Report

Open distributed systems

Executable UML

Geir Melby

Content

1 Introduction

1.1 Background

Ericsson has for long time used formal languages to make executable models. AXE 10, the most successful line switch ever made was originally described using FDL (Functional Description Language) to describe the behavior and structure of the system. During the nineties the AXE development started to use SDL (Specification and Description Language) an ITU standard for design, simulation, verification and 100% code generation to the target platforms.

The background for this formal approach is motivated by the complexity, quality requirements, complexity of behavior and the size of the products. At the end of the nineties Ericsson supported the development of UML, which is now the preferred specification and design language to be used in new product development in Ericsson.

Earlier versions of UML are lacking the formalism needed to make executable models of complex real time systems. Ericsson has therefore in the RFP for UML2.0 made proposals for more advanced architectural and behavior concepts. Ericsson has also requested a more powerful language for interactions based on the MSC language standardized of ITU.

NorARC (Norwegian Applied Research Center) has developed a prototype of an execution framework for a role-oriented approach, called ActorFrame. ActorFrame is implemented in Java and it is based on JavaFrame that is an execution platform for communicating state machines according to the proposed UML 2.0 standard. ActorFrame is used to make a framework for creation and execution of services. It is called ServiceFrame and it can be used both to develop traditional telecom services, Internet like applications and a combination of these.

Object Management Group (OMG) has changed the focus from Corba standard as a platform and language independent middleware for integration of systems, to a Model Driven Architecture approach (MDA) where UML is the core. From platform independent models (PIM) they foresee transformation to platform dependent models from where code can be generated to different platforms and middleware. This is an approach that is supported by Ericsson. There are books on market that describes how to use UML (version 1.4 or earlier) to make executable models with UML.

1.2 Objectives

The main objectives for this study are

- To identify the main improvements of UML 2.0
- To make an executable UML model for a simple application based on ActorFrame principles
- To evaluate if UML 2.0 can be used to make advanced executable framework models like ActorFrame

1.3 Convention

Substantives in italics and starting with a capital letter are classes in UML. For instance *Actor* is a class Actor and *anActor* is an instance of class Actor. Substantives with ordinary format and starting with a capital letter, represent concepts or names that are used in the context of this report. For instance ActorFrame is the name of a framework used in the ServiceFrame.

1.4 Readers guideline

This report presents in chapter 2 the new concepts in the current proposal to new UML2.0 standard. In chapter 3 is ServiceFrame described with focus on those part that are most relevant for the Chat example that will be modelled. An UML model of the the Chat example and the results from the simulation of the Chat model are presented in chapter 5. The experiences from the study are presented in chapter 6. At last in chapter 7 is the conclusion from the study presented. The Tau 2.0 UML tool from Telelogic will be used in this evaluation.

The report does not give detailed descriptions of UML and ServiceFrame and the report is written for readers that in advance are familiar with those concepts that are used in the report. A whitepaper about ServiceFrame is found in [4]. It may also be useful for the readers that are not familiar with statemachines and advanced interactions diagrames to read the SDL [5] and the MSC [6] standard.

2 UML 2.0

2.1 History and current status of the language

Rational with Ivar Jacobsson, James Rumbough and Grady Booch made UML 1.0 in 1997. It was based on these author's earlier methodology and language work. A brief history of the language can be found in [ref]. A revised version of UML 1.1 was offered to OMG, which took the responsibility for the further development of the language. During fall 1998 OMG Revision Task Force (RTF) released version 1.3, which have up to now been the version that tools have supported.

Version 1.4 was released late 2001. It included an action language for definition of behavior used in operations and state machines definitions. This was an important step towards a more precise language. A constraint language called Object Constraint Language (OCL) is also a part of UML language. OCL is up to now mainly been used to make the metamodel of UML more precise. Designer may also use OCL to make constraints in the model. Supported by tools, OCL and the action language, can significantly improve the preciseness of the model.

A new Request For Proposal (RFP) was submitted in 1999. The work was split in work group for minor a new revision version 1.4 and a major revision of the language called UML2.0.

The RFP has defined following goals that are based on more than 20 proposals:

1. Restructure and refine the language to make it easier to apply, implement and customize by

   Increase precision

   Reuse packages

2. Infrastructure part

   Enable reuse of Core constructs used to define UML2 in other Model-Driven Architecture standards

   Provide more powerful mechanisms to customize UML

3. Superstructure part

   Support component-based development using platform independent components and platform specific components

   Provide architectural structures that allow hierarchical composition of parts and interfaces

   Allow separate semantics for activities and state machines

   Support composition of sequence diagrams

   Refine other constructs and notations (e.g. use cases, relationships)

Ericsson is participating in a consortium named U2 partners (http://www.u2-partners.org). The main actors are Rational, IBM, I-Logix, Motorola, Oracle, Telelogic, Unisys, HP, Business Software, IONA, Alcatel, CA, ENEA, Jaczone, Kabira, Unisys and WebGain. In addition has the consortium a group of supporters. Together they form a good mix of users and tool vendors. There are other contributors to the final proposal, but there is no real competition on the main parts of contribution from U2. A beta version of the proposal from U2 consortium was released in September 2002 and it is planned to submit a final release to OMG RTF during january 2003.

Ericsson got most of the wanted requirements into the RFP (marked bold in the goals described above) and so long they have had a strong influence on the parts of the standard that are important for Ericsson.

The beta releases [2,3] from September 2002 are used in this report.

2.2 Superstructure and Infrastructure parts of UML

The work in U2 consortium has been organized in to groups according to the 2 RFP, Infrastructure and Superstructure. There are also groups for OCL language and XMI. The Infrastructure group has been working mainly with the kernel package to make it compliant with the Meta Object Language (MOF). At the same time the MOF is undergoing similar revisions to have MOF 2.0, so alignment is not trivial While the Superstructure group has worked with that part of the metamodel for introducing elements representing structural architecture and behavior features. In this context is the Superstructure work that is most interesting because their work will define the new concepts that the designer will use.

The final Class concept in UML is very central when it comes to structural behavior and architectural concepts. Important metaclasses are *Classifier* and its subtypes *StructuredClassifier* that introduce the parts and *BehavioredClassifiers* that introduce inheritance of behavior.

2.3 Improvements in UML 2.0

The superstructure improvements are most interesting as they define those concepts to be used by designers. These are

- Changes of internal architectural structure and behavior as introduction of parts, connectors, ports and generalization.
- Components with improved encapsulation through ports and with internal structure of parts with connectors between parts.
- Activities use flow semantics instead of state machines. Action semantics and activities are supposed to be merged.
- Interactions are improved with better architectural and control concepts as composition, sequence diagram references, exceptions, loops and alternatives.

2.4 Architectural concepts

There have been introduced new elements that are representing structural architecture of the meta class *StructuredClassifier*. The motivation for these changes is that some elements shall only exist in a specific context. In addition there is a need for better encapsulation of *Classifier*. Good encapsulation concepts are important for specification of independent components and classes. The *Part*, *Connector* and *Port* are proposed concepts to be used to achieve this.

2.4.1 Parts and Connectors

The example used in the proposal is illustrated in Figure 1. The class model describes a *Car* that consists of *Axle*, *Wheel* and an *Engine*. Each *Axle* is connected to at least two and at most four *Wheel* (may be three wheels). Each *Axle* is also connected to an *Engine*. The *Boat* consists of an *Engine* and *Propellers*. From this model it is possible that the same instance of *Engine* is connected to *aAxle* in a car and *aPropeller* in a boat at the same time.

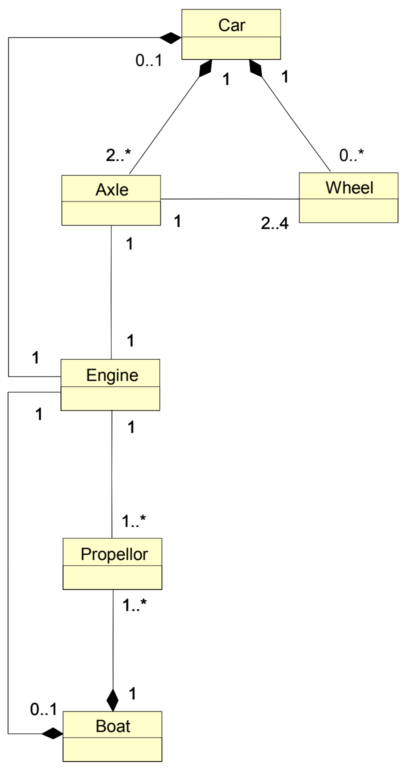


Figure 1 Composition versus parts

And that is obviously not what the model should express. The model should describe the *Engine* independent of its use (encapsulation) and describe more precisely that an instance of *Engine* is part of *aCar* and is connected to *aAxle* of that *Car*. Another instance of *Engine* is part of *aBoat* and connected to an *aPropellor* of that boat. This is main motivation for introducing the Part and Connector concept.

In Figure 2 the *Car* has got an internal structure of parts of other classes and connectors that are connecting the different parts together through ports. These internal parts and connectors will only exist as a part of an instance of the class Car. So when an *aCar* is created, *e:Engine* is connected to *d:Axle* that has 2 instances of *dw:Wheel*. The car has also one or more sets of *Axle where* each *n:Axle* has exactly one pair of *nw:wheel* connected. It also describes that the same instance of class *Engine* cannot be parts of both a car and a boat, which is possible according to the model shown in Figure 2.

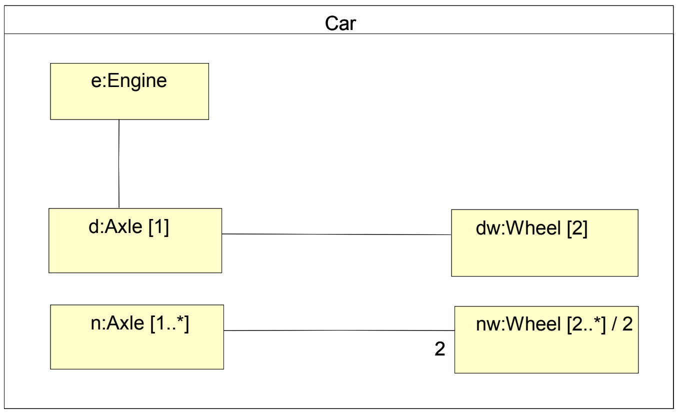


Figure 2 Class with internal structure

2.4.2 Ports

When instances shall be connected together, the connection point has to be described formally. The concept of Port describes an entry and an exit point for a class as described in Figure 3. A port encapsulates communication for an *EncapsulatedClassifier*. It contains a provided interface that specifies services offered by the classifier and its environment and a required interface that describes services the classifier expects from its environment.

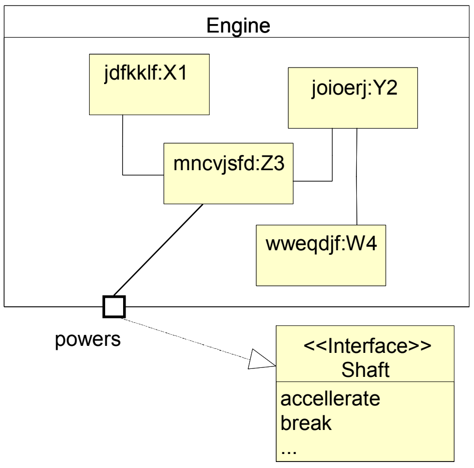


Figure 3 Ports connected to classes

Figure 4 shows a class *Engine* with a port on its environment. All communication to or from the class has go through this port named powers. Use of ports enables specification of a class without knowing anything about the environment where the class may be used. Classes can send and receive signals via ports, and a class can expose operations through a port. Ports also relay communication along connectors.

2.5 Behavior concepts

The major changes of the state machines in UML are

- Composite state with entry/exit points that increases the scalability and independence of behavior specification.

- State machine generalization that enables inheritance and specialization of behavior.
- Protocol state machines that enables specifying of allowed sequences of signals and operation calls.
- State machine for operations that enables procedural like calls on state machines.
- State groups that enable common behavior of events in different states.

Entry/exit and generalization concepts are described below.

2.5.1 Entry/Exit points

Earlier versions of UML have no limitations on how to entry and exit composite states (Substates). It is still legal to enter directly into a new state of the composite state and it is therefore difficult to specify behavior components that may be reused in another state machines. An example of a definition of a Substate with entry and exit points is shown in Figure 5. And an example on how to use entry and exit points of the Substate ReadAmountSM is shown in Figure 5.

For example if the state machine *ATM* gets an input signal *rejectTransaction* in state *VerifyTransaction*, the transition enter the substate *ReadAmount* through the port *again*. This port is connected to the *EnterAmount* state.

An unnamed entry or exit point represents default behavior. You may have several entry and exit points.

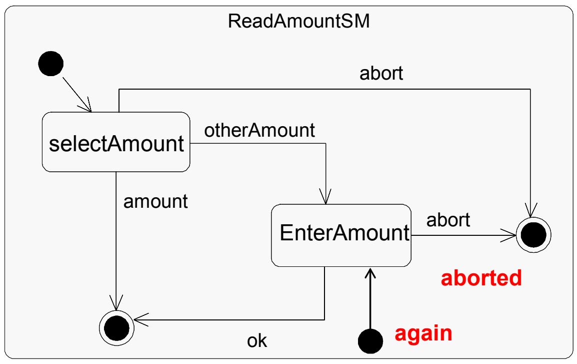


Figure 4 Definition of Exit / Entry points

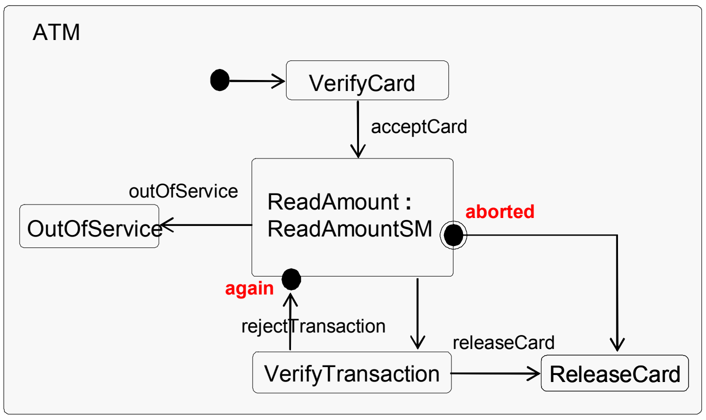


Figure 5 Use of Exit / Entry points

2.5.2 Generalization

The generalization and specialization concepts have been an important part of the UML language. There has not been possible until now to inherit behavior of state machines. As shown in Figure 6 this has now been added to UML in the same way as ordinary inheritance of classes is done. New state machine types can also be specified using inheritance independently of classes. New behavior can be added and parts of existing behavior can be redefined as follows:

- States and transitions can be added
- States and state machines can be extended
- Effect actions, that is behavior specified in the transitions, may be replaced
- Targets of transitions can be replaced
- Sub machine states can be redefined adding entry/exit points and replacing the substate machine.

Examples of these concepts are shown in chapter 6.2.

