Programmer l'interaction avec des machines à états hiérarchiques

Renaud Blanch

Laboratoire de Recherche en Informatique & INRIA Futurs [*]
Bat 490, Université Paris-Sud
91405 Orsay cedex, France
renaud.blanch@lri.fr

RESUME
Le développement de la partie interactive d'un logiciel conduit, du fait de structures de contrôle inadaptées, à du code difficilement maintenable et réutilisable. Pour décrire et spécifier l'interaction, des formalismes adaptés existent. Cet article propose d'ajouter au niveau du langage une structure de contrôle issue de l'un de ces formalismes, les machines à états hiérarchiques, pour programmer l'interaction.

MOTS CLES : architecture logicielle, machines à états hiérarchiques, techniques d'interaction avancées.

ABSTRACT
Writing interactive software leads, due to the lack of adapted control structures, to a code that is difficult to maintain and reuse. Formalisms adapted to the description and to the specification of interactions do exist. This paper proposes to extend imperative programming languages with a control structure borrowed from one of those formalisms : the hierarchical state machines.

KEYWORDS : software architecture, hierarchical state machines, advanced interaction techniques.

INTRODUCTION

Les langages de programmations impératifs usuels ne sont pas adaptés à la programmation des interactions. Ils ont été créés pour écrire des algorithmes qui se "coupent du monde pendant leur exécution" [18]. L'inaptitude de ces langages à exprimer les interactions est mise en lumière par les formalismes développés pour les décrire et les spécifier. De même qu'on ne décrit pas un algorithme récursif à l'aide de structures de contrôle du type *goto* (bien qu'elles soient l'essence du code interprété par la machine) et qu'il a fallu étoffer les langages évolués de structures de plus haut niveau, il faut étendre les langages actuels pour offrir des moyens simples de programmer des interactions, de les réutiliser, de les raffiner et d'en tester facilement de nouvelles.

L'inadaptation des langages actuels apparaît notamment dans le code produit pour gérer les entrées d'une application interactive. Il prend en général deux formes. Chaque interaction peut être découpée au sein de *callbacks* élémentaires dont l'interdépendance est très vite incompréhensible [16]. C'est à ce type de code "spaghetti" que conduit l'utilisation de boîtes à outils telles que Java Swing ou Tcl/Tk. L'alternative est une concentration de l'ensemble du code gérant l'interaction au sein d'une boucle principale recevant des événements, tout aussi délicate à maintenir que la première forme. Il est de plus difficile d'intégrer au sein d'une même application des bibliothèques qui nécessitent chacune leur boucle principale [7].

Le problème devient particulièrement aigu dès lors que l'on veut utiliser des techniques d'interaction avancées mettant en jeu plusieurs périphériques de pointage ou plusieurs modalités. Par exemple, l'interaction bimanuelle combinée, qui met en jeu deux pointeurs coopérant pour réaliser une tâche [6], ne peut s'implémenter que grâce à un développement *ad hoc* [4].

Le développement de méthodes visuelles [2, 8, 12] pour supporter l'interaction illustre la pertinence de la problématique et le besoin de nouvelles techniques pour dépasser les limitations des langages de programmation. Nous explorons ici une approche différente : l'extension syntaxique du langage de programmation, ce qui permet d'intégrer le formalisme utilisé au code produit.

Cet article propose d'utiliser les machines à états hiérarchiques, extension des machines à états qui les rapproche des *Statecharts* [12], pour étendre un langage existant et ainsi programmer l'interaction. Ce formalisme fait des interactions des objets concrets, représentés et manipulables grâce à une structure de contrôle adaptée. Ainsi, il offre pour le développement une modularité favorisant la réutilisation, et pour l'exécution, la possibilité de manipuler les interactions pour les adapter au contexte.

IHM 2002, November 26-29, 2002, Poitiers, FRANCE.

[*] projet In Situ, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, Ecole Polytechnique, INRIA, Université Paris-Sud

L'article présente les machines à états hiérarchiques et motive le choix effectué au regard d'autres formalismes. Après une description de leur syntaxe et de leur sémantique, nous donnons des exemples d'utilisation. Enfin nous apportons quelques précisions sur l'environnement au sein duquel nous avons intégré les machines à états hiérarchiques.

TRAVAUX ANTERIEURS

Les formalismes utilisés pour spécifier ou décrire l'interaction sont nombreux. Ils sont aussi utilisés pour tenter de simplifier la programmation des interactions. Ils ne répondent cependant que partiellement aux besoins de réutilisation et d'extension qui nous préoccupent.

