Component-based Chat Room Development in SVM (Statechart Virtual Machine)

Thomas Huining Feng and Hans Vangheluwe
MSDL, McGill
`http://msdl.cs.mcgill.ca/`

Introduction

With this case study (chat room), we demonstrate component-based model design in UML, and discuss the consistency problems in stages of the development process.

Component-based Design

- *Modularity*.
- *Reusability*.

Consistency

- *Intra-consistency*. Artifacts in the development process of a model must be consistent.
- *Inter-consistency*. All the components of a model must be consistently function together.

In this case study, we focus on intra-consistency.

Outline

Part I – An Introduction to SVM

- Statechart basics.
- SVM extensions to statecharts.

Part II – Chat Room Model Design

- Communication protocol of the chat room model.
- Class design.
- Sequence diagrams.
- Statecharts.
- Model execution in SVM.
- Conclusion.

Part I

An Introduction to SVM

SVM Design

The goal is to build a generalized simulator capable of executing statechart models.

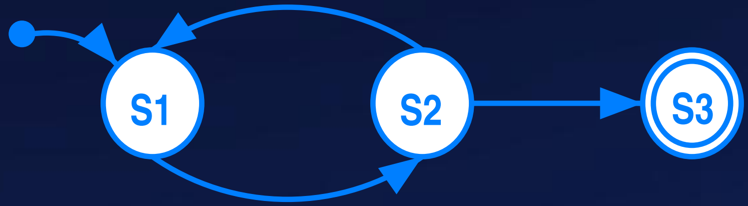
Design considerations:

- **Interpretation vs Compilation**. SVM is a statechart *interpreter*.
- **Virtual-time Simulation and Real-time Execution**. The same model can be simulated for analysis purpose and executed as a final product.
- **Model Specification**. A statechart model is specified in a text file, which is easy to handle.
- **Portability**. SVM is implemented in Python and Jython. It is portable to many operating systems and architectures.
- **Functionality**. SVM is capable of interpreting a model specified in the extended statechart formalism.

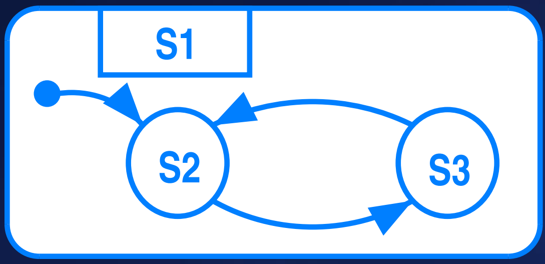
Statechart Introduction

Statechart (a discrete-event formalism) is a powerful tool to describe both software systems and physical systems.

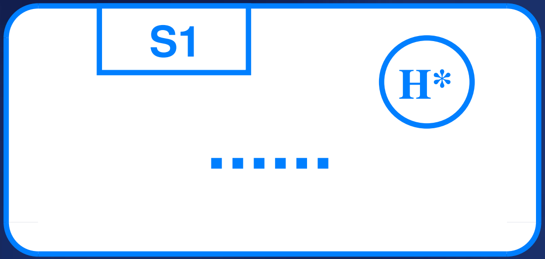
Statechart Elements



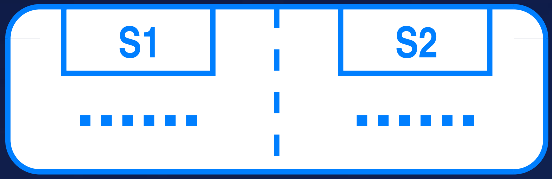
finite state automata



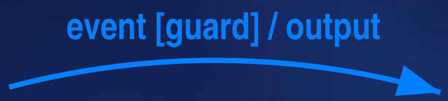
hierarchy



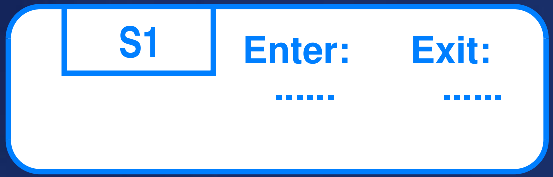
history



orthogonal components



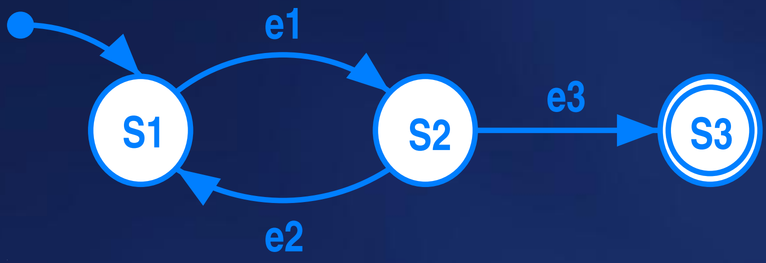
transition



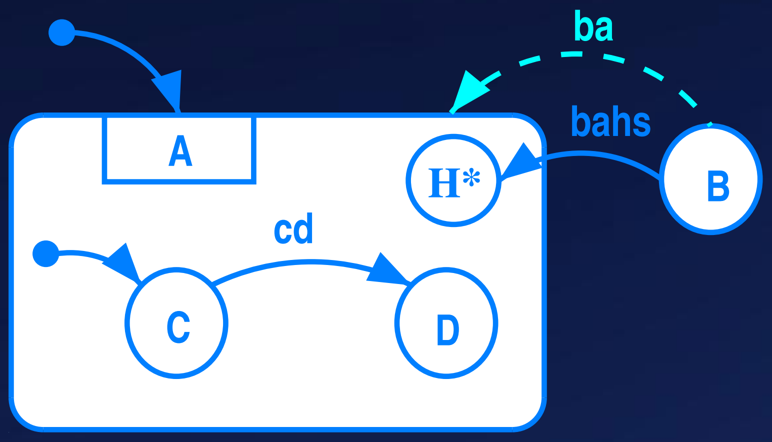
enter/exit actions

Simple Statechart Model

```
STATECHART:
  S1 [DS]
  S2
  S3 [FS]

TRANSITION:
  S: S1
  N: S2
  E: e1
  O: [DUMP("e1 is triggered")]

TRANSITION:
  S: S2
  N: S1
  E: e2
  O: [DUMP("e2 is triggered")]

TRANSITION:
  S: S2
  N: S3
  E: e3
  O: [DUMP("finish")]
```



Hierarchical Statechart Model



```
STATECHART:
  A [DS] [HS*]
    C [DS]
    D
  B
......
TRANSITION:
  S: A.C
  N: A.D
  E: cd

TRANSITION: [HS]
  S: B
  N: A
  E: bahs
......
```

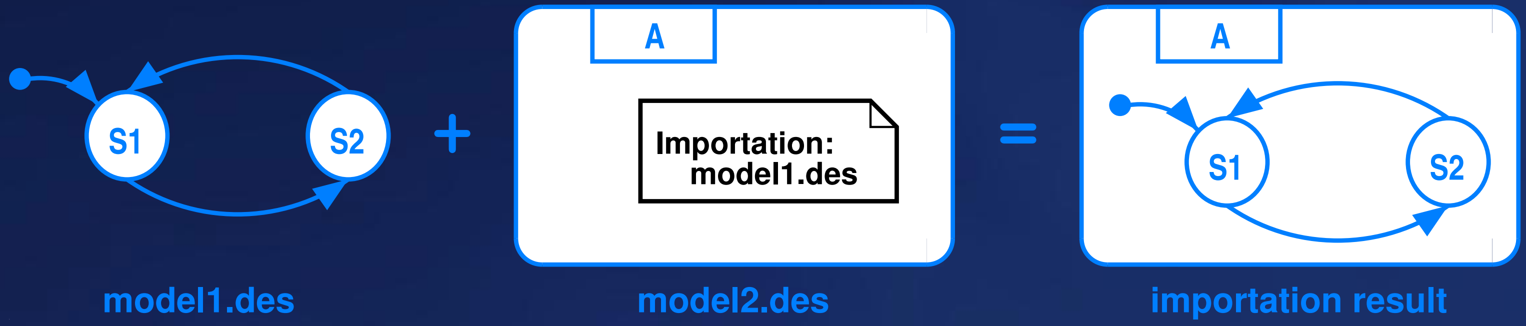
Extension 1: Model Importation

Motive

Statecharts are not modular and thus hard to reuse. The complexity of a statechart model exhibits itself even in solving small problems. It is desirable to divide a large model into smaller parts and assemble them after designing separately. Individual parts can also be reused.

SVM Extension

SVM presents a general idea of model importation. An imported model is a full-function model in its own right. When imported, all its states and transitions are placed in a state of the importing model.



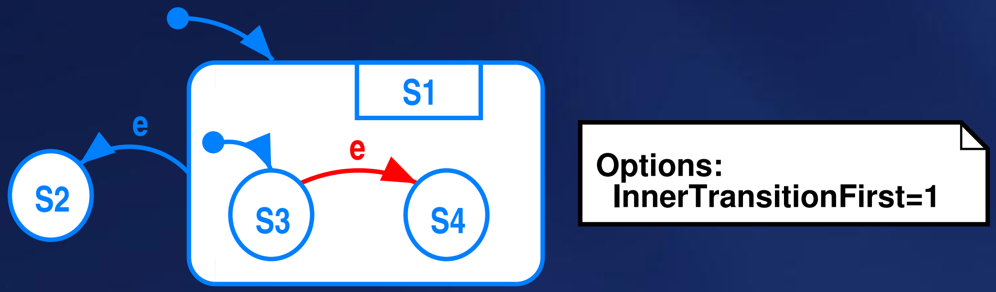
Extension 2: Transition Priorities

Motive

When two or more transitions are enabled by the same event, there is a conflict. In UML, if the source state of a transition is a substate of the source state of the other, it gets higher priority (inner-first); however, in the STATEMATE semantics, it gets lower priority (outer-first). It is desirable to enable both of these schemes.

SVM Extension A.

Every model has a global option: `InnerTransitionFirst`.



Options:
InnerTransitionFirst=1

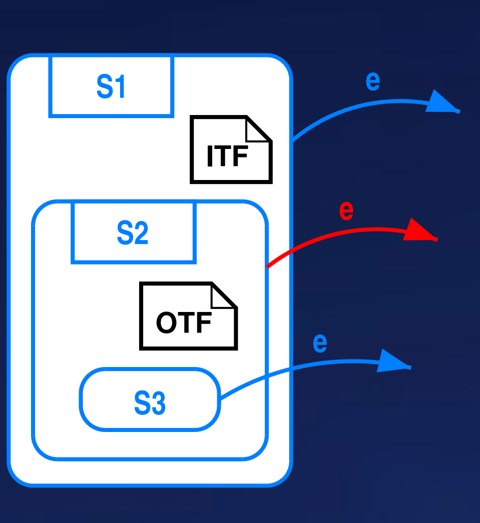
If the current state is S1.S3 and event e occurs, the new state will be S1.S4.

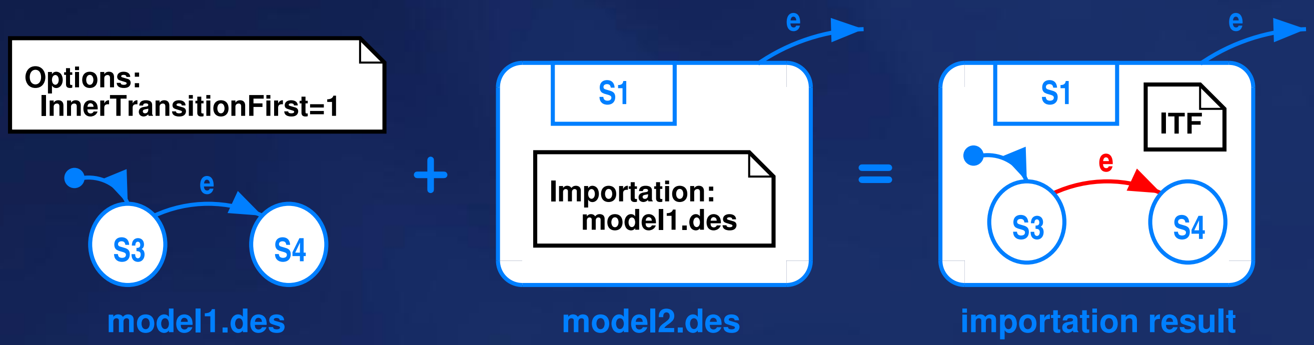
SVM Extension B.

Every state can be associated with one of the following properties:

- **ITF**. Inner transition first.
- **OTF**. Outer transition first.
- **RTO**. Reverse transition order. (If its parent state is ITF, it is OTF; vice versa.)

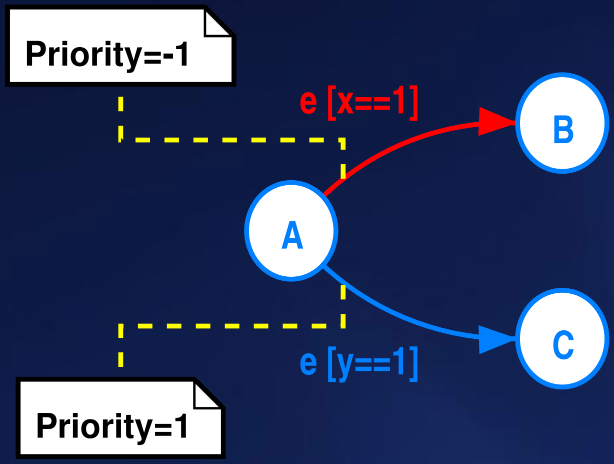
The property of a state override the setting of its parent *in its scope*.





SVM Extension C.

Every transition can be associated with an integer priority number (by default, it is 0). For conflicts which cannot be solved by extensions A and B, a transition with the smallest priority number is fired.



When e occurs, if the model is in state A and both conditions are true,

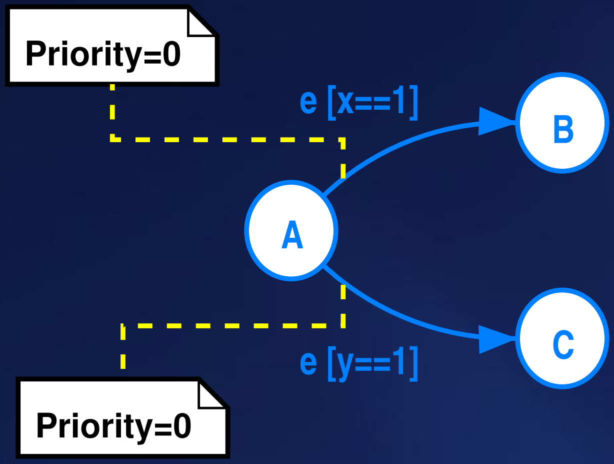
$$\begin{cases} x = 1 \\ y = 1 \end{cases}$$

the state will change to B.

SVM Extension D.

If conflicts still exist at run-time, which cannot be solved by extensions A, B and C, the choice is strictly random according to a uniform distribution.

This is usually caused by a design flaw, in which case the designer cannot foresee a potential conflict in the model.



Extension 3: Parametrized Model Templates

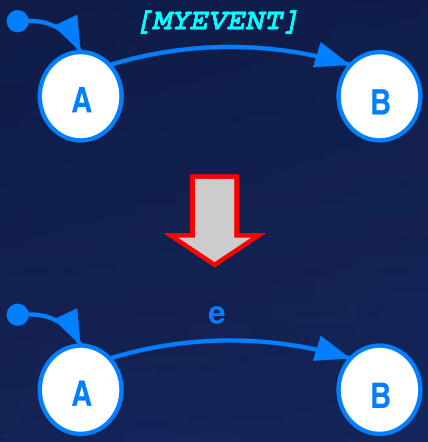
Motive

Usually a design cannot be reused in a new system without any change or customization. The importing model should be able to customize the imported model before placing it in one of its states.

The customization should be restricted and modular.

SVM Extension

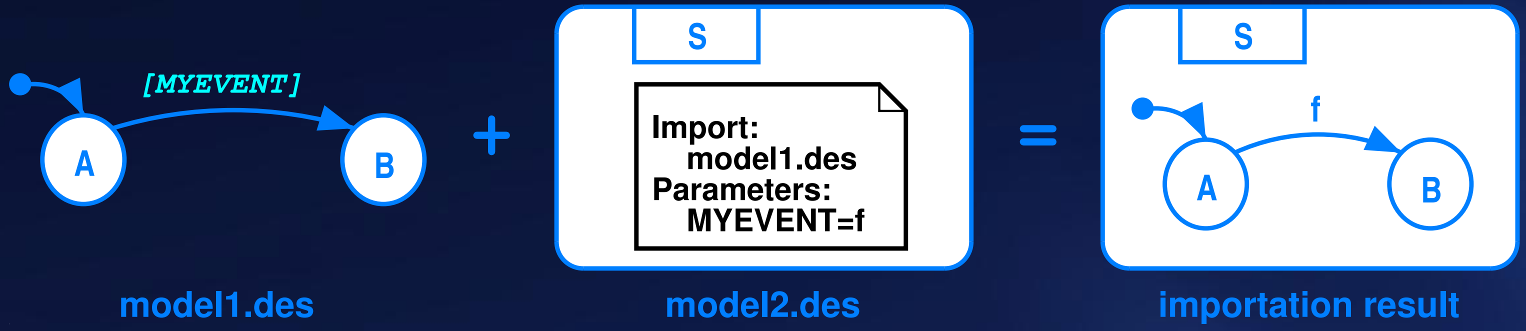
Macros can be defined in a macro and used anywhere in its description.



```
MACRO:
  MYEVENT = e
......
TRANSITION:
  S: A
  N: B
  E: [MYEVENT]
......
```

SVM Extension (Continued)

The designer is allowed to redefine the macros when reusing a model.

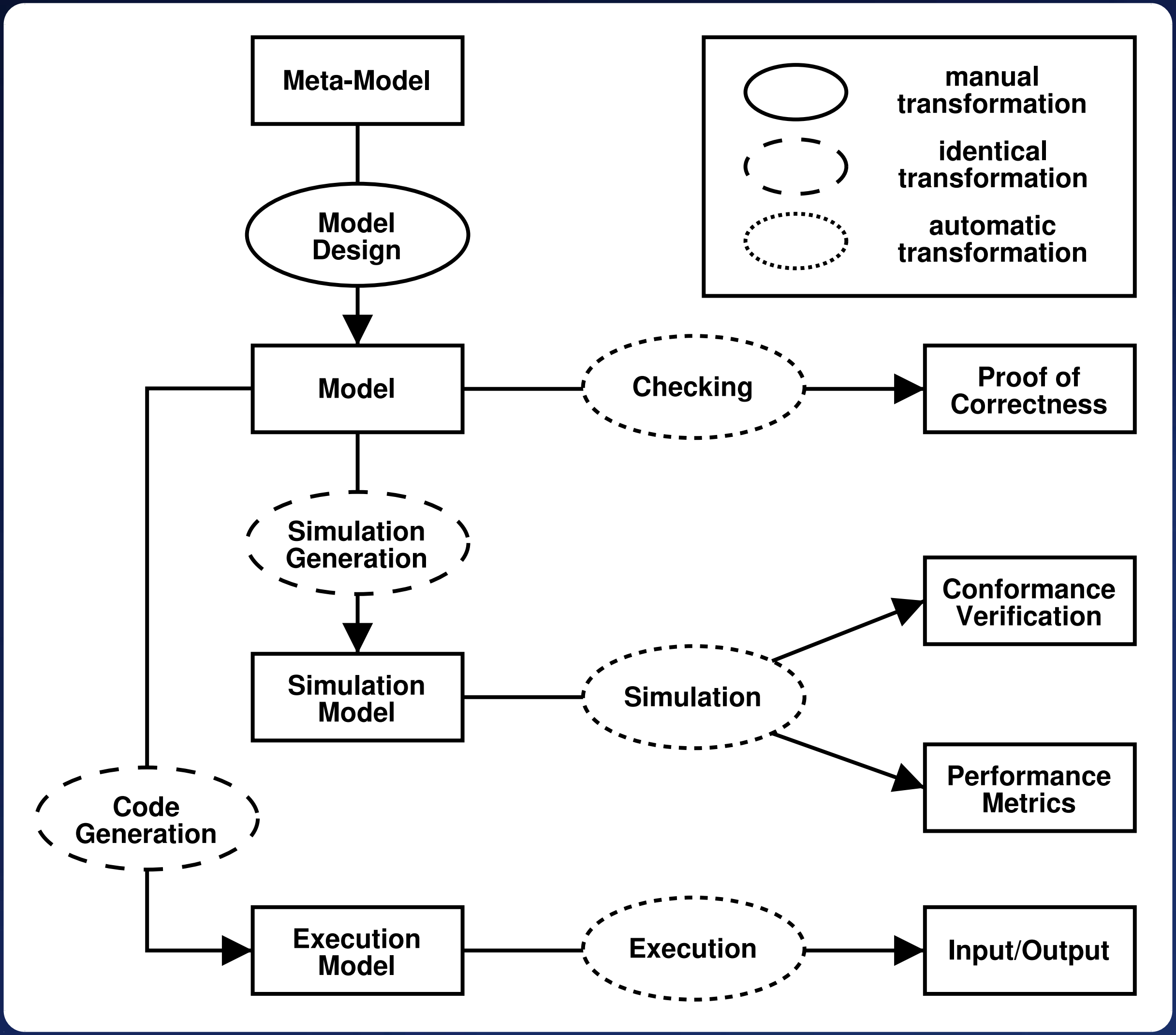


The outside world is able to modify the behavior of a model only by parameters, which is defined in the model with its consent. There is no way to modify its hard-coded parts.

Part II

Chat Room Model Design

Model-based Development Process



Communication Protocol

1. 5 clients and 2 chat rooms in the system. Initially clients not connected. They try to connect to a random chat room every 1 to 3 seconds. No delay for requests.
2. A chat room accepts at most 3 clients. It accepts a connection request if and only if its capacity is not exceeded.
3. The requesting client receives an acceptance or rejection notice immediately.
4. A client must be accepted by a chat room before it may send chat messages.
5. When connected, a client sends random messages to its chat room every 1 to 5 seconds. No delay for messages. The chat room takes 1 second to process a message and broadcast it to all *other* clients connected to it.
6. No delay for the broadcast.

Class Design

- `ChatRoom`. 2 instances are required. Each of them receives incoming requests and messages.

```
    request(clientID, roomID)
    send(clientID, roomID, msg)
```

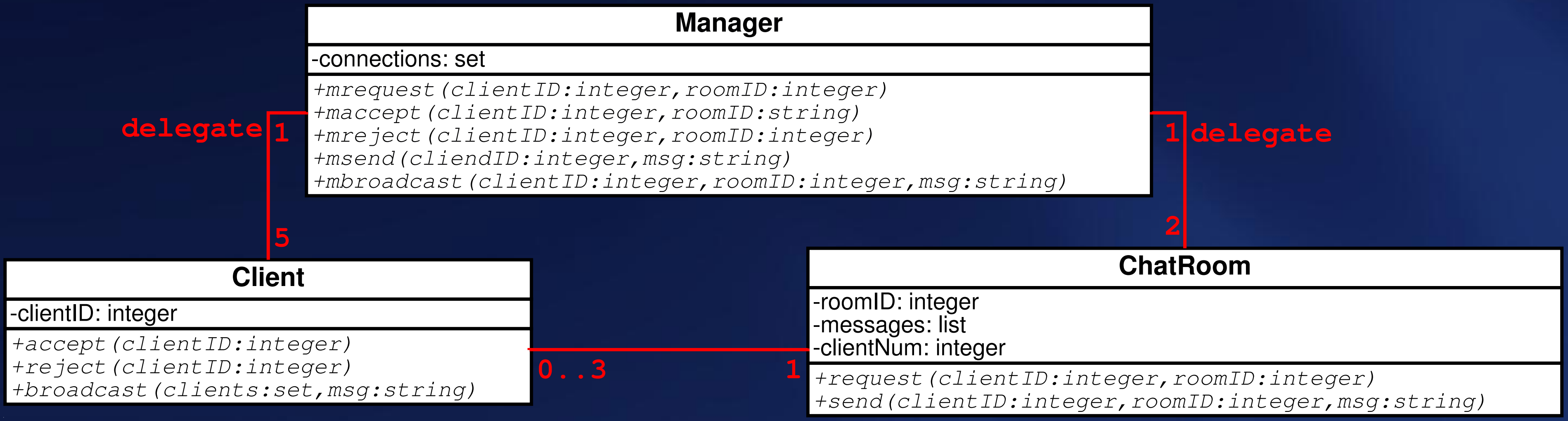
- `Client`. 5 instances.

```
    accept(clientID)
    reject(clientID)
    broadcast(clients, msg)
```

- `Manager`. 1 instance that relays all the events between clients and chat rooms.

```
    mbroadcast(clientID, roomID, msg)
```

Class Design (Continued)


Manager
-connections: set
+mrequest(clientID:integer,roomID:integer)
+maccept(clientID:integer,roomID:string)
+mreject(clientID:integer,roomID:integer)
+msend(cliendID:integer,msg:string)
+mbroadcast(clientID:integer,roomID:integer,msg:string)
delegate
1
5
1
delegate
2
Client
-clientID: integer
+accept(clientID:integer)
+reject(clientID:integer)
+broadcast(clients:set,msg:string)
0..3
1
ChatRoom
-roomID: integer
-messages: list
-clientNum: integer
+request(clientID:integer,roomID:integer)
+send(clientID:integer,roomID:integer,msg:string)

MSDL

Verification 1: Class Diagram → Protocol

Though this API definition is not functional, the behavior behind the interface is easily understood. Checking its consistency with the protocol is however difficult or even impossible because of the following reasons:

- Behavior is hidden behind the interface.
- The protocol is specified in natural language.
- For a well-defined system there can be a number of interface designs. They may differ substantially.

Sequence Diagrams

The sequence diagrams bring the design to a lower level of abstraction (higher level of detail) than the class diagram.

Timing

In the protocol description, more than one action can happen at the same time, though they may be causally related.

`request at time 1; accept at time 1` $\surd$

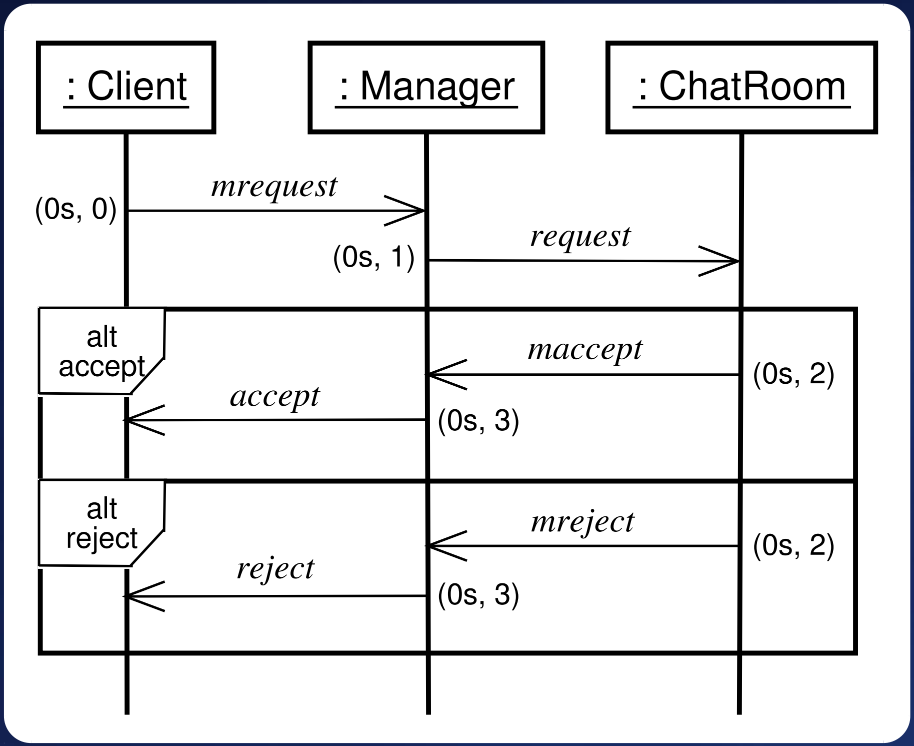
`accept at time 1; request at time 1` $\times$

A tuple (t, s) is used to represent time.

`request at time (1.0s, 0); accept at time (1.0s, 1)` $\surd$

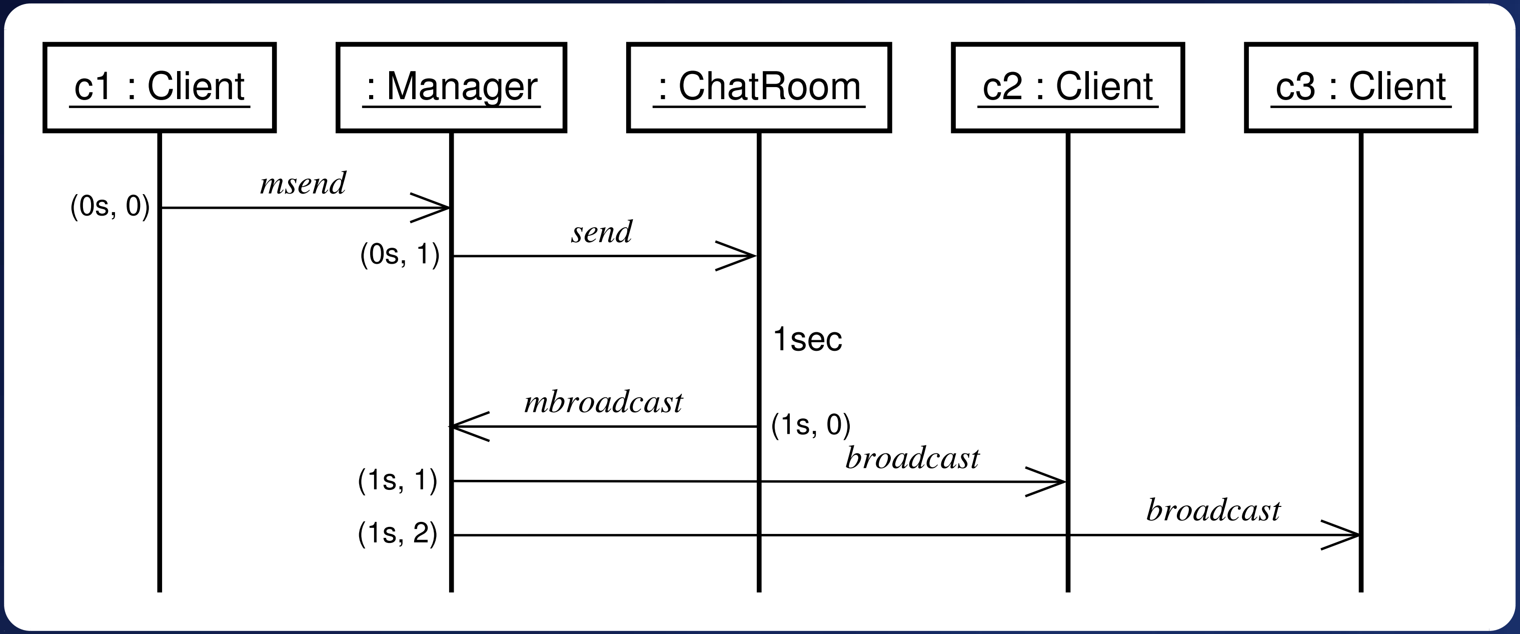
`accept at time (1.0s, 0); request at time (1.0s, 1)` $\times$

Sequence Diagrams (Continued)



Request pattern

Sequence Diagrams (Continued)



Message pattern

MSDL

Verification 2: Sequence Diagrams $\longrightarrow$ Class Diagram

Collect all the method calls (or incoming events) of a component and check if they have corresponding definitions in its class design.

For example:

In the request pattern, `Manager` *receives events* `mrequest`, `maccept` *and* `mreject`. *In the message pattern, it receives* `msend` *and* `mbroadcast`. *These 2 patterns cover all usage. So, its class design must have (and only have) definition for the corresponding 5 public methods.*

This process can be automated.

Verification 3: Sequence Diagrams $\longrightarrow$ Protocol

Consistency with the original protocol can only be partly checked.

For example:

In the request pattern, if a `ChatRoom` *receives a request at time 0, it accepts or rejects the* `Client` *at time 0. The absolute values of the two times are not important. Important is that the reply is sent back at exactly the same time, as specified in the protocol.*

A rule-based approach will be introduced in the latter part. (First convert the protocol into extended REs, then use the REs to check the sequence diagrams.) It can speed up this checking process.

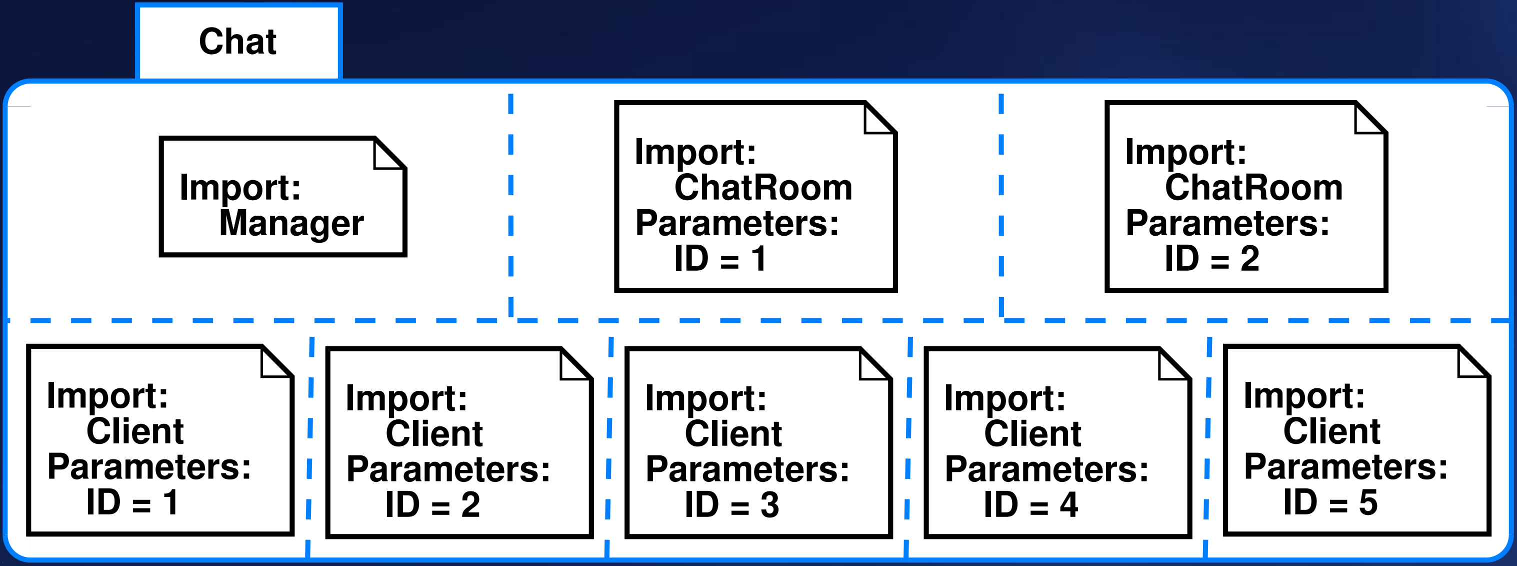
Verification 3: Sequence Diagrams $\longrightarrow$ Protocol (Continued)

However, checking is quite limited because of the expressiveness of sequence diagrams. Eg.:

- Sequence diagrams cannot describe "what should *not* happen at a certain time or in a certain period."
- In the request pattern, *if* a client sends an `mrequest`, *then* the manager sends a `request` without time advance, *then* the chat room sends `maccept` or `mreject` . . . Unfortunately, an inert client that does not send any request cannot be detected as a problem.

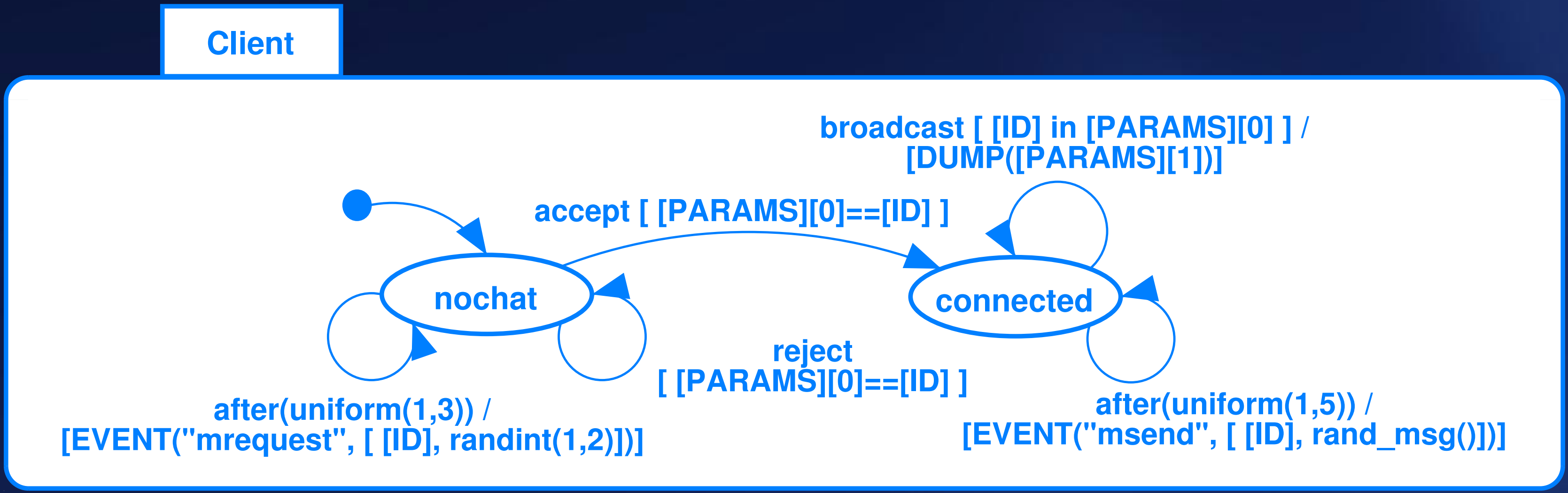
Statechart Design

Components Client, ChatRoom and Manager are designed in separate statecharts. Model Chat (the final system) imports five instances of Client, two instances of ChatRoom and one Manager.


Chat
Import: Manager
Import: ChatRoom Parameters: ID = 1
Import: ChatRoom Parameters: ID = 2
Import: Client Parameters: ID = 1
Import: Client Parameters: ID = 2
Import: Client Parameters: ID = 3
Import: Client Parameters: ID = 4
Import: Client Parameters: ID = 5

Client Component

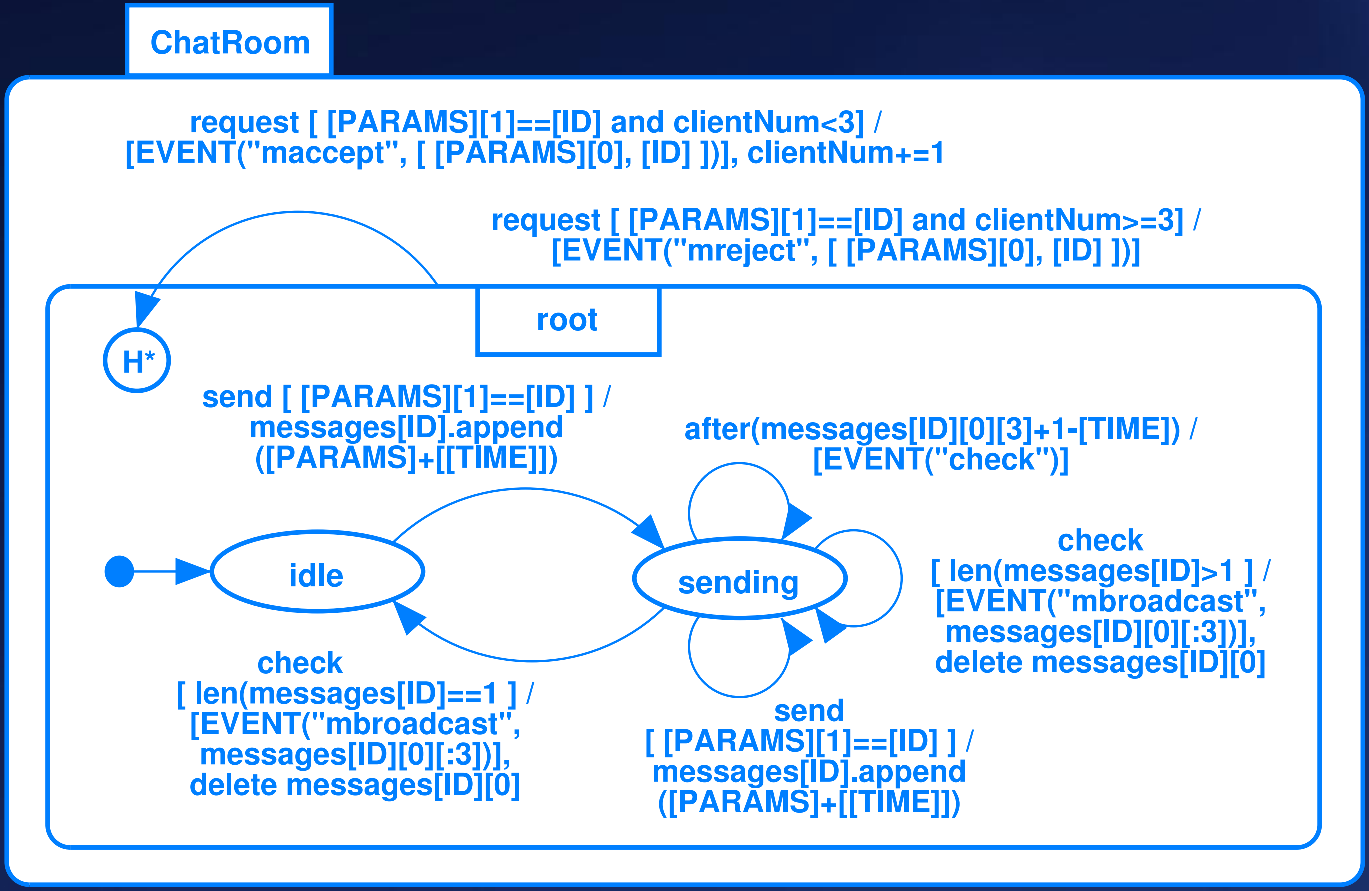
Initially, it is in the nochat state. It repeatedly tries to connect to the chat room via the manager by raising an mrequest event every 1 to 3 seconds (uniformly distributed), until the request is accepted.


Client
broadcast [[ID] in [PARAMS][0]] / [DUMP([PARAMS][1])]
accept [[PARAMS][0]==[ID]]
nochat
connected
reject [[PARAMS][0]==[ID]]
after(uniform(1,3)) / [EVENT("mrequest", [[ID], randint(1,2)])]
after(uniform(1,5)) / [EVENT("msend", [[ID], rand_msg()])]

- `uniform` is a Python function which returns a random real number in a range.
- `randint` returns a random integer.
- `[EVENT(...)]`, `[PARAMS]` and `[DUMP(...)]` are pre-defined. `[ID]` is user-defined.
- `after` event is raised at a certain time after a state is entered.
- `accept`, `reject` and `broadcast` are incoming events.
- `mrequest` and `msend` are outgoing events.

ChatRoom Component

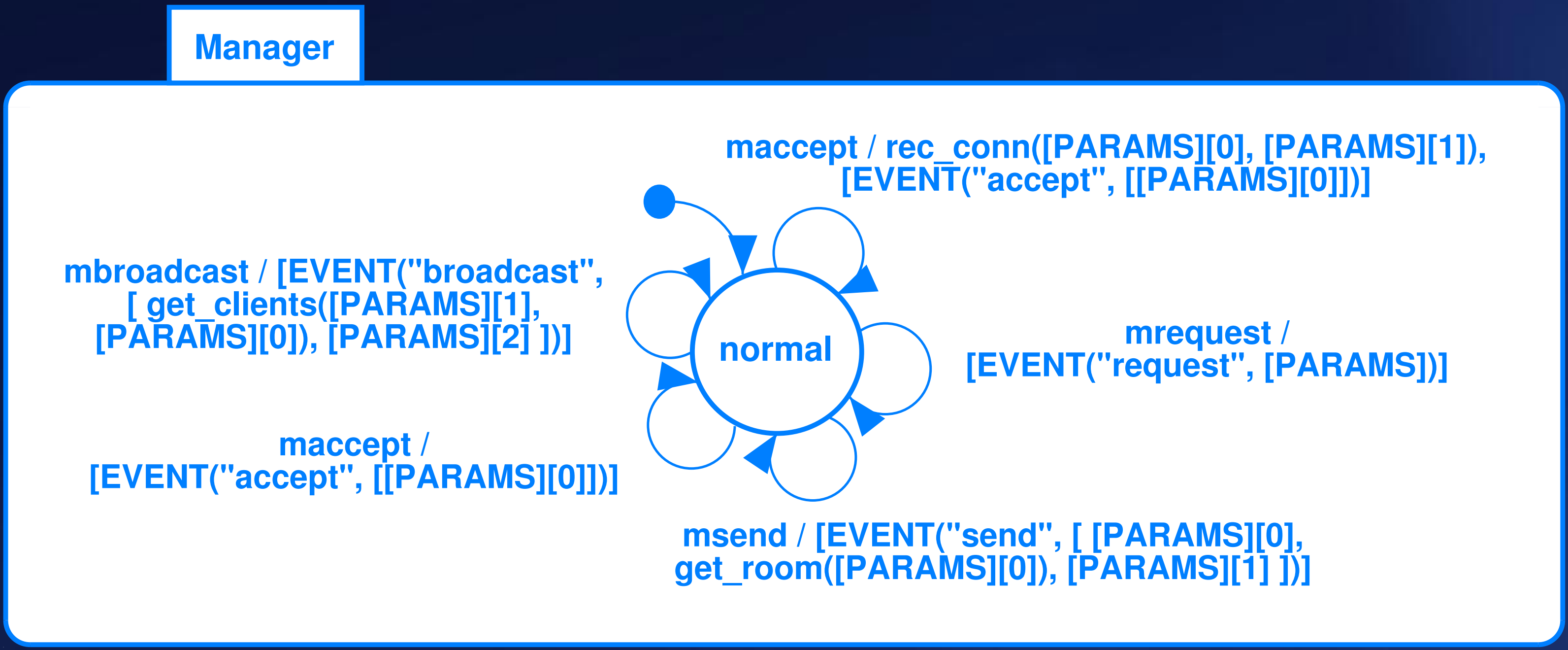
It uses a list messages[ID] to queue incoming messages. This means every chat room with a unique ID has its own queue.


ChatRoom
request [[PARAMS][1]==[ID] and clientNum<3] / [EVENT("maccept", [[PARAMS][0], [ID]])], clientNum+=1
request [[PARAMS][1]==[ID] and clientNum>=3] / [EVENT("mreject", [[PARAMS][0], [ID]])]
root
H*
send [[PARAMS][1]==[ID]] / messages[ID].append ([PARAMS]+[[TIME]])
after(messages[ID][0][3]+1-[TIME]) / [EVENT("check")]
idle
sending
check [len(messages[ID]>1] / [EVENT("mbroadcast", messages[ID][0][:3])], delete messages[ID][0]
check [len(messages[ID]==1] / [EVENT("mbroadcast", messages[ID][0][:3])], delete messages[ID][0]
send [[PARAMS][1]==[ID]] / messages[ID].append ([PARAMS]+[[TIME]])

MSDL

Manager Component

The Manager component simply relays messages.


Manager
maccept / rec_conn([PARAMS][0], [PARAMS][1]), [EVENT("accept", [[PARAMS][0]])]
mbroadcast / [EVENT("broadcast", [get_clients([PARAMS][1], [PARAMS][0]), [PARAMS][2]])]
normal
mrequest / [EVENT("request", [PARAMS])]
maccept / [EVENT("accept", [[PARAMS][0]])]
msend / [EVENT("send", [[PARAMS][0], get_room([PARAMS][0]), [PARAMS][1]])]

Function `rec_comm(client, room)` records a connection in a list when a chat room accepts a client. `get_clients(room, client)` looks up the list and returns all the clients in chat room `room`, except `client`. `get_room(client)` returns the room ID for `client`.

Verification 4: Statecharts $\rightarrow$ Class Diagram

If event senders are inconsistent with receivers . . .

A program can be written which automatically checks sender-receiver consistency of all the method calls. For example:

`Manager` *accepts event* `maccept`*. This means it provides method* `maccept` *in its class definition.*

```
maccept /
    rec_conn([PARAMS][0],[PARAMS][1]),[EVENT("accept",[[PARAMS][0]])]
```

In the guard and output of the transition that handles this event, `[PARAMS][0]` *and* `[PARAMS][1]` *are used, so it requires* at least *two parameters.*

In the whole `Chat` *model, this method is only called (asynchronously) by the* `ChatRoom` *component. The call provides exactly two parameters.*

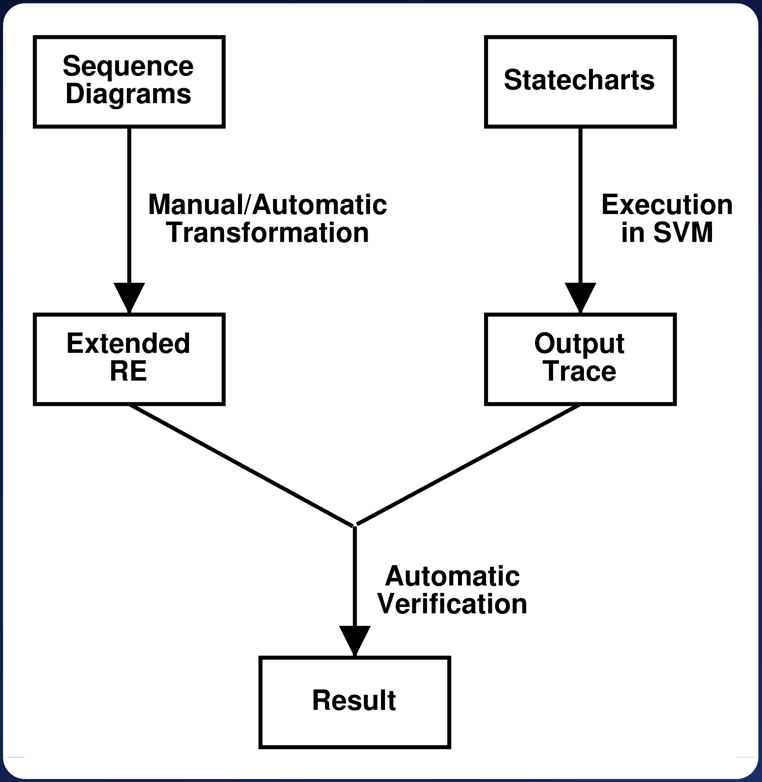
```
request [ [PARAMS][1]==[ID] and clientNum<3 ] /
    [EVENT("maccept",[[PARAMS][0],[ID]])],clientNum+=1
```

Model Execution

The statechart model is executed by SVM. Output is dumped to screen or a file as a list of messages. Each message contains: the time written as a tuple (t, s), the sender or receiver with its unique ID, and the message body.

```
. . . . . .
CLOCK: (10.5s,0)
Client 0
Says "Hello!" to ChatRoom 1
. . . . . .
CLOCK: (11.5s,0)
ChatRoom 1
Broadcasts "Hello!" to all clients except Client 0
. . . . . .
CLOCK: (11.5s,2)
Client 1
Receives "Hello!" from Client 0
. . . . . .
```

Verification 5: Output Trace $\longrightarrow$ Sequence Diagrams



Verification 5: Output Trace → Sequence Diagrams (Continued)

Extended RE (Regular Expression)

A rule contains 4 parts:

- Pre-condition, a regular expression used to match a part of the output trace.
- Post-condition, another RE to be found in the output.
- Guard (optional), a boolean expression defining the applicable condition.
- Counter-rule property (optional).

Example: ***"the sender of a message does NOT receive the broadcast after 1 second"***

pre-condition	`CLOCK: \((\d+\.{0,1}\d*)s,(\d+\.{0,1}\d*)\)\n\Client (\d+)\nSays "(.*?)" to ChatRoom (\d+)\n`
post-condition	`CLOCK: \([(\1+1)]s,(\d+\.{0,1}\d*)\)\nClient [(\3)]\n Receives "[(\4)]" from Client [(\3)]\n`
guard	`[(\1+1)]<50`
counter-rule	`true`

Verification 6: Output Trace $\longrightarrow$ Protocol

It is difficult, if not impossible, to prove the model is completely consistent with the protocol.

Rule-based approach does not work, because it is hard to transform the protocol (described in natural language) into formal representation.

A series of steps are used to achieve the final, executable design. Information is lost while converting a design into another in a different formalism. Checking between intermediate steps does not guarantee the correctness of the final product.

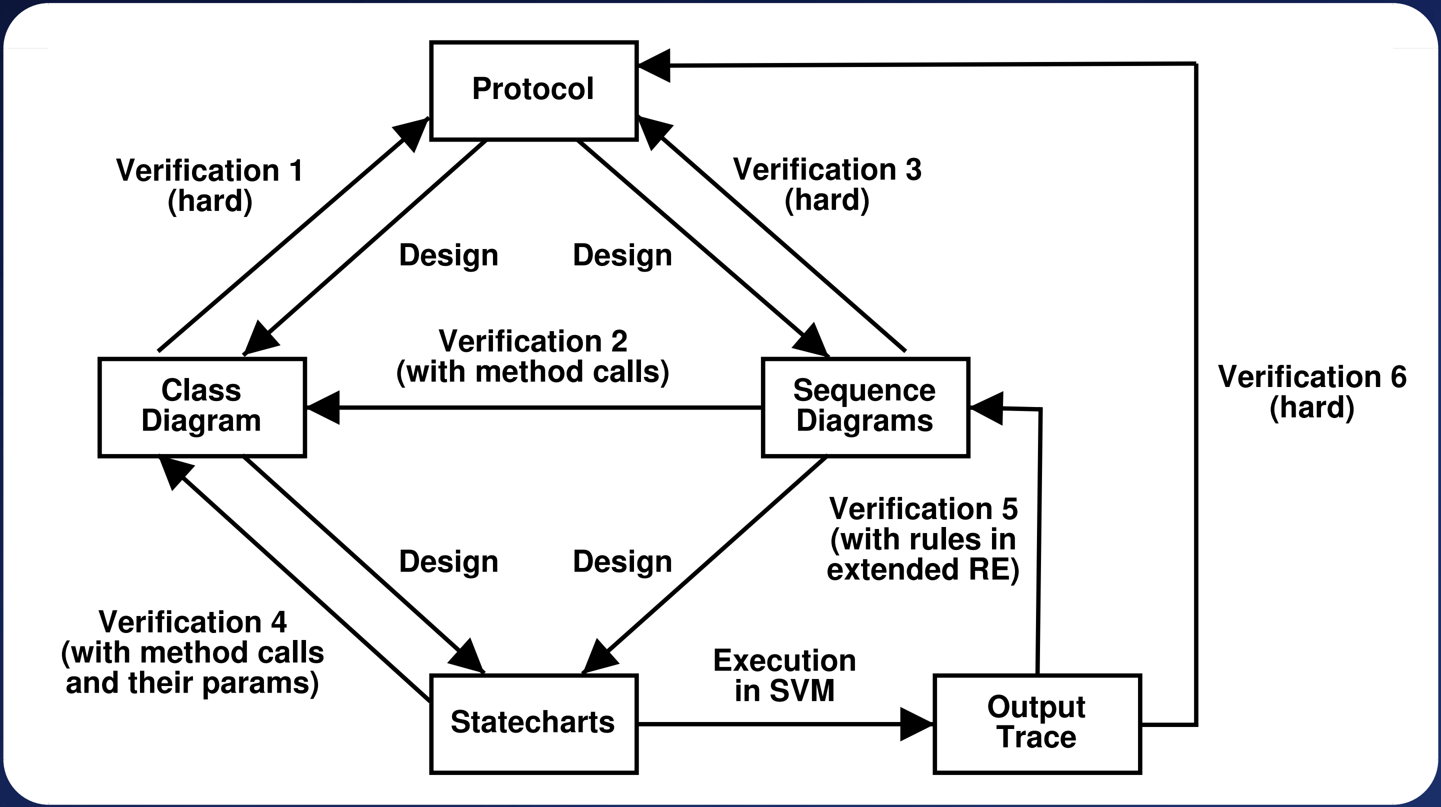
“What can we do?”

Conclusion

Part I

SVM is a flexible and modular tool for the extended statechart formalism.

Part II


Protocol
Verification 1 (hard)
Design
Design
Verification 3 (hard)
Verification 2 (with method calls)
Class Diagram
Sequence Diagrams
Verification 6 (hard)
Verification 5 (with rules in extended RE)
Design
Design
Verification 4 (with method calls and their params)
Execution in SVM
Statecharts
Output Trace