[BeGo88], modélise les événements par des *signaux*. Dans un programme ESTEREL, on peut réagir instantanément à la *présence* et à l'*absence* d'un signal. La réaction instantanée à l'absence, introduit des problèmes de cohérence connus sous le nom de *cycles de causalité*. L'exemple générique est le programme qui réagit à l'absence d'un signal en émettant ce même signal. Une passe de pré-compilation est nécessaire pour rejeter de tels programmes.

Pour éviter ces problèmes de cohérence, nous interdisons de réagir instantanément à l'absence des événements, en reportant cette réaction à l'instant suivant. Supposons par exemple que l'on veuille réagir à la présence d'un événement $E1$ uniquement dans le cas où un autre événement $E2$ n'est pas engendré en même temps (c'est à dire au cours du même instant). Dans notre approche, on ne peut savoir qu'à la fin de l'instant que $E2$ n'a pas été engendré et ainsi la réaction est nécessairement reportée à l'instant suivant.

Pour que cette réaction à l'absence paraisse "instantanée", nous définissons un mode particulier d'exécution qui consiste à réactiver le système *immédiatement* à la fin de chaque instant. Ainsi ce mode de réexécution permanente est une approximation de la réaction instantanée qui évite les problèmes de cohérence.

Le modèle proposé utilise la diffusion des événements pour permettre une programmation modulaire : la communication avec de nouveaux objets ajoutés au système est possible sans aucune modification des objets déjà existants. Cela ne peut être le cas dans une communication dirigée où il est nécessaire de désigner un destinataire pour lui communiquer une information.

Après avoir présenté le modèle d'objets réactifs, nous décrivons une implémentation particulière fondée sur les langages **C++**[Stou86] et **RC**[Bous91a].

1 Les objets du modèle

Dans la Programmation par Objets Réactifs un programme est constitué d'objets qui s'exécutent en parallèle. Les objets sont des entités indépendantes qui réagissent à des événements diffusés dans le système et qui communiquent par ces événements.

L'approche réactive permet de définir une notion d'instant : un instant est terminé lorsque tous les objets ont fini de réagir à tous les événements.

Les événements

Les événements peuvent être engendrés à tout moment par l'environnement (*événement externe*) ou par le système lui-même (*événement interne*). Un événement n'est pas une interruption car il n'a pas de procédure associée qui serait exécutée lors de sa génération. Ce n'est pas non plus un message comme dans le modèle des acteurs[Ag86], car il n'a pas de destinataire.

Les événement peuvent avoir des valeurs, être hiérarchisés ou même être composés d'autres événements. Les événements sont *éphémères*: leur présence varie dans le temps. Un événement peut apparaître, disparaître et puis réapparaître de nouveau, ... Toutefois, si les objets peuvent

à tout moment engendrer des événements, il leur est impossible d'en supprimer. La disparition de tous les événements est uniquement provoquée par le système à la fin de chaque instant.

Objets réactifs

Les objets réactifs possèdent une méthode particulière qui spécifie comment l'objet doit réagir aux événements. Cette méthode réactive est une *procédure réactive* au sens de [BoDo91], qui est activée à chaque fois que l'objet doit réagir. On peut considérer les objets réactifs comme des processus indépendants en attente d'événements, et le système comme un système d'exploitation chargé de les activer. Les objets réactifs peuvent être crées ou détruits dynamiquement à tout moment de l'exécution.

Le modèle proposé est décrit par la figure 1.

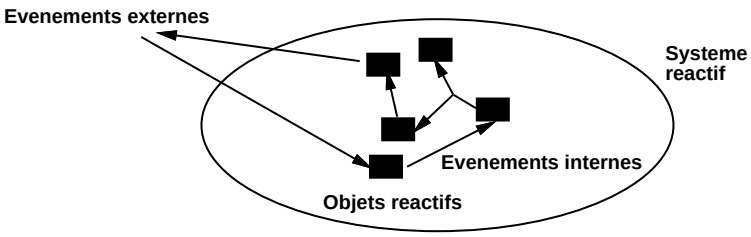

Figure 1: Un système réactif

2 Modèle d'exécution

Le modèle d'exécution de la POR est fondé sur la notion d'*activation permanente* des systèmes que l'on va décrire maintenant.

Exécution permanente

Les objets réactifs qui composent un système, réagissent instantanément à la présence des événements, et de manière retardée à leur absence. Cette contrainte est naturelle dès que l'on s'interdit de faire des hypothèses sur l'absence des événements : on ne peut savoir qu'un événement n'apparaît pas dans un certain intervalle de temps qu'à la fin de celui-ci.

Les réactions à l'absence d'un événement sont ainsi retardées à l'activation suivante. Pour conserver un aspect réactif, il est nécessaire que ce retard ne soit pas "trop important", c'est pourquoi le système active de façon ininterrompue les objets. La réaction à l'absence d'un événement devient alors "presque synchrone" : elle est reportée à la prochaine activation qui est immédiatement provoquée par le système à la fin de l'instant courant.