Les formalismes

Les *dataflows* et les modèles réactifs synchrones, qui fonctionnent par propagation de grandeurs dans un circuit à la manière d'un réseau électrique, mettent en jeu des principes sous-jacents radicalement différents de ceux des langages usuels et ne peuvent donc pas être intégrés facilement dans ceux-ci [19]. Des langages complets comme Lustre ou Esterel [5] ont dû être créés pour mettre en œuvre ce type de formalismes. Les réseaux de Petri sont très expressifs et permettent de formaliser l'interaction [2]. Cependant, cette expressivité se paie par la relative complexité du formalisme: une version textuelle d'un tel réseau n'est envisageable qu'à titre de représentation interne et il n'est pas possible de l'utiliser directement comme syntaxe d'un langage de programmation. Les *Statecharts* [12] souffrent, par rapport à cet objectif, des mêmes problèmes.

Ces formalismes sont par ailleurs utilisés dans des outils de programmation visuelle auxquels leur sont plus adaptés [9,22].

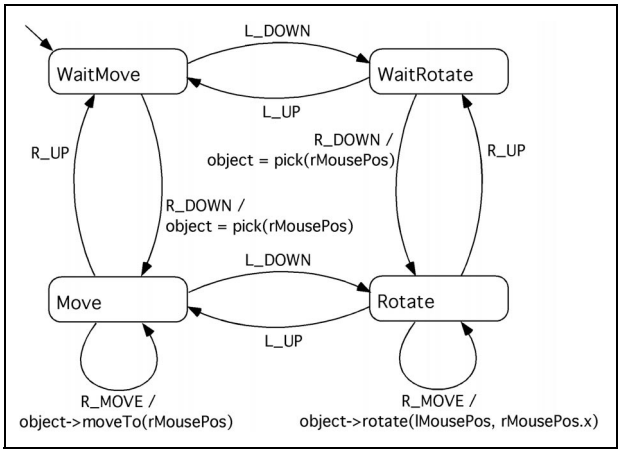

Figure 1: Machine à états décrivant une tâche de navigation bimanuelle (version graphique).

Les machines à états offrent moins d'expressivité que les réseaux de Petri mais permettent d'exprimer simplement des interactions relativement complexes [11]. A titre d'exemple, considérons une tâche de déplacement bimanuel d'un objet [13]: le déplacement, bouton enfoncé, de la souris droite bouge l'objet sélectionné mais si le boutton de la souris de gauche est enfoncé, il tourne l'objet autour de la position fixée par cette dernière. La figure 1 montre une représentation graphique de la machine à états décrivant cette interaction et la figure 2 en donne une version textuelle. Cette représentation est lisible, donc facilement éditable. Les machines à états peuvent être implémentées simplement dans un langage à objets [10]. Cependant, il est impossible de combiner des machines préexistantes pour étendre une interaction sans modifier en profondeur les transitions.

```
stateMachine Bimanual {
  local object
  state WaitMove {
    on L_DOWN -> WaitRotate
    on R_DOWN do {
      object = pick(rMousePos)
    } -> Move
  }
  state WaitRotate {
    on L_UP -> WaitMove
    on R_DOWN do {
      object = pick(rMousePos)
    } -> Rotate
  }
  state Move {
    on L_DOWN -> Rotate
    on R_UP -> WaitMove
    on R_MOVE do {
      oject->moveTo(rMousePos)
    }
  }
  state Rotate {
    on L_UP -> Move
    on R_UP -> WaitRotate
    on R_MOVE do {
      object->rotate(lMousePos, rMousePos.x)
    }
  }
}
```

Figure 2: Version textuelle de la machine à états de la figure1.

Les machines à états hiérarchiques sont une extension classique des machines à états. Chacun des états d'une machine peut être constitué d'une sous-machine. Cela offre la possibilité, en conservant la simplicité et l'expressivité des machines à états [1], de réutiliser des sous-parties existantes au sein de nouvelles machines ou de factoriser des états et des transitions. C'est pour sa simplicité, son relatif pouvoir d'expression pour les applications qui nous intéressent et sa modularité que nous avons choisi ce modèle.

Utilisation de ces formalismes pour programmer

Libero [20] est un environnement graphique qui permet de développer des applications ayant une structure de machine à états. Il génère du code dans une variété de langages cibles allant des langages interprétés aux

langages à objets. Libero permet par exemple de développer des scripts de configuration, d'implémenter des protocoles ou des dialogues. Cependant, il ne propose pas de structure hiérarchique et aucun support pour les entrées, à part la console, n'est fourni.

