Implementing the Model Communication DEVS and Statechart

Thomas Feng

April 2, 2003

Email: thomas@email.com.cn

Homepage: http://moncs.cs.mcgill.ca/people/tfeng/

▶ Overview

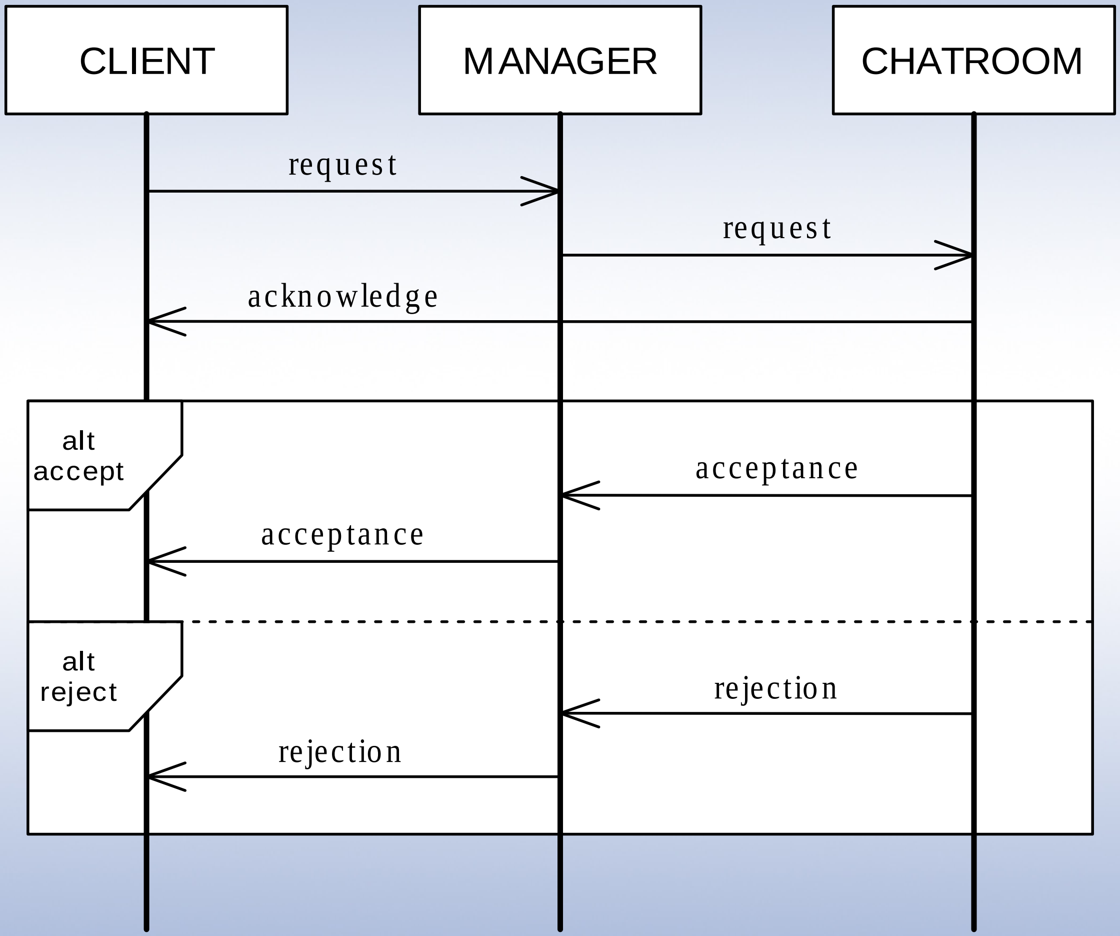
- A communication protocol
- Sequence diagram to illustrate the scheme
- A Non-realtime DEVS implementation
- A realtime DEVS implementation
- A realtime Statechart implementation
- A Non-realtime Statechart implementation

The following process is applied to each model implementation: design, coding, simulation, automatic validation (according to a rule file).