Bien entendu, cette approche n'a de sens que si les réactions des objets sont suffisamment rapides. Le domaine de validité du modèle est de fait, l'ensemble des applications où la durée des réactions est négligeable vis-à-vis de la fréquence d'apparition des événements.

Les méthodes réactives

Plusieurs primitives permettent de définir les méthodes réactives. Nous allons en décrire quelques unes brièvement.

La plus simple :

```
await_event(evt);
```

attend que l'événement `evt` soit produit pour poursuivre l'exécution. Cette primitive permet aux objets de se synchroniser.

On peut tester la présence et l'absence d'un événement par la primitive :

```
present_event (evt) instr1; else instr2;
```

celle-ci n'exécute l'instruction `instr1` que si l'événement `evt` est engendré dans l'instant courant. S'il ne l'est pas, l'instruction `instr2` est exécutée à l'activation suivante.

La primitive `watching_event(evt)` permet d'interrompre une réaction par la présence d'un événement. Lorsque l'événement `evt` est présent alors que le corps du `watching_event` n'est pas terminé, la terminaison de celui-ci est forcée à la réaction suivante.

La primitive `every_event(evt)` permet de relancer une réaction à chaque fois que l'événement `evt` est présent. Cette relance se fait également à la réaction suivante.

Pulsation et déterminisme

Un système réactif est dit *stable*, lorsque tous ses objets sont en attente d'événements : toutes les méthodes réactives des objets sont bloquées sur des `await_event`. L'activation d'un système stable sans événement extérieur est donc sans effet, puisqu'elle ne peut débloquer aucun objets.

Nous considérerons que dans la plupart des applications, les systèmes retrouvent rapidement leur stabilité après être devenus instables par l'apparition d'un ou de plusieurs événements externes. On appelle *pulsation*, la suite d'instants qui permet à un système instable de retrouver sa stabilité.

Le mode d'exécution permanente des systèmes réactifs ainsi que le choix des primitives de description des méthodes réactives font que le comportement global d'un système est *déterministe* : un système exécuté plusieurs fois avec la même séquence d'événements externes réagit toujours de la même façon. Cette propriété permet de faciliter les phases de mise au point et de déboggage.

En absence d'événement extérieur, l'activation d'un système stable est sans effet. Un tel système peut donc être mis en *attente passive* (au sens des système d'exploitation) et n'être ré¡veillé qu'à l'apparition d'un événement externe. C'est ainsi que fonctionne l'implémentation réalisée.

3 Implémentation

Les objets réactifs et les événements sont implémentés par deux classes pré-définies `Reactive` et `Event`. Les mécanismes classiques d'héritage permettent de les utiliser dans toute application à travers des sous-classes spécifiquement dédiées. L'implémentation que nous présentons est séquentielle et est fondée sur **C++**, cependant tout autre langage orienté objet aurait pu aussi bien être utilisé. Les méthodes réactives ainsi que le mécanisme d'exécution des systèmes sont décrits en **RC**.

Les événements

Les événements sont implémentés à travers la classe `Event`. Celle ci admet deux primitives `generate` et `vanish` qui permettent respectivement d'engendrer et de faire disparaître un événement. Seule la primitive `generate` est accessible à l'utilisateur, l'autre étant réservée au système. A partir de cette classe, on peut définir de nouvelles sous-classes d'événements avec valeur, typées, etc...

```
class Event {
protected:
   bool present, absent
   void vanish();
public:
   void generate();
};
```

Les booléens `present` et `absent` permettent de savoir si un événement est respectivement présent ou absent dans un instant. Au début de chaque instant, les valeurs des booléens `present` et `absent` de tous les événements sont mises à `false`, sauf au premier instant pour les événements externes qui ont amorcés la réaction et dont les booléens `present` sont alors mis à `true`. Dès qu'un événement est engendré, le booléen `present` est mis à `true`. Seul le système modifie les valeurs des booléens `absent` en mettant à `false` à la fin de chaque instant ceux correspondant à des événements non présents.

Les objets réactifs

La classe `Reactive` permet de construire un objet réactif à partir d'une méthode réactive décrite en **RC**. Celle-ci est activée par la fonction `activate` à chaque fois que l'objet doit réagir.

```
class Reactive {
private:
   rprocType behavior;
   void *param;
public:
   Reactive(Rproc, void *);
   void activate();
};
```

Le champ `param` contient les paramètres d'appel de la procédure réactive.

Les primitives de programmation des méthodes réactives sont implémentées en **RC**. Par exemple, "`present_event(evt) instr1; else instr2;`" est réalisée par le code **RC** suivant :

```
while (!(evt.present || evt.absent)) suspend;
if (evt.present)
  instr1;
else
  {stop; instr2;}
```

L'exécution est suspendue tant qu'il n'est pas possible de décider si l'événement est présent ou absent. On remarque la présence de l'instruction `stop` qui dans le cas où l'événement est absent, bloque l'exécution jusqu'à l'instant suivant.

