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

Thomas Huining Feng and Hans Vangheluwe
Modelling, Simulation and Design Lab
McGill University
http://msdl.cs.mcgill.ca/

Abstract

This article describes a case study, where a model of a chat room application is built from initial requirements. UML class diagrams, sequence diagrams and statecharts are used in different stages of the development process. Consistency problems are identified and methods, most notably simulation, are proposed, to ensure consistency between some aspects of the models. We focus on intra-consistency, the consistency among artifacts within a given model.

1 Introduction

The development process of a software system is usually divided into steps, in which different UML diagrams are involved. As modelled systems becomes more and more complex, consistency problems become more prominent. Two types or problems are apparent. The first, intra-consistency problems, are concerned with consistency among artifacts within a given model. The second, inter-consistency problems, are concerned with consistency between different models evolved during the course of the software development process. In the sequel, we focus mostly on intra-consistency.

Various formal methods have been explored in the literature to automatically check consistency and discover design problems. In the following sections, steps of the development process of a chat room model are studied. The case is inspired by a project report on "Executable UML" by Geir Melby at Agder University College (`http://fag.grm.hia.no/ikt2340/year2002/`). Potential consistency problems in the model are shown. For some of them, automatic consistency checking methods are proposed.

In this case study, a model of a chat room application is developed from requirements. A protocol specifying communication between clients, a manager object, and chat rooms is given in section 2, and treated as initial requirements. Section 3 studies a possible class design. It defines interfaces conforming to the protocol. The sequence diagrams in section 4 further refine and illustrate the inter-ccomponent communication, consistent with the class design. Section 5 uses statecharts to further specify the application's behaviour. This specification can be either simulated or executed in real-time in our SVM (Statechart Virtual Machine) environment. In section 6, consistency of simulation traces with the protocol specification is discussed. Section 7 concludes this case study.

2 The Communication Protocol

The chat room application to be built features a client-server configuration. Clients try to connect to random chat rooms. After a client is accepted by a chat room, it sends messages to its chat room. The chat room broadcasts each message so that every client connected to it, except the sender, receives a copy.

A specific, simplified use case is described below:

- There are 5 clients and 2 chat rooms in the system. Initially, the clients are not connected. They try to connect to a random chat room every 1 to 3 seconds (uniformly distributed). The requested chat room instantaneously receives the request (no network delay, and reliable communication are assumed).
- A chat room accepts at most 3 clients. It accepts a connection request if and only if its capacity is not exceeded.
- The requesting client receives an acceptance or rejection notice immediately.
- A client must be accepted by a chat room before it may send chat messages.
- When connected, a client sends random messages to the chat room it is connected to every 1 to 5 seconds (uniformly distributed). The chat room immediately receives the messages. It takes 1 second to process a

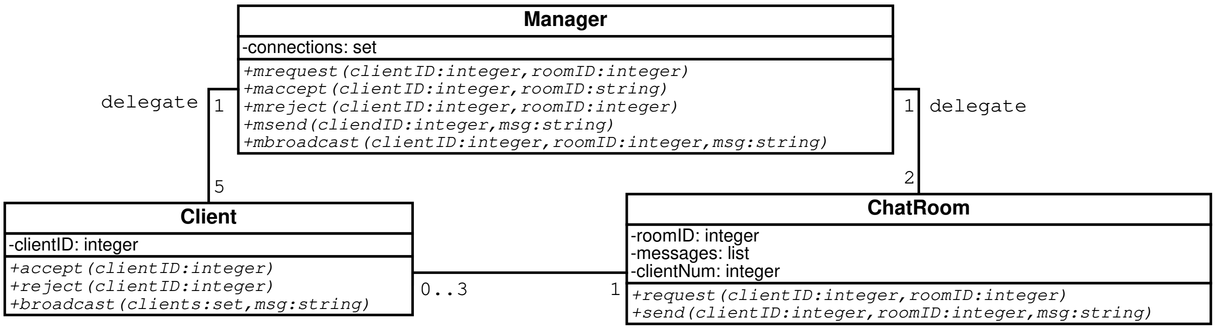

Figure 1: Class design

message and broadcast it to all the clients connected to it, except the sender.