Jacob [13] propose un langage visuel qui permet de coder les interactions. Une machine à états sélectionne le mode de l'interaction et dans chacun d'entre-eux, une approche *dataflow* connecte des grandeurs d'entrée aux grandeurs manipulées. L'organisation se fait donc en deux niveaux ayant chacun leur formalisme. L'approche *dataflow* peut être simulée à l'aide d'une machine à états synchrone. Par exemple, dans l'état *Move* de la figure 2, la position de l'objet est mise à jour de manière synchrone quand le pointeur bouge, ce qui revient à connecter la position de l'objet à celle du pointeur. Les machines à états hiérarchiques permettent, de ne conserver qu'un formalisme et d'étendre la structure en niveaux offerte pour programmer les techniques d'interaction.

ICON [8] permet à l'utilisateur, grâce à une interface graphique, de reconfigurer l'interaction en fonction des périphériques particuliers dont il dispose. Il permet ainsi d'adapter une application à des besoins divers, de ceux du graphiste à ceux d'une personne handicapée. Cette approche instrumentale de l'interaction [3] qui consiste à découpler l'interaction d'un instrument avec l'objet manipulé (service offert par l'application) de l'action de l'utilisateur sur l'instrument au travers des périphériques présents est l'une des motivations de notre approche. Un système comme ICON peut être développé grâce aux machines à états hiérarchiques utilisées comme structure de contrôle dans un langage.

Des compilateurs de machines à états (hiérarchiques ou non) existent par ailleurs [14, 21]. Cependant, ils n'offrent pas de support pour les périphériques d'entrée et les syntaxes adoptées sont souvent peu lisibles.

LES MACHINES A ETATS HIERARCHIQUES

Nous présentons l'exemple précédant reformulé à l'aide des machines à états hiérarchiques dans une syntaxe simplifiée. La syntaxe réelle et le fonctionnement du formalisme adopté sont présentés ensuite.

Exemple

Les figures 3 et 4 montrent la version hiérarchique de la machine à états des figures 1 et 2. Les deux interactions élémentaires sont rendues indépendantes et peuvent maintenant être modifiées sans interférences. Elles sont combinées au sein d'une machine de plus haut niveau qui choisit laquelle est active. L'interaction qui permet de passer de l'une à l'autre (action sur le boutton du périphérique de gauche) n'est plus intégrée dans les deux machines ce qui permet de la remplacer facilement. On pourrait par exemple la remplacer par l'action sur une palette d'outils sélectionnant explicitement le mode d'interaction.

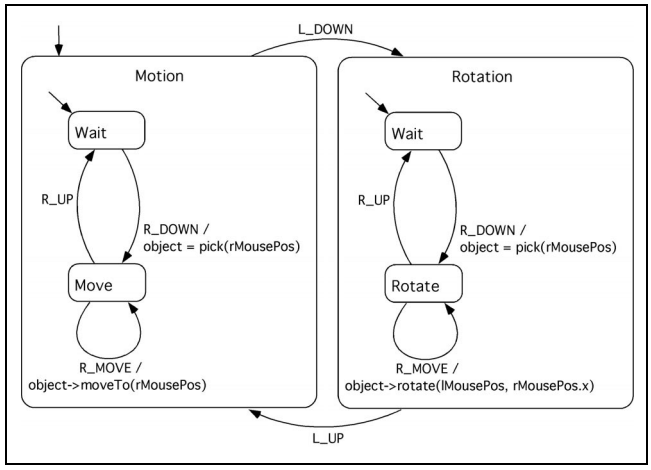

Figure 3: Version hiérarchique de la machine à états de la figure 1 (les machines ne sont pas équivalentes, ce point est discuté plus loin).

```
state Motion {
  local object
  state Wait {
    on R_DOWN do {
      object = pick(rMousePos)
    } -> Move
  }
  state Move {
    on R_UP -> Wait
    on R_MOVE do {
      object->moveTo(rMousePos)
    }
  }
}

state Rotation {
  local object
  state Wait {
    on R_DOWN do {
      object = pick(rMousePos)
    } -> Rotate
  }
  state Rotate {
    on R_UP -> Wait
    on R_MOVE do {
      object->rotate(lMousePos, rMousePos.x)
    }
  }
}

machine Bimanual {
  state Motion
  state Rotation
  on L_DOWN -> Rotation
  on L_UP -> Motion
}
```

Figure 4: Version textuelle de la machine à états hiérarchique de la figure 3 (la syntaxe utilisée est une version allégée de celle présentée ensuite).

