A Case Study: Consistency Problems in a UML Model of a Chat Room

Thomas Huining Feng and Hans Vangheluwe

MSDL, McGill

http://msdl.cs.mcgill.ca/

Introduction

Goal

- Demonstrate modelling (and simulation) based design using the UML.
- Discuss consistency problems.

Model/Component-based Design

- *Modularity*.
- *Reusability*.

Consistency

- **Intra-consistency**. Among artifacts within a given model.
- *Inter-consistency*. Between different models evolved during the software development process.

Outline

Part I – An Introduction to SVM

- Statechart basics.
- SVM extensions to statecharts.

Part II – Chat Room Model Design

- Requirements (Use Cases/Protocol).
- Class design.
- Sequence diagrams.
- Statecharts.
- Model execution in SVM.

Conclusions

Part I

An Introduction to SVM

SVM Design

Goals

- Extend the Statechart formalism.
- Implement a virtual machine for simulation and RT execution.

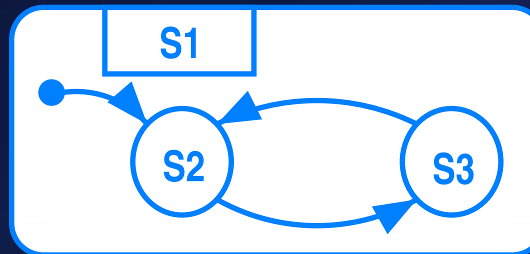
General considerations

- **Interpretation vs. Compilation**.
- **Virtual-time Simulation and Real-time Execution**.
- **Textual vs. Visual**.
- **Portability**. Python.