- The clients instantaneously receive the broadcast.

For simplicity, disconnection is not discussed.

3 The Class Design

According to the above specification, two classes are obviously required: `Client` and `ChatRoom`. At this early stage in the development process, so that no user intervention is needed for a simulation, we explicitly model client behaviour (connection requests and chat messages) as a random process. Later, this model of the client will be replaced by real human clients interacting with the software. When the simulation is started, 5 instances of `Client` and 2 instances of `ChatRoom` are initialized.

A singleton class `Manager` is added. The `Manager` acts as a mediator and relays all the communication between components. This central control facility helps to intercept all the messages passed in the system, with which correctness of the model can be checked.

Figure 1 shows the UML class diagram featuring the three classes.

- A `ChatRoom` provides two methods to handle incoming events. `request` handles incoming requests, each of which has parameters `clientID` and `roomID`. The `ChatRoom` sends back an acceptance or rejection notice to the sender with a global ID `clientID`. It also uses the `roomID` parameter to decide whether the request is sent to itself or to another chat room[1]. The `send` method receives a `msg` sent by client `clientID`. This `msg` will be broadcast 1 second later.

[1] As UML statecharts (pre UML 2.0) are used in later steps, which do not allow one to specify the receiver of an event, all the chat rooms receive the same request even if only one of them will handle it.

- Methods `accept` and `reject` of `Client` handle incoming acceptance and rejection notices. Parameter `clientID` is used to identify the target client. When a `Client` receives a `broadcast` event, it checks if itself is in the set of `clients`. If so, message `msg` is printed to the output.

- The `Manager` relays connection requests, acceptance and rejection notices, messages sent from clients and broadcasts from chat rooms. For example, when it receives broadcasts from chat rooms, the three parameters tell it the original sender (client) of the message, the broadcaster (chat room) and the message string. It then sends the message to all the clients connected to this chat room, with the exception of the original sender.

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, which can only be interpreted by human understanding.

- The protocol is specified in natural language, which a program cannot easily process.

- For a well-defined system there can be a number of interface designs. They may differ substantially. For example, in this design a `Manager` class is used to intercept communication. Another design may use `RequestManager` to intercept requests, acceptances and rejections, and use `MessageManager` to intercept messages and broadcasts. Yet another design may not use any manager at all.

4 Sequence Diagrams

The sequence diagrams discussed in this section bring the design to a lower level of abstraction (higher level of