Le système

Les systèmes réactifs sont implémentés par la classe abstraite `Reactive`. Un système est un ensemble d'objets réactifs. Les cycles de pulsation du système sont produits par la primitive `pulse`.

```
class Reactive {
private:
   static int numberOfObject;
   static Reactive *objects;
public:
   static void pulse();
};
```

La primitive `pulse` active les méthodes réactives de tout les objets réactifs jusqu'à ce que toutes leurs méthodes soient suspendues en attente d'un événement. Lorsqu'elles sont toutes suspendues, cette primitive déclare absent tous les événements non produits dans l'instant (elle met à `false` tous les booléens `absent` de ces événements). Puis elle relance les objets suspendus pour préparer leur réaction à l'absence lors de l'activation suivante. Le cycle recommence alors et les objets peuvent maintenant réagir à l'absence des événements lors de l'instant précédent. La primitive `pulse` ne rend la main que lorsque tous les objets sont arrêtés sur des `await_event`. Un système peut diverger en n'atteignant jamais un état de stabilité, toutefois de tels systèmes divergents ne peuvent plus être considérés comme réactifs.

Les pulsations

Le noyau du moteur d'exécution du système est codé en **RC**. L'attente passive d'événements externes est réalisée par la fonction C `select_ext_event()` qui bloque le contrôle jusqu'à ce qu'un événement soit présent. Le moteur d'exécution est le suivant :

```
for(;;) {
   select_ext_event();
   rwhile (!all_object_suspend_on_await_event)
      Reactive::pulse();
}
```

L'implémentation de la procédure `pulse` peut être plus ou moins optimisée. Nous allons décrire la plus simple, elle aussi codée en **RC**. Dans cette implémentation, le booléen `has_react` permet de savoir si un objet quelconque a réagit. Le code est le suivant :

```
catch ("end_reaction")
  close
    merge {
      /* activation de tous les objets   */
      for (i = 0; i < Reactive::numberOfObject; i++)
         object[i].activate();
      }
      loop {
        /* les objets ne réagissent plus */
        if (!has_react)
          raise "end_reaction";
        suspend;
      }
  /* fixation de tous les evenements *
   * fin de la reaction en cours     */
  fix_event_absent();
  /* activation de tous les objets *
   * les evenements etant fixes    */
  for (i = 0; i < Reactive::numberOfObject; i++)
     object[i].activate();
```

Variations du modèle

On peut envisager plusieurs variations du modèle précédent. Dans une première variation, on peut par exemple supposer que les événements externes qui activent le système, restent présents durant toute la pulsation qu'ils provoquent. Cette approche permet de définir de nouvelles primitives `watching_ext_event` et `every_ext_event` dédiées aux événements externes et dont l'action respective d'interruption et de relance est immédiate.

Dans une seconde variation, on peut accepter de prendre en considération de nouveaux événements externes alors que la pulsation n'est pas encore terminée. Cela risque de se produire si les réactions du système ne sont pas suffisamment rapide pour permettre d'obtenir la stabilité avant l'apparition de nouveaux événements externes.

Enfin, on peut supprimer la communication par diffusion des événements et ne considérer

qu'une communication par envoi de messages sur des files de type FIFO. On retrouve alors un modèle de type acteurs réactifs. Cette approche est celle qui a été choisie dans le modèle des *réseaux de processus réactifs* [Bous91b].

Conclusion

Nous avons présenté un nouveau modèle d'objets réactifs. Dans ce modèle la communication entre objets concurrents se fait par diffusion d'événements. Lorsqu'il est activé, un système se met à "pulser" jusqu'à retrouver un état de stabilité dans lequel il est en attente de nouveaux événements. Nous avons décrit les grandes lignes d'une implémentation de ce modèle fondée sur une approche réactive.

References

[Ag86] G. Agha *A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass, 1986.

[BeGo88] G. Berry, G. Gonthier, 'The Esterel Synchronous Programming Language: Design, Semantics, Implementation', INRIA Report **842** (1988), à paraitre dans *Science of Computer Programming*.

[Bous91a] F. Boussinot, 'Reactive C: An Extension of C to Program Reactive Systems', *Software-Practice and Experience*, vol. **21**(4), 401-428 (1991).

[BoDo91] F. Boussinot, G. Doumenc, 'Le langage Reactive C', Rapport de Recherche ENSMP-CMA 9/91 (1991).

[Bous91b] F. Boussinot, 'Réseaux de processus réactifs', Rapport de Recherche INRIA **1588** (1992)

[DMN68] O.J. Dahl, B. Myrhaag, K. Nygaard, *Simula 67 Common Base Language*, Norwegian Computing Center, (1968)

[GoRo83] A. Goldberg, D. Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, (1988)

[Stou86] B. Stroustrup, *The C++ programming language*, Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Company (1986).

[X3J13] D.G. Bobrow, L. DeMichiel, R.P. Gabriel, G. Kickzales, D. Moon, S. Keene, *The Common Lisp Object System Specifi¡cation: Chapters 1 and 2*, Technical Report 88-002R, X3J13 standards comitee document, 1988.