Multiple Inheritance in AgentSpeak(L)-style Programming Languages

Akshat Dhaon
Rem Collier

UCD School of Computer Science & Informatics,
University College Dublin, Belfield, Dublin 4, Ireland.

Overview

- Agent-Oriented Programming is a high-level programming paradigm for implementing intelligent distributed systems.
- Most of the AOP languages focus on the provision of support for intelligent decision making.
- Language design concerns such as modularity, reuse, code structure and performance have been neglected.
  - Existing AOP languages are rarely used, partially because large implementations are difficult to understand, maintain and reuse.
- This paper presents an abstract model of multiple inheritance for AgentSpeak(L) style languages.
  - Agent programs are decomposed into a set of inter-related agent classes.
  - Focus on AgentSpeak(L) because MI requires a run-time apparatus for rule selection and establishing relationships between agent classes.

AgentSpeak(L)

- Attempts to bridge the gap between theory and practice by mapping the BDI (Beliefs, Desires, Intentions) model to an event-driven language:
  - Plans – basic abilities of an agent.
  - Intentions – Plans chosen by the agent for execution.
  - Beliefs, Events & Intentions cumulatively form the state.

- Agent processes events relating to its internal decision making (goals) or its external environment (beliefs).
  - Two types of triggering events – related to the addition and deletion of beliefs and/or goals.

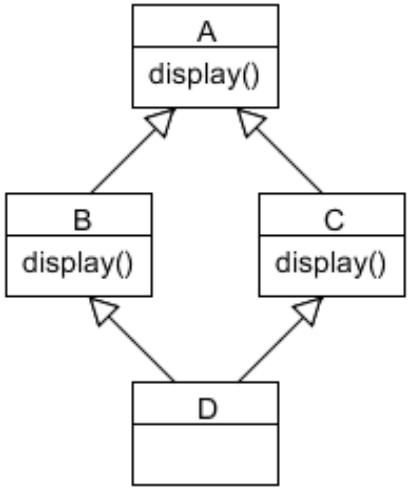
- Execution is governed by the dynamics of Event Selection, Rule Selection and Intention Execution.
  - Events are processed in order of occurrence, through their (contextual) matching to a plan which is then either adopted as an intention or appended to an existing intention.

Possible Approaches

- Single Inheritance
  - A class is derived from a single base/parent class.
- Multiple Inheritance
  - A class is derived from multiple parent classes.
- Mixins
  - Abstract classes that implement self-contained behaviours which can be “mixed in” to other classes as necessary.
- Interfaces
  - Defining a set of abstract methods that are to be implemented in any class that uses the interface.
- Traits
  - Groups of methods that serve as building blocks for classes and are primitive units of code reuse.
- Automated Delegation
  - Automate the forwarding of messages to contained classes.

MI Issues

- The Diamond Problem
  - Both B & C have their own implementation of display(). Which one should D inherit?



- Surprising Method Bindings
  - Inherited classes and superclasses ordered, based on priority.
  - May have surprising effects in large hierarchy.
  - Names may become ambiguous.
  - Ambiguously named methods compete for selection.
  - Selected definition may be non-deterministically chosen.

Overview of the Approach

- Agent class is a container for plan rules and a set of initial beliefs and goals.

Definition: Agent Class

`Class = <name, P, IS, R>`

- `name` is the name of the class.
- `P` is the list of parents of the class.
- `IS` is the initial state of the class.
- `R` is an ordered list of rules associated with the class.

Definition: Program

`Program = <C, A>`

- `C` is a set of agent classes associated with the program.
- `A` is a set of agents associated with the program.

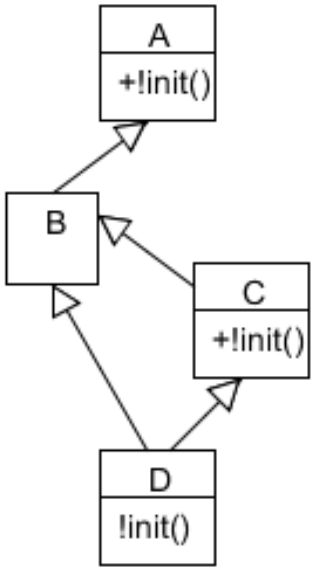
Definition: Agent

`Agent = <name, type, B, I, E>`

- `name` is the agent identifier.
- `type` is the name of the type (class) of the agent.
- `B` is the set of beliefs that the agent has.
- `I` is the set of intentions which the agent has.
- `E` is the event queue associated with the agent.