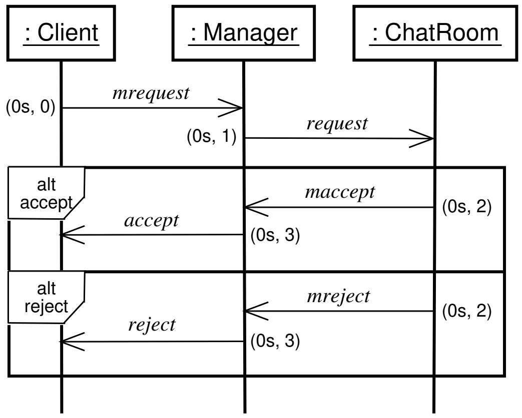

Figure 2: Sequence diagram of the request pattern

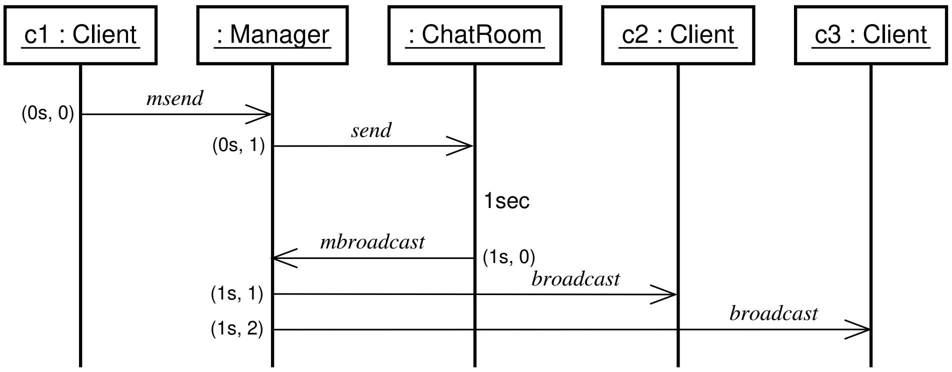

Figure 3: Sequence diagram of the message pattern

detail) than the class diagram. The sequence diagrams must clearly reflect the interaction between components.

4.1 Timing

Timing issues make conversion of the protocol into sequence diagrams and later statecharts difficult. In the protocol description, more than one action can happen at the same time, though they may be causally related. For example, a chat room should not send an acceptance or rejection before it receives a request, though time is not advanced. Hence, "`request at time 1; accept at time 1`" in the output trace is correct, while "`accept at time 1; request at time 1`" is not.

One possible solution is to use a tuple (t, s) to represent time, where t is float-point time in seconds and s is an integer sequence number. In this way, the correct output can be written as "`request at time (1.0s, 0); accept at time (1.0s, 1)`". The sequence "`accept at time (1.0s, 0); request at time (1.0s, 1)`" is incorrect.

4.2 The Request and Message Patterns

The request pattern is shown in Figure 2. A `Client` first invokes the `mrequest` method of the `Manager`. The `Manager` then relays the request by calling `request` of a `ChatRoom`. The `ChatRoom` immediately responds and calls back the `maccept` or `mreject` method of the `Manager`. The requesting `Client` then receives the relayed reply from the `Manager`.

Figure 3 shows the message pattern where a random message is generated and passed in the system. Note that the `ChatRoom` deliberately delays 1 second after it receives a request. No other time delay is shown in the two sequence diagrams.

4.3 Consistency with the Class Diagram and the Protocol

Consistency with the class diagram can be easily checked by collecting all the method calls that a component receives. For example, in the request pattern, the `Manager` receives `mrequest`, `maccept` and `mreject`. In the message pattern it receives `msend` and `mbroadcast`. These five methods are defined in the class diagram, and no other public methods are defined. As parameters are not shown in the sequence diagram, there is no need to check their parameters. This checking process can be automated.

Consistency with the protocol can be partially checked. One can easily see that according to the request sequence diagram, if a chat room 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. In this way the designer can check "what should happen at a certain time" manually. If the rule-based approach discussed later is used, limited automatic checking is also possible.

Note how basic sequence diagrams are unable to express what should *not* happen at a certain time or in a certain period. For example, it is implied in the protocol that a chat room does not accept or reject a client without a request. This information is missing in the sequence diagrams. This may introduce design errors into the model, and affect the correctness of later development steps.

Another possibility for an erroneous design is due to the semantics of sequence diagrams. For example, in the request pattern, the sequence diagram describes: *if* a client sends an `mrequest`, *then* the manager sends a `request` without time advance, *then* the chat room sends an `maccept` or `mreject` without time advance, *then* the manager sends `accept` or `reject` accordingly. Unfortunately, an inert client that does not send any request, which is obviously a problem in the system, can not be detected by checking the sequence diagram. In the worst case, no client tries to connect, and thus the system halts forever.

To compensate for the loss of information, designs in other UML formalisms are needed which do not completely depend on sequence diagrams, or the sequence diagram formalism must be extended. An excellent example of the extension and use of sequence diagrams are Live Sequence Charts, as desiscribed by Harel [1].

5 Statechart Design

Statecharts are used to implement the behavior behind the class definitions. They can be executed in our SVM (Statechart Virtual Machine) [2] [3], an interpreter for an extended statechart formalism written in Python.