Syntaxe et sémantique

La syntaxe adoptée pour déclarer une machine à états hiérarchique se veut proche de l'esprit du C++. Elle peut néanmoins être adaptée à d'autres langages. Un exemple réel de code d'une machine hiérarchique est donné figure5. L'environnement au sein duquel les machines à états hiérarchiques sont intégrées fournit un système de notification. Chaque composant (machine à états, abstraction d'un périphérique, *etc*.) peut émettre des évènements à destination des composants qui lui sont abonnés et peut recevoir des évènements en les traitant de manière synchrone ou asynchrone. Cet environnement est décrit dans la section implémentation.

```
machine MultiClick {

  // déclaration explicite d'une entrée
  in button;
  enter { cout << "MultiClick entered.\n"; }
  leave { cout << "MultiClick leaved.\n"; }

  // déclaration d'un sous-état
  // et de son entrée
  state NoClick(button)

  // définition d'un sous-état
  state Click {
    in button;
    var unsigned int nbClicks;
    var unsigned int delay;
    after delay {} send(MCLICK, nbClicks)
      -> NoClick
    on DOWN from button { nbClicks++; }
      -> Click(delay = 150)
  }
}

// définition du sous-état déjà déclaré
state NoClick {
  in but;
  on DOWN from but {}
    -> Click(delay = 300, nbClicks = 1)
}
```

Figure 5: Exemple réel de machine à états hiérarchique.
La machine synthétise un évènement contenant le nombre de clics successifs d'un boutton séparés de moins de 300 ms pour les deux premiers et de moins de 150 ms pour les suivants.

Machines. La machine reçoit les évènements et les traite en franchissant, si possible, une transition. La définition d'une machine se fait à la manière de la déclaration d'une classe en utilisant le mot clé *machine* suivi de son nom puis de son corps.

Le corps contient les déclarations ou les définitions des états de la machine et la définition de ses transitions. Celles-ci doivent avoir pour cible l'un des états de la machine. Lorsqu'une transition est franchie, sa cible devient l'état courant. Si un évènement n'est pas traité par la machine (c'est-à-dire qu'il ne permet pas de franchir une transition), elle le passe à son état courant.

Etats. Un état se définit de la même façon qu'une machine mais en utilisant le mot clé *state*. Il peut comporter des sous-états et des transitions. Celles-ci doivent avoir pour cible un état frère (c'est à dire déclaré au même niveau et englobé par la même machine ou le même état comme le sont *NoClick* et *Click*, figure 5). Un état ne reçoit d'évènements que de l'état ou de la machine qui l'englobe au niveau immédiatement supérieur. Il propage de la même manière qu'une machine les évènements non traités à son sous-état courant s'il a des sous-états.

Un état peut être déclaré sans être défini si sa définition est connue à la compilation. Sa déclaration doit comporter les sources d'évènements qu'il utilise (voir la déclaration de l'état *NoClick*, figure 5, qui spécifie qu'il utilisera l'entrée *button* de *MultiClick* pour son entrée *but*).

Sources d'évènements. Les sources des évènements attendus par une machine ou un état doivent être déclarées au début du corps de la définition en les faisant précéder du mot clé *in*. Elles peuvent être définies explicitement dans le corps de la machine ou de l'état. La déclaration d'une source prend la forme suivante:

```
in name [= definition];
```

Si une source n'est pas donnée lors de la déclaration d'une machine ou d'un état, elle sera passée en paramètre lors de l'instanciation de la machine. L'instanciation de la machine de la figure 5 qui ne définit pas la source *button* se réalise par exemple ainsi:

```
State *clicks = new MultiClick::State(aButton);
```

Etat initial. L'état initial d'une machine (ou le sous-état initial d'un état) est, par défaut, le premier déclaré dans son corps. Cependant, il peut être souhaitable de le déterminer au moment de l'activation de la machine, et ce en fonction du contexte dans lequel celle-ci a lieu. En effet, dans la machine de la figure 3, enfoncer le bouton gauche lorsque l'on est dans l'état *Move* fait passer dans le sous-état *Wait* de *Rotation* et non dans *Rotate* comme le faisait la version non hiérarchique (figure 1). En accédant au contexte (état du bouton droit) on peut choisir le bon état initial (*Wait* ou *Rotate*).