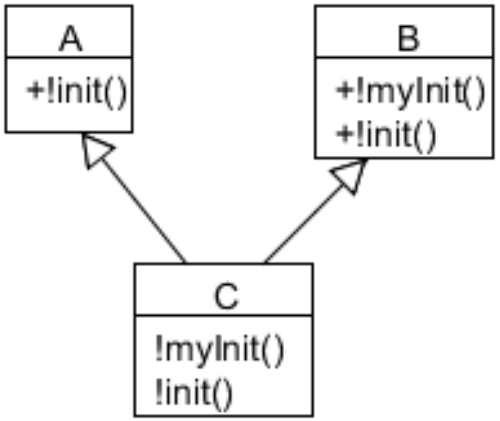
The MI Approach

- If a matching rule is found in the implementing class, that rule is used.
  - Otherwise the interpreter searches the hierarchy for a matching rule



- If multiple implementations of a rule exist in the hierarchy, preference is given to one which is at largest distance from the root of the hierarchy (closest to the calling agent class).

The MI Approach (contd.)



- If multiple implementations of plan rules exist at same level in the hierarchy, preference is given to one that is provided by the class on left in the "extends" line.
  - This is the default conflict resolution strategy and similar to the C ++ "virtual" functionality.
- Scope operator provided to restrict the scope of invocation.

Updated Syntax For AgentSpeak(L)

```
A ::= [agent <agent-name>
      [extends <agent-name> (, <agent-name>)*]<c>*
c ::= <initial> | <rule>
initial ::= <belief> | <goal>

rule ::= <event>: <context> <- (<statement>)*

statement ::= ? <belief> | <update> | <goal> | <action> | <scoped-goal> | <scoped-update>

update ::= +<belief> | -<belief>

scoped-goal ::= <agent-name> :: <goal>

scoped-belief ::= <agent-name> :: <update>
```

Simple Program:
(Fibonacci Number Generator)

```
agent Fibby {
    fib(1,1)
    fib(2,1)

    +!fib(X, N) : X < N &
         fib(X-1, Y) & fib(X-2, Z) <-
    +fib(X, Y+Z);
    !fib(X+1, N).

    +!fib(X, N) : X == N &
         fib(X-1, Y) & fib(X-2, Z) <-
        +fib(X, Y+Z).
}

agent 50Fibs extends Fibby {
    !fib(3,50)
}
```

Handling Scoped Events

- Scoping requires a change to the event model and an updated option selection algorithm:

```
Algorithm: selectOption(P, A, e)
    if e.type = ϕ then
        t <- A.type
    else
        t <- e.type
    endif

    classes <- getLinearization(P, t)
    while (classes != []) do
        cls <- head(classes)
        classes <- tail(classes)
        o <- selectOptionForClass(P,A,cls,e)
        if (o != ϕ) then
            return o
        endif
    endwhile
    return ϕ
```

Definition: Event

`Event = <te, i, type>`

- `te` is the triggering event.
- `i` is the source of the event (an intention or ϕ meaning it is an external source).
- `type` is the name of the scoped class or ϕ if not scoped.

Simple Examples

Extending Behaviours

```
agent A {
        +!init() <-
                println("Hi from A");
        }

        agent B extends A {
        +!init() <-
                println("Hi from B");
                A::!init();
        }
```

Combining Behaviors

```
agent A {
+!init() <-
        println("Hi from A");
}

agent B {
+!init() <-
        println("Hi from B");
}

agent C extends A, B {
+!init() <-
        A::!init();
        B::!init();
}
```

Simple Examples (contd.)

Defensive Programming

```
agent A {
count(0);

+!init() : count(x) & x<50<-
        A::!inc();
        //Do something
        !init();

+!inc() : count(x) <-
        -count(x);
        +count(x+1);
}
```

```
agent B extends A {
+!inc() : count(x) <-
        -count(x);
        +count(x-1);
}

agent C extends A, B {
        !init();
}
```

- Ability to extend behaviour can introduce unexpected side-effects.
- Can be restricted by the scope operator: Run-time “final” functionality.