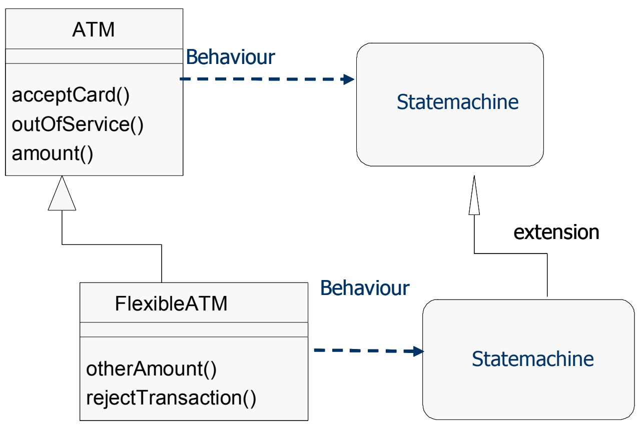


Figure 6 Specialization by extension

2.6 Interaction diagrams

An important element of specification of systems is modeling of the behavior, especially the interaction between objects (parts). In complex systems it can be a significant amount of sequences of wanted and not wanted interactions. The Sequence Diagram (SD) in previous versions of UML has lacked language constructs to describe interactions in a more manageable and compact model. This is the motivation behind the significant improvements of the Sequence Diagrams in UML. The most important improvements are:

- References that can be used to refere to other interactions that increase modularization and reuse of the interactions.
- Combinations that express alternatives, exceptions, loops etc that makes the diagram more compact.
- Lifelines may be detailed recursively (Decomposition) enabling an abstraction of the interaction details.
- Better overview of combinations of interactions enabling high-level interactions where lifelines and individual messages are hidden.
- Gates that gives connection points between references and their environments.
- Dynamic creation and destruction of lifelines.

Decomposition and diagram references are described below.

2.6.1 References

The Figure 7 shows examples on how to use references in an interaction diagram. The referenced diagram *Authorization* is bound to a containing diagram via gates. The gates have no names, which may make it more difficult to understand the diagram. The gates are bound via the names of the signals. The interaction diagram shows also a definitions of an CombinedFragment with operator “opt”, which express a specific option that may happen.

Figure 8 shows the definition of the referenced diagram *Authorization*. It also specifies a creation and later a destruction of a lifeline or the part *Autorizer*.

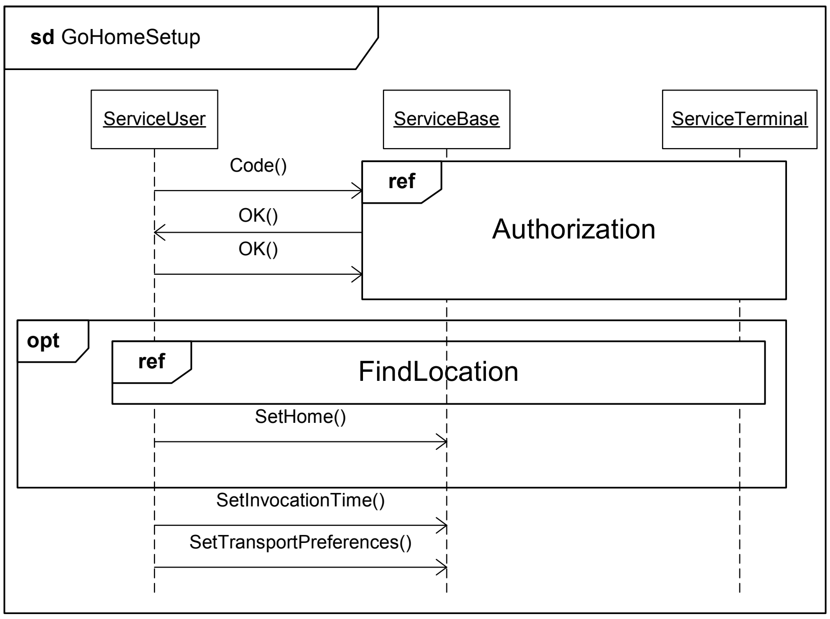
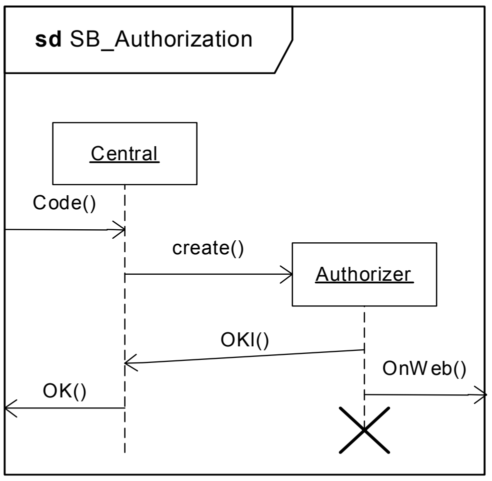


Figure 7 References and gates



Feil!

Figure 8 Gates and creation / destruction

2.6.2 Decomposition

Figure 9 shows how to describe a decomposition of a lifeline (part). *Servicebase* refers to an another interaction diagram named “SB_Authorization” in Figure 8, that shows what happens inside the part.The signals at border of “SB_Authorization” are the same as the signals that are connected to the lifeline *ServiceBase* in Figure 9.

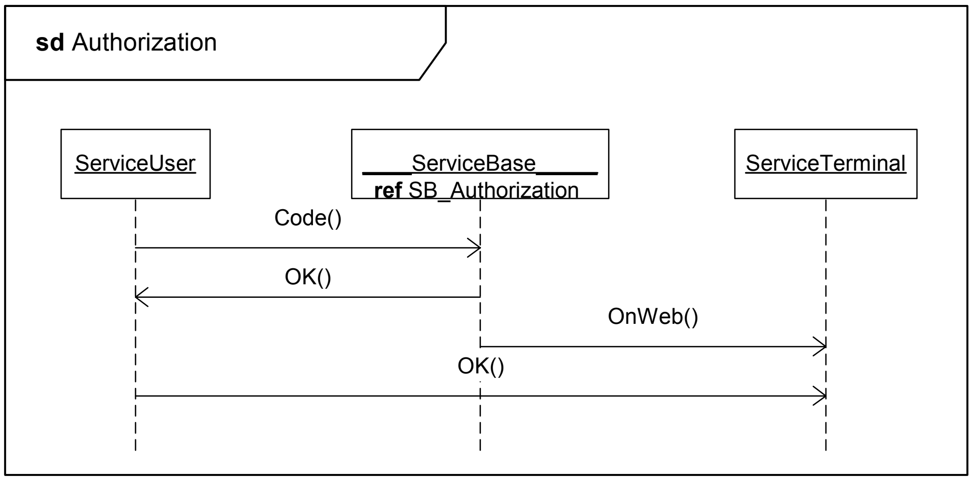


Figure 9 Decomposition of lifelines

3 ServiceFrame

3.1 ServiceFrame overview

ServiceFrame provides a set of domain given actors, with generic attributes and behaviour. Services are applications of ServiceFrame that are defined by specializing and instantiating the Actors and by defining and deploying Roles.

This chapter will describe those parts of the ServiceFrame that is used in the modeling of the chat application. A description of the ServiceFrame can be found in [SERVICEFRAME].

3.1.1 ServiceFrame

ServiceFrame provides architectural support for service creation, service deployment and service execution. Services are realized by ServiceFrame applications that are defined by specializing and instantiating framework classes. The idea is that service developers shall be able to concentrate on modeling the service functionality and be relieved from considering technicalities that are not service specific.

To this end ServiceFrame provides architectural support for modeling and for implementation in terms of domain concepts. In addition it has mechanisms that support incremental development and deployment of services.

The architectural support is provided in three layers, as illustrated in Figure 10

ServiceFrame itself is an application of ActorFrame, which is a generic application framework supporting Actors and Roles. Both are implemented in Java using JavaFrame [JavaFrame], which provide support for state machines and asynchronous communication according to UML2.0 running on a Java Virtual Machine.

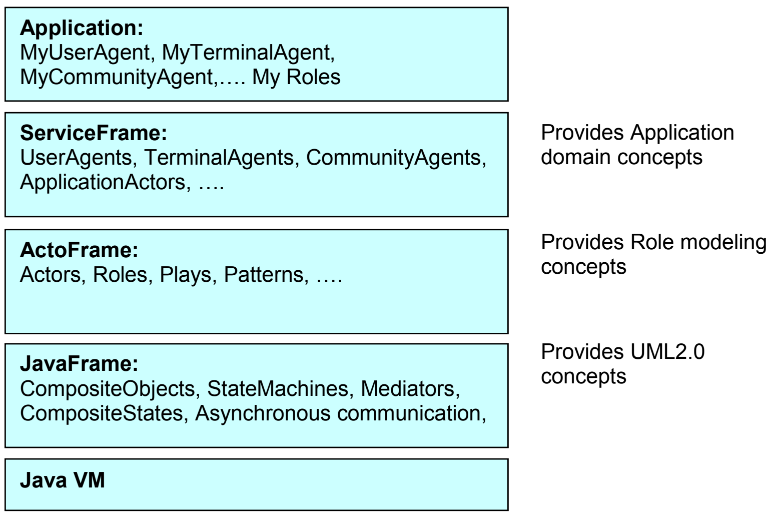

Application:
MyUserAgent, MyTerminalAgent, MyCommunityAgent,…. My Roles
ServiceFrame:
UserAgents, TerminalAgents, CommunityAgents, ApplicationActors, ….
Provides Application domain concepts
ActoFrame:
Actors, Roles, Plays, Patterns, ….
Provides Role modeling concepts
JavaFrame:
CompositeObjects, StateMachines, Mediators, CompositeStates, Asynchronous communication,
Provides UML2.0 concepts
Java VM

Figure 10 ServiceFrame layers

3.1.2 ActorFrame

The Actor illustrated in Figure 11, is the core concept of ActorFrame. An Actor is an object having a state machine and an optional inner structure of Actors. Some of these inner Actors are static, having the same lifetime as the enclosing Actor, and others are dynamically created and deleted during the lifetime of the enclosing Actor. The state machine of an Actor will behave according to generic actor behavior, common to all actors, and a Role type, which is bound when the Actor is instantiated. If the Actor shall play several Roles, this is accomplished by creating several inner Actors each playing one of the desired roles.

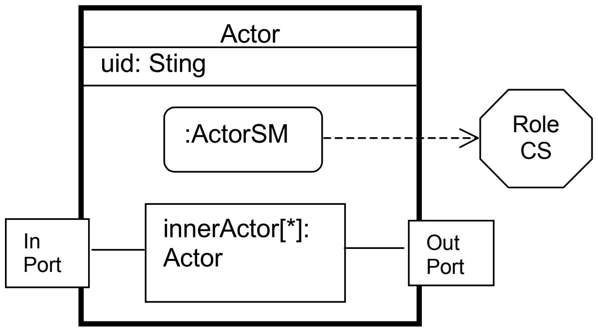


Figure 11 The Actor and the dynamic Actor context

Communication between the Actor and its environment takes place via an Inport and an Outport. Internal communication among the inner actors is also routed via the ports.

The Actor has a generic behavior, inherited from the base Actor type, that provides management functionality. It manages the inner structure of Actors and the Roles they play. It knows the available Roles and the rules for Role invocation. The generic behavior handles role requests as described in Figure 12. It will either deny the request or invoke an Actor to play the requested role or an acceptable alternative role. The generic behavior also has the capability to add and remove roles, and to perform other Actor management functions.

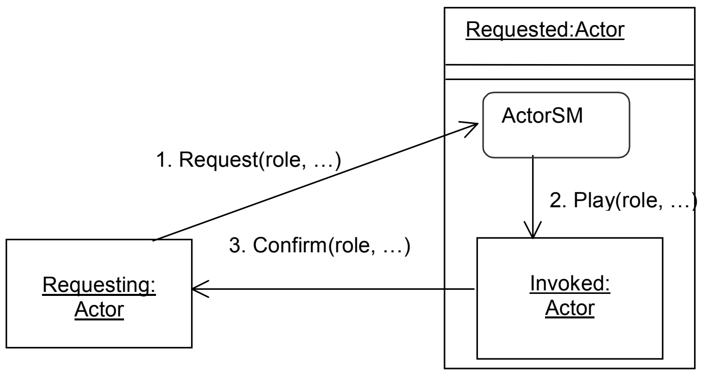


Figure 12 RoleRequest protocol

3.1.3 ActorFrame protocol

ActorFrame has protocols for role requests and role releases. New roles can be created dynamically and initiated on requests. The idea is that an Actor can request another Actor initiate new roles (Actors) to do the requested services. The Sequence Diagram for a role request is shown in Figure 13.

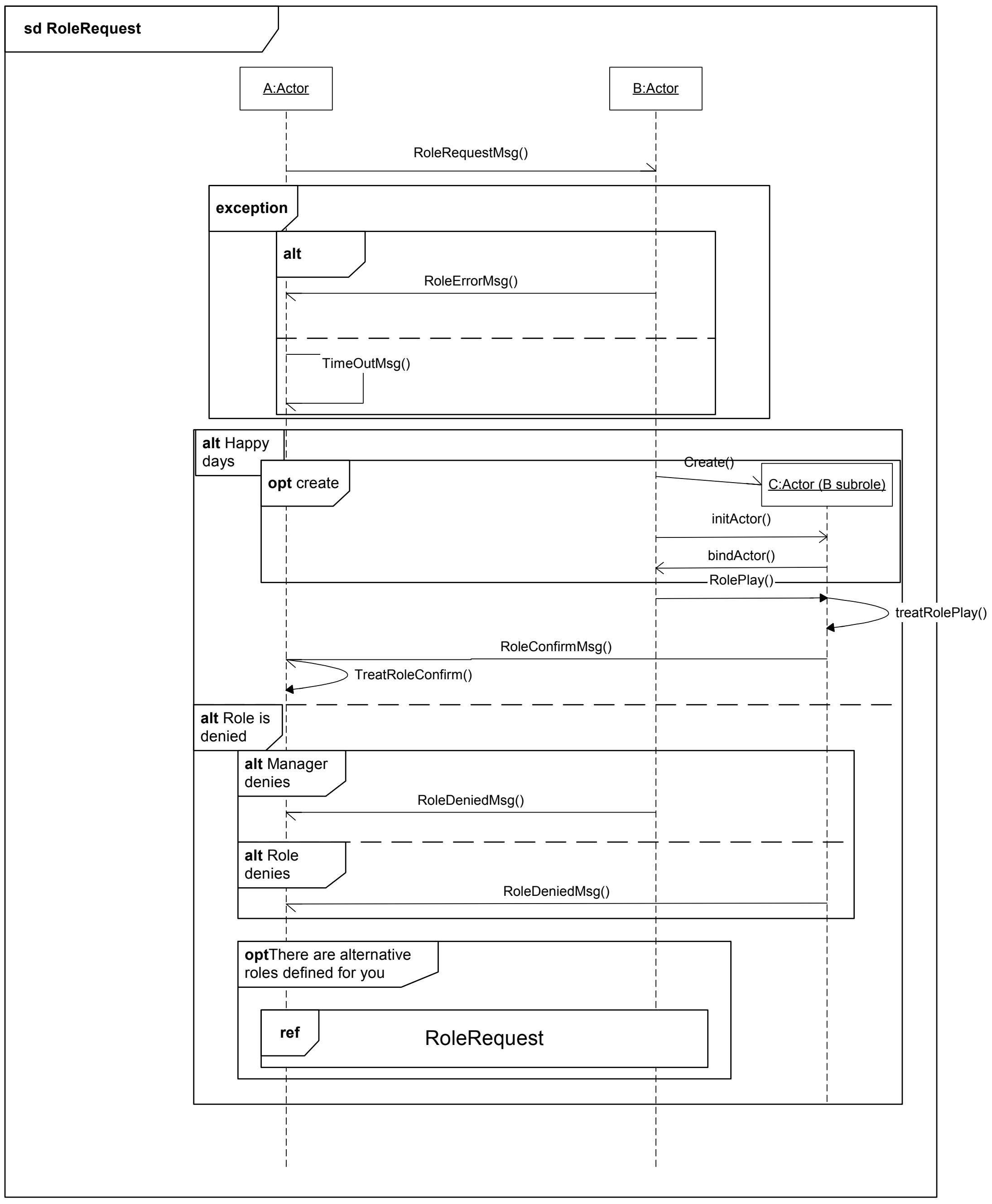


Figure 13 RoleRequest pattern

As shown in Figure 14 an actor may request several other actors and several other actors may request one actor. All actors are running in parallel. An actor may play several roles in parallel. If a requested role is released from all requestors the requested actor will delete the role. If an actor is defined but it does not exist, it will be created.