La sélection se fait en examinant les règles inclues dans le corps dans l'ordre où elles apparaissent. Elles comportent une condition booléenne calculable dans le contexte de l'état et le nom de l'état initial lorsque cette condition est valide. Elles prennent donc cette forme :

```
(condition) : InitialState
```

Variables d'état. Un état ou une machine peut comporter un ensemble de variables déclarées dans leur corps. Il est possible de leur donner une valeur initiale prise à l'instanciation de la machine. Les variables ne sont visibles qu'à l'intérieur du corps de l'état ou de la machine (et pas à l'intérieur de leurs sous-états). La

valeur d'une variable peut être fixée lors d'une transition arrivant sur l'état (voir la description des transitions).

Une variable d'état est déclarée à l'intérieur du corps d'un état à l'aide du mot clé *var* suivi de la déclaration (ou de la définition) de celle-ci dans le langage cible:

```
var declaration [= val];
```

Transitions. La première partie d'une transition détermine la condition du franchissement: expiration d'un délai ou réception d'un évènement d'un type donné vérifiant une condition sur son expéditeur et toute autre expression booléenne qui peut être exprimée dans le contexte de l'état. Pour chaque évènement reçu, les transitions issues de l'état courant sont examinées dans l'ordre de leurs définitions et la première transition dont toutes les conditions sont vérifiées est franchie. Si aucune transition n'est possible, l'évènement est passé à l'état courant du niveau inférieur.

Ces deux possibilités prennent la forme suivante:

```
after ms { action } ...
on EVT_TYPE [from sender] [with (condition)] {
  action
} ...
```

Vient ensuite l'action de la transition, exprimée dans le langage cible. Elle est effectuée dans le contexte de l'état avant transition. Une fois l'action effectuée il est possible d'envoyer un évènement aux composants abonnés à la machine à états. Ceci permet de l'utiliser comme une source d'évènements de haut niveau filtrant ses entrées et synthétisant un périphérique virtuel (par exemple, la machine de la figure 5 regroupe des clics successifs en évènements de plus haut niveau *MCLICK*).

Enfin, la cible de la transition peut être spécifiée. Si elle ne l'est pas, la machine reste dans le même état. Dans le cas contraire, l'état courant est quitté puis l'état cible est activé. Il est possible de fixer de nouvelles valeurs pour les variables d'état de la cible qui sont affectées avant l'entrée dans celle-ci.

La seconde partie d'une transition prend donc la forme suivante:

```
[send(EVT_TYPE [,data] )]
[-> NextState [(variable = value, ...)] ]
```

On peut propager l'évènement qui provoque la transition au nouvel état courant en remplacant la flèche simple (**->**) par une flèche double (**=>**). Enfin, lorsqu'un état est activé, il réinitialise son sous-état courant, s'il en a un, à l'état initial. Ce comportement peut-être inhibé pour permettre la persistance du sous-état courant d'une activation à l'autre (*history* des *Statecharts*) en employant les flêches **-h->** ou **=h=>**

Actions. Un état peut définir des actions particulières qui sont exécutées lorsque l'état est quitté ou entré. Le comportement par défaut d'un état lorsqu'il est entré est d'initialiser son *timer* si l'une de ses transitions le nécessite et d'entrer son propre sous-état initial s'il en a un. De même, lorsqu'il est quitté, il désactive son *timer* et interrompt son état courant. Quitter et interrompre sont deux actions différentes, la première est volontaire (un évènement a déclenché une des transitions de l'état qui est donc quitté) alors que la seconde est imposée par un sur-état (un des sur-états est quitté, il somme son sous-état courant de s'interrompre). Enfin, un état interrompu interrompt son sous-état courant.

Il est possible d'ajouter une action supplémentaire exécutée lorsque l'on entre, quitte ou interrompt un état. Elle est effectuée avant le comportement par défaut pour l'entrée et après celle-ci pour les deux autres cas. Cette action est précédée du mot clé *enter, leave* ou *escape*.

Discussion

Les machines à états hiérarchiques, comme l'illustre l'exemple des figures 1 et 3, invitent plus à structurer le code que les simples machines à états. Cette structuration plus rigide que celle des *Statecharts*, puisque les transitions entre niveaux différents sont proscrites, permet limiter les interdépendances et favorise donc la réutilisation de fragments déjà écrits.