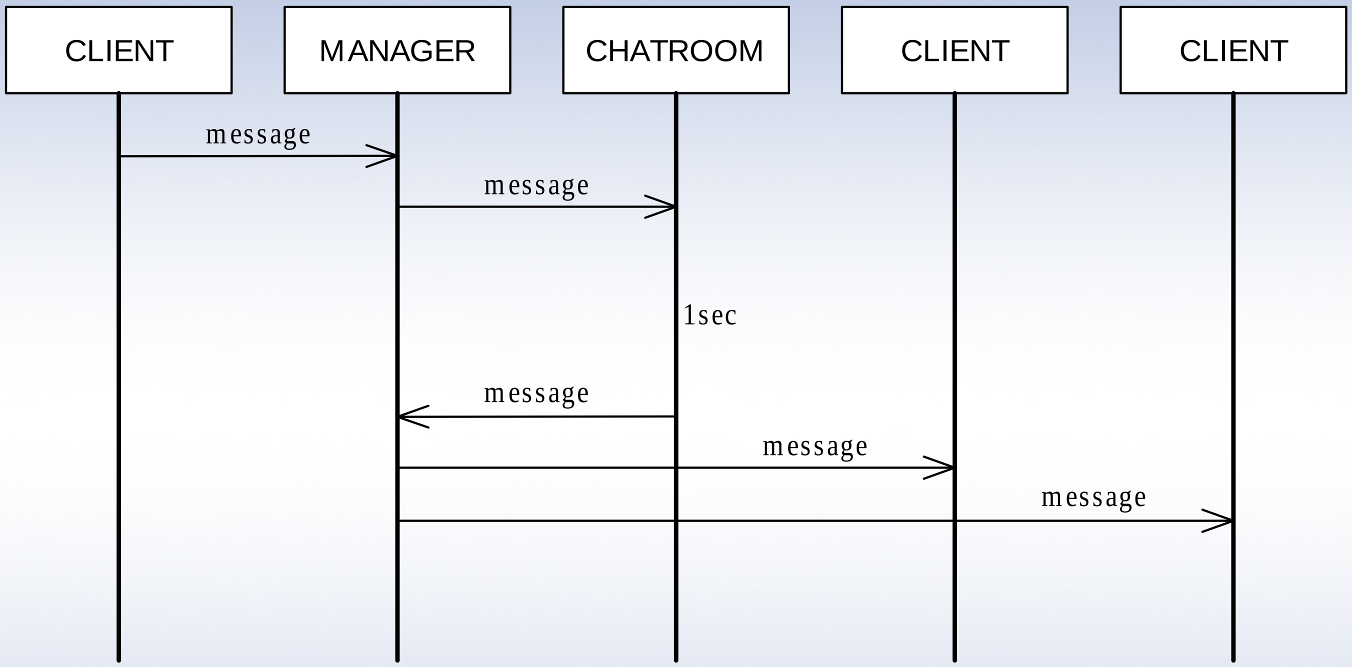
▶ COMMUNICATION PROTOCOL

- There are 5 clients and 2 chat rooms in the system. The clients must be connected to the chat rooms before they can randomly send messages.
- Initially, the clients are not connected. They try to connect to random chat rooms every 1 to 3 seconds (uniformly distributed).
- A chat room can accept at most 3 clients. It accepts a connection request iff its capacity is not exceeded.
- When connected, a client sends random messages to the chat room every 1 to 5 seconds (uniformly distributed). It takes 1 second for the chat room to process the message and broadcast it to all the other clients connecting to it. (The sender will not receive this message.)
- The clients immediately receive the broadcast messages.

▶ Chat Sequence Diagram (1)



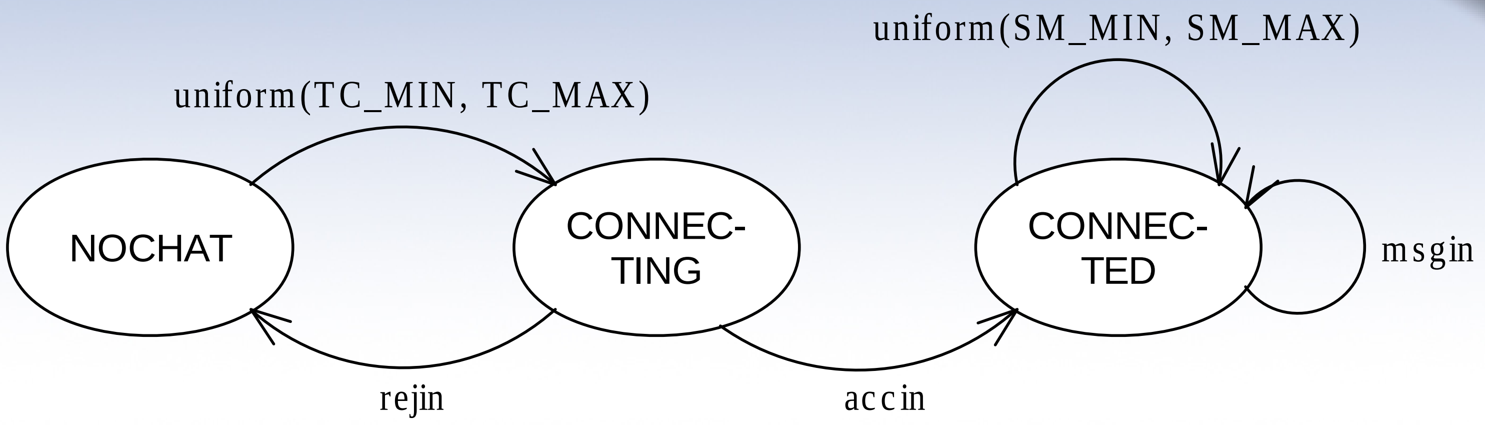
▶ Chat Sequence Diagram (2)



▶ DEVS Model Design

- CLIENT. An atomic DEVS repeatedly requesting for a connection every 1 to 3 seconds (uniformly distributed). Once connected, it repeatedly sends random messages every 1 to 5 seconds.
- CHATROOM. An atomic DEVS passively waiting for connection requests and messages. It accepts a connection request from a client iff it has less than 3 clients at that time.
- MANAGER. An atomic DEVS maintaining all the connections. CHATROOMs and CLIENTs are connected to only one MANAGER, which relays all the messages. When a CHATROOM broadcasts a message, the MANAGER sends the message to all the clients connected to this CHATROOM, except the sender.
- CHAT. A coupled DEVS hosting all these components.

▸ DEVS Model: CLIENT (1)