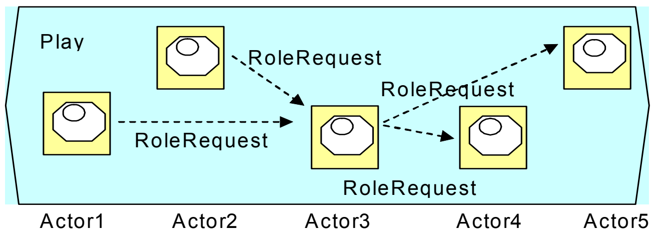


Figure 14 Multiple Roles and actors

The protocol must also be able to handle errors that may occur. It may not be possible to create an actor if the actor template does not exist or available resource is limited. The requested actors may be distributed to different machines, so hardware errors must be handled.

3.2 Service specification

The basic feature of the protocol is to allow an actor (requestor) to request another actor to play a specific role and to allow the actors to interact to perform a service or a play. The protocol includes also a protocol to release a requested role. This is shown in Figure 15.

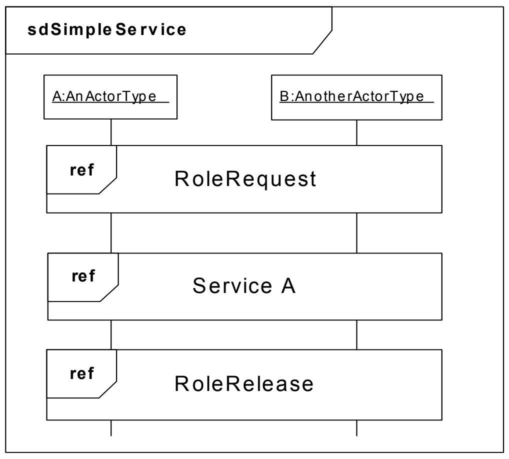


Figure 15 A simple service

The referenced sequence diagram ServiceA may again include the RoleRequest and the RoleRelease protocols. In chapter 5 is the chat example described.

3.2.1 ActorFrame behavior

The behavior of an Actor is described in Figure 16 and the composite state *Playing* is described in Figure 17. This implementation uses advanced object oriented concepts as inheritance of behavior and architectural concepts as sub states. The ActorFrame protocol is also implemented with use of polymorphism.

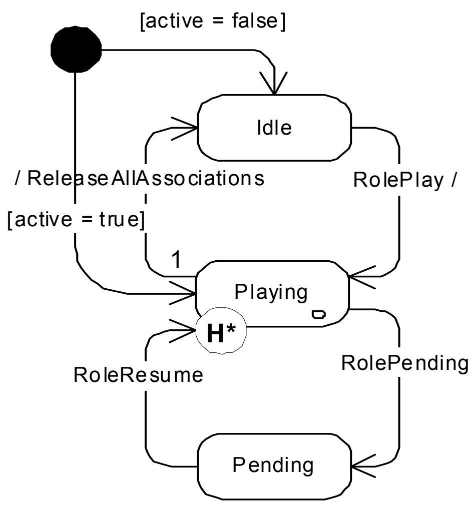


Figure 16 Bevaior of Actor

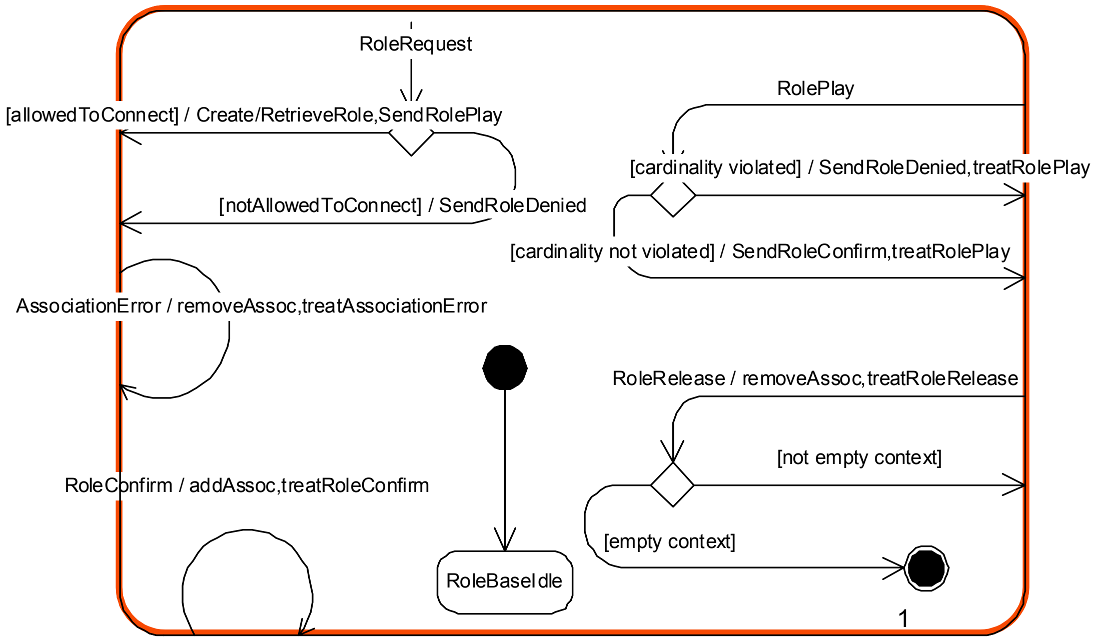

RoleRequest
[allowedToConnect] / Create/RetrieveRole,SendRolePlay
RolePlay
[cardinality violated] / SendRoleDenied,treatRolePlay
[notAllowedToConnect] / SendRoleDenied
[cardinality not violated] / SendRoleConfirm,treatRolePlay
AssociationError / removeAssoc,treatAssociationError
RoleRelease / removeAssoc,treatRoleRelease
[not empty context]
RoleConfirm / addAssoc,treatRoleConfirm
[empty context]
RoleBaseIdle
1

Figure 17 Playing composite state

3.3 Implementation of new Actors

ServiceFrame is implemented in Java and new Actors are implemented by specialization of generic Actor classes with behavior as described earlier a shown in Figure 18. An Actor class represents a complete behavior and instances of Actors may be created without any specialization. It then has all behavior of a generic Actor that enabling lifecycle management and handling of the RoleRequest protocol.

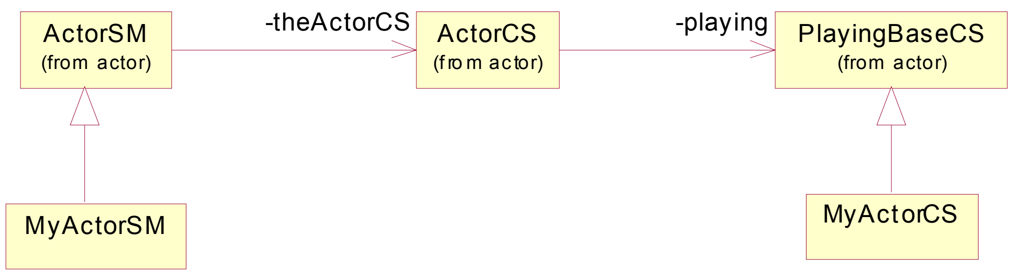


Figure 18 New Actors from specialization of ActorFrame classes

4 Tool – Tau 2.0

Telelogic has for many years been building Case tools for making executing models mostly for real time systems. They have supported the ITU languages SDL and MSC enabling designers to build complete models describing both architectural and behavior aspects of a system. Telelogic is now one of the active U2 partners that are making proposals for UML2.0.

Telelogic has now released a new generation of its Case tool called Tau 2.0. The tool supports UML2.0. The tool has support for:

- Designing of UML models - This include both specification of system architecture and behavior including an action language for specification of transitions and operations.
- Analyzing of UML models - This includes both syntax and semantic checks of the UML models.
- Automatic code generation – 100% code is generated for the target language C/C++. It also supports executable UML models with behavioral specifications.
- Dynamic model verification – The UML models can be simulated and the functionality can be verified through a rich set of debugging trace features.

The version Tau 2.0 does not have support for the whole UML2.0. It has either support for full generalization / specializations nor has it polymorphism. It has also some features that are not part of UML as a special data type PID that is used as a references to a state machine. Reason for this, is probably that the code generator and simulator, is built on an earlier toolset that supported SDL. This limitations in the tool have led to some problems when comes to the design of the *ActorFrame,* which is implemented in Java with advanced object oriented techniques. Comments on this will be given in chapter 6.1.

However this tool supports the basic idea of this Model Driven Architecture approach.

5 Example of UML model – a chat application

The idea of study was to make an UML model of the *ActorFrame* described in chapter 3.1.2 and then to make an application using the *ActorFrame* framework. But because of lacking of support of full UML and limitations in the tool, this was not possible.

This chapter will however describe a chat example where part of the *ActorFrame* protocol is used. But as the example will show, the seperation of the *ActorFrame* and the chat example is not clean. This example also shows the difference between a reuse-oriented approach that *ActorFrame* supports and the implementation of this chat example where little of the *ActorFrame* behavior can be reused.

This chapter describes the chat example with some comments. In chapter 6 this will be followed up with comments on the use of UML 2.0 concepts. This chapter will also describe the results from a simulation of the model showing that the model conforms to the specification.

There are two models for the chat application. The first one is a domain model showing the basic functionality of the chat functionality without showing how the chat application is implemented. This model is a Platform Independent Model. The other model is a design model or Platform Specific Model where the model shows how the *ActorFrame* protocol is used to model the chat application.

This chapter will describe only parts of the model. A complete listing of the model is given in Appendix A – Chat Application model.

5.1 Domain model

There is only one use case of this chat example that is *Chatting* as shown in Figure 19. The domain model shown in Figure 22, has the following domain concepts:

- *ChatService* that is a manager of the chat rooms and that gives access to user of the chat service.
- *User* that is the user of the chat service.
- *Client* that represent the device that the user uses to type and read chat text.
- *ChatRoom* that is modeling a chat room that sends received chat text to clients.

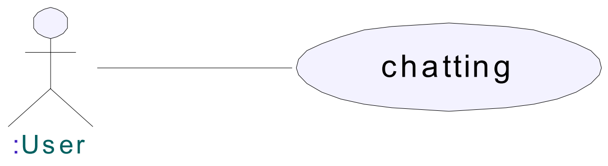


Figure 19 Chat use case

The ChatService may have many chat rooms and clients connected to a chat room. One client may only be connected to one chat room. Chat rooms are created dynamically on request from the users. When no clients are connected to a chat room, the chat room is deleted.

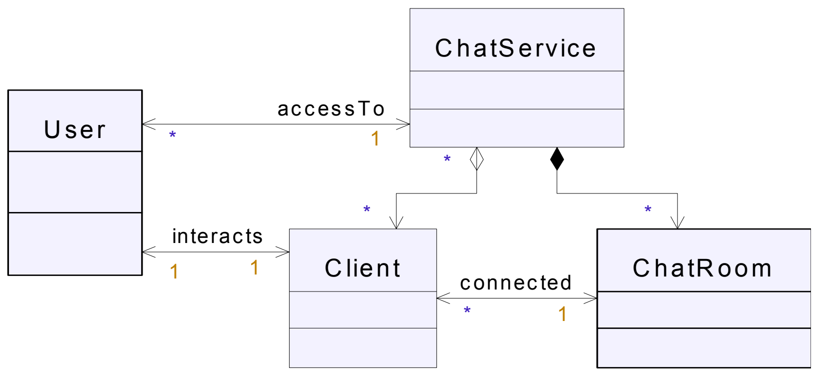


Figure 20 Domain model

A typical interaction of the chat service is shown in Figure 21. The message *ChatFromClient* contains the input text from a user and the message *ChatToClients* contains the text that is sent from *aChatRoom* to *aClient* that presents the text to the user.

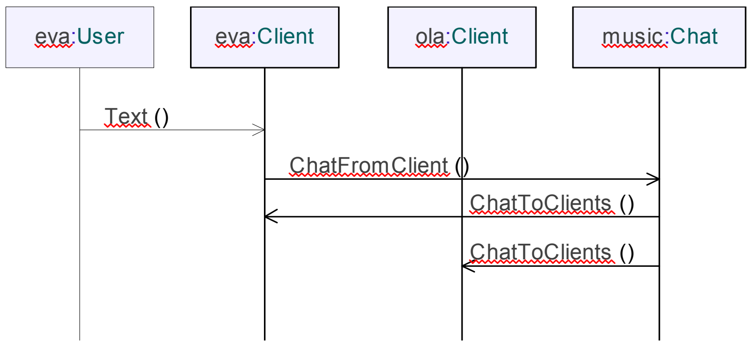


Figure 21 Interaction diagram for chatting

The sequence diagram does not show the RoleRequest and RoleRelease sequences. The use of ActorFrame protocol messages is not a part of a domain model. They will be included in the design model (PSM).

5.2 Design model

The design model is the base for the implementation of the application. The design model uses the basic classes from the domain model, refines these classes with more details, and adds system or platform specific classes. In this case we will use the ActorFrame protocol to dynamically create and delete chat rooms. This put requirements on the participating classes or Actors as they are called in the ActorFrame terminology.

5.2.1 Class model

The design adjusted class model is shown in Figure 22. Active classes (classes that have state machines) should normally be sub classes of *Actor* such that they could participate in the ActorFrame protocol.

In this model the ActorFrame protocol is only partly implemented in the different classes. This simplification can be done because the *Client* and *Chat* classes do not contain other classes and therefore these classes do not have to respond to the *RoleRequest* message. The new class *RoleManager* that may contain both instances of the classes *Client* and *Chat,* must however respond to *RoleRequest* message to be able to create instances of the contained classes.

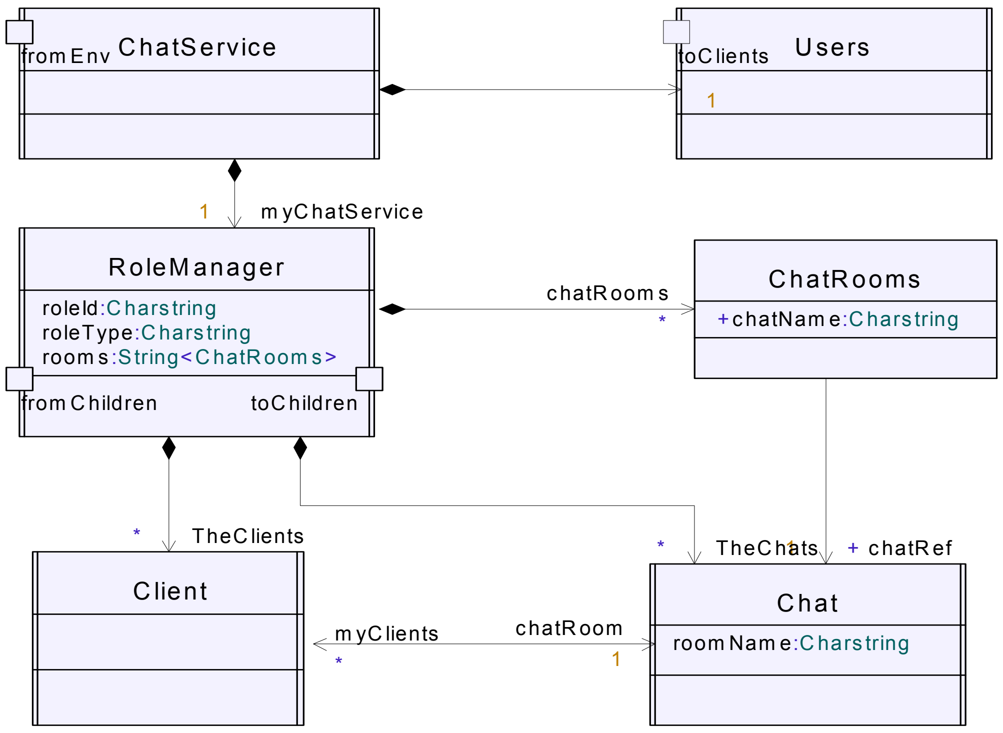


Figure 22 Class model

The class model in Figure 22 shows that the *RoleManager* has a composition relation to *Client* and *Chat*. In UML 2.0 it is introduced a new diagram type that can specify the internal structure of classes. Figure 23 describes the internal structure of *RoleManager*. This diagram specifies the instances with the cardinality on the set and how the different sets are connected. Connectors connect the ports together and they also specify the direction of and the signals that the connectors may convey.

