Spécification et implémentation de comportement d'agent

Quentin Baert, Maxime Morge
Univ. Lille, CNRS, Centrale Lille
UMR 9189 - CRIStAL
F-59000 Lille, France
Email: {Quentin.Baert,Maxime.Morge}@univ-lille.fr

25 mars 2020

L'approche centrée individu nécessite la mise en œuvre d'agents autonomes. Inspiré du modèle d'acteurs de Hewitt (1977), nous considérons qu'un agent :

1. possède une adresse unique pour communiquer avec ses pairs par envoi de messages ;
2. peut créer d'autres agents ;
3. est muni d'un état mental ;
4. réagit aux messages reçus selon un comportement.

Message. Un agent possède une file d'attente qui contient l'ensemble des messages qu'il a reçus. Nous considérons ici une version simplifiée du modèle de transmission asynchrone de messages pour la programmation concurrente (Clinger, 1981). En d'autres termes, nous suppose que :

1. le délai de transmission des messages est arbitraire mais non négligeable ;
2. l'ordre d'émission/réception des messages est identique par pair émetteur-récepteur ;
3. la distribution des messages est garantie.

Contrairement au modèle de Clinger (1981), nous faisons l'hypothèse que les messages sont livrés exactement un fois.

Création d'agents. Un agent peut être composé de plusieurs agents composants avec lesquels il communique. Une telle architecture composite d'agent permet d'isoler les différentes activités d'un agent pour ainsi faciliter sa conception, l'intelligibilité de son comportement et sa mise en œuvre.

État mental. Un agent est muni d'une base de connaissances et/ou de croyances. Cette mémoire constitue une représentation explicite, locale et

subjective de la perception par l'agent du système multi-agents mais également de son environnement physique. En particulier, la liste de ses accointances est la liste des pairs avec lesquels l'agent est capable de communiquer directement par envoi de message.

Comportement. Pour déterminer sa prochaine action, un agent retire le premier message de sa file d'attente et réagit en fonction de son état courant.

Même si tous les agents utilisent le même comportement, chaque agent est indépendant. D'une part, chacun possède sa propre file de messages, son propre état courant et son propre état mental. Ces éléments sont inaccessibles directement par les autres agents : c'est le principe d'encapsulation. D'autre part, la mise en œuvre indépendante, concurrente et asynchrone de ces comportements permet au système d'exécuter l'algorithme multi-agents de manière décentralisée.

Spécification d'un comportement d'agent

Le comportement d'un agent peut être décrit par un automate fini déterministe (en anglais, *finite state machine*) qui spécifie comment l'agent doit réagir au premier message de sa file en fonction de son état courant et de son état mental (cf. figure 1).

Les actions possibles d'un agent sont :

— l'exécution d'actions sur l'environnement ;
— la création d'un autre agent ;
— la mise à jour de son état mental ;
— l'envoi de messages (dénoté par l'opérateur `!`).

Un agent peut envoyer un message à l'un de ses pairs ou à lui-même (dénoté par la suite `self`). Dans ce dernier cas, un agent est en mesure de déclencher son propre comportement.

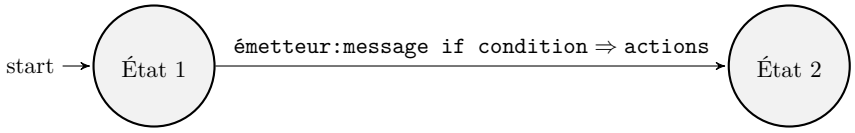

FIGURE 1 – Automate fini déterministe spécifiant le comportement d'un agent. Quand il reçoit un `message` de la part d'un `émetteur` dans l'`état 1` et que la `condition` est remplie, l'agent effectuent les `actions` et passe dans l'`état 2`.

Exemple 1 (Spécification d'un comportement d'agent) *L'automate de la figure 2 spécifie le comportement de l'agent Bruce lors de ses interactions avec un agent* `reckless` *(téméraire). L'agent Bruce possède un état mental constité d'une unique variable* `patience` *qui représente son niveau de*

patience. L'état initial de l'agent Bruce est l'état Banner*. Tant qu'il lui reste de la* `patience`*, l'agent Bruce accepte de recevoir le message* `Flick` *(pichenette) de la part de l'agent* `reckless` *mais perd progressivement* `patience`*. En revanche, lorsque sa* `patience` *est nulle et qu'il reçoit une* `Flick`*, l'agent Bruce passe dans l'état* Hulk *et envoie un message* `Warning` *(avertissement) à l'agent* `reckless`*. Dans l'état* Hulk*, l'agent Bruce répond systématiquement à une* `Flick` *par un* `HulkSmash`*. Après avoir délivré un* `HulkSmash`*, l'agent Bruce s'envoie un message* `Calm` *qui, lorsqu'il le traite, déclenche la réinitialisation de sa variable* `patience` *et l'agent repasse dans l'état* Banner*.*