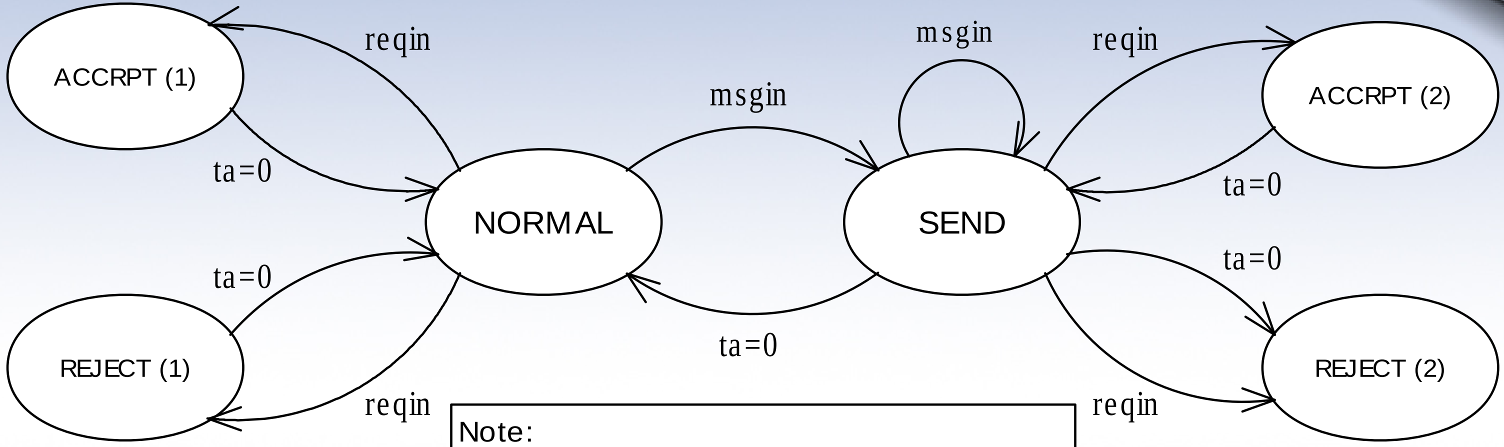
$$\begin{aligned}
S &= \{\text{NC, CNTING, CNTED}\}\\
\delta_{int}(\text{NC}) &= \text{CNTING}\\
\delta_{int}(\text{CNTING}) &= \text{CNTING}\\
\delta_{int}(\text{CNTED}) &= \text{CNTED}\\
ta(\text{NC}) &= \text{uniform(TC\_MIN, TC\_MAX)}\\
ta(\text{CNTING}) &= \infty\\
ta(\text{CNTED}) &= \text{uniform(SM\_MIN, SM\_MAX)}\\
\text{X} &= \{\text{accin, rejin, msgin}\}\\
\text{Y} &= \{\text{reqout, msgout}\}
\end{aligned}$$

MSDL

▶ DEVS Model: CLIENT (2)

$$\begin{aligned}
\delta_{ext}((\mathsf{CNTING},e),\mathsf{accin}) &= \mathsf{CNTED}\\
\delta_{ext}((\mathsf{CNTING},e),\mathsf{rejin}) &= \mathsf{NC}\\
\delta_{ext}((\mathsf{CNTED},e),\mathsf{msgin}) &= \mathsf{CNTED}\\
\lambda(\mathsf{CNTED}) &= \mathsf{msgout}\\
\lambda(\mathsf{NC}) &= \mathsf{reqout}
\end{aligned}$$

▶ DEVS Model: CHATROOM (1)



Note:
There're two reqin external transitions for NORMAL state and SEND state because the other states, messages, mcnt and clientnum, are not shown for simplicity. The model is deterministic.

$$S = \{\text{A1,R1,NORM,SEND,A2,R2}\} \times \text{messages} \times \text{mcnt} \times \text{clientnum}$$

$$\delta_{int}((\text{A1/R1}, \ldots)) = (\text{NORM}, \ldots)$$

$$\delta_{int}((\text{A2/R2}, \ldots)) = (\text{SEND}, \ldots)$$

$$\delta_{int}((\text{SEND,messages,mcnt}, \ldots)) = \begin{cases} (\text{SEND,messages,mcnt}+1, \ldots) & \textit{if mcnt<len(messages)-1} \\ (\text{NORM,[ ],0}, \ldots) & \textit{otherwise} \end{cases}$$

MSDL

McGill

▶ DEVS Model: CHATROOM (2)

