Introduction To Rebeca

Ehsan Khamespanah

Fall 2022

Actor Model

- A reference model for concurrent computation
- Consisting of concurrent, distributed active objects
  - Proposed by Hewitt as an agent-based language (MIT, 1971)
  - Developed by Agha as a concurrent object-based language (UIUC, since 1984)
  - Formalized by Talcott (with Agha, Mason and Smith), Towards a Theory of Actor Computation (SRI, 1992)

Rebeca

- The story of Rebeca … but unfortunately not this Rebeca!

- **Re**active o**bjec**t l**a**nguage
  (Sirjani-Movaghar, Sharif U. of Technology, 2001)

Rebeca

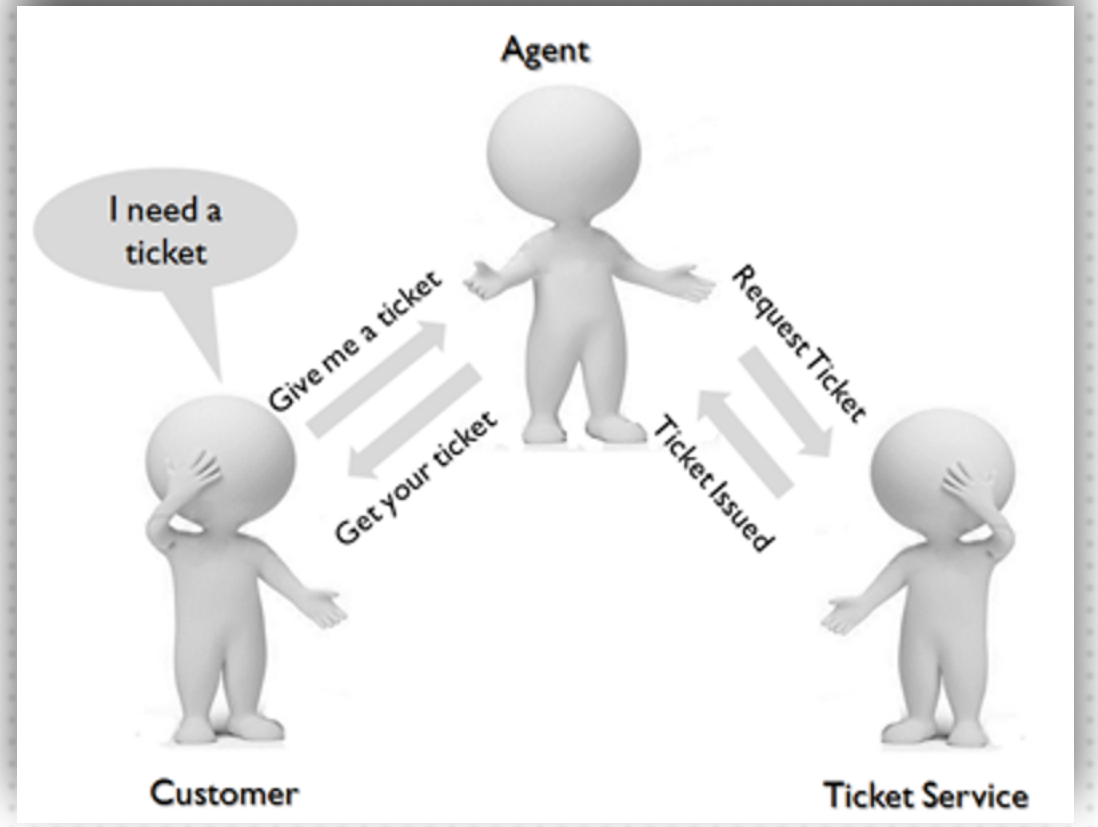
- Imperative Actor-based language
  - Concurrent reactive objects (OO)
  - Java like syntax
  - Simple core
  - Many features and extensions
  - Supported by a set of model checking and other analysis tools
- Conforms Hewitt-Agha's Actors

Inside a Rebeca Model

- Communication:
  - Asynchronous message passing: non-blocking send
  - Unbounded message queue for each rebec
  - No explicit receive
- Computation:
  - Take a message from top of the queue and execute it
  - Event-driven
  - Non-preemptive (atomic execution)

A Simple Rebeca Model

- Example of a ticket service system
- A Customer wants to buy a ticket



```
reactiveclass Customer (2) {
    knownrebecs {
        Agent a;
    }
    statevars {
        byte id;
    }
    Customer (byte myId) {
        id = myId;
        self.try();
    }
    msgsrv try() {
        a.giveTicket();
    }
    msgsrv ticketReady(byte id) {
        self.try();
    }
}
```

```
reactiveclass Customer (2) {
    knownrebecs {
        Agent a;
    }
    statevars {
        byte id;
    }
    Customer (byte myId) {
        id = myId;
        self.try();
    }
    msgsrv try() {
        a.giveTicket();
    }
    msgsrv ticketReady(byte id) {
        self.try();
    }
}
```

```
reactiveclass Customer (2) {
    knownrebecs {
        Agent a;
    }
    statevars {
        byte id;
    }
    Customer (byte myId) {
        id = myId;
        self.try();
    }
    msgsrv try() {
        a.giveTicket();
    }
    msgsrv ticketReady(byte id) {
        self.try();
    }
}
```

Actor type and its message servers

Assign value to state variables

```
reactiveclass Customer (2) {
    knownrebecs {
        Agent a;
    }
    statevars {
        byte id;
    }
    Customer (byte myId) {
        id = myId;
        self.try();
    }
    msgsrv try() {
        a.giveTicket();
    }
    msgsrv ticketRead
        self.try();
    }
}
```

Actor type and its message servers

Assign value to state variables

Asynchronous message sending

```
reactiveclass Customer (2) {
    knownrebecs {
        Agent a;
    }
    statevars {
        byte id;
    }
    Customer (byte myId) {
        id = myId;
        self.try();
    }
    msgsrv try() {
        a.giveTicket();
    }
    msgsrv ticketReady(byte id) {
        self.try();
    }
}
```

```
reactiveclass Agent (2) {
    knownrebecs {
        TicketService ts;
        Customer c;
    }
    msgsrv giveTicket() {
        ts.requestTicket();
    }
    msgsrv ticketIssued(byte id) {
        c.ticketReady(id);
    }
}
```

```
reactiveclass TicketService (3) {
    knownrebecs {
        Agent a;
    }
    statevars {
        int issuedTicket;
    }
    msgsrv requestTicket() {
        a.ticketIssued(1);
        issuedTicket++;
    }
}
main {
    Agent a(ts, c):();
    TicketService ts(a):();
    Customer c(a):(1);
}
```

Analysing Rebeca Models

- We developed a model checking toolset
  - Automatic deadlock and queue size analysis
  - Checking for satisfaction of assertions
  - CTL and LTL model checking
- We also developed some reduction techniques
  - Atomic execution are used to reduce the size of state spaces
  - Partial order and symmetry reductions