Benefits of the MI Approach

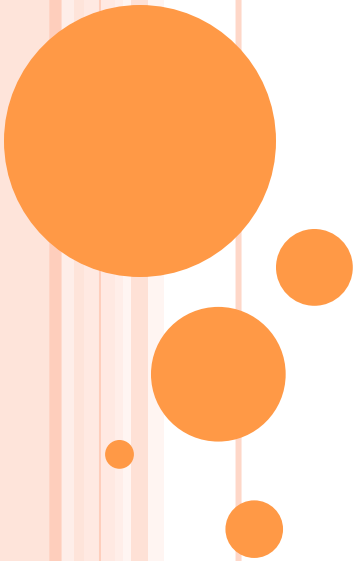
As shown in the simple case study presented in the paper, Multiple Inheritance demonstrates various benefits:

- Improved Quality of Code
  - Reusing existing (and tested) piece of code allows focus on only the new code as against the entire codebase. Makes the cycle of development, maintenance and testing simpler.
- Ability to decouple interaction logic from business logic
  - If interaction logic is not dependent on the business logic, then both can be developed, refactored and enhanced independently without having concerns over one affecting the other.
- Maintain clear link between design and implementation
  - Makes it possible to maintain the more natural role-based decomposition of methodologies through the mapping of roles to classes that are then combined into the concrete agent classes that are instantiated.
  - Promotes consistency between design and implementation.

Conclusion

- First attempt to provide support for MI in AOP.
- AOP hierarchies expected to be less complex than their OOP counterparts.
  - Agents are coarse grained entities inhabiting upper layers of complex systems (objects expected to be the building blocks of those systems).
  - Agents are intended to provide high level decision-making and coordination infrastructures.
- Focus on providing support to roles at runtime.
  - Roles are a common feature of Agent methodologies.
  - Provide better levels of abstraction.
  - Can be applied to multiple agents and vice-versa.
  - Often abstracted out of final design due to absence of clear mappings.
- Reference implementation: ASTRA
  - Typed variables; integrated with EIS and CArtAgO; extended set of plan operators
  - Available as Eclipse Plugin from: http://astralanguage.com

Questions?



Appendix

```
getLinearization(P, t)
        if (linearization = []) then
                queue <- queue+[t]
                while (queue != []) do
                        class <- head(queue)
                        queue <- tail(queue)
                        if (class.parents != []) then
                                while (class.parents != []) do
                                        if (!queue.contains(head(class.parents) &
                                        !priorQ.contains(head(class.parents)) then
                                                queue <- head(class.parents)
                                        endif
                                        class.parents <- tail(class.parents)
                                endwhile
                                priorQ <- addToPriorQueue(class, priorQ)
                        endif
                endwhile
                while (priorQ != []) do
                        if (!linearization.contains(head(priorQ)))
                                linearization <- [head(priorQ)]+linearization
                        endif
                        priorQ <- tail(priorQ)
                endwhile
        endif
        return linearization
```

Appendix (Contd.)

```
addToPriorQueue(class, priorQ)
        if (priorQ == []) then
                return [class]
        endif
        tempQ <- priorQ
        priorQ <- []
        inserted <- false
        while (tempQ != []) do
                claz <- head(tempQ)
                tempQ <- tail(tempQ)
                if (!inserted && (getDistance(class) > getDistance(claz))) then
                        priorQ <- priorQ + [class]
                        inserted <- true
                else
                        priorQ <- priorQ + [claz]
                endif
        endwhile
                if (!inserted) then
                        priorQ <- priorQ + [class]
                endif
        return priorQ
```

Appendix (Contd.)

```
getDistance(class)
        if (distFromRoot = -1) then
                maxDist <- 0
                tempPar <- parents
                while (tempPar != []) do
                        parent <- head(tempPar)
                        tempPar <- tail(tempPar)
                        d <- getDistance(parent)
                        if (d > maxDist) then
                                maxDist <- d
                        endif
                endwhile
                distFromRoot <- maxDist+1
        endif
        return distFromRoot
```

Appendix (Contd.)

```
agent Election {
   rule +!bully_election()
         : score( int score ) & participants( list agents ) {
      +holding("election");
      if (leader( string X )) -leader(X);

      forall (string receiver : agents)
         if (receiver ~= system.name() | failed_election(receiver))
            send (request, receiver, elect(system.name(), score));

      wait_for_deadline();

      if (holding("election"))
         forall (string agt : agents)
            send(inform, agt, elected(system.name()));

      foreach (failed_election(string name))
          -failed_election(name);
   }
```

Appendix (Contd.)

```
   rule @message(request, string sender, elect(string name, int score))
         : score( int my_score ) {
      if (score < my_score) {
         +failed_election(name);
         send ( inform, sender, result("ok") );
         if (~holding("election"))
            Election::!!bully_election();
      }
   }

   rule @message(inform, string sender, result("ok"))
         : holding("election") {
      -holding("election");
   }

   rule @message(inform, string N, elected(N)) {
      +leader(N);
   }

   plan wait_for_deadline() {
      system.sleep(2000);
   }
}
```