En revanche, les *Statecharts* ne font pas de distinction entre les manières de quitter un état. Le mécanisme d'interruption de l'interaction permet, à la manière des exceptions, de dépiler les contextes en effectuant les éventuelles actions nécessaires pour terminer ou annuler les opérations en cours. Il permet par exemple de gérer simplement des problèmes comme la fermeture de menus ouverts en cascade lorsque l'action est annulée. Cette distinction peut être trouvée dans la machine à état qui régit les interacteurs de Myers [14].

La concurrence et le modèle de synchronisation du traitement des évènements ne sont pas donnés par le formalisme mais sont supportés au niveau de l'environnement au sein duquel les machines ont été intégrées (voir la section implémentation).

EXEMPLES D'UTILISATION

Nous présentons ici plusieurs exemples illustrant les possibilités offertes par les machines à états hiérarchiques dans des utilisations simples et typiques.

Filtrage

Nous avons utilisé une machine à états comme un filtre agissant sur une entrée. La machine devient alors une nouvelle source d'évènements qui peut être utilisée à son tour comme une entrée.

Ainsi, on peut filtrer la position du stylet sur une tablette graphique en l'amortissant pour gommer les irrégularités des courbes tracées à main levée dans un logiciel de dessin. Un exemple de machine réalisant un tel amortissement exponentiel est donné figure 6. L'évènement *CHANGE* est reçu à chaque fois que la

position d'entrée est modifiée. La position amortie est alors calculée. Cette position est accessible grâce au mécanisme général d'introspection des composants qui n'est pas détaillé ici et peut être utilisée en lieu et place de la position de la souris par exemple.

```
state Smooth {
  in position;
  // point qui stocke la position amortie
  // nom : smooth, dimension : 2 (x et y)
  var Point p = Point("smooth", 2);
  var float k = 0.1;
  enter { p = position; }
  on CHANGE from position {
    p = k*position + (1–k)*p;
  }
}
```

Figure 6: Filtrage d'une position en amortissant ses évolutions.

Le principe du filtrage permet de réaliser des fonctions plus complexes. En utilisant un algorithme de reconnaissance de trace comme celui de Rubine [17] qui se prête bien à un traitement sur un flux de positions puisqu'il est incrémental, on peut réaliser un filtre qui transforme les variations d'une position en évènements de plus haut niveau précisant la trace reconnue.

Spécialisation d'une technique existante

L'utilisation des machines à états hiérarchiques permet de raffiner une technique d'interaction existante. L'exemple proposé ici est celui d'une interaction de "glisser" dans une application de dessin (figure7). Elle est enrichie en permettant de contraindre le mouvement (sur l'horizontale ou la verticale par exemple) lorsqu'une touche est enfoncée (figure8).

```
machine Motion {
  in position;
  in button;
  on DOWN from button {} -> Move
  on UP from button {} -> Idle
  state Idle {}
  state Move {
    in position;
    var Object *object = NULL;
    enter { object = pick(position); }
    on CHANGE from position {
      object->moveTo(position);
    }
  }
}
```

Figure 7: Interaction originale de "glisser".

Les états *NormalMove* et *ConstMove* réutilisent chacun l'état *Move* défini précédemment en lui donnant en entrée la position réelle du pointeur ou une position calculée à partir de celle-ci qui respecte les contraintes voulues. Les deux états définissent par ailleurs les transitions qui vont permettre de basculer de l'un à l'autre au cours de l'interaction. La position contrainte est mise à jour, comme dans le cas du filtrage par amortissement, à chaque modification de la position originale signalée par le *CHANGE*. Cette mise à jour provoque alors l'émission d'un évènement *CHANGE* par la position contrainte. Le sous-état *Move* de l'état *ConstMove* a pour entrée cette position calculée, il reçoit donc les évènements qui en proviennent et fait suivre la position contrainte à l'objet déplacé.

```
machine ExtMotion {
  in position;
  in button;
  on DOWN from button {} -> ExtMove
  on UP from button {} -> Idle
  state Idle {}
  state ExtMove {
    state NormalMove {
      in switch; in position;
      on DOWN from switch {} -> ConstMove
      state Move(position)
    }
    state ConstMove {
      in switch; in position;
      var Point p = Point("const", 2);
      on CHANGE from position {
        p = constraint(position);
      }
      on UP from switch {} -> NormalMove
      state Move(p)
    }
  }
}
```

Figure 8: "glisser" contraint réalisé en réutilisant le *Move* de la version originale.