The two ports in Figure 23 with a state like symbol connected to it, represent ports to the state machine of the containing class. For instance from port *toManager* of the set *myChat* is connected to the port *fromChildren* that belongs to containing class *RoleManager.* The signal *RoleEnd* may be sent from *myChat* to *RoleManager* on connector *c4*. In the specification of the instance set, the cardinality can be a fixed number of instances or a range of instances. For example *chatClient:Client[2..4]* specifies that two instances of *Client* is created when an instance of the containing class is created and two more instances of *Client* may be created at runtime.

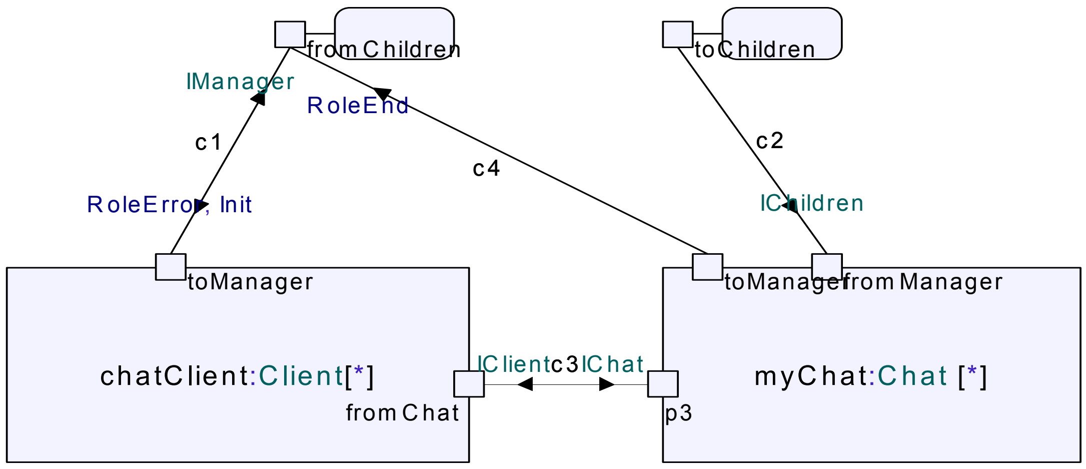


Figure 23 Architectural diagram for class RoleManager

5.2.2 Interaction diagram

A design adjusted interaction diagram of the use case Chat is shown in Figure 24. Here is the prerequisite that the two instances *ola:Client* and *eva:Client* are already created.

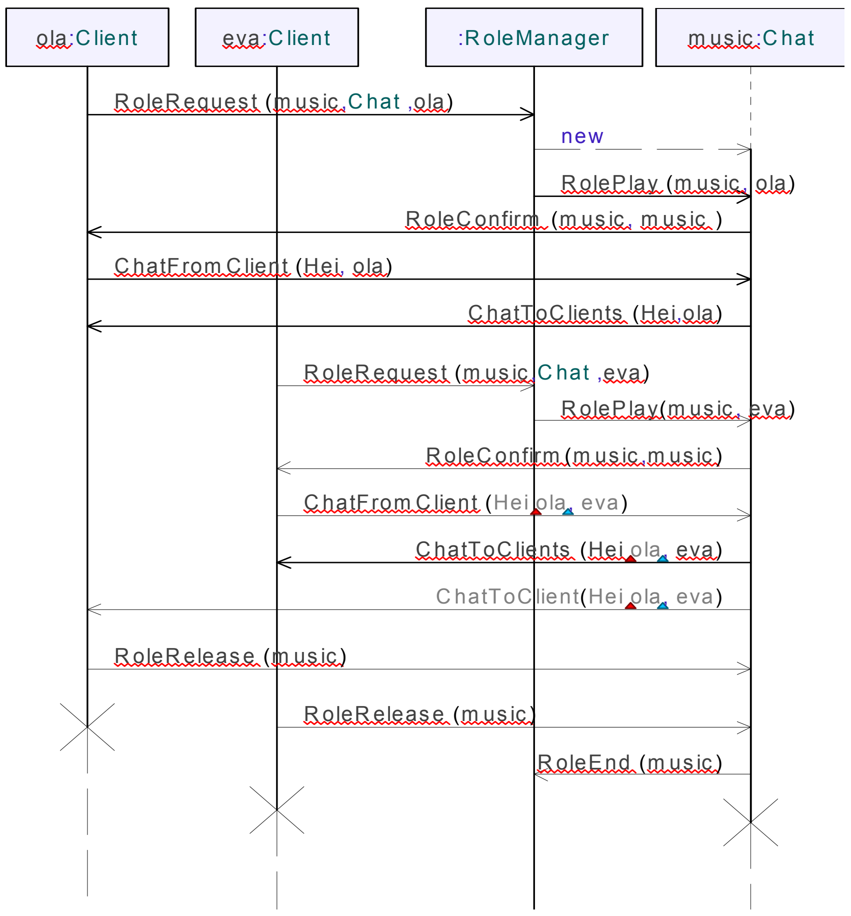

ola:Client
eva:Client
:RoleManager
music:Chat
RoleRequest (music,Chat ,ola)
new
RolePlay (music, ola)
RoleConfirm (music, music)
ChatFromClient (Hei, ola)
ChatToClients (Hei,ola)
RoleRequest (music,Chat ,eva)
RolePlay(music, eva)
RoleConfirm (music,music)
ChatFromClient (Hei,ola, eva)
ChatToClients (Hei,ola, eva)
ChatToClient(Hei,ola, eva)
RoleRelease (music)
RoleRelease (music)
RoleEnd (music)

Figure 24 Normal chatting sequence

In this diagram the RoleRequest and RoleRelease patterns from the ActorFrame protocol are shown in the sequence diagram. These patterns are normally described in separate sequence diagrams where the protocols are specified with and all exceptions and alternatives. In UML2.0 these diagrams can be referenced to such that the sequence diagrams become more abstract and easy to understand for application designers. The sequence diagram in Figure 25 is the same as shown in Figure 21, but now is the ActorFrame protocol only referenced to in the diagram.

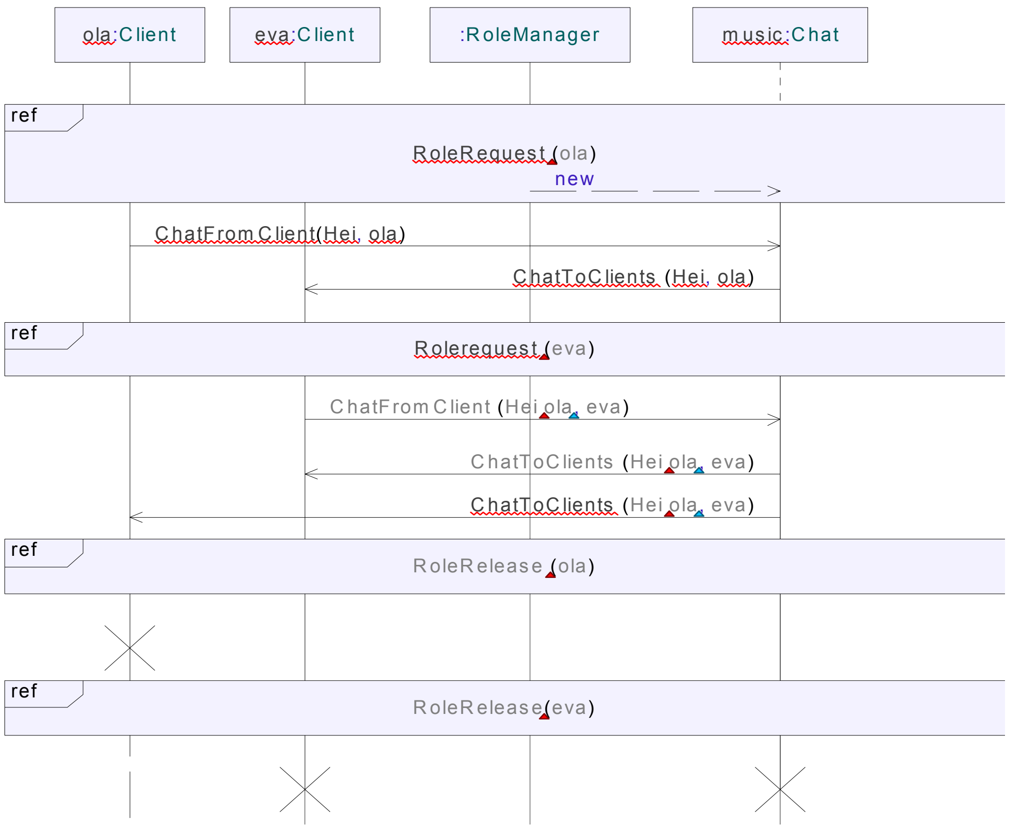


Figure 25 Interaction diagram with use of references

5.2.3 Classes with architecture and behavior diagrams

5.2.3.1 ChatService

The *ChatService* class contains one *Geir:Users* and one *chatManager:RoleManager* as shown in Figure 26. *Geir:Users* has a state machine that sends at creation time signals of type *StartClient* to *chatManager*. The *chatManager* then creates clients and initiate the clients with names.

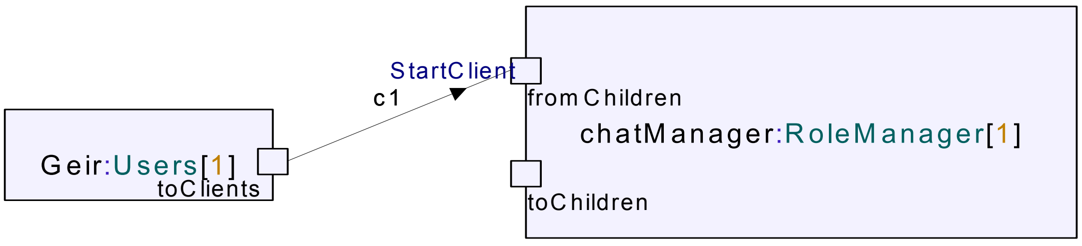


Figure 26 ChatService

5.2.3.2 Class RoleManager

The internal structure of *RoleManager* is shown in Figure 24. It contains a *Client* set that represents interaction point with the users of the chat service. A *chatClient:Client,* makes a RoleRequest to its containing *RoleManager* for a chat role. The *chatManager* creates then an instance of *Chat* if not the chat room already exists. This sequence is shown in the sequence diagram in Figure 24 and Figure 25.

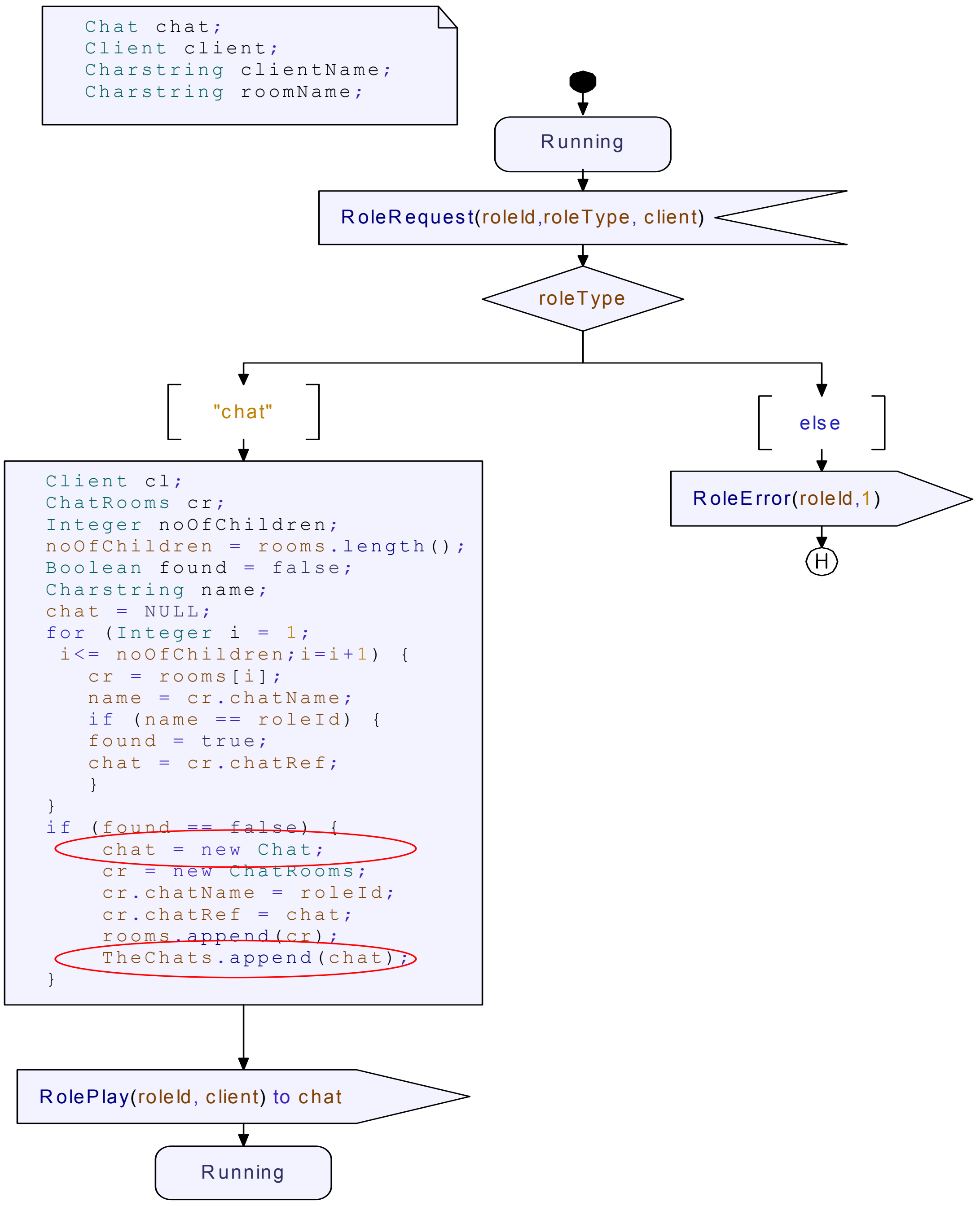
Part of the behavior of *RoleManager* is shown in Figure 27. The state chart diagram describes the transition when *RoleRequest* signal is received in state Running . This diagram shows both a new graphical representation of actions in UML and the textual representation of the action semantics defined in UML. Upon the reception of the signal *RoleRequest* the actual parameters of the signal *roleId, roleType* and *client* are updated with information contained in the signal. The next symbol is a decision where the variable *roleType* is tested against the guard "chat". In this case this is the chat role that the client requested. If the test failed, the "else" guard is selected.