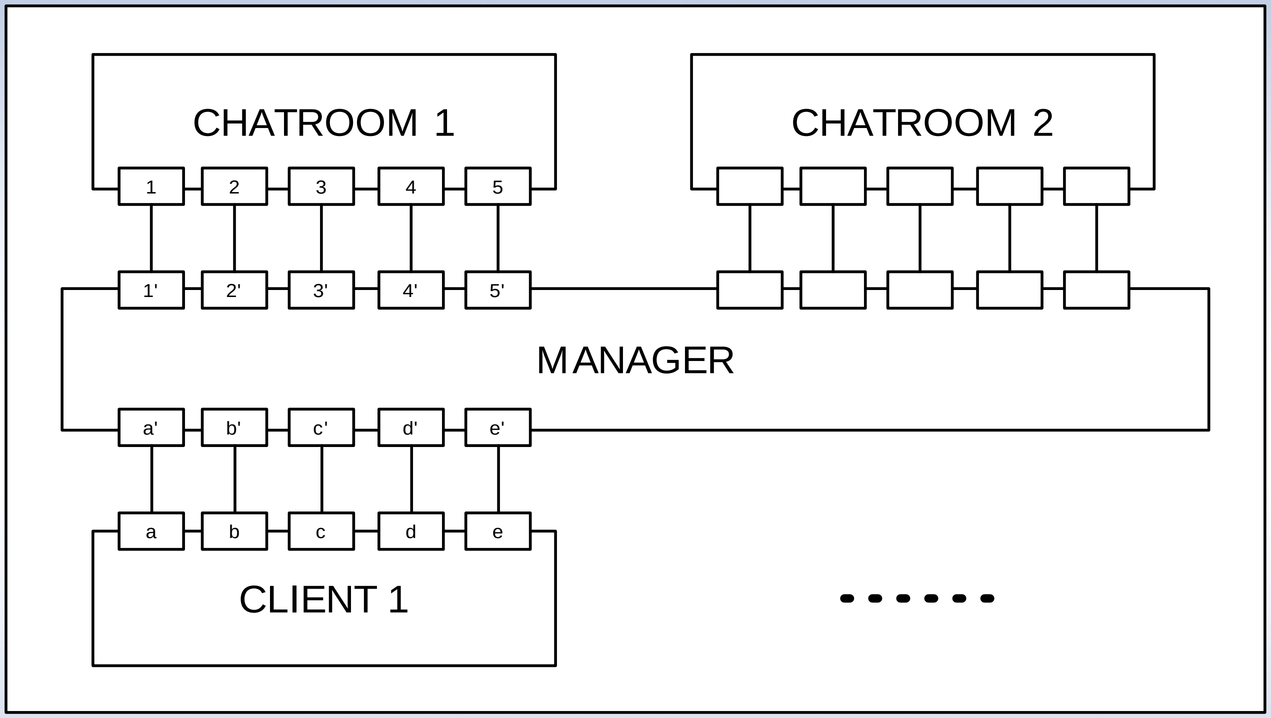
$$\begin{aligned}
ta((\text{A1/R1}, \ldots)) &= 0\\
ta((\text{A2/R2}, \ldots)) &= 0\\
ta((\text{SEND,messages,mcnt}, \ldots)) &= (\text{messages[mcnt]}.t_{rec}+1)\text{-}t_{cur}\\
&\qquad \textit{if } mcnt<len(messages)\\
&\quad 0 \textit{ otherwise}\\
ta((\text{NORM}, \ldots)) &= \infty\\
\text{X} &= \{\text{reqin, msgin}\}\\
\text{Y} &= \{\text{accout, rejout, msgout}\}\\
\delta_{ext}(((\text{NORM,clientnum}, \ldots),e),\text{reqin}) &= (\text{A1,clientnum+1} \ldots)\ \textit{if clientnum}<3\\
&\quad (\text{R1,clientnum} \ldots)\ \textit{otherwise}\\
\delta_{ext}(((\text{SEND,clientnum}, \ldots),e),\text{reqin}) &= (\text{A2,clientnum+1} \ldots)\ \textit{if clientnum}<3\\
&\quad (\text{R2,clientnum} \ldots)\ \textit{otherwise}\\
\delta_{ext}(((\text{NORM,messages}, \ldots),e),\text{msgin}) &= (\text{SEND,[msgin.msg]}, \ldots)\\
\delta_{ext}(((\text{SEND,messages}, \ldots),e),\text{msgin}) &= (\text{SEND,messages+[msgin.msg]}, \ldots)\\
\lambda((\text{SEND}, \ldots)) &= \text{msgout}\\
\lambda((\text{A1/A2}, \ldots)) &= \text{accout}\\
\lambda((\text{R1/R2}, \ldots)) &= \text{rejout}
\end{aligned}$$

MSDL

▶ DEVS Model: MANAGER

$$
\begin{array}{rcl}
\text{S} & = & \text{cons}\times\text{acts} \\
\lambda((\text{cons,acts})) & = & \text{carry out the first action } \textit{if len(acts)>0} \\
\delta_{int}((\text{cons,acts})) & = & (\text{cons,acts-[acts[0]]}) \\
ta((\text{cons,acts})) & = & 0 \textit{ if len(acts)>0} \\
 & & \infty \textit{ otherwise} \\
\text{X} & = & \{\text{reqin, rmsgin, cmsgin, accin, rejin}\} \\
\text{Y} & = & \{\text{reqout, rmsgout, cmsgout, accout, rejout}\} \\
\delta_{ext}(((\text{cons,acts}),e),\text{reqin}) & = & (\text{cons, acts+[send reqout to reqin.roomno]}) \\
\delta_{ext}(((\text{cons,acts}),e),\text{accin}) & = & (\text{cons+[accin.roomno,accin.clientno]}, \\
 & & \quad \text{acts+[send accout to client accin.clientno]}) \\
\delta_{ext}(((\text{cons,acts}),e),\text{rejin}) & = & (\text{cons, acts+[send rejout to client accin.clientno]}) \\
\delta_{ext}(((\text{cons,acts}),e),\text{cmsgin}) & = & (\text{cons, acts+[send rmsgout to chat room]}) \\
\delta_{ext}(((\text{cons,acts}),e),\text{rmsgin}) & = & (\text{cons, acts+[send cmsgout to client } x \text{ for any} \\
 & & x\in\text{cons[rmsgin.roomno] and } x\neq\text{rmsgin.sender]})
\end{array}
$$

▶ DEVS Model: CHAT



1.	msgout	1'.	rmsgin	a.	msgout	a'.	cmsgin
2.	msgin	2'.	rmsgout	b.	msgin	b'.	cmsgout
3.	reqin	3'.	reqout	c.	reqout	c'.	reqin
4.	accout	4'.	accin	d.	accin	d'.	accout
5.	rejout	5'.	rejin	e.	rejin	e'.	rejout

McGill

▶ Validating the Execution (1)

To validate the execution trace, we identify the following rules, formally convert them into regular expressions, and use automatic tools to validate the output.

1. When a client sends a connection request to a chat room, the chat room immediately responses by outputing a message.
2. On receiving a connection request, the chat room immediately makes a decision whether to accept the client or reject it.
3. When a chat room accepts a client, the client immediately receives the acceptance and dumps to the output.
4. When a chat room rejects a client, the client also immediately dumps the rejection.

MSDL

▶ Validating the Execution (2)

5. When a client sends a message, the chat room immediately receives it and output the receipt.
6. Exactly 1 second after a chat room receives a message, it broadcasts it. (This rule is not valid for the last second before end of the execution.)
7. The sender cannot receive its own message 1 second after it sends it.