Adaptation aux périphériques d'entrée

Le dernier exemple montre comment adapter une technique d'interaction aux périphériques disponibles. Une palette d'outils classique est utilisable, en interaction bimanuelle, comme une *toolglass*. Il faut cependant adapter l'interaction de sélection dans la palette pour qu'elle déclenche directement l'action de l'outil par un "clic au travers".

La première machine de la figure9 code l'interaction qui permet la sélection classique d'un outil dans la palette. Dans l'état *Choose*, un clic sur un outil fait passer dans l'état qui code l'interaction de celui-ci. Dans cet état, les clics sur la palette sont interceptés et redirigés vers l'état *Choose* pour changer d'outil (transition de l'état *DoTool1*). Les autres évènements sont passés normalement à l'outil.

Dans le cas de la *toolglass*, le clic qui sélectionne l'outil dans l'état *Choose* est transmis à l'état sélectionné pour pouvoir débuter l'interaction directement. Comme précédemment, pendant une interaction, les clics sur la palette sont interceptés et redirigés vers l'état *Choose* pour pouvoir changer d'outil.

```
state Palette {
  in pos;
  in but;
  state Choose {
    in pos; in but;
    on DOWN from but with (in(pos, tool1))
      {} -> DoTool1
    on DOWN from but with (in(pos, tool2))
      {} -> DoTool2
  }
  state DoTool1 {
    in pos; in but;
    on DOWN from but with (in(pos, pal))
      {} => Choose
    state Tool1(pos, but)
  }
  state DoTool2 { [...] }
}
state ToolGlass {
  in pos;
  in but;
  state Choose {
    in pos; in but;
    on DOWN from but with (in(pos, tool1))
      {} => DoTool1
    [...]
  }
  state DoTool1 {
    in pos; in but;
    on DOWN from but
      with (in(pos, pal) && !in(pos, tool1))
      {} => Choose
    state Tool1(pos, but)
  }
  state DoTool2 { [...] }
}
machine SelectInteraction {
  in switch;
  in position;
  in button;
  on DOWN from switch {} -> ToolGlass
  on UP from switch {} -> Palette
  state Palette(position, button)
  state ToolGlass(position, button)
}
```

Figure 9: Utilisation d'une palette d'outils comme *toolglass* dans le cas bimanuel (seul le choix de l'outil est traité ici, pas le déplacement de la palette).

Enfin, la machine *SelectInteraction* fait basculer d'une technique à l'autre en fonction des évènements émis par *switch*. Ces derniers sont générés par le gestionnaire de périphériques de l'environnement sur lequel repose notre implémentation. Il notifie de l'ajout et du retrait à chaud d'un périphérique.

IMPLEMENTATION

L'implémentation des machines à états hiérarchiques repose sur un environnement constitué de classes C++ (langage choisi pour ses performances et l'intégration avec OpenGL) qui fournit divers services d'abstraction donnés ici. La compilation d'une machine repose sur un préprocesseur qui génère du code C++.

Contexte

Nous avons implémenté le modèle des machines à états hiérarchiques présenté dans le cadre d'un environnement de développement plus large fournissant une abstraction du système d'exploitation[1], du gestionnaire de fenêtres et des périphériques d'entrée. Cet environnement offre un système d'évènements sans imposer une méthode d'aiguillage, chaque composant pouvant envoyer des messages aux composants qui lui sont abonnés ainsi qu'à une cible précise. Les évènements peuvent être traités dans l'ordre de leurs arrivées, de manière synchrone ou asynchrone par les composants. D'autres stratégies de traitement (queues avec priorités, *etc*.) peuvent être ajoutées simplement puisqu'elles sont implémentées séparemment des composants.

Les périphériques sont encapsulés dans des composants qui maintiennent un état, peuvent être interrogés sur leur structure et peuvent notifier les composants abonnés de leurs changements d'état à divers niveaux de granularité. Un composant notifie des changements à chaud de périphériques (branchement et débranchement d'un périphérique USB par exemple), ce qui permet d'élaborer des mécanismes de reconfiguration de l'interaction en cours d'utilisation d'un logiciel.

Traduction

La syntaxe des machines à états hiérarchiques est décrite par une grammaire LL(1). La description d'une machine est traduite en code C++. Chaque état est implémenté par une classe héritant ses comportements par défaut de l'état vide. Plusieurs versions d'une machine peuvent donc être instanciées de manière concurrente, chacune pouvant recevoir ses entrées de composants différents.