The task symbol after the guard "chat" contains actions written in the syntax of the action language that this tool support. The textual syntax of this language is tool specific, but it may be transformed to a representation in XMI that is part of UML standard. The textual syntax is very much like C#, C++ and Java. It is also possible to use this action language to specify a complete state machine.

After the task symbol it comes an action of type signal output. At last the transition ends in the state Running. After the output action of *RoleError* signal the transition ends in a circle with an *H.* This is a shortcut for writing the state name of the origin of this transition.

It may however be use to comment on the actions circled in red in Figure 27. The statement marked with a circle in red in Figure 27, "chat = new Chat" creates a new instance of class Chat including initialization of its internal behavior. The variable *chat* is a reference to the new instance and the reference can be send as a parameter of signal to another state machine. This is exactly what happens in the RoleRequest sequence when *aChat* receives the *RolePlay* signal and it sends a *RoleConfirm* signal to requestor by using the "to reference" clause in the output action.

The other circle contains the statement "TheChats.append(chat)". The variable *TheChats* is the assosiation end named *TheChat* of the composition association from *RoleManager* to *Chat* shown the class model in Figure 22. The variable *TheChat* may contain zero or more references depending of how many instances of *Chat* has been created. Rest of the actions should be easy to understand.


Running
RoleRequest(roleId,roleType, client)
roleType
"chat"
else
RoleError(roleId,1)
H
RolePlay(roleId, client) to chat
Running

```
Chat chat;
Client client;
Charstring clientName;
Charstring roomName;
```

```
Client cl;
ChatRooms cr;
Integer noOfChildren;
noOfChildren = rooms.length();
Boolean found = false;
Charstring name;
chat = NULL;
for (Integer i = 1;
 i<= noOfChildren;i=i+1) {
   cr = rooms[i];
   name = cr.chatName;
   if (name == roleId) {
   found = true;
   chat = cr.chatRef;
   }
}
if (found == false) {
    chat = new Chat;
    cr = new ChatRooms;
    cr.chatName = roleId;
    cr.chatRef = chat;
    rooms.append(cr);
    TheChats.append(chat);
}
```

Figure 27 Behavior of class RoleManager

5.2.3.3 Class Client

The behavior of *Client* is shown in Figure 28. The *Init* signal is used to initiate the class variables with values as name of the user (clientName), the id of the requested role (roleId) and the text the user has typed (startText). An alternative way to initiate the variables of this class is to make a constructor of the class with parameters. When the instance is created the new operator is called with actual parameters. The cross symbol circled with red at the end of the diagram, means that this instance is destructed.

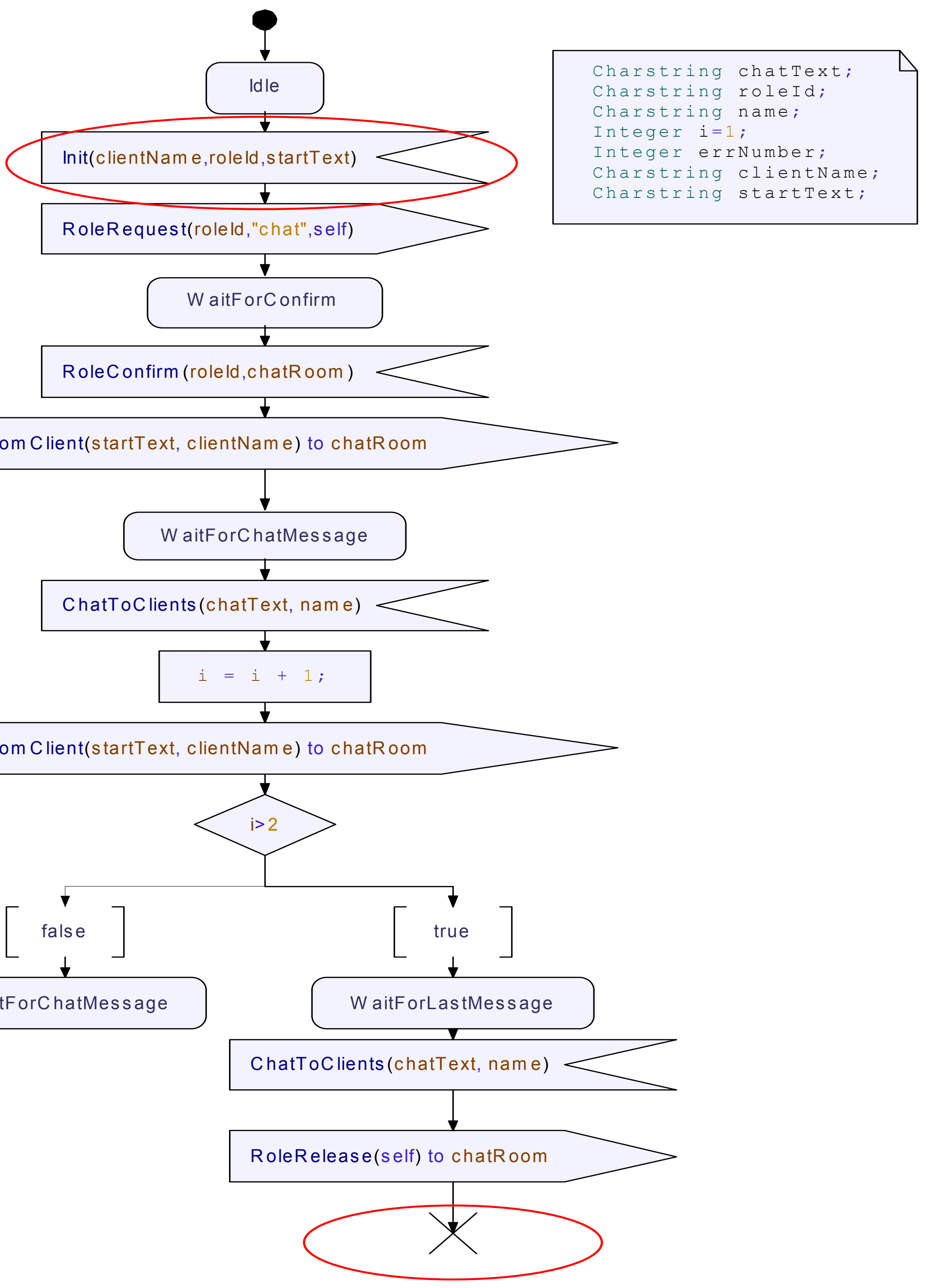

Idle
Init(clientName,roleId,startText)
RoleRequest(roleId,"chat",self)
WaitForConfirm
RoleConfirm(roleId,chatRoom)
ChatFromClient(startText, clientName) to chatRoom
WaitForChatMessage
ChatToClients(chatText, name)
i = i + 1;
ChatFromClient(startText, clientName) to chatRoom
i>2
false
true
WaitForChatMessage
WaitForLastMessage
ChatToClients(chatText, name)
RoleRelease(self) to chatRoom
Charstring chatText;
Charstring roleId;
Charstring name;
Integer i=1;
Integer errNumber;
Charstring clientName;
Charstring startText;

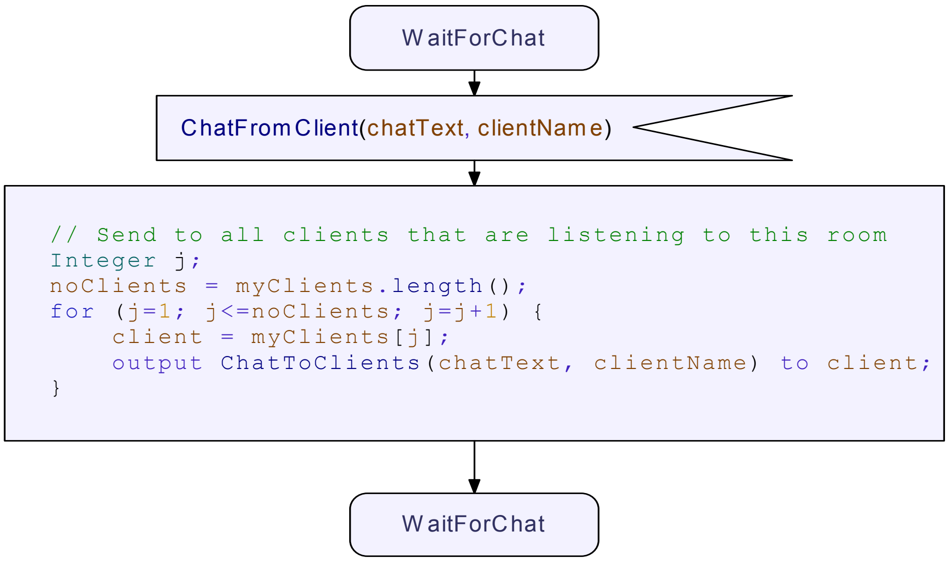
Figure 28 Behavior of class Client

5.2.3.4 Class Chat

The behavior diagram for the *Chat* in Figure 29, shows a transition where signals to all instances of *Client* are send. References to the clients are stored in the variable *myClients*. There is several ways to specify the receiving state machine:

- By reference - that is the way it is done her in this transition.
- By ports – where the port is specified to carry the actual signal.

- By connector – where the connector can carry the signal in the right direction.

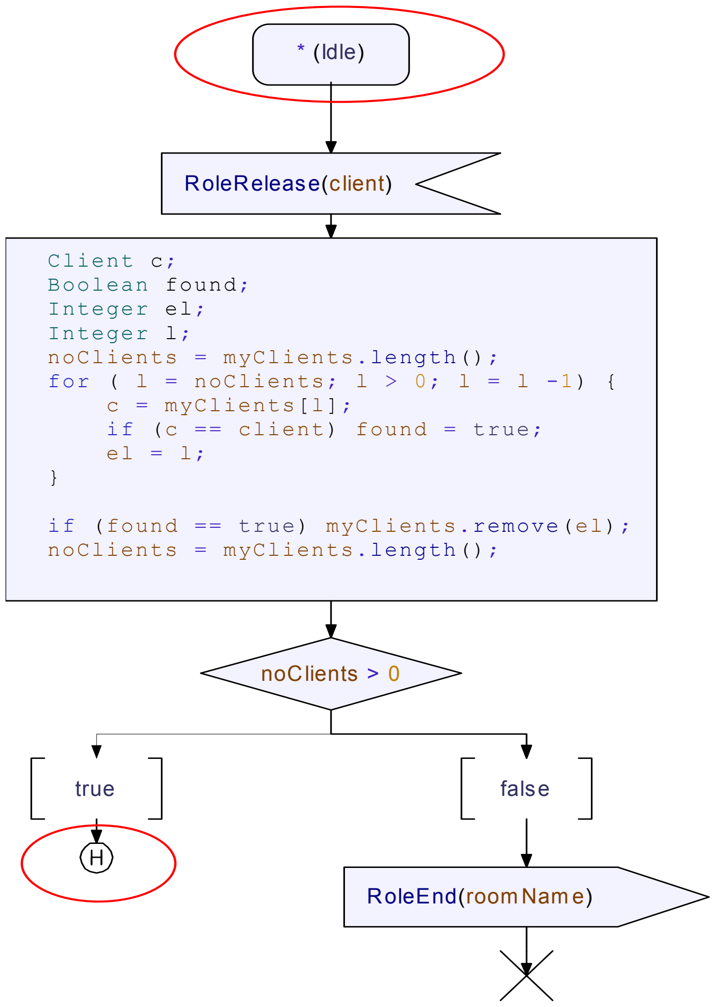


```
// Send to all clients that are listening to this room
Integer j;
noClients = myClients.length();
for (j=1; j<=noClients; j=j+1) {
    client = myClients[j];
    output ChatToClients(chatText, clientName) to client;
}
```

Figure 29 Behavior of class Chat

In Figure 30 behavior is specified for *aClient* to send a *RoleRelease* signal to *Chat*. *Chat* checks if more clients are associated with this chat room. If none, it sends a *PlayEnd* signal to the containing *RoleManager*.

The state symbol is named with a star and in parenthesis the state *Idle*. This is a shortcut for specifying that this state machine may receive the signal *RoleRelease* in all defined states except the *Idle* state. The history symbol at the end of the transition means that the next state is the same state as it received the signal. This mechanism is powerful for specifying of transitions that may happen in more than one state. It is also possible to specify a group of states in the state symbol.



```
Client c;
Boolean found;
Integer el;
Integer l;
noClients = myClients.length();
for ( l = noClients; l > 0; l = l -1) {
    c = myClients[l];
    if (c == client) found = true;
    el = l;
}

if (found == true) myClients.remove(el);
noClients = myClients.length();
```

Figure 30 RoleRelease in class Chat

5.2.4 Validation of the model

This model may be simulated for validation of functionality of the model conforms to the specification of chat. The sequence diagram in Figure 31 shows the initial startup when Geir:Users sends two *StartClient* signals to chat*Manager*. The *chatManager* then creates two *Clients* which each makes a *RoleRequest* to *RoleManager* for the same chat room. The rest of the trace is shown in the appendix B where it can be verified that after three rounds with chatting the *Clients* and the *Chat* is deleted.