5.1 SVM Conventions

Before the statechart designs can be easily understood, some SVM conventions must be introduced beforehand.

SVM interprets models in the extended statechart formalism. New features are added. Though expressiveness is not enhanced[2], ease of use is greatly improved.

Component-based design is possible in SVM, though original statecharts are not modular. This is necessary for the chat model, where components such as clients and chat rooms are designed separately, but work together in the final system. Components are reused by importation. A larger component imports (an instance of) a smaller one into one of its states. The result is as if all the (hierarchical) states and transitions of the imported component were directly written inside that state.

SVM models are written in text files. *Macros* are a concept introduced in SVM. Macros are defined in the `MACRO` section of an SVM source file. Once defined, they can be used in brackets throughout the text file. For example, with `PREFIX=state` defined, `[PREFIX]` can be used to literally substitute string "state", and thus `[PREFIX]1` is equivalent to `state1`.

Some of the predefined macros are used later on. `[EVENT(event, param)]` raises an event. It carries a string `event` and an optional list `param` as the parameters that travels along with it. `[PARAM]` is used to retrieve parameters of the event being handled. It is usually used in the guard or output of a transition. `[DUMP(msg)]` prints debugging messages to the screen or records them in a text file.

Macros also serve as parameters when a component is imported. The importing component may *redefine* some or all of the macros originally defined in the imported component, including predefined macros. As a continuation of the previous example, if the importing component specifies `PREFIX=mystate` as an importation parameter, `[PREFIX]1` within the imported component is interpreted as `mystate1` instead.

It is easy to show that these extensions do not increase the expressive power of statecharts.

5.2 The Chat Room Model in the Extended Statechart Formalism

Components `Client`, `ChatRoom` and `Manager` are designed in separate statecharts. As Figure 4 shows, model `Chat` imports five instances of `Client`, two instances of `ChatRoom` and one `Manager`. Each instance of the same type has a unique ID parameter. Instances of different types can have the same ID since their sets of acceptable events are disjoint. This model can be simulated or executed in the SVM environment.

The `Client` component is shown in Figure 5. Initially, it is in the `nochat` state. It repeatedly tries to connect to the

[2]More extensions can be made to enhance expressiveness, but checking the correctness of a model becomes much more difficult.

chatroom via the manager by raising an `mrequest` event every 1 to 3 seconds (uniformly distributed), until the request is accepted (the `accept` event is received. `uniform` is a Python function which returns a random real number in a range, and `randint` returns a random integer. The event's first parameter gives the client's unique ID. The event's second parameter gives the destination chatroom (randomly chosen from 1 or 2). Then, the client moves to state `connected` and starts sending messages and receiving broadcasts. Since parameters of events are sent as a list, `[PARAMS][0]` gives access to the first parameter, and so on. Note that when the content between square brackets is not a macro name or a Python index, it is a guard as defined in the original statechart formalism [4].

User-defined macro `[ID]` gives a unique ID to each `Client`. Its definition `ID=0` implies that the default value is 0. It is changed by the importing component (the `Chat` model in this case) to a unique number. ID of a component is important. Since the whole system can be viewed as one large statechart after importation, all broadcast events are received by every orthogonal component. Thus, the only way to send an event to a specific client is to give the receiver's ID in the parameter list. Each client checks if its ID matches before handling an event.

Compared to the `Client` component, `ChatRoom` is much more complicated. It uses a list `messages[ID]` to queue incoming messages. This means every chat room with a unique ID has its own queue. (For `ID=0`, `messages[ID]` is equivalent to `messages0`.) If a message comes when it is busy processing a previous message (it takes 1 second), the new one is added to the list. The time when the message is received is also recorded so that even if a message is queued, its processing time is still 1 second since its arrival.

The `Manager` component simply relays messages. 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`.

The message queue of chat rooms and the connection list of manager are examples of variables. They help to record the state of the model. Strictly speaking this is also an extension to original statecharts, where states must be explicitly specified. The discussion of variables is outside the scope of this case study.