CONCLUSION ET PERSPECTIVES

Dans cet article nous avons proposé d'enrichir un langage de programmation avec des machines à états hiérarchiques pour mieux supporter le développement de techniques d'interaction. Ceci permet d'offrir une structure adaptée aux particularités des mécanismes mis en jeu dans l'interaction en facilitant la modularisation du code. Il devient ainsi plus aisé à maintenir et à réutiliser. Par ailleurs cette modularité permet d'envisager une reconfiguration dynamique de l'interaction tenant compte des périphériques disponibles et des choix de l'utilisateur.

Pour aller plus loin dans ces directions, nous prévoyons d'ajouter la possibilité d'étendre un état existant par un mécanisme d'héritage. Cela permettrait de réutiliser une technique d'interaction en l'adaptant simplement à un nouveau contexte légèrement différent de celui pour lequel elle a été pensée. Les machines à états hiérarchiques permettront alors de développer simplement et de tester rapidement de nouvelles techniques d'interaction.

[1] Les systèmes supportés sont actuellement MacOsX et Win32.

REMERCIEMENTS

Merci à Michel Beaudouin-Lafon, Olivier Beaudoux, Stéphane Conversy et Nicolas Roussel pour leur lecture attentive de ce travail et les discussions qui l'ont alimenté.

BIBLIOGRAPHIE

1. R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Proc. 26th International Colloquium on Automata, Languages, and Programming (ICALP'99)*, pp. 169–178, Prague - Czech Republic, July 1999.

2. R. Bastide, and Ph. Palanque. A Petri Net Based Environment for the Design of EventDriven Interfaces. In *Proc. 16th International Conference on Application and Theory of Petri Nets (ATPN'95)*, Torino - Italy, June 1995.

3. M. Beaudouin-Lafon. Instrumental interaction : An interaction model for designing post-WIMP user interfaces. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI'2000)*, The Hague - Netherlands, April 2000. ACM Press.

4. M. Beaudouin-Lafon, and H. M. Lassen. The architecture and implementation of CPN2000, a post-WIMP graphical application. In *Proc. ACM Symposium on User Interface Software and Technology (UIST'2000)*, Boston - USA, 2000.

5. G. Berry, and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, Vol. 19, No. 2, pp. 87–152, 1992.

6. S. Chatty. Extending a graphical toolkit for two-handed interaction. In *Proc. ACM Symposium on User Interface Software and Technology (UIST'94)*, pp. 195–204, Marina del Rey - USA, 1994.

7. S. Conversy, P. Janecek, and N. Roussel. Factorisons la gestion des évènements des applications interactives. In *Travaux préparatoires des dixièmes journées francophones sur l'Interaction Homme Machine (IHM'98)*, pp. 141–144, Nantes - France, September 1998.

8. P. Dragicevic, and J.-D. Fekete. Input device selection and interaction configuration with ICON. In *Joint proceedings of HCI 2001 and IHM 2001 (IHM–HCI'2001)*, pp. 543–558, Lille - France, September 2001.

9. O. Esteban , S. Chatty, and P. Palanque. Whizz'Ed : a visual environment for building highly interactive software. In *Proc. 5th IFIP International Conference on Human-Computer Interaction (Interact'95)*, pp. 121-126, Lillehammer - Norway, June 1995.

10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

11. M. Green. A survey of three dialogue models. *ACM Transaction on Graphics*, Vol. 5, No. 3, pp. 244–275, July 1986.

12. D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, Vol. 8, No. 3, pp. 231–274, 1987.

13. R. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, Vol. 6, No. 1, pp. 1–46, March 1999.

14. P. J. Lucas. An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines. *Technical Report UIUCDCS-R-94-1868*, Department of Computer Science, University of Illinois, August 1994.

15. B. A. Myers. A New Model for Handling Input. *ACM Transactions on Information Systems*, Vol. 8, No. 3, pp. 289–320, July 1990.

16. B. A. Myers. Separating application code from toolkits : Eliminating the spaghetti of call-backs. In *Proc. ACM Symposium on User Interface Software and Technology (UIST'91)*, pp. 211–220, Hilton Head – USA, 1991.

17. D. Rubine. Specifying gestures by example. *Computer Graphics*, Vol. 25, No. 4, pp. 329–337, 1991.

18. P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, Vol. 40, No. 5, pp. 80–91, May 1997.

19. R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, Vol. 30, No. 4, pp. 459–527, December 1998.

20. Libero Home page. http://www.imatix.com/html/libero/

21. SMC Home page. http://smc.sourceforge.net/

22. xjCharts Home page. http://www.xjtek.com/products/xjcharts/