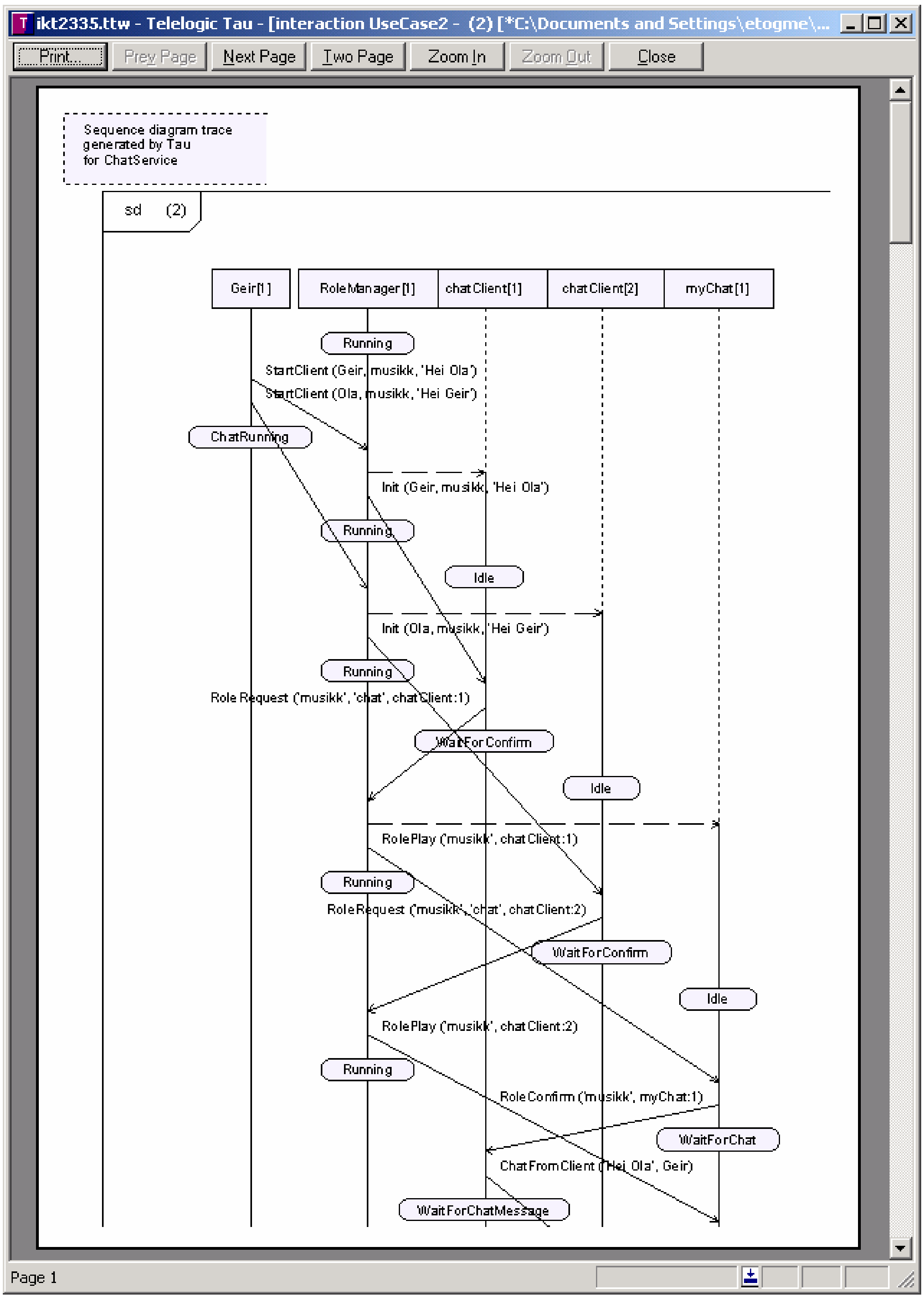
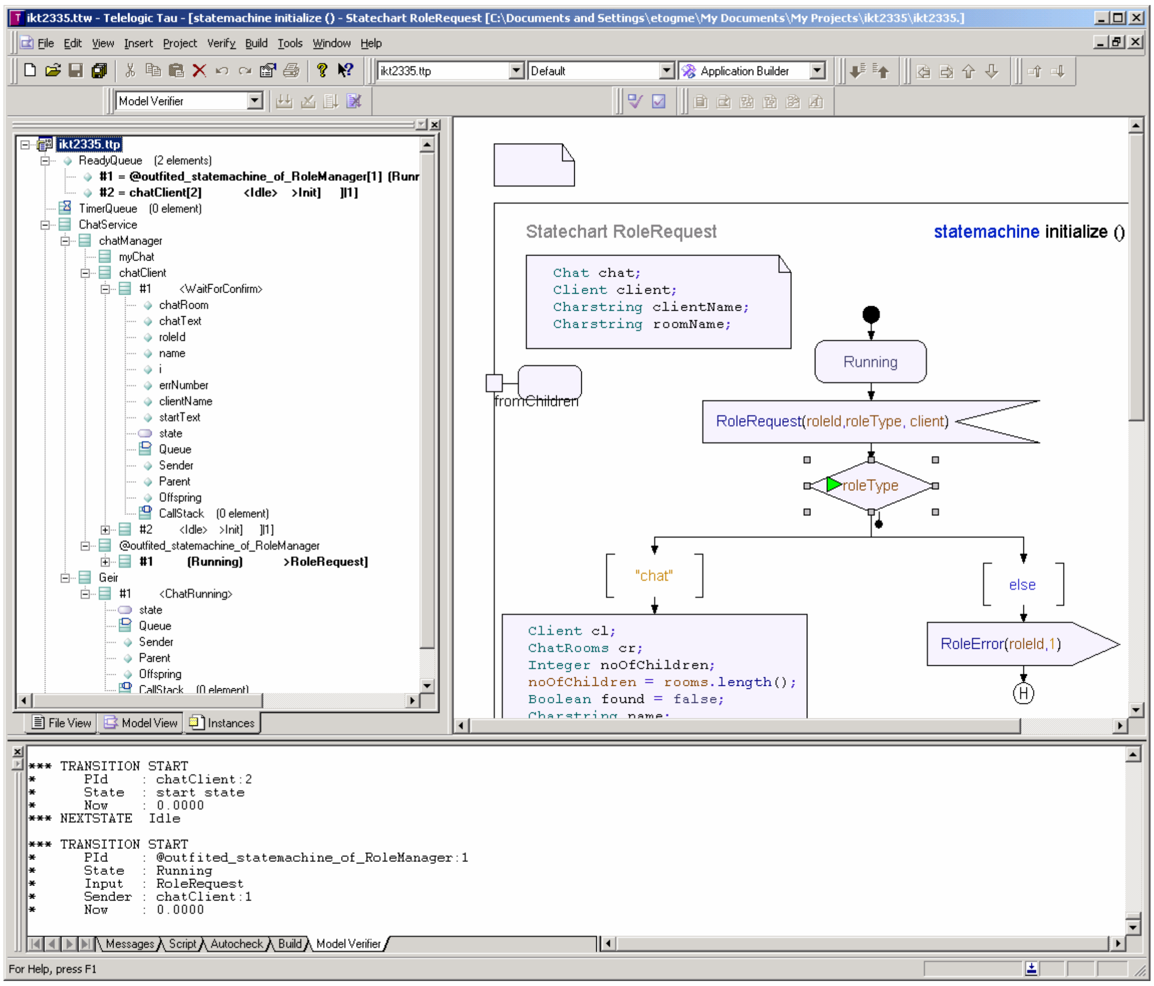

ikt2335.ttw - Telelogic Tau - [interaction UseCase2 - (2) [*C:\Documents and Settings\etogme\...
Print...
Prev Page
Next Page
Two Page
Zoom In
Zoom Out
Close
Sequence diagram trace generated by Tau for ChatService
sd (2)
Geir[1]
RoleManager[1]
chatClient[1]
chatClient[2]
myChat[1]
Running
StartClient (Geir, musikk, 'Hei Ola')
StartClient (Ola, musikk, 'Hei Geir')
ChatRunning
Init (Geir, musikk, 'Hei Ola')
Running
Idle
Init (Ola, musikk, 'Hei Geir')
Running
RoleRequest ('musikk', 'chat', chatClient:1)
WaitForConfirm
Idle
RolePlay ('musikk', chatClient:1)
Running
RoleRequest ('musikk', 'chat', chatClient:2)
WaitForConfirm
Idle
RolePlay ('musikk', chatClient:2)
Running
RoleConfirm ('musikk', myChat:1)
WaitForChat
ChatFromClient ('Hei Ola', Geir)
WaitForChatMessage
Page 1

Figure 31 Trace of initiating of chatting

It is possible to test a UML model with the validation tool in Tau in the same way as ordinary Integrated Development Environments have. A typical window is shown in Figure 32. It is possible to make single steps, trace to breakpoints, inspect values of variables, states, status of state machine etc. A textual output of the execution is also available as shown in Figure 33.


ikt2335.ttw - Telelogic Tau - [statemachine initialize () - Statechart RoleRequest [C:\Documents and Settings\etogme\My Documents\My Projects\ikt2335\ikt2335.]
File Edit View Insert Project Verify Build Tools Window Help
ikt2335.ttp
Default
Application Builder
Model Verifier
File View
Model View
Instances
Messages
Script
Autocheck
Build
Model Verifier
For Help, press F1
Statechart RoleRequest
statemachine initialize ()
Running
fromChildren
RoleRequest(roleId,roleType, client)
roleType
"chat"
else
RoleError(roleId,1)
H

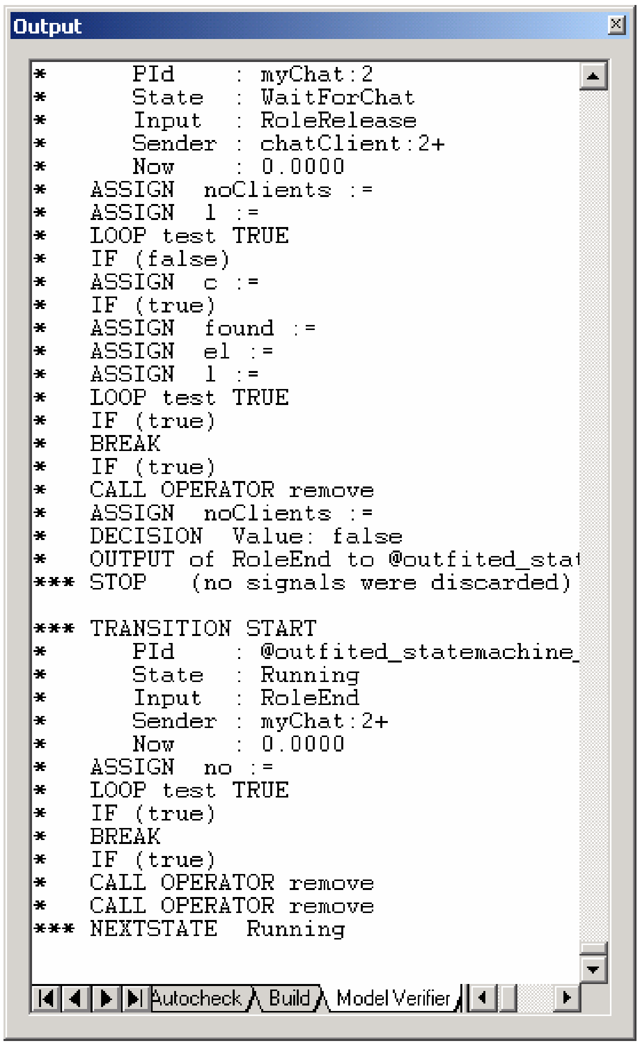
```
ikt2335.ttp
  ReadyQueue    (2 elements)
    #1 = @outfited_statemachine_of_RoleManager[1] (Runr
    #2 = chatClient[2]        <Idle>    >Init]    ]|1]
  TimerQueue    (0 element)
  ChatService
    chatManager
      myChat
      chatClient
        #1        <WaitForConfirm>
          chatRoom
          chatText
          roleId
          name
          i
          errNumber
          clientName
          startText
          state
          Queue
          Sender
          Parent
          Offspring
          CallStack    (0 element)
        #2        <Idle>    >Init]    ]|1]
      @outfited_statemachine_of_RoleManager
        #1        (Running)        >RoleRequest]
    Geir
      #1        <ChatRunning>
        state
        Queue
        Sender
        Parent
        Offspring
        CallStack    (0 element)
```

```
Chat chat;
Client client;
Charstring clientName;
Charstring roomName;
```

```
Client cl;
ChatRooms cr;
Integer noOfChildren;
noOfChildren = rooms.length();
Boolean found = false;
Charstring name;
```

```
*** TRANSITION START
*     PId    : chatClient:2
*     State  : start state
*     Now    : 0.0000
*** NEXTSTATE  Idle

*** TRANSITION START
*     PId    : @outfited_statemachine_of_RoleManager:1
*     State  : Running
*     Input  : RoleRequest
*     Sender : chatClient:1
*     Now    : 0.0000
```

Figure 32 Debugging in the Model Validator



```
*       PId     : myChat:2
*       State   : WaitForChat
*       Input   : RoleRelease
*       Sender  : chatClient:2+
*       Now     : 0.0000
*    ASSIGN  noClients :=
*    ASSIGN  l :=
*    LOOP test TRUE
*    IF (false)
*    ASSIGN  c :=
*    IF (true)
*    ASSIGN  found :=
*    ASSIGN  el :=
*    ASSIGN  l :=
*    LOOP test TRUE
*    IF (true)
*    BREAK
*    IF (true)
*    CALL OPERATOR remove
*    ASSIGN  noClients :=
*    DECISION  Value: false
*    OUTPUT of RoleEnd to @outfited_stat
***  STOP   (no signals were discarded)

***  TRANSITION START
*       PId     : @outfited_statemachine_
*       State   : Running
*       Input   : RoleEnd
*       Sender  : myChat:2+
*       Now     : 0.0000
*    ASSIGN  no :=
*    LOOP test TRUE
*    IF (true)
*    BREAK
*    IF (true)
*    CALL OPERATOR remove
*    CALL OPERATOR remove
***  NEXTSTATE  Running
```

Figure 33 Textual trace of execution

6 Experiences

The first attempt to make a UML model consisting of ActorFrame classes and to make a chat application by extending ActorFrame classes, failed. The reason for that is partly

- Limitations in the tool
- Limitations of the proposed UML2.0 standard
- Lack of knowledge of both the language and the tool

The first two reasons are commented below. The intention of study is to look at concepts in UML2.0 and not eventually additional concepts that the tool supports.

6.1 UML2.0 support in the tool

As mentions in chapter 4, Tau is partly built on a earlier toolset that supported mainly the SDL language. SDL has defined many of the language concepts that UML has lacked. SDL has in fact heavily influenced the specification of UML 2.0 especially the architectural and structural concepts (ports, parts, interactions, behavior specialization).

The tool supports also some concepts that are not part of UML 2.0 as e.g. a reference data type and predefined operations to get sender of the signal, parent and offsprings of a state machine. Especially the lack of a data type in UML to keep references to state machines that is not bound to a special class, is a serious limitation when it comes to make general classes of frameworks. For instance in *ActorFrame* protocol the sender of the *RoleRequest* signal have to be included in signal as a parameter. In UML2.0 this is not possible because the parameter containing the reference to sender of the signal has to be of the sender class. The data type *Pid* in SDL can have references to state machine of different classes.

But the tool did not support important concepts that are part of UML as (sjekk dette)

- Specialization / generalization of signals that caused that it is not possible to specify a general signal that all ActorFrame signal inherits. This led to the problem of not be able to make a general Actor class with ports that conveys signal of the generic type. The problem with the port could have been solved if generalization of ports had been supported.
- Polymorphism is an important technique when it comes to make general frameworks that may be reused by extended and redefining of the general classes of the framework. This is used in implementation of ActorFrame to obtain the general management behavior that all subtypes of *Actor* inherit. The Tau tool does not support polymorphism. This reduces the possibility to make general frameworks.

6.2 UML 2.0 standard

The changes in UML2.0 from previous versions have increased the power and expressiveness of the language significantly. It has many of the concepts needed to model complex systems. Use of architectural diagrams to specify internal structure of classes seems to be a better way to describe hierarchies than using composite associations. The port concept with internal operations and behavior makes UML2.0 to a better language for describing interfaces of classes and components.

Specialization of behavior was not used in the chat application. The reason for that is that there were not many similarities between the different classes when it was not possible to model a general ActorFrame behavior of *Actor*. But inheritance of behavior is achieved by normal generalization as shown in Figure 34. The behavior of the Actor class is shown in Figure 35.

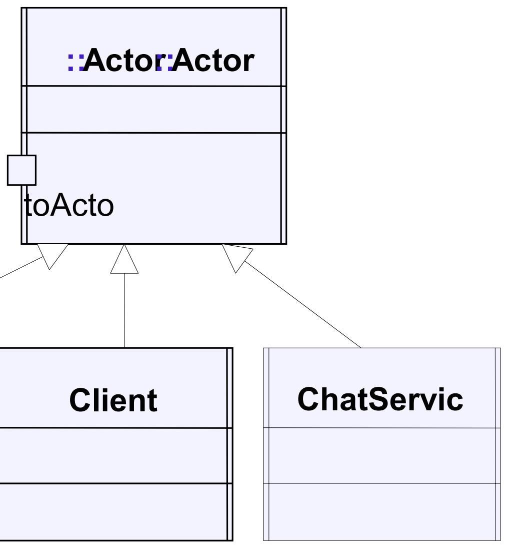


Figure 34 Class inheritance

The subclasses may extend behavior of *Actor* Figure 35 as shown in Figure 36. The *RoleBaseIdle* state is extended with new transitions and the state machine is extended with new states. So the *Client* contains the behavior of *Actor* extended with behavior specified in *Client*. This is obviously a more efficient way of structuring the model. All active classes that are sub classes of *Actor* inherit the behavior of *Actor* that is *ActorFrame* functionality as earlier presented in chapter 3.1.2.