5.3 Consistency with the Class Diagram

This component-based design should strictly conform to the class design in Figure 1. Otherwise, a component may send an event to a receiver, who cannot handle it. Or, the sender may provide less parameters than required. The result can be a fatal run-time error.

A program can be written which automatically checks sender-receiver consistency of all the method calls. Not how at the code-level this might be checked by a type-checker and/or linker. For example, `Manager` accepts event `maccept`. This means it provides method `maccept` in its class definition. 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. The checker then looks through the whole `Chat` model and finds that this method is only called (asynchronously) by the `ChatRoom` component. The call `[EVENT("maccept", [[PARAMS][0], [ID]])]` provides exactly two parameters (`[PARAMS][0]`[3] and `[ID]`). The checking of this call is successful.

Similarly, the consistency of all the method calls in the model can be checked against the class diagram.

6 Consistency Checking by Model Execution

The `Chat` model is simulated or possibly executed in real-time (needed in case of a human in the loop) by the SVM interpreter. The output produced in the execution is dumped to screen and a to text file. As mentioned above, human intervention is not needed if all user interaction is explicitly modelled. The output trace is the only means by which we validate the execution. Consistency of the trace with all the design artifacts discussed above must be checked.

Consistency with the class diagram was studied in the previous section. The checker formally checks the statechart design. Model execution is not needed.

Consistency with the sequence diagrams is checked by validating the output trace of experiments. Although correctness can in many cases not be proved (as it would require the exploration of a large or possibly infinite state-space of possible behaviours), *confidence* in the final product is greatly increased.

Consistency with the statecharts is implied provided that the SVM execution environment is correct.

Proving consistency with the original protocol is not easy, because it contains much more information than the sequence diagrams does. It is also hard to be processed by a checker program.

[3] `[PARAMS][0]` here refers to the first parameter of event `request`, which is handled by the `ChatRoom` component. This parameter is further passed on in event `maccept` as the latter's first parameter.