▶ Limitations

- The time interval between trying connections and sending messages cannot be checked.
- The receiver of the broadcast cannot be checked, since this means the checker must know the run-time connection of each chat room.
- Messages are identified by time and content, not instance identity. This may cause trouble in rare cases.
- Rule checking is literal, which is not appropriate in realtime applications.
- And more . . .

▶ The Format of Rules

Each rule may consist of 4 parts in separate lines:

1. Pre-condition. The regular expression pattern that must appear somewhere in the output to enable the rule.
2. Post-condition. The regular expression pattern that must be found somewhere in the output to make the rule satisfied.
3. Rule validation (optional). The condition that must be satisfied to make the rule valid.
4. Counter (optional). The property that makes the rule a counter-rule.

▶ Examples

The rule to validate that the chat room receives the connection request at the same time a client sends it:

```
\* \* \* \* \* \* \* \* \* \* CLOCK: (\d+\.{0,1}\d*)\W*\n\n\(Client (\d+\.{0,1}
    \d*)\) A connection request is sent to chat room (\d+\.{0,1}\d*)\.\n
\* \* \* \* \* \* \* \* \* \* CLOCK: ([(\1)])\W*\n\n\(Chat Room [(\3)]\) Received
    connection request from client [(\2)]\.\n
```

The rule to validate that after 1 second from a chat room receives a message, it broadcasts it:

```
\* \* \* \* \* \* \* \* \* \* CLOCK: (\d+\.{0,1}\d*)\W*\n\n\(Chat Room
    (\d+\.{0,1}\d*)\) Received message "(.*?)" from client (\d+\.{0,1}\d*)\.\n
\* \* \* \* \* \* \* \* \* \* CLOCK: [(\1+1)]\W*\n\n\(Chat Room [(\2)]\)
    Sent message "[(\3)]" to all connected clients except client [(\4)]\.\n
[(\1+1)]<10
```

McGill

▶ Executing the DEVS Model

The `Chat.py` is a DEVS Model (both realtime and non-realtime). It first runs the model for 10 (virtual or real) seconds, and then invokes the rule checker to check the output according to the file `Chat.rules`.

Non-realtime version (PythonDEVS):

```
python Chat.py
```

Realtime version (RealtimeDEVS):

```
python Chat.py -realtime
```

Do expect to find a failure in the second run, because the rules cannot check realtime applications.

MSDL

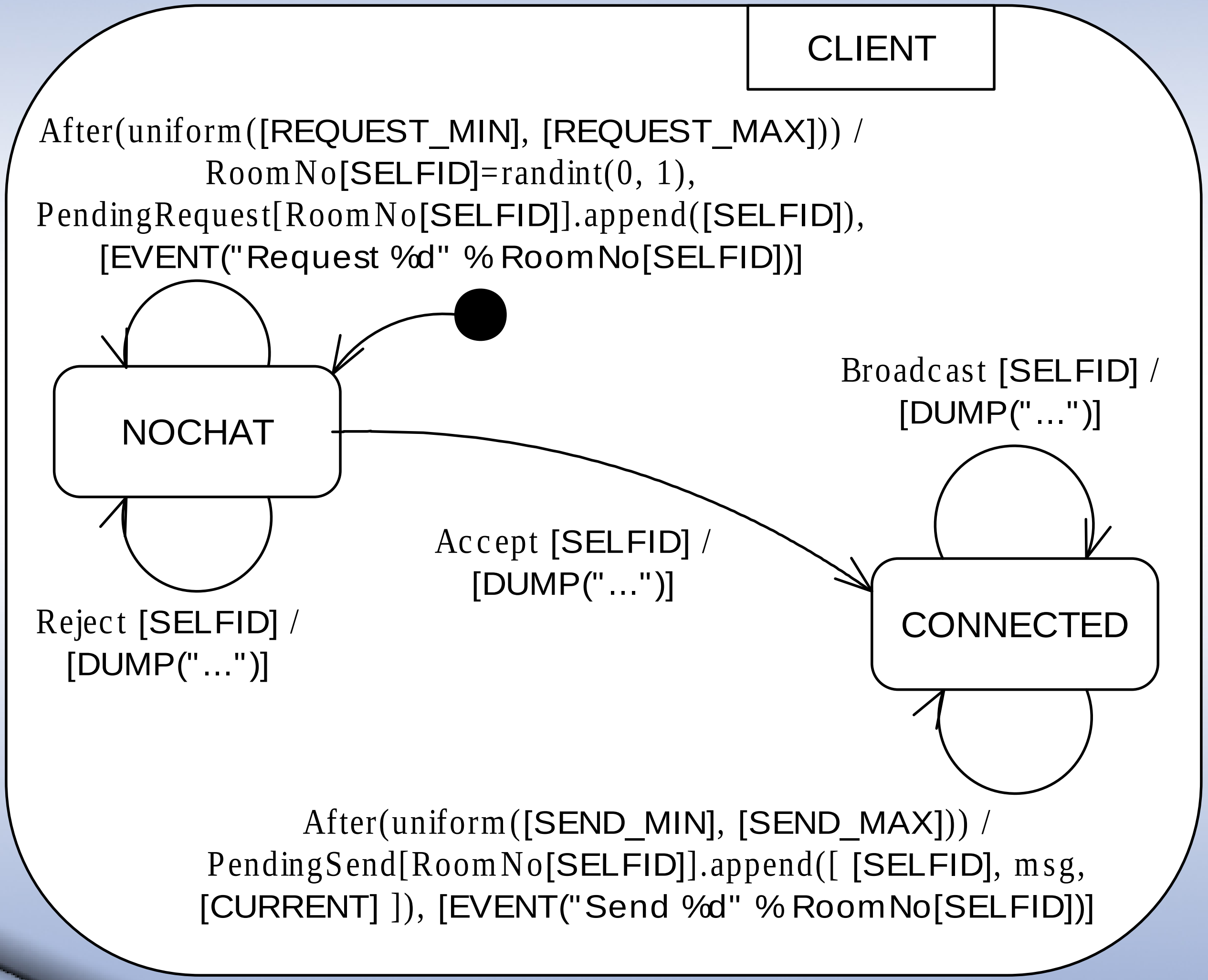
▶ Statechart Design

For comparison, the same protocol is simulated by statechart models, which run in SVM (Statechart Virtual Machine).