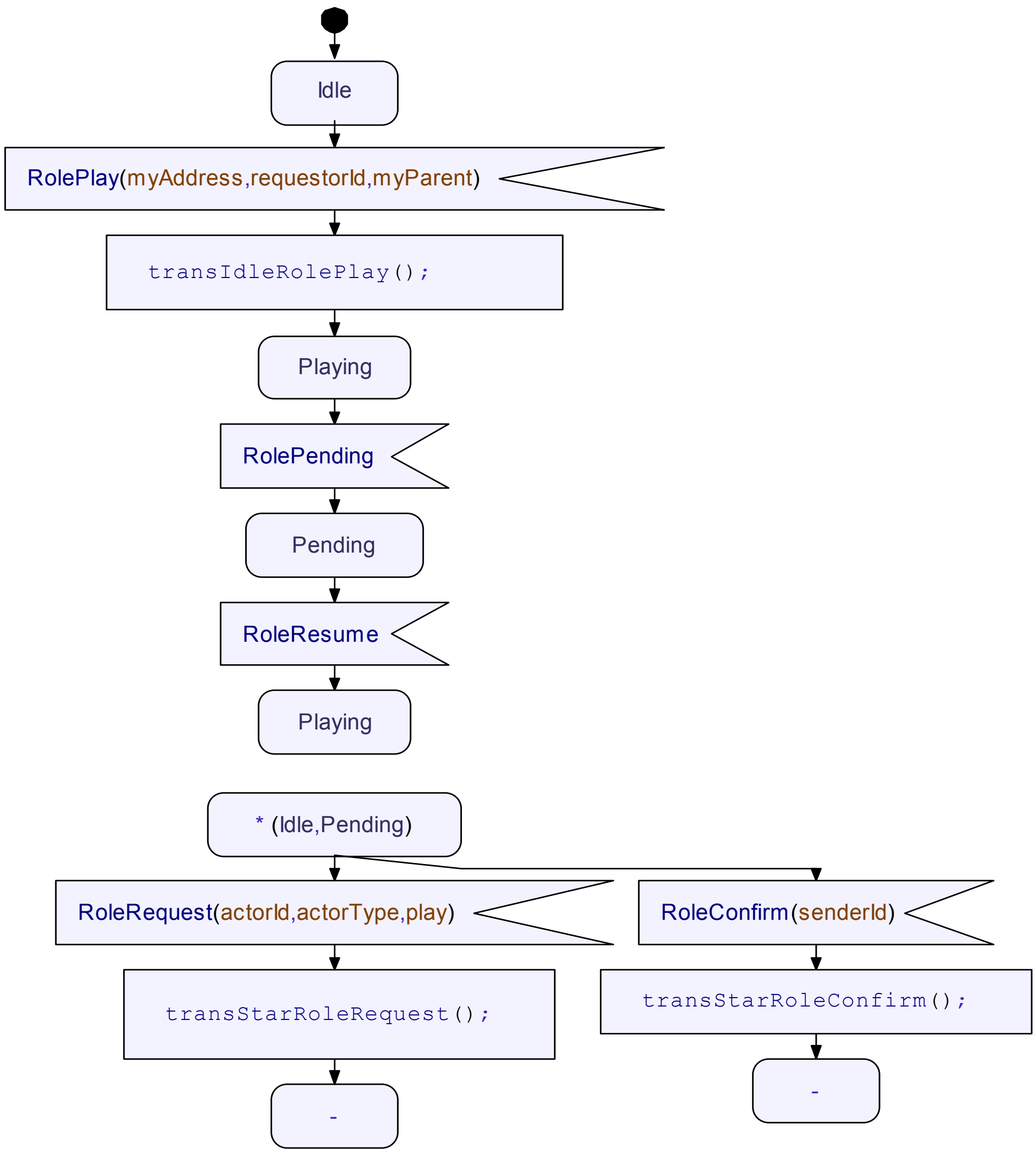
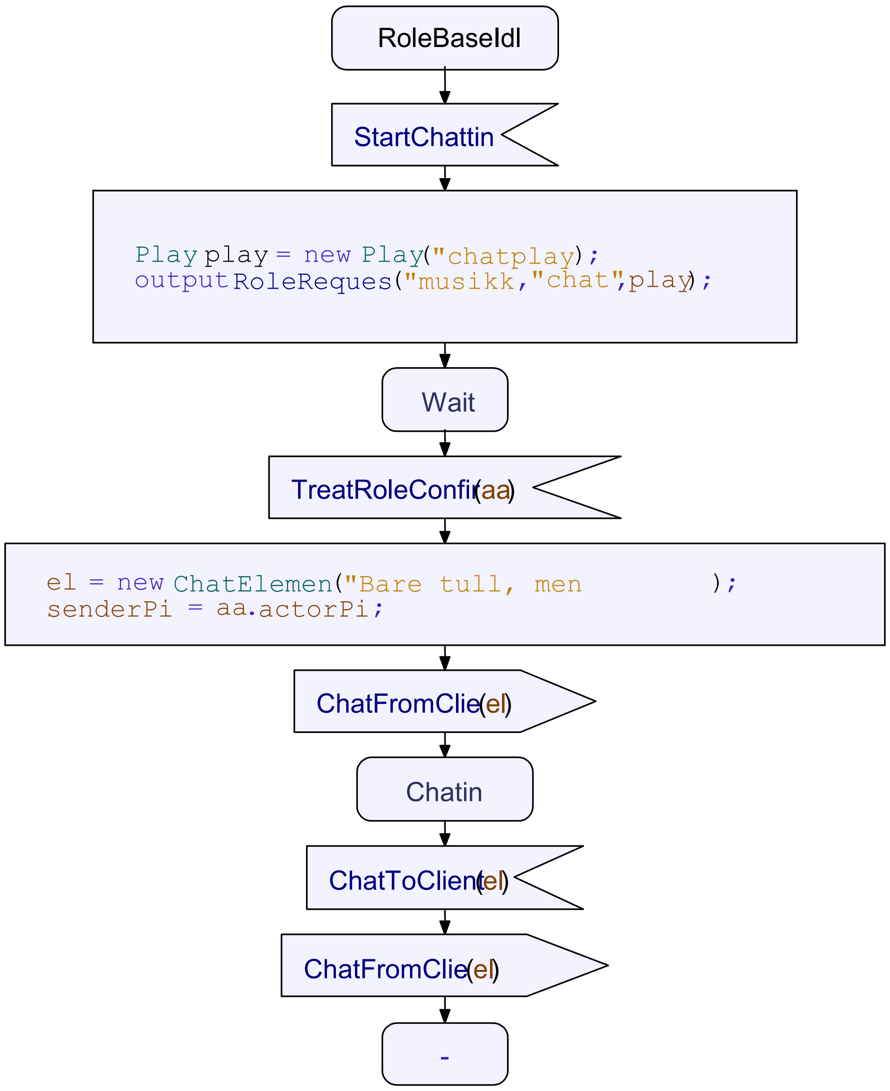


Figure 35 Behavior of class Actor



```
Play play = new Play("chatplay);
output RoleReques("musikk,"chat",play);
```

```
el = new ChatElemen("Bare tull, men           );
senderPi = aa.actorPi;
```

Figure 36 Behavior of subclass Client

Other problems in UML2.0 that still remains to be solved, are that received signals are not available after the trigging of the transition. This makes it more complicated to model for instance the functionality of a router. All signals have to be defined in the state machine, to enable a forwarding of the signal. If the signal has been available in the transition, it would have been possible to define a rather generic router that receives for instance Actor signals, inspect the generic part of the signal and then forwards the same signal. It is at moment not clear that if this is possible or not in UML.

6.3 Tool

The Tau tool from Telelogic was just released when the study started in September. There was no problem with the installation and it was easy to follow the tutorial to make the first model. The tool is very fast, has a good design and it is supported with many useful trace and debugging possibilities. It is possible to a make a UML model, analyze it and then do model verifying.

However did the study expose some weaknesses in the tool as

- Restrictions in the action languages in how to use the language. Generally it was most safe only to write one statement on each line.

- Features that was allowed to write in model, but that was not supported behind the scene, created some trouble. For instance it was allowed to make generalization hierarchies of signals, but it was not supported in model verification.
- Though the tool has an excellent help facility, the relevant content was difficult to find. There were to many hits and many of them were connected to the code generator that the user normally does not work with.
- Missing error explanations that made it difficult to find errors. For instance gave the tools many runtime errors of type “memory exception” without an explanation of where these errors where located. These errors also some times corrupted the application model and an earlier version had to be loaded which caused loss of data.

6.4 MDA approach

Although the book [1] was read as part of the study it did not give very much new to the concept of executing models. However has this study shown that it is possible to make executable models in UML, also where the new concepts in UML2.0 were used. In overall has this study followed the main steps in the book. But the book did not cover UML2.0, which has been the focus of this study.

7 Conclusions

This study has done an evaluation of the UML2.0 language as it was in September 2002. The language has been significantly improved by new concepts that supports modeling of large and complex real time systems as most of telecom systems are. Examples of improvements are better concepts for architecture of classes and structuring of behavior.

This study has used many of the new concepts in modeling the chat application. A model of ActorFrame was partly made, but the study got problems when it tried to model ActorFrame using the same advanced concepts that were used in the Java implementation. The reason form this was partly that the tool did not support all concepts in UML2.0. An example was that the tool did not support generalization / specialization of signals and polymorphism. The study has also shown that UML needs a better support for referencing to state machines as a data type that could contain references to state machines.

However the study also did show that it was possible to make executable UML models with behavior specification and that the models may be verified through simulation.

Earlier experiences with other specification languages have however shown that it is complicated to make complete models of complex systems that are so complete that it is possible to be verified, analyzed, and code generation to target platforms. But this is for sure possible, but it will take some time to the enough experiences with both the language and the tools to be in large scale able to achieve that.

The future may be bright but have patiance!

8 References

1. Stephen J. Mellor, Marc J. Balcer: *Executable UML – Model-Driven Architecture*: Addison-Wesley

2. *Unified Modeling Language: Superstructure*: version 2; Updated submission to OMG RFP ad/00-09-01; http://cgi.omg.org/cgi-bin/doc?ad/02-09-02

3. *Unified Modeling Language: Infrastructure*: version 2: Updated submission to OMG RFP ad/00-09-01; http://cgi.omg.org/cgi-bin/doc?ad/02-09-01

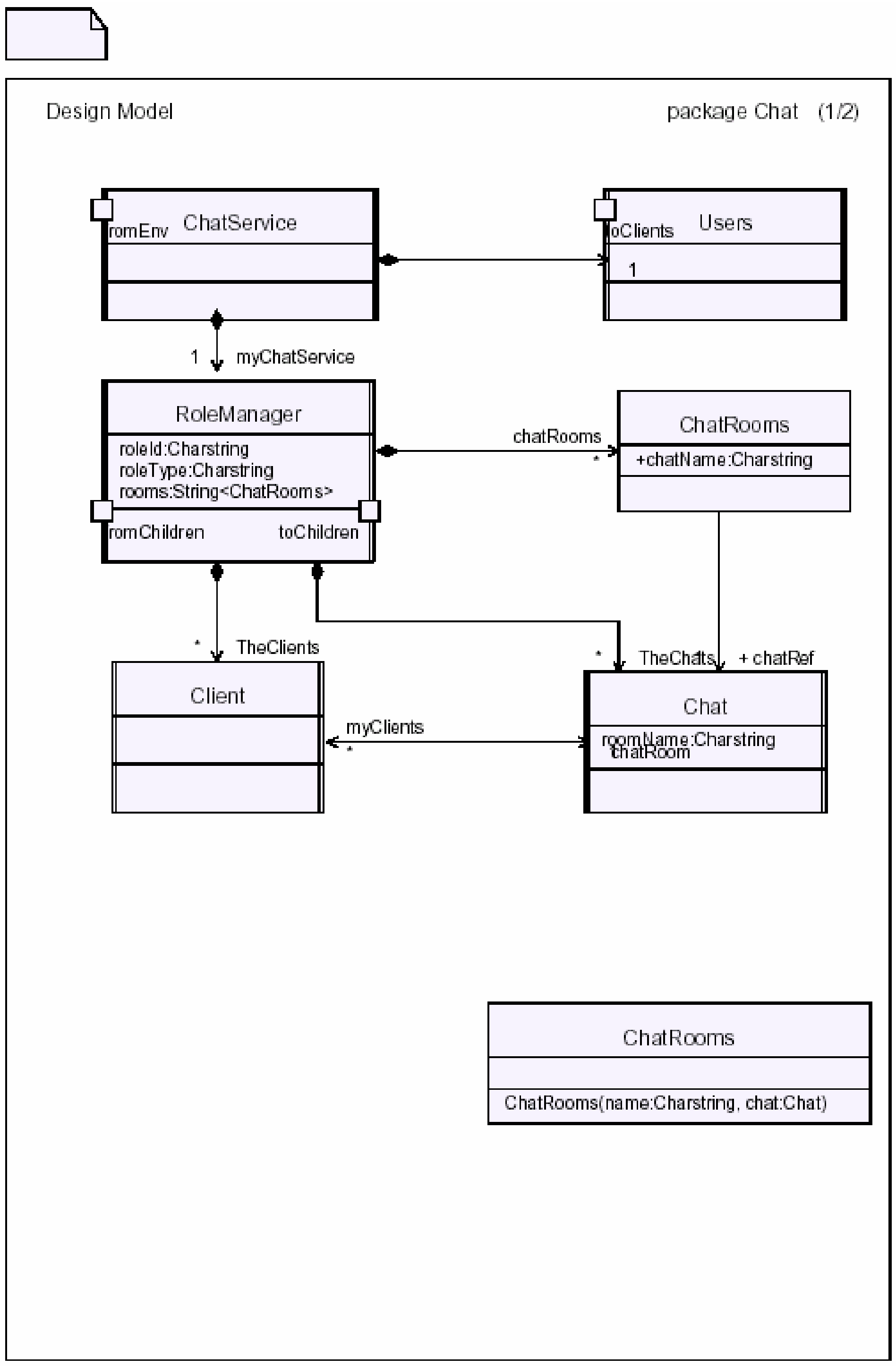
4. Rolv Bræk, Knut Eilif Husa, Geir Melby: *SeriviceFrame*; Whitepaper; Ericsson