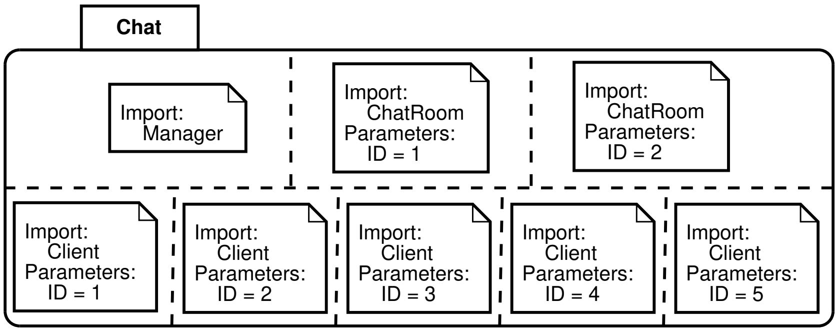

Figure 4: Chat model

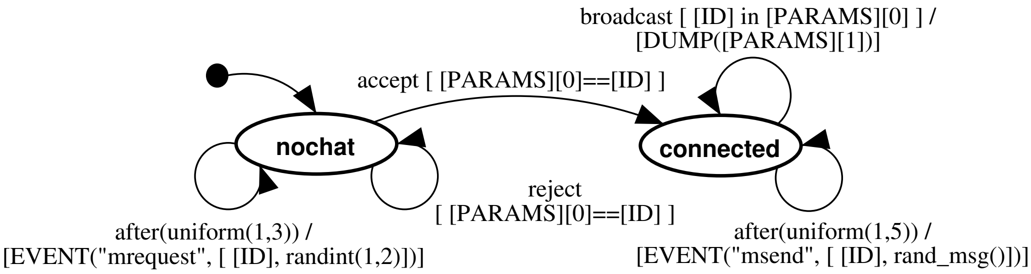

Figure 5: Client component

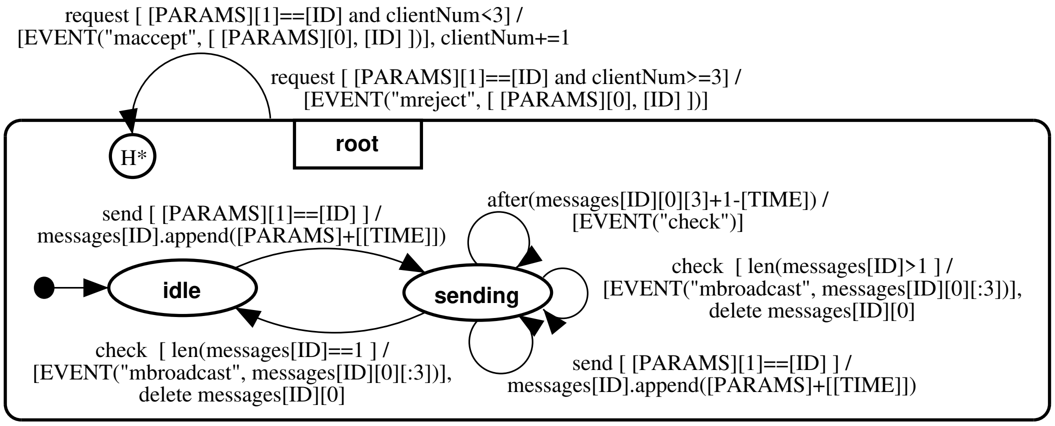

Figure 6: Chat room component

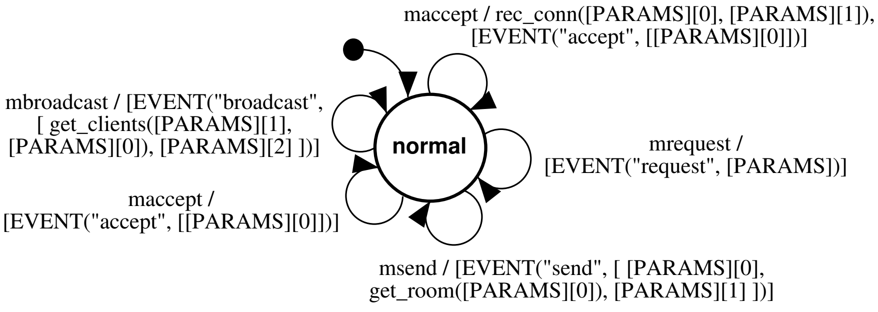

Figure 7: Manager component

6.1 Output Trace

`[DUMP(msg)]` macro is used to record messages `msg` in a file, until the execution is finished (either automatically or by manual control by the debugger). Each message consists of three parts: the time written as a tuple (t, s), the sender or receiver with its unique ID, and the message body. The following is taken from the output:

```
......
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
......
```

This output is produced by the manager, which has access to all relevant information in the communication. At time 10.5 a message is sent by the client with ID 0. According to the protocol, chat room 1 broadcasts this message after 1 second. Another client (client 1), which is also connected to chat room 1, receives the broadcast at the same time. The original sender of the message from is also shown.

6.2 Consistency with the Sequence Diagrams

Consistency with the sequence diagrams can be checked in a rule-based approach. A set of rules are defined and written in a text file. A checker reads the file and checks if the output trace satisfies every rule.

Regular expression are extended to describe rules. A rule consists of four parts: pre-condition, post-condition, guard (optional) and counter-rule property (optional). *Pre-condition* is a regular expression used to match a part of the output trace. It, combined with the *guard* (a boolean expression), defines when the rule is applicable. When it is applicable and the *counter-rule property* is `false`, the *post-condition* (another regular expression) must be found in the output; if counter-rule is `true`, the post-condition must *not* be found.