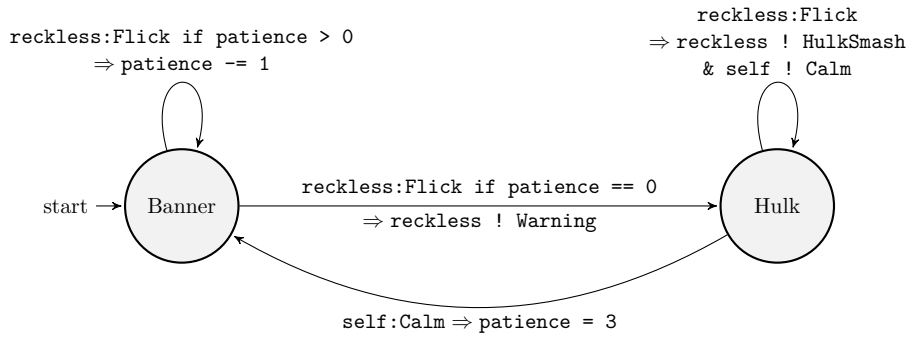

FIGURE 2 – Spécification du comportement de l'agent Bruce issu de l'exemple 1.

Les automates permettent une spécification complète et formelle du comportement des agents.

Implémentation d'un comportement d'agent

Les comportements d'agents peuvent être implémentés grâce à la boîte à outils Akka (Lightbend, Inc) qui permet la construction d'applications réactives, concurrentes et distribuées.

Pour illustrer la relation directe entre la spécification d'un comportement par un automate fini détermiste et son implémentation en grâce à la boîte à outils Akka, la figure 3 présente l'implémentation du comportement de l'agent Bruce de l'exemple 1. Selon l'API Akka, La classe `ActorSystem` permet de déployer un système multi-agents. Comme évoquée à la ligne 26, la classe `Actor` définit les méthodes/propriétés d'un agent et la classe `FSM` permet décrire son comportement grâce à un automate fini déterministe (en anglais, *finite state machine*). D'abord est énumérée la liste exhaustive des types de messages échangés (lignes 8 à 12) puis celle des états courants possibles pour l'agent (lignes 17 à 19). L'état mental de l'agent contient l'unique variable `patience` initialisée à 3 (ligne 25). Un agent `Bruce` est

associé à un état courant et à un état mental (ligne 30). L'agent est initialisé (ligne 52) dans l'état courant `Banner` avec un nouvel état mental `Mind` (ligne 3). Dans cet état courant (ligne 35), il peut recevoir un message `Flick` : soit sa patience est positive et il reste dans le même état en décrémentant sa patience (ligne 36-38) ; soit sa patience est nulle et il passe dans l'état `Hulk` en émettant le message `Warning` (ligne 39-41). Dans l'état courant `Hulk`, l'agent peut recevoir un message `Flick` ou `Calm`. Dans le premier cas (lignes 46-50), il reste dans le même état courant en répondant par le message `HulkSmash` et en s'envoyant à lui-même le message `Calm`. Dans le second cas (ligne 51-452), l'agent retourne dans l'état courant `Banner` avec un état mental réinitialisé.

En résumé l'automate qui spécifie le comportement d'un agent s'implémente naturellement et directement grâce à Akka.

Références

William Douglas Clinger. *Foundations of actor semantics.* PhD thesis, Massachusetts Institute of Technology, 1981.

Carl Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3) :323–364, 1977.

Lightbend, Inc. Akka toolkit. `https://akka.io`, visited 2019-12-17.

```scala
//Copyright (C) Quentin BAERT and Maxime MORGE 2019
package cristalsmac.mas4data.sample
import akka.actor._
import akka.actor.Actor
/**
  * The messages
  */
sealed trait HulkMessage
object Flick extends HulkMessage
object Warning extends HulkMessage
object HulkSmash extends HulkMessage
object Calm extends HulkMessage

/**
  * The state of Bruce
  */
sealed trait BruceState
case object Banner extends BruceState
case object Hulk extends BruceState

/**
  * The mind of Bruce
  * @param patience is an integer representation of his memory
  */
final case class Mind(patience: Int = 3)

/**
  * Bruce is an agent, i.e. an actor with a behaviour and a state of mind
  */
class Bruce extends Actor with FSM[BruceState, Mind] {
  // Bruce starts as Banner with patience
  startWith(Banner, Mind())

  // When Bruce is Banner, the flicks decrease his patience until he becomes Hulk
  when(Banner) {
    case Event(Flick, mind) if mind.patience > 0 =>
      println("Bruce: -Hum...")
      stay using Mind(mind.patience-1)
    case Event(Flick, mind) if mind.patience == 0 =>
      println("Bruce: -Grr...")
      goto(Hulk) using Mind(0) replying Warning
  }

  // When Bruce is Hulk, flicks imply a smash and he calms down, i.e. become Banner
  when(Hulk) {
    case Event(Flick, _) =>
      stay replying{
        sender ! HulkSmash
        self ! Calm
      }
    case Event(Calm, _) =>
      goto(Banner) using Mind()
  }

  // Monitoring
  onTransition {
    case from -> to=> println(s"Bruce was $from and he becomes $to")
  }

  // Initialize Bruce
  initialize()
}
```

FIGURE 3 – Code qui implément le comportement illustré par la figure 2.