5. SDL, Z-100 2000, ITU

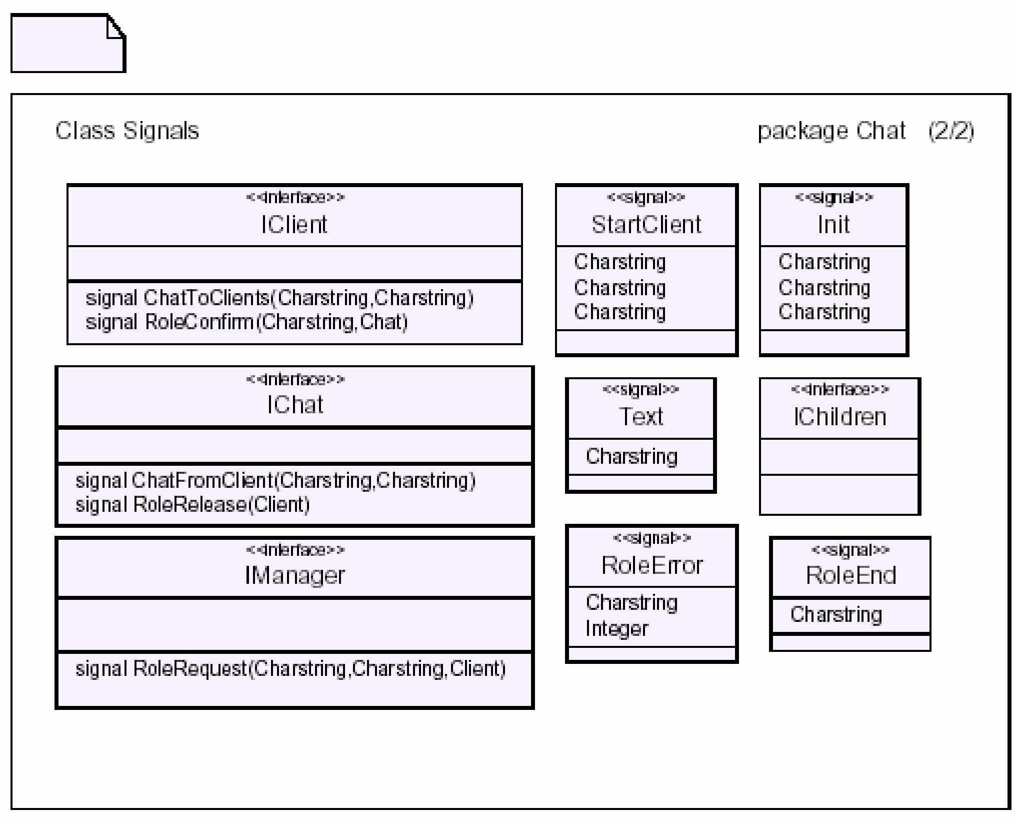
6. MSC, Z-120 2000, ITU

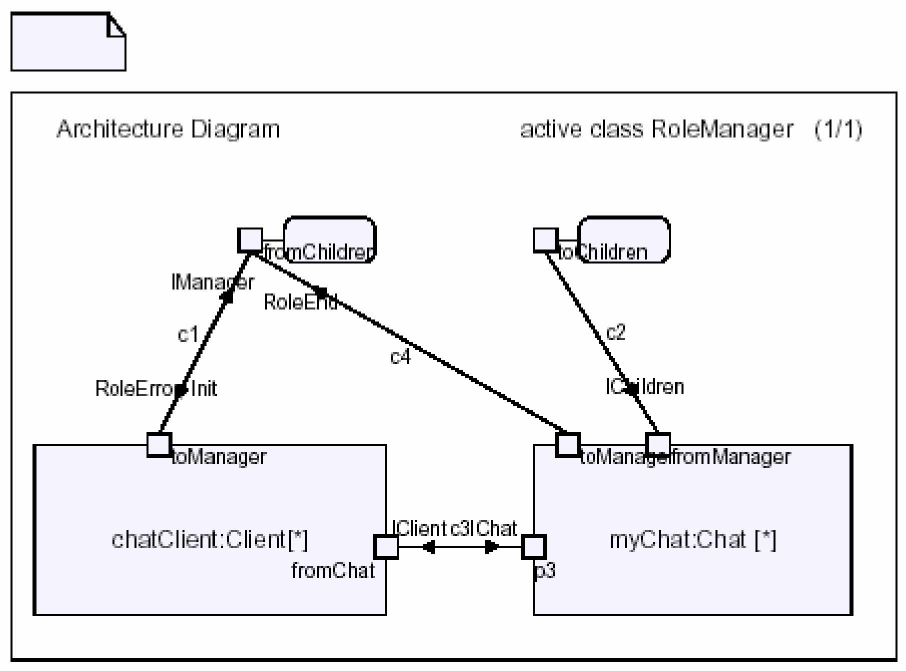
Appendix A – Chat Application model

8.1 Class model

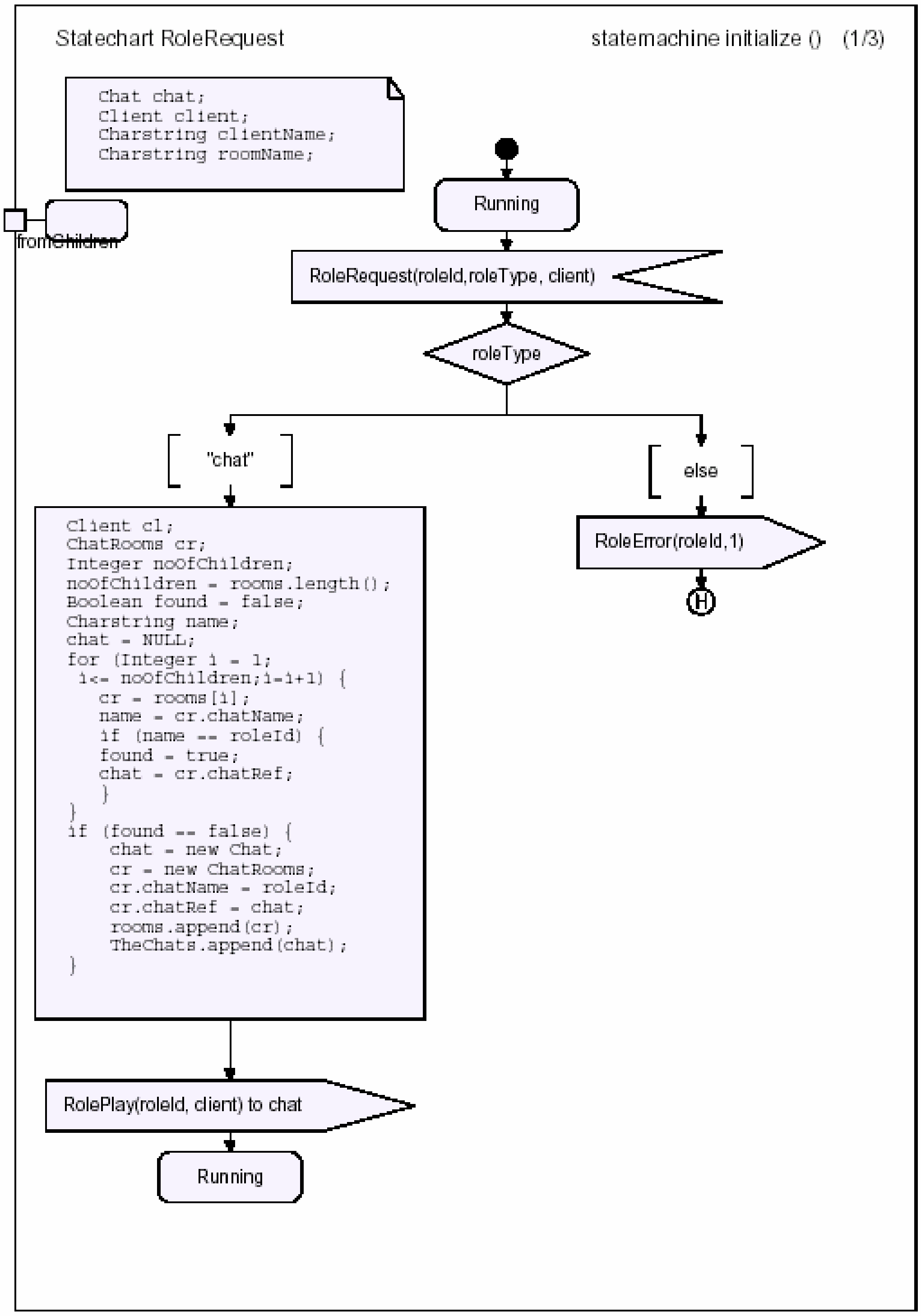


8.2 Signals definition





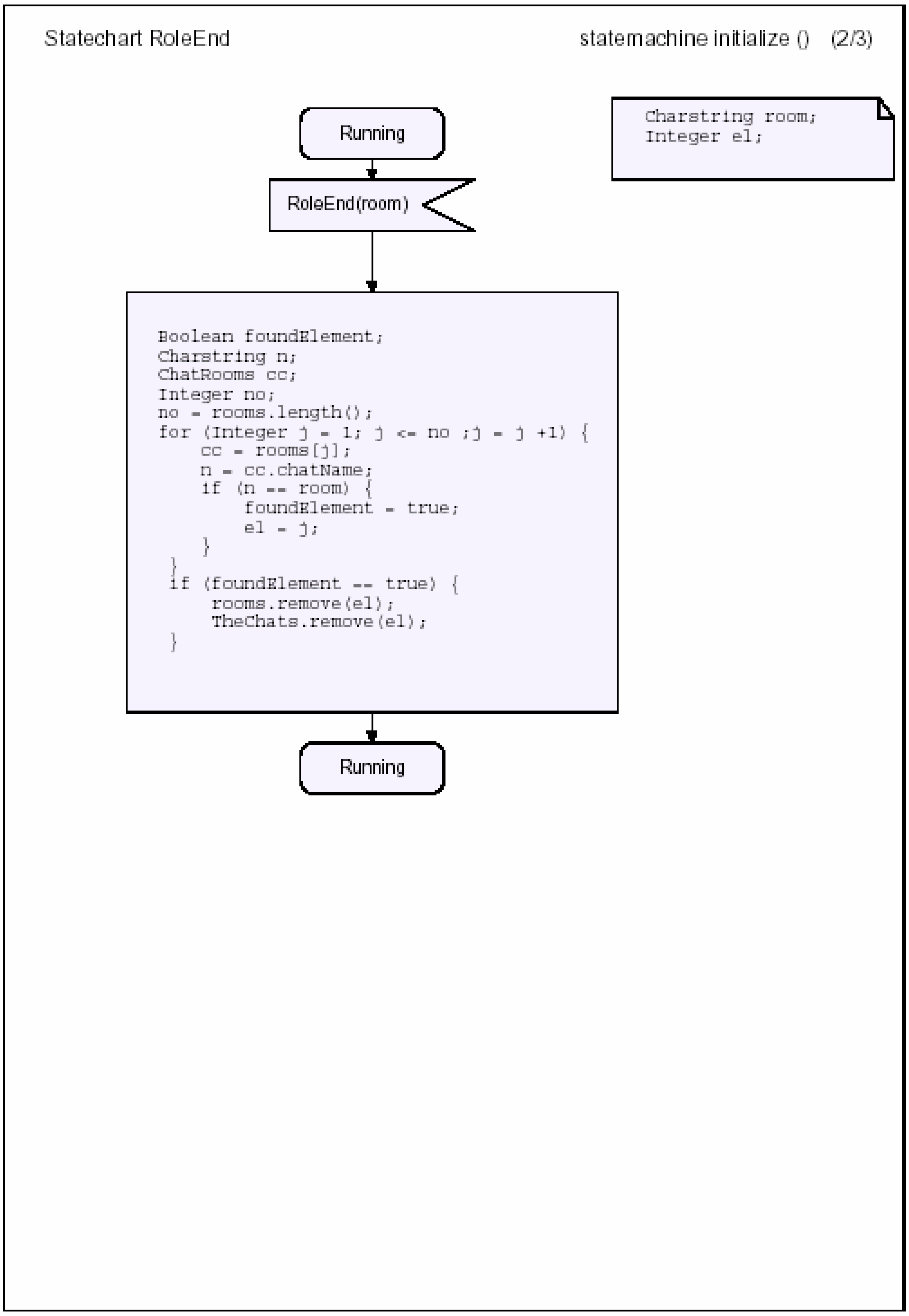
8.3 RoleManager - RoleRequest


Statechart RoleRequest
statemachine initialize () (1/3)
fromChildren
Running
RoleRequest(roleId,roleType, client)
roleType
"chat"
else
RoleError(roleId,1)
H
RolePlay(roleId, client) to chat
Running

```
Chat chat;
Client client;
Charstring clientName;
Charstring roomName;
```

```
Client cl;
ChatRooms cr;
Integer noOfChildren;
noOfChildren = rooms.length();
Boolean found = false;
Charstring name;
chat = NULL;
for (Integer i = 1;
 i<= noOfChildren;i=i+1) {
   cr = rooms[i];
   name = cr.chatName;
   if (name == roleId) {
   found = true;
   chat = cr.chatRef;
   }
}
if (found == false) {
    chat = new Chat;
    cr = new ChatRooms;
    cr.chatName = roleId;
    cr.chatRef = chat;
    rooms.append(cr);
    TheChats.append(chat);
}
```

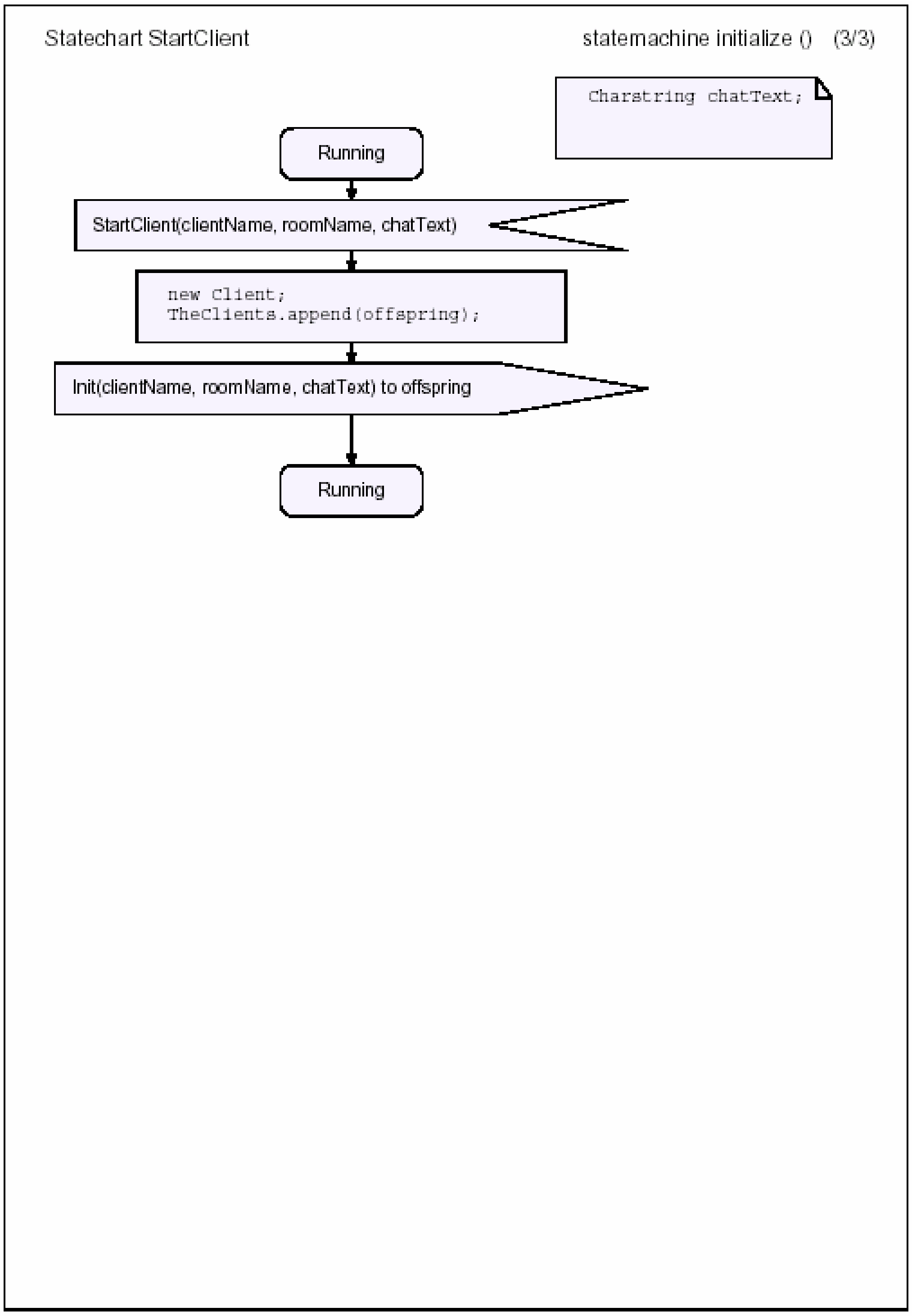
8.4 RoleManager - RoleEnd