For example, the following rule expresses the fact that 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`

In the pre-condition, five expression *groups* are defined in parentheses. They are numbered 1 to 5. Group 1 matches the floating-point time. Group 2 matches the sequence number. They constitute a time tuple. Group 3 matches the integer client ID of the sender. Group 4 matches the message, which is an arbitrary string. Group 5 matches the chat room which the sender is in.

In the post-condition, `[(...)]` contains an expression, where values of groups can be cited with their index numbers behind "\". Thus, `[(\1+1)]` is the value of the first group plus 1. `[(\3)]` is equal to group 3. More about the regular expressions used can be found in [5].

Suppose the execution stops at simulated time 50. The checking should not exceed time 50. Without additional conditions, if a message is sent to a chat room at time 49.5, the checker would expect a corresponding broadcast at time 50.5. To cope with this, a guard `[(\1+1)]<50` is added. This tells the checker that the rule is applicable only when the value of group 1 (floating-point time) plus 1 is less than 50.

Since a client should not receive its own message, this is a counter-rule.

6.3 Consistency with the Protocol

It is difficult, if not impossible, to prove the model is completely consistent with the protocol. The protocol, also regarded as a set of requirements, is described in natural language. Its interpretation is the main obstacle for the development of an automatic checker.

One may argue that the protocol can be transformed into a set of rules. With the rule-based method described above, consistency with the protocol can be checked. However, it is hard to transform the complete meaning of the protocol into a formal representation, which is easily processed by a program. Obvious facts implied in the protocol and common knowledge are usually lost. As an interface between human beings and computer programs, a natural language processing technique is required.

In this case study, 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.

On the one hand, checking intermediate steps is not strong enough. On the other hand, it is extremely hard to check the model directly against the original protocol. "How to prove the correctness of a final design" is the last and largest open question in this case study.

7 Conclusion

A concrete example is discussed in this case study. An executable model is developed from initial requirements. Steps are gone through and designs at different levels of abstraction are studied. A component-based approach is chosen to make the model modular and manageable. A class diagram defines the interface of components. Sequence diagrams formalize the communication and make automatic checking possible, though they only partly illustrate the requirements. The component-based model is modelled in the extended statechart formalism. This model is directly interpreted by the SVM execution environment.

Development of the chat room model gives rise to a series of consistency problems. For some of them, automatic checking is successfully applied.

1. Consistency between the *sequence diagrams* and the *class diagram* is checked. A checker verifies all the required methods are correctly specified in the interface.
2. Consistency between the *statecharts* and the *class diagram* is also checked in a similar way. The sender of an event always provides enough parameters to the receiver.
3. Consistency between the *statecharts* and the *sequence diagrams* is checked with a rule-based checker. Regular expressions are extended to specify pre-conditions, post-conditions, guards and counter-rule properties.

However, other consistency problems remain unsolved.

1. Consistency between the *class diagram* and the *protocol* (initial requirement) is not checked. Design flaws may be discovered in later steps or be hidden in the final product.
2. Consistency between the *sequence diagrams* and the *protocol* is only checked manually. Though the sequence diagrams are just a formalization of the protocol, it is not easy to check their correctness with a program.
3. It is even harder to check the consistency between the final design in *extended statecharts* and the *protocol*. This checking is necessary, as information is lost in intermediate steps.

Attention must be paid to these open questions which mostly pertain to inter-consistency. It is believed that consistency checking should be an integral part of the development process and of software development tools.

References

[1] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. Technical Report MSC01-15, The Weizmann Institute of Science, 2001.

[2] Thomas Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S147 – S166. The Society for Computer Modelling and Simulation, July 2003. Montréal, Canada.

[3] Thomas Feng. Statechart Virtual Machine (SVM), 2003. MSDL, McGill University, `http://moncs.cs.mcgill.ca/people/tfeng/?research=svm`.

[4] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[5] Python 2.2.3 documentation, May 2003. `http://www.python.org/doc/2.2.3/`.

[6] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] Michael von der Beeck. A structured operational semantics for UML statecharts. *Software and Systems Modeling*, 1(2), 2002.

[8] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.