Conventions:

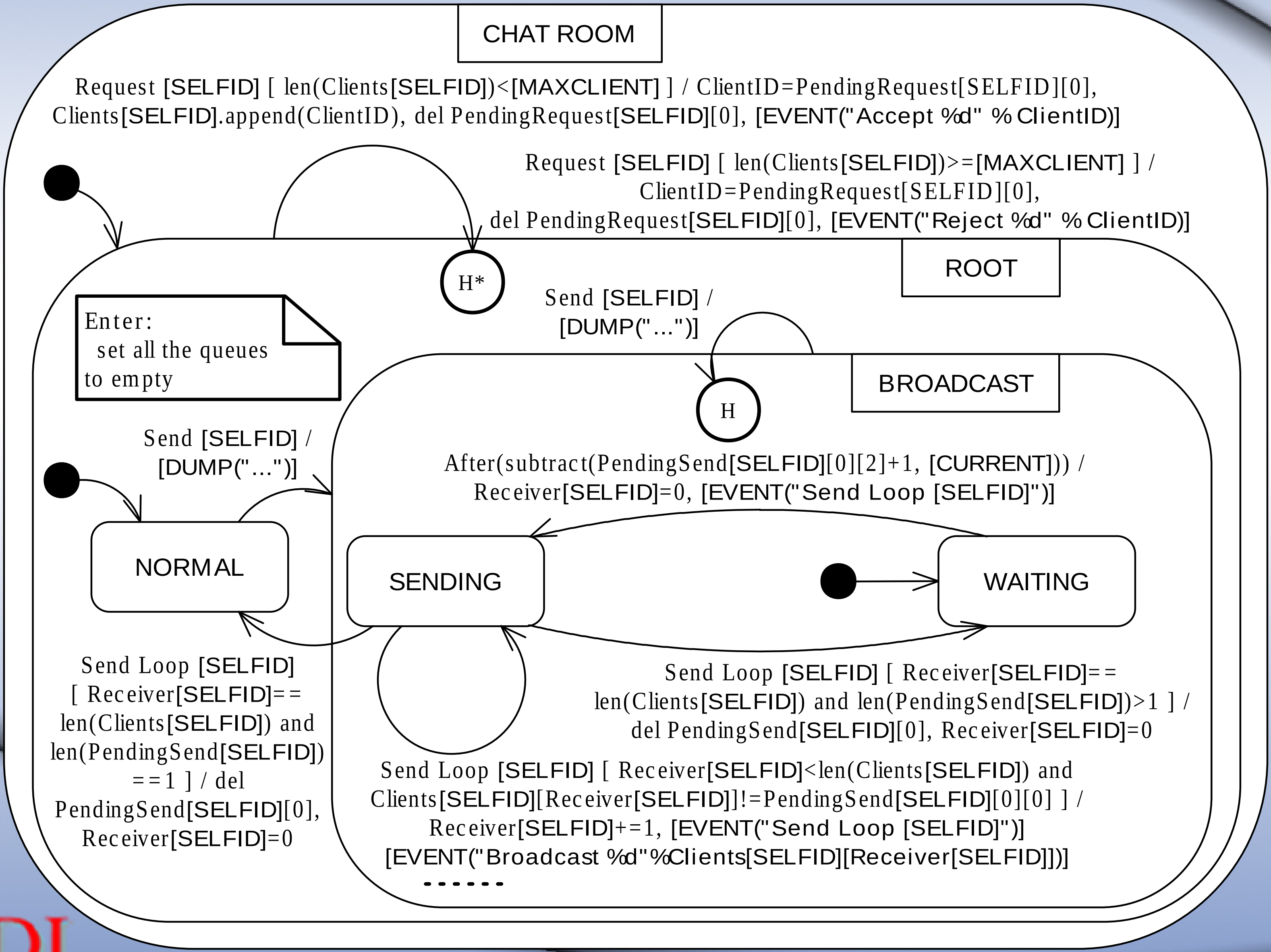
- Besides all the semantic elements in original statechart formalism, SVM also supports model initializer, finalizer and parameterized model importation. Some of these parts are not shown in the following statecharts, though they are necessary for the model execution.
- Macros can be defined in a model, and redefined by the reusing model or from the command line. Macros are placed in square brackets. They can carry parameters in parentheses. In the following charts, macros are shown in uppercase, bold Arial font to distinguish from guards and lists.

▶ Statechart: CLIENT


CLIENT
After(uniform([REQUEST_MIN], [REQUEST_MAX])) /
RoomNo[SELFID]=randint(0, 1),
PendingRequest[RoomNo[SELFID]].append([SELFID]),
[EVENT("Request %d" % RoomNo[SELFID])]
NOCHAT
Broadcast [SELFID] /
[DUMP("...")]
Accept [SELFID] /
[DUMP("...")]
CONNECTED
Reject [SELFID] /
[DUMP("...")]
After(uniform([SEND_MIN], [SEND_MAX])) /
PendingSend[RoomNo[SELFID]].append([[SELFID], msg,
[CURRENT]]), [EVENT("Send %d" % RoomNo[SELFID])]

MSDL

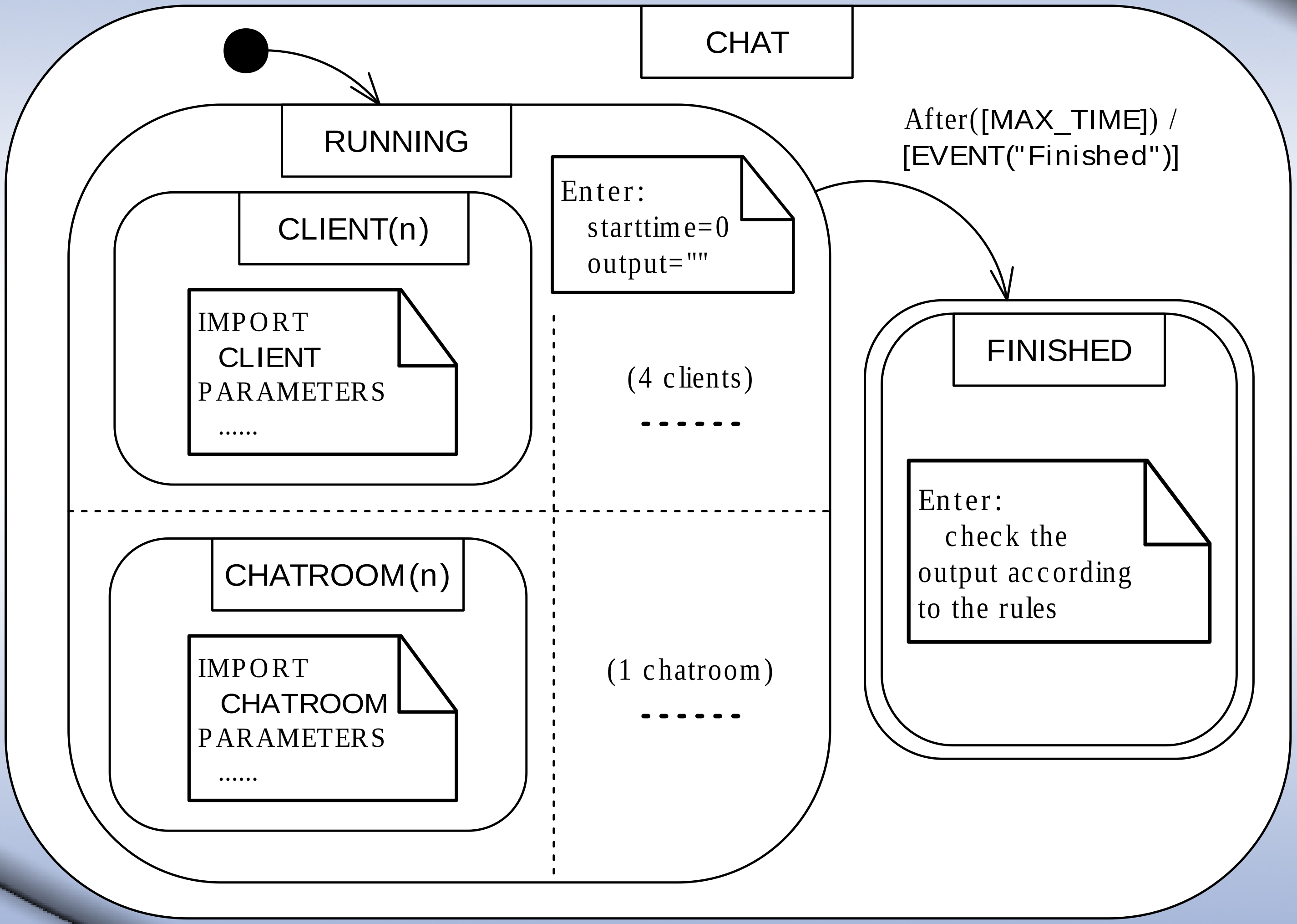
▶ Statechart: CHATROOM



CHAT ROOM
Request [SELFID] [len(Clients[SELFID])<[MAXCLIENT]] / ClientID=PendingRequest[SELFID][0], Clients[SELFID].append(ClientID), del PendingRequest[SELFID][0], [EVENT("Accept %d" % ClientID)]
Request [SELFID] [len(Clients[SELFID])>=[MAXCLIENT]] / ClientID=PendingRequest[SELFID][0], del PendingRequest[SELFID][0], [EVENT("Reject %d" % ClientID)]
H*
ROOT
Enter: set all the queues to empty
Send [SELFID] / [DUMP("...")]
H
BROADCAST
Send [SELFID] / [DUMP("...")]
After(subtract(PendingSend[SELFID][0][2]+1, [CURRENT])) / Receiver[SELFID]=0, [EVENT("Send Loop [SELFID]")]
NORMAL
SENDING
WAITING
Send Loop [SELFID] [Receiver[SELFID]== len(Clients[SELFID]) and len(PendingSend[SELFID]) ==1] / del PendingSend[SELFID][0], Receiver[SELFID]=0
Send Loop [SELFID] [Receiver[SELFID]== len(Clients[SELFID]) and len(PendingSend[SELFID])>1] / del PendingSend[SELFID][0], Receiver[SELFID]=0
Send Loop [SELFID] [Receiver[SELFID]<len(Clients[SELFID]) and Clients[SELFID][Receiver[SELFID]]!=PendingSend[SELFID][0][0]] / Receiver[SELFID]+=1, [EVENT("Send Loop [SELFID]")] [EVENT("Broadcast %d"%Clients[SELFID][Receiver[SELFID]])]

MSDL

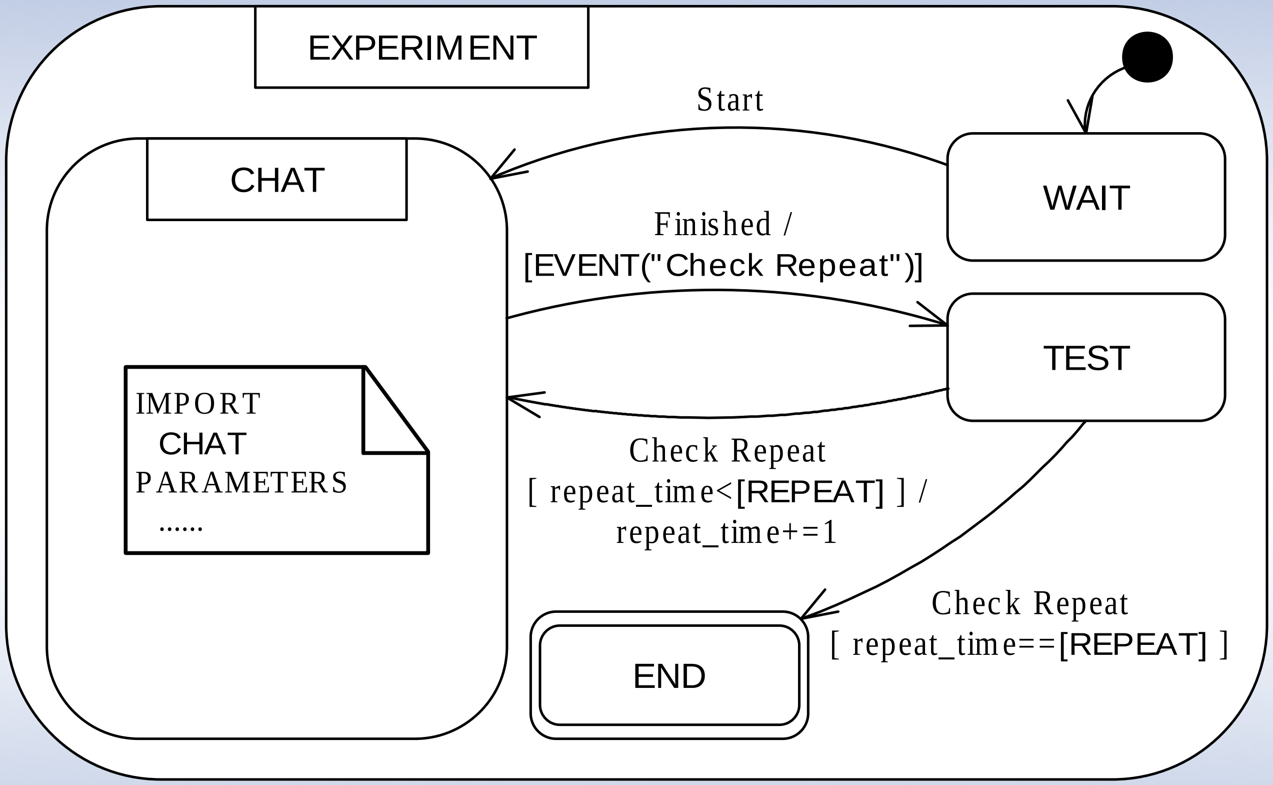
▶ Statechart: CHAT


CHAT
RUNNING
CLIENT(n)
IMPORT CLIENT PARAMETERS ......
Enter: starttime=0 output=""
(4 clients)

CHATROOM(n)
IMPORT CHATROOM PARAMETERS ......
(1 chatroom)

After([MAX_TIME]) / [EVENT("Finished")]
FINISHED
Enter: check the output according to the rules

▶ Statechart: EXPERIMENT



Run the model (do expect to get a failure):

```
./svm Chat/Experiment.des
```

▶ Non-realtime Statechart (1)

The fourth model described below is a non-realtime statechart implementation of the same protocol. It differs from the realtime version in:

- Instead of scheduling realtime events (after a certain time), the CLIENT and CHATROOM models schedule virtual time events, and wait for the "Time Advance" event from the CHAT model.
- CLOCK is added to the CHAT model as another orthogonal component. The "Check Time" event is triggered *whenever* the other threads are all waiting. Then, if it is not time to finish, the CLOCK advances the time by sending a "Time Advance" event. It must also set the current time to the minimum scheduled time.

▶ Non-realtime Statechart (2)

- All the other components receive the "Time Advance" event.
  The first thing they must do is to compare their last scheduled time with the current time, to determine if it is their turn.

- When the two times coincide for a transition, it is fired. It may schedule another event.

- The EXPERIMENT model is exactly the same, except that it loads another CHAT model.

Run the model:

```
./svm Chat/Experiment.des "TYPE=nrt"
```

▶ Summary (1)

Communication modelling with DEVS:

- A manager is set up to manage all the connections and relay all the messages between the sender and receiver.
- A message must carry the receiver's ID (if it is not broadcast).
- To establish the connections, the manager must know the format of some of the messages (i.e., the acceptance and rejection messages).

▶ Summary (2)

Communication modelling with Statechart:

- There is no manager. The clients and chat rooms themselves record their own connections.
- Messages are sent as events, which are global. So the receiver's IDs must be attached to the events.
- It is necessary for a non-realtime application to explicitly model the event scheduling (with a CLOCK orthogonal component in this example).

Validation based on rules is very strict, but it is applicable only to non-realtime applications.

- So much for today...

Thank you for your attendance!

Any problems or concerns, please email: thomas@email.com.cn