Statecharts = State Machines + Hierarchy + Orthogonality + Broadcast

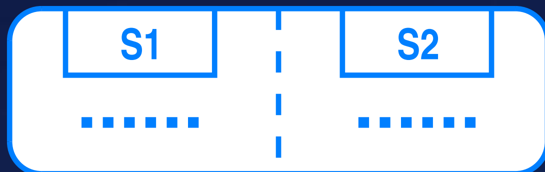
Statechart Elements

finite state automata



hierarchy

history



orthogonal components

transition

enter/exit actions

A Simple Statechart Model

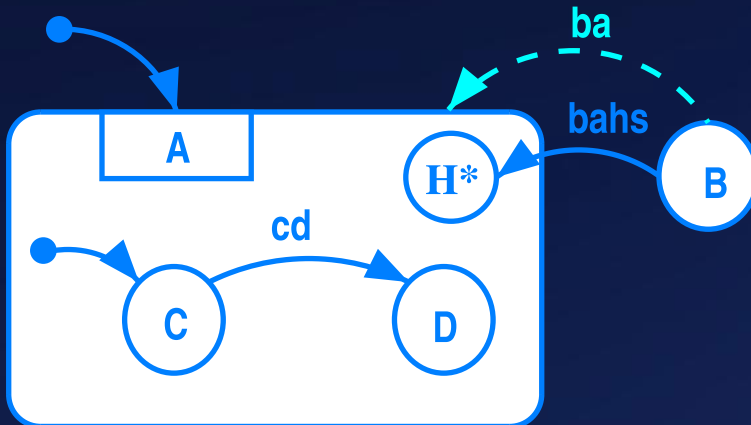
```
STATECHART:
  S1 [DS]
  S2
  S3 [FS]

TRANSITION:
  S: S1
  N: S2
  E: e1
  O: [DUMP("e1 triggers transition")]

TRANSITION:
  S: S2
  N: S1
  E: e2
  O: [DUMP("e2 triggers transition")]

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: Importing Models (Dynamically)

Motivation

Divide a large model into smaller parts and assemble them when needed.

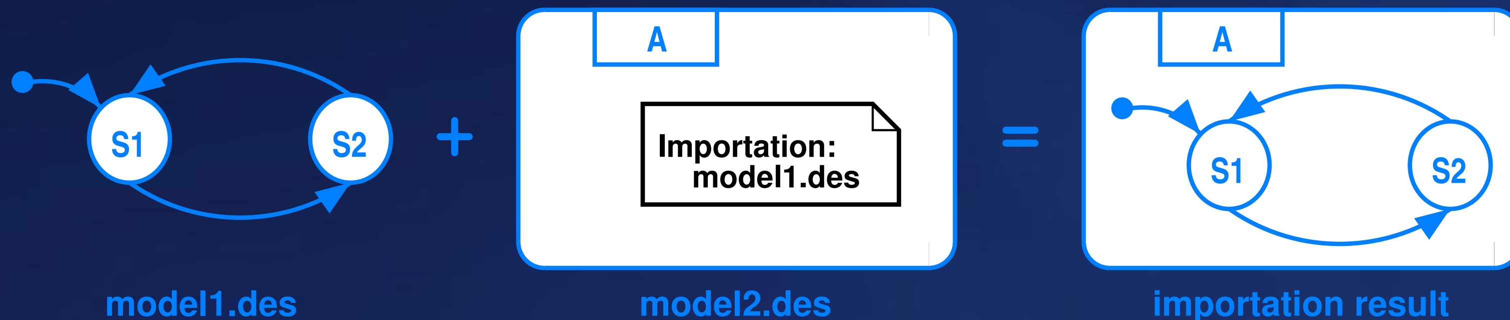
Note: Statecharts were non-modular (but are in UML 2.0).

SVM: importing models

An imported model is a Statechart model in its own right.

Place all states and transitions in a state of the importing model.

Only import model when needed (e.g., in entry action) !



Extension 2: Transition Priorities

Motivation

Two or more transitions enabled by the same event → **conflict**.

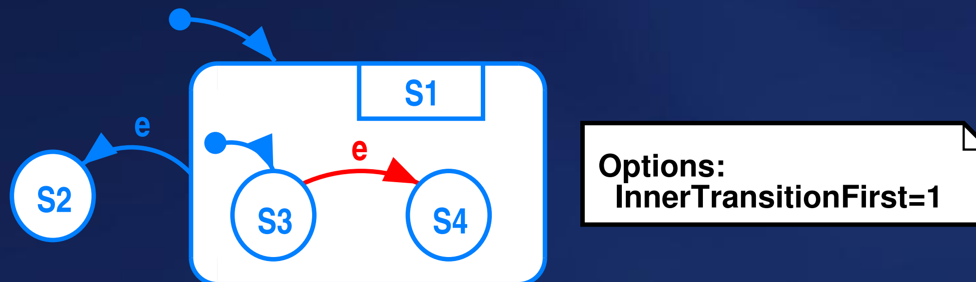
UML semantics: source state of transition is substate of source state of the other → gets higher priority (inner-first).

STATEMATE semantics: it gets lower priority (outer-first).

Desirable to support both of these schemes (and more).

SVM Extension A.

Every model has a global option: `InnerTransitionFirst`.



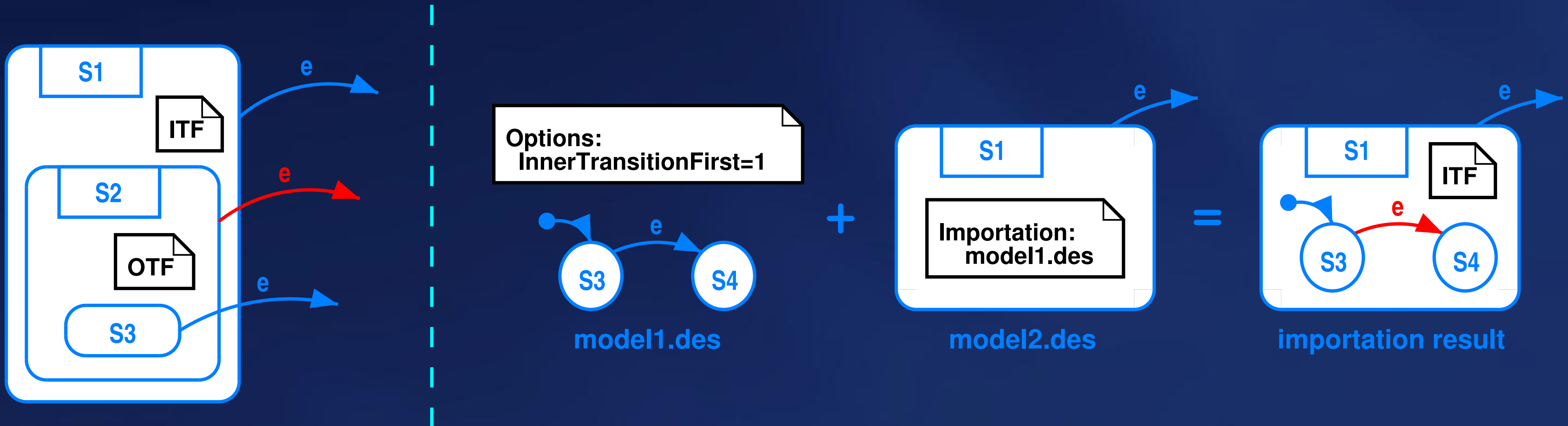
If the current state is S1.S3 and event e occurs, the new state will be S1.S4.

SVM Extension B.

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

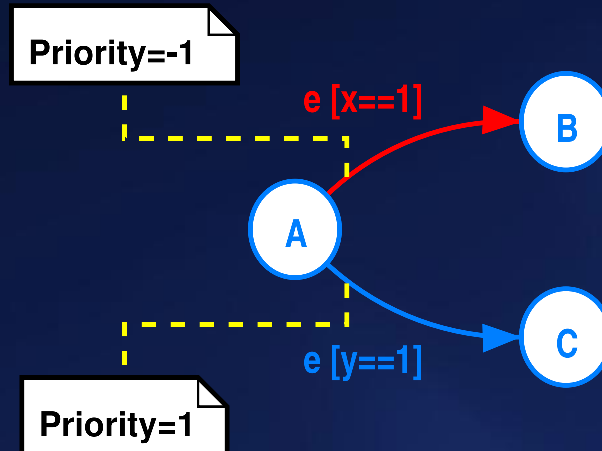
- **ITF**. Inner transition first.
- **OTF**. Outer transition first.
- **RTO**. Reverse transition order. (If parent state is ITF, it is OTF and vice versa.)

The property of a state overrides 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

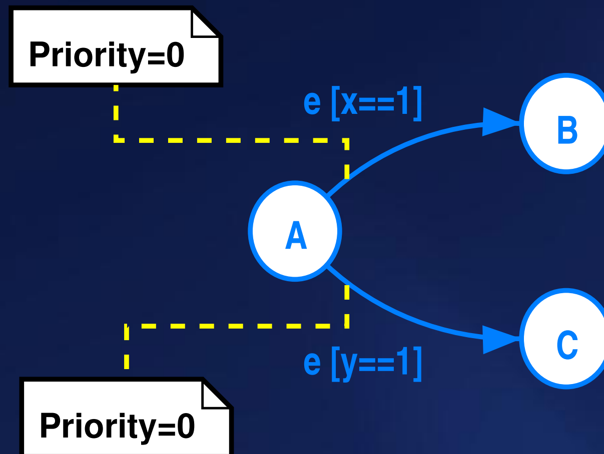
$$\begin{cases} x = 1 \\ y = 1 \end{cases}$$

the state will change to B.

SVM Extension D.

If unresolved (by extensions A, B and C) conflicts still remain at run-time, the choice is random according to a uniform distribution.

Note: This usually indicates a design flaw. The designer did not foresee a potential conflict in the model.



Extension 3: Parametrized Model Templates

Motivation

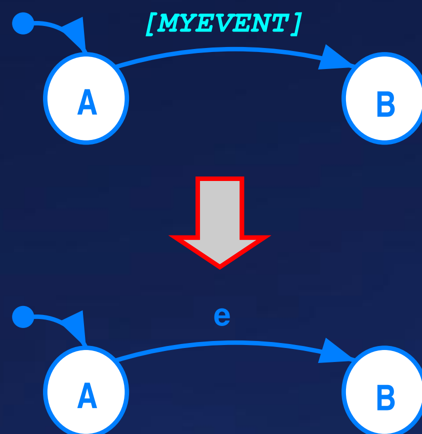
Re-use of a design usually requires change or customization.

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

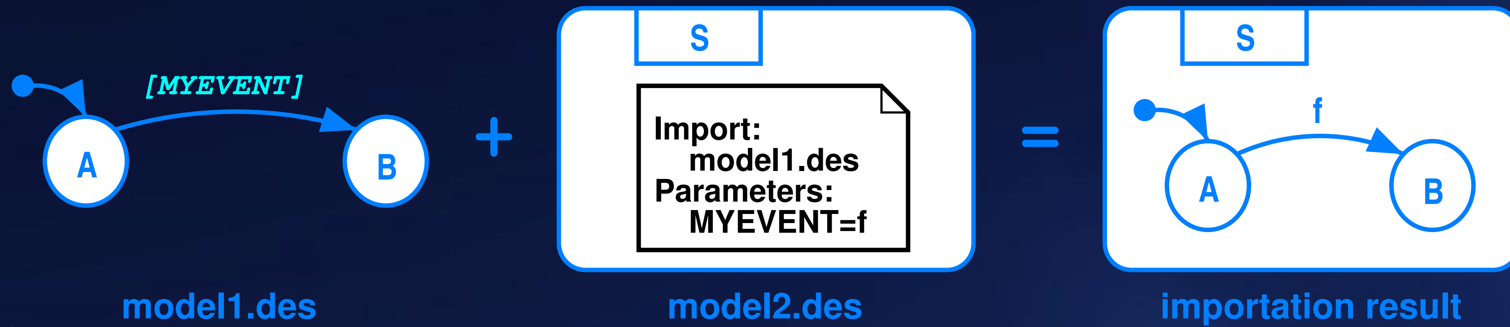
Macros can be defined and used anywhere in its description.




```
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.



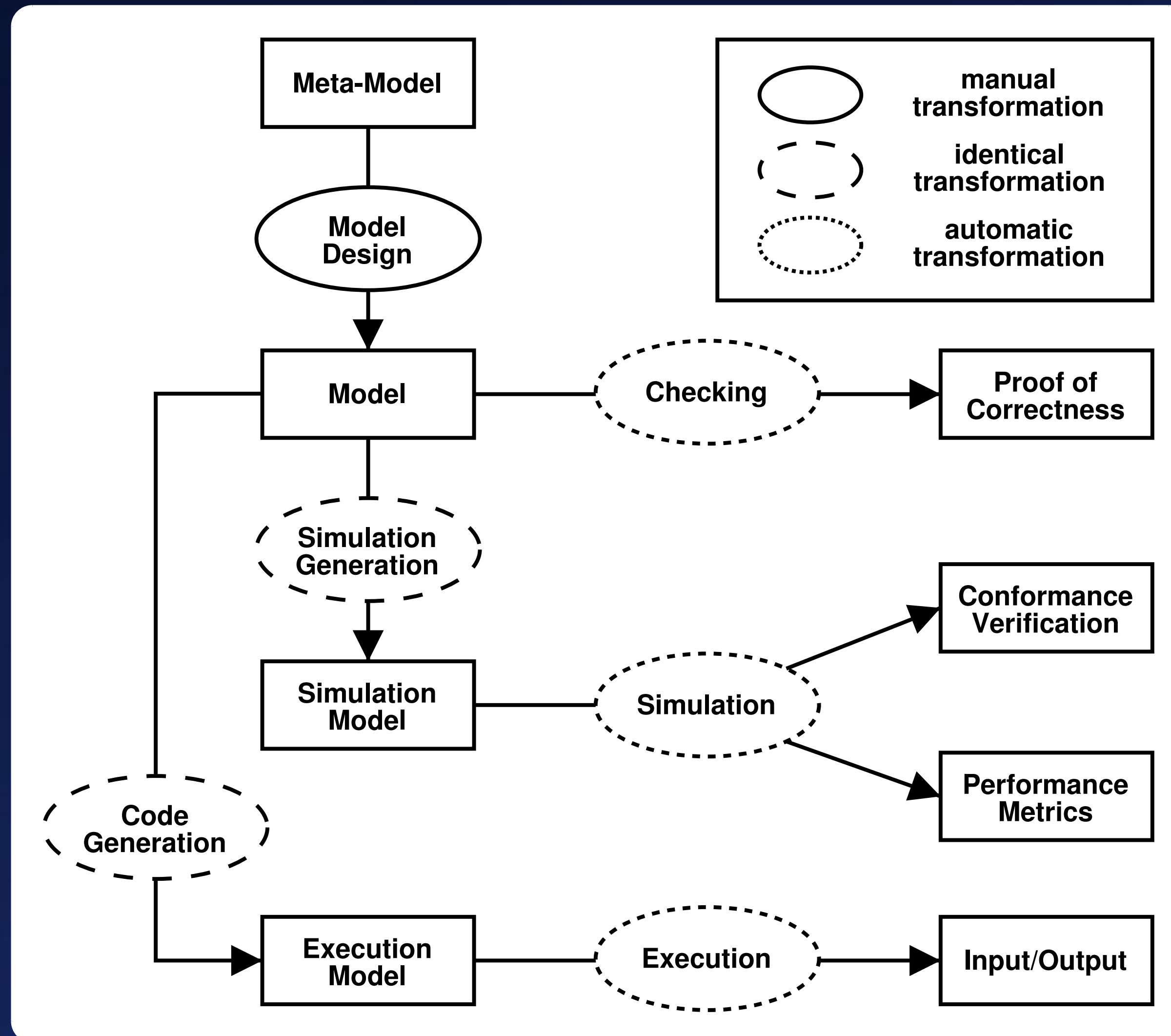
The outside world is able to modify the behavior of a model *only* through parameters.

There is *no other way* to modify a model.

Part II

Chat Room Model Design

The Model-based Development Process



Use Case: The Communication Protocol

1. 5 clients and 2 chat rooms. Initially, clients are 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.

Design: Classes

- `ChatRoom`. 2 instances.

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

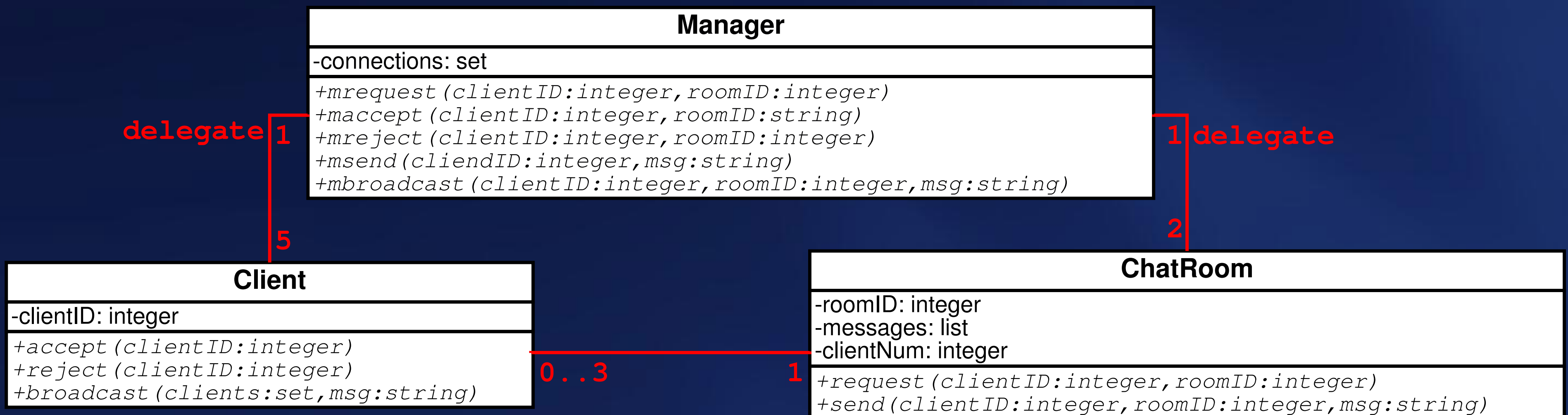
- `Client`. 5 instances.

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

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

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

Design: Class Diagram



Consistency Check 1: Class Diagram $\rightarrow$ Use Case

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

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

Design: Sequence Diagrams

Sequence diagrams specify *constraints* on communication between class instances.

Indirectly, this imposes constraints on allowed *method* implementations.

Timing

According to the use case description, more than one action may happen at the same time.

There may be a *causal relationship* however.

`request at time 1; accept at time 1` √

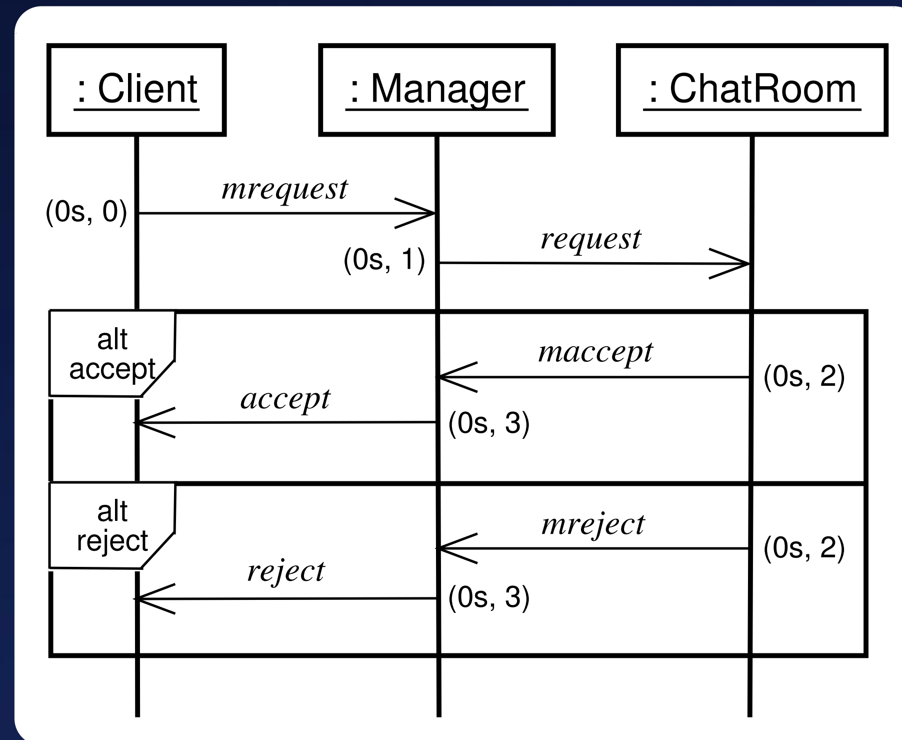
`accept at time 1; request at time 1` ×

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

`request at time (1.0s, 0); accept at time (1.0s, 1)` √

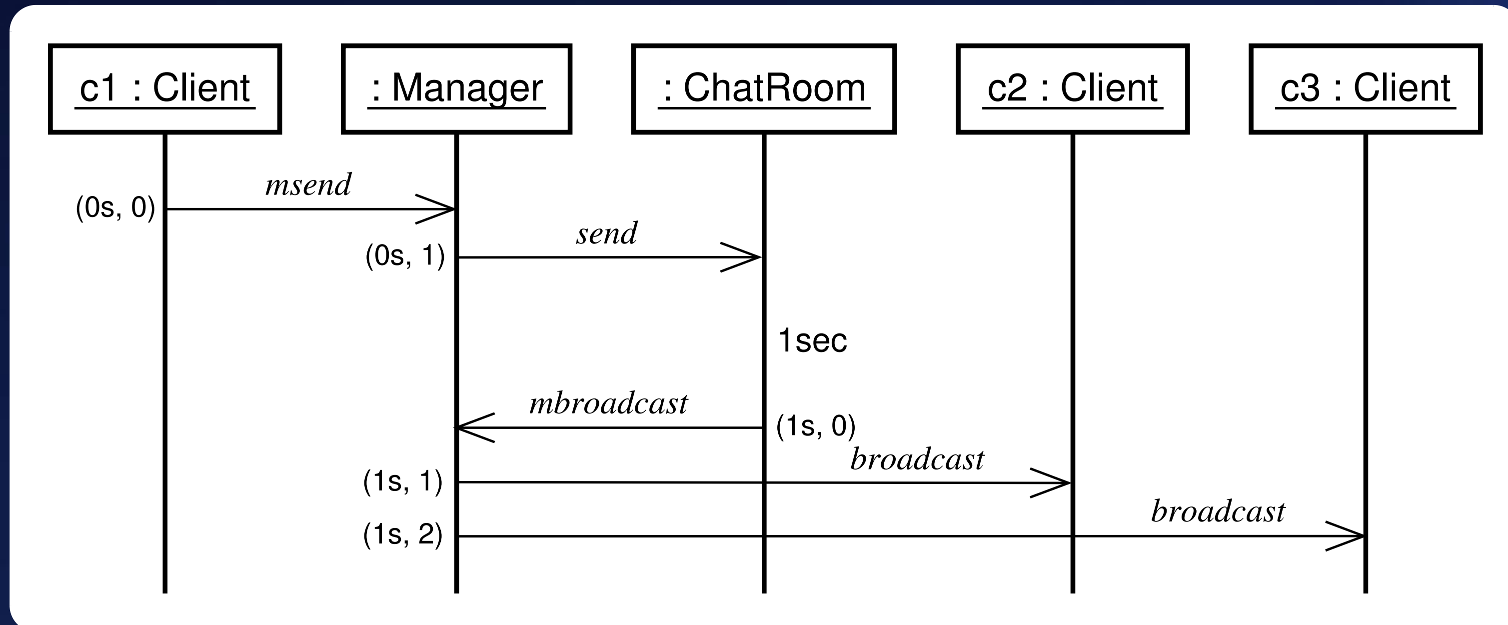
`accept at time (1.0s, 0); request at time (1.0s, 1)` ×

Sequence Diagrams (Continued)



Request pattern

Sequence Diagrams (Continued)



Message pattern

Consistency Check 2: Sequence Diagrams → Class Diagram

Components must be instances of existing classes.

Collect all method calls (or incoming events) of a component and check if they have corresponding definitions in the component's 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 possible uses of* `Manager`*. So, its class design must have (and only have) definitions for the corresponding 5 public methods.*

This consistency check can be automated.

Consistency Check 3: Sequence Diagrams $\longrightarrow$ Use Case

Consistency with the use case can only be partially 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 requirements.*

A rule-based approach will be introduced later (convert the use case/protocol into extended REs, then use the REs to check the sequence diagrams.)

Consistency Check 3: Sequence Diagrams $\rightarrow$ Use Case (Continued)

However, checking is limited due to expressiveness of sequence diagrams.

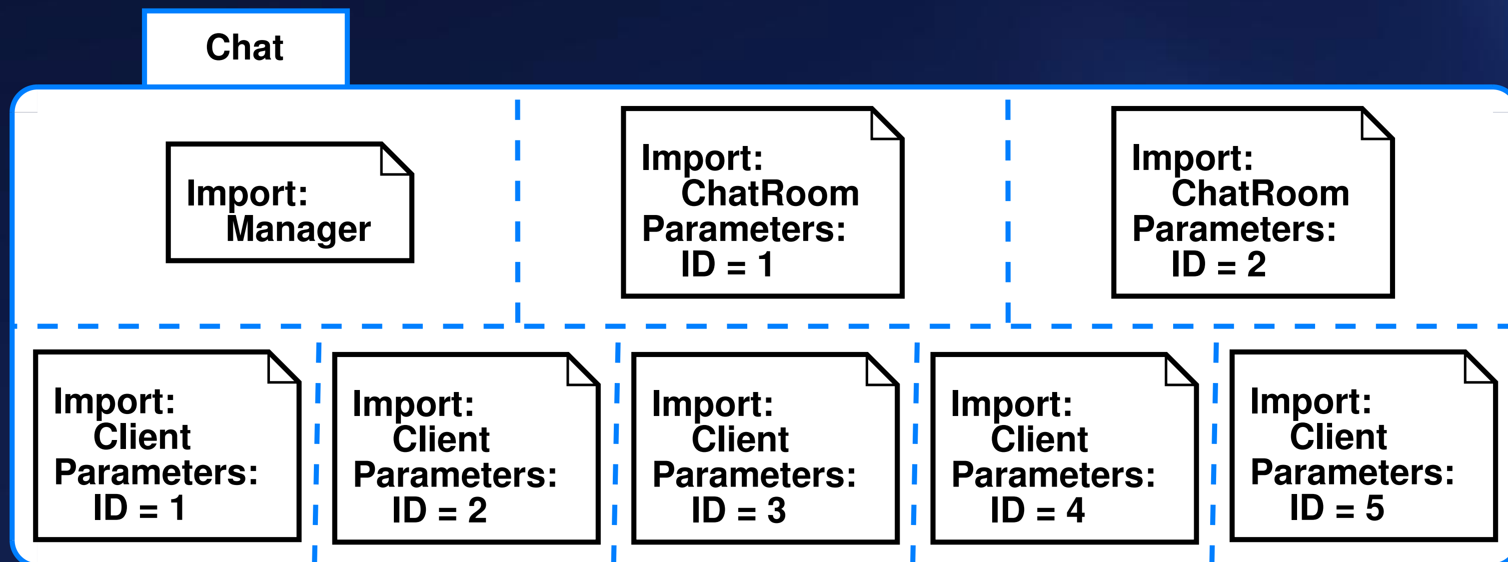
For example:

- 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, a "dead" client which does not send any request cannot be detected as a problem.

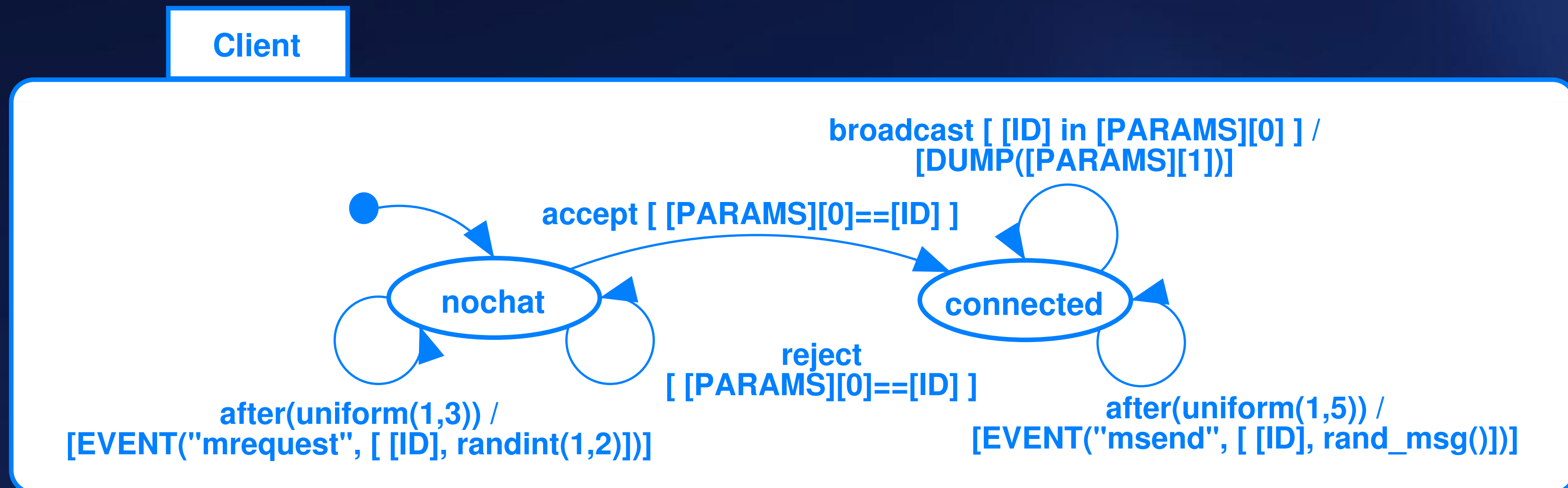
Design: Statecharts

The detailed behaviour of components `Client`, `ChatRoom` and `Manager` is described in separate statecharts. Model `Chat` (the complete system) imports five instances of `Client`, two instances of `ChatRoom` and one `Manager`.



Client Component

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

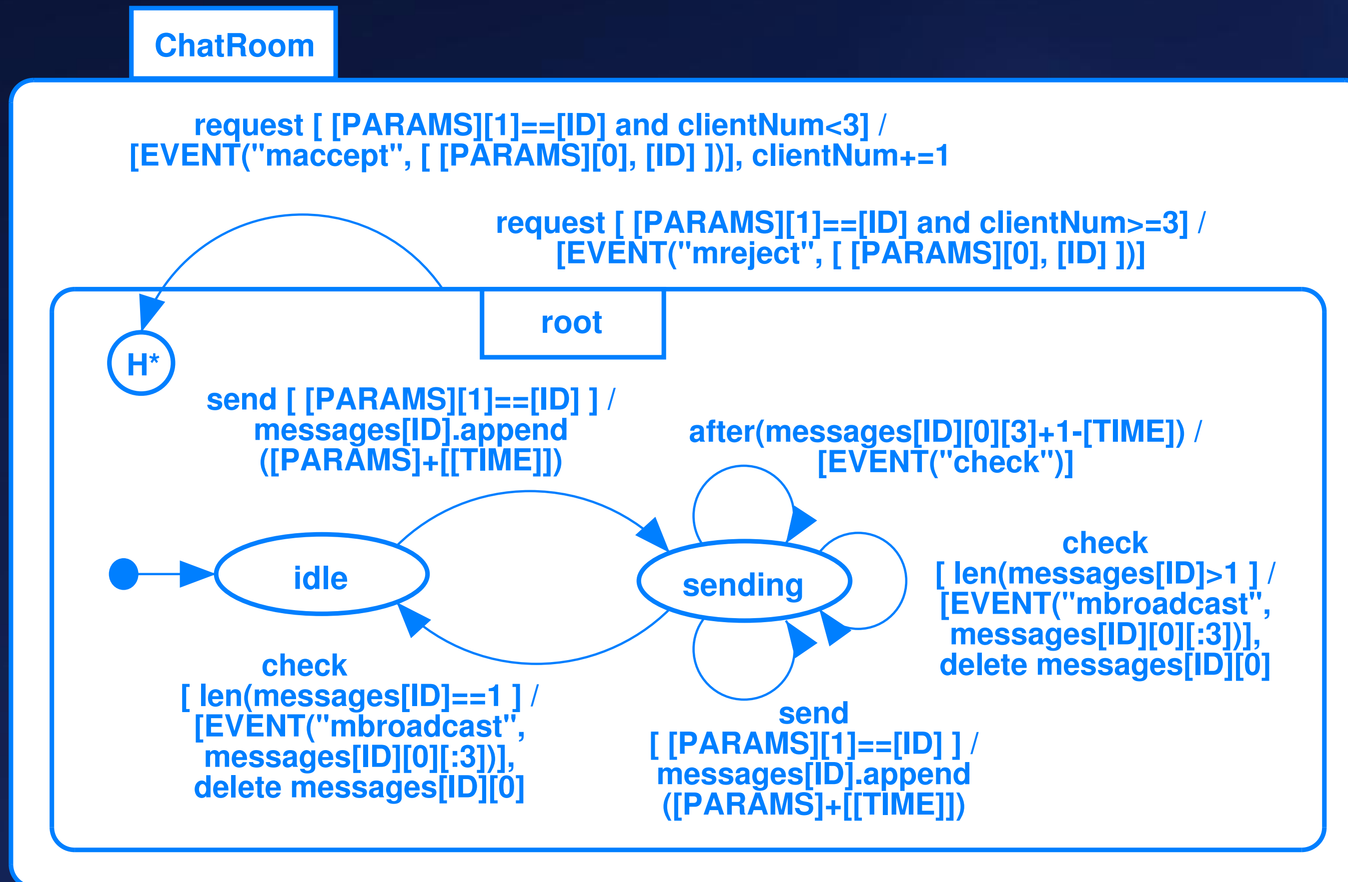


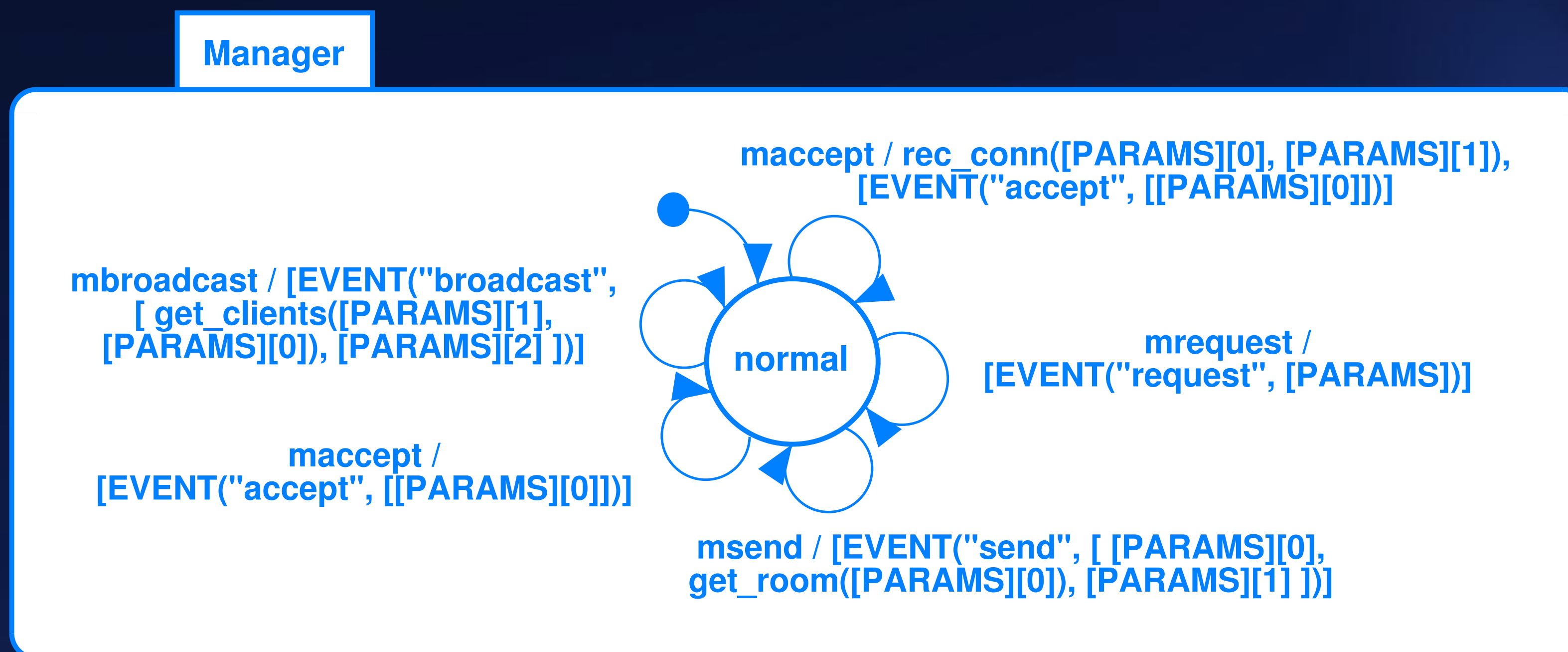
- `uniform` is a Python function which returns a random real number in a range.
- `randint` returns a random integer.
- `[EVENT(...)]`, `[PARAMS]` and `[DUMP(...)]` are pre-defined. `[ID]` is user-defined.
- `after` event is raised at a certain time after a state is entered.
- `accept`, `reject` and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

ChatRoom Component

Uses a list `messages[ID]` to queue incoming messages.

Every chat room has its own queue.





`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`.

Consistency Check 4: Statecharts → Class Diagram

Sender-receiver consistency of all the method calls can be checked automatically.

For example:

`Manager` *accepts event* `maccept`. *This means it must provide 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* `maccept` *requires* at least *two parameters.*

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

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

Note the relationship between method/event/message

Model Execution

The statechart model is executed using SVM. The trace is saved as a list of messages. Each message contains: the time as a tuple (t, s), the sender or receiver with their 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
......
```

Model Execution in SVM

State Hierarchy

Statechart Virtual Machine - Thomas (Nov. 2003)

ChatSequentialTime.des

```
  - connected
  - nochat
+ client4
  - connected
  - nochat
+ client5
  - connected
  - nochat
+ clock
  - NORMAL
+ manager
  - normal
+ room1
  + root
    - idle
    - sending
+ room2
  + root
    - idle
    - sending
```

Enabled Events

```
accept
broadcast
check
gettime
maccept
mbroadcast
mreject
mrequest
msend
notify
reject
request
schedule
send
```

Output

```
CLOCK: (4.445668s,0)
Client 5
Sends "Hello, everyone!"

CLOCK: (4.445668s,1)
ChatRoom 1
Receives "Hello, everyone!" from Client5

CLOCK: (4.489383s,0)
ChatRoom 1
Broadcasts "I feel very well today!" to clients except Cl

[EVENT] > debug
CLOCK: (4.489383s,1)
Client 1
```

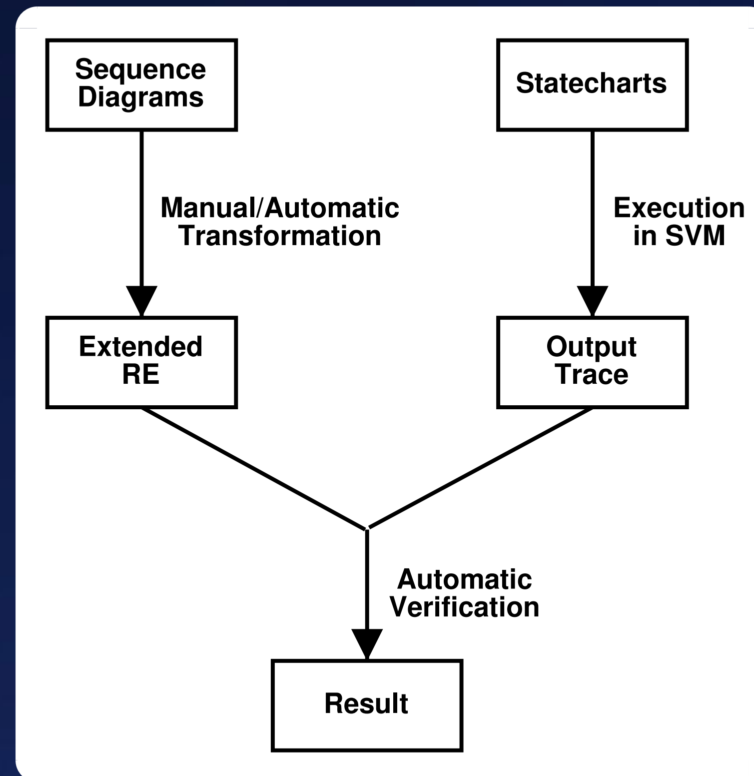
Command

```
if eventhandler.is_in_state("client4.connected"):
  print "Client 4 is connected"
```

Show State Hierarchy | Snapshot | About SVM | Exit

```
['clock.NORMAL', 'client2.nochat', 'manager.normal', 'client4.connected', 'client5.co
```

Consistency Check 5: Output Trace → Sequence Diagrams



Consistency Check 5: Output Trace $\longrightarrow$ Sequence Diagrams

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).

Consistency Check 5: Output Trace → Sequence Diagrams

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

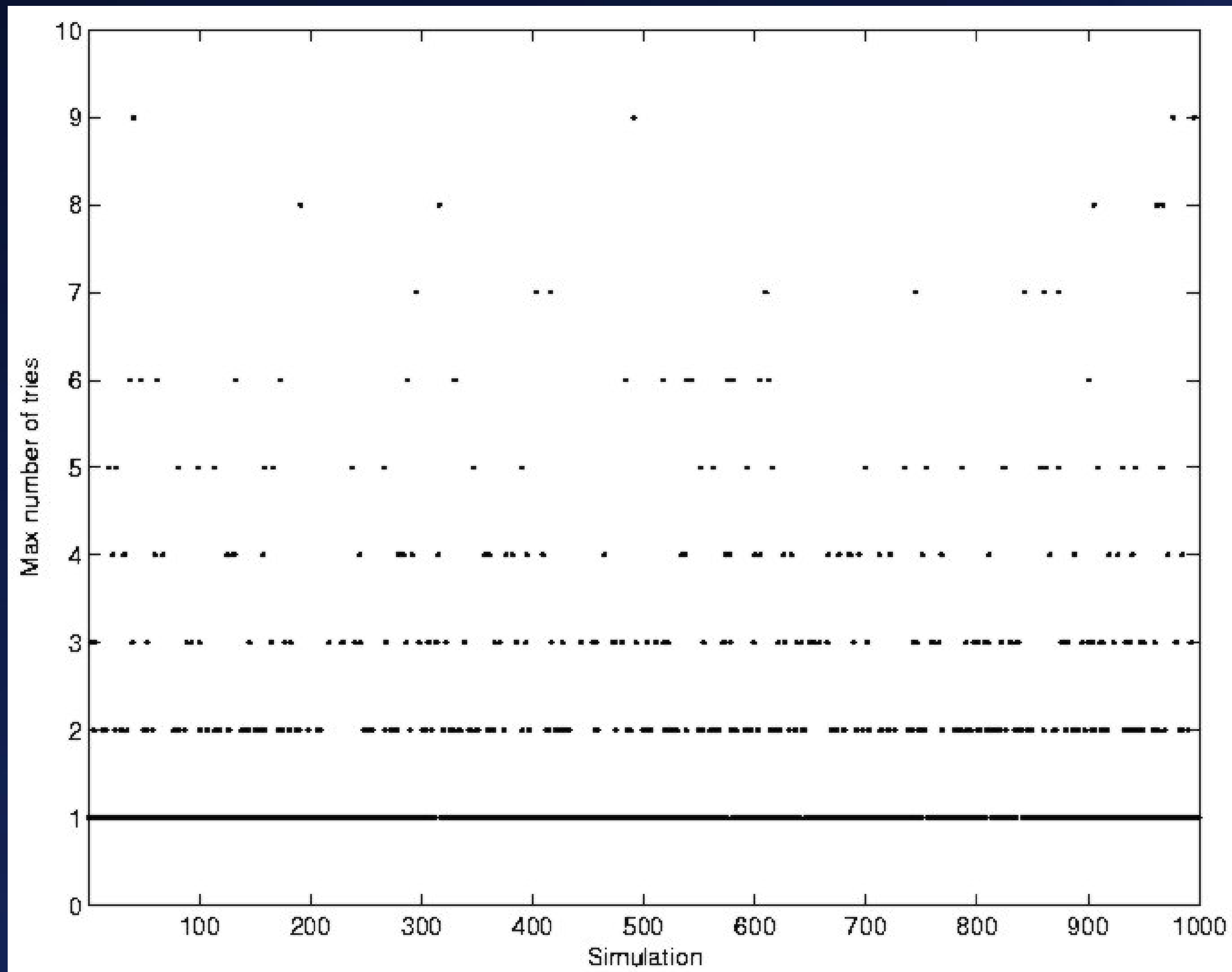
pre-condition	`CLOCK: \((\d+\.{0,1}\d*)s,(\d+\.{0,1}\d*)\)\n\Client (\d+)\nSays "(.*?)" to ChatRoom (\d+)\n`
post-condition	`CLOCK: \([(\1+1)]s,(\d+\.{0,1}\d*)\)\nClient [(\3)]\n Receives "[(\4)]" from Client [(\3)]\n`
guard	`[(\1+1)]<50`
counter-rule	`true`

Consistency Check 6: Output Trace $\longrightarrow$ Use Case

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

A rule-based approach does not work, as it is hard to transform the use case/protocol (described in natural language) into a formal representation.

Performance Analysis

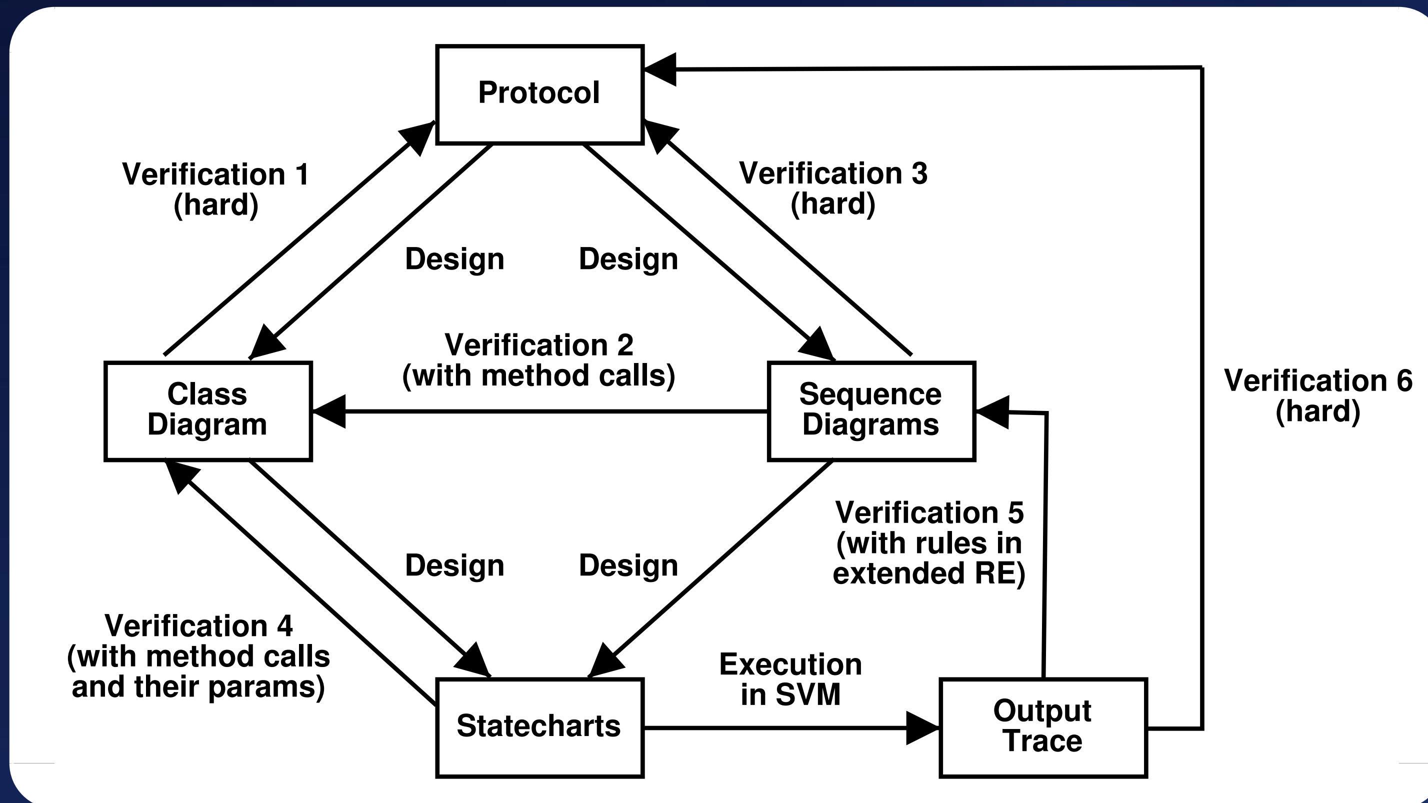


Conclusion

Part I

SVM implements an extended statechart formalism simulator/executor.

Part II



Future Work

- Use most appropriate formalisms/techniques (e.g., Petri Nets)
- Meta-model the formalisms → syntax check, visual environment
- Use graph grammars to model consistent transformations

`http://msdl.cs.mcgill.ca/`

Meta-Modelling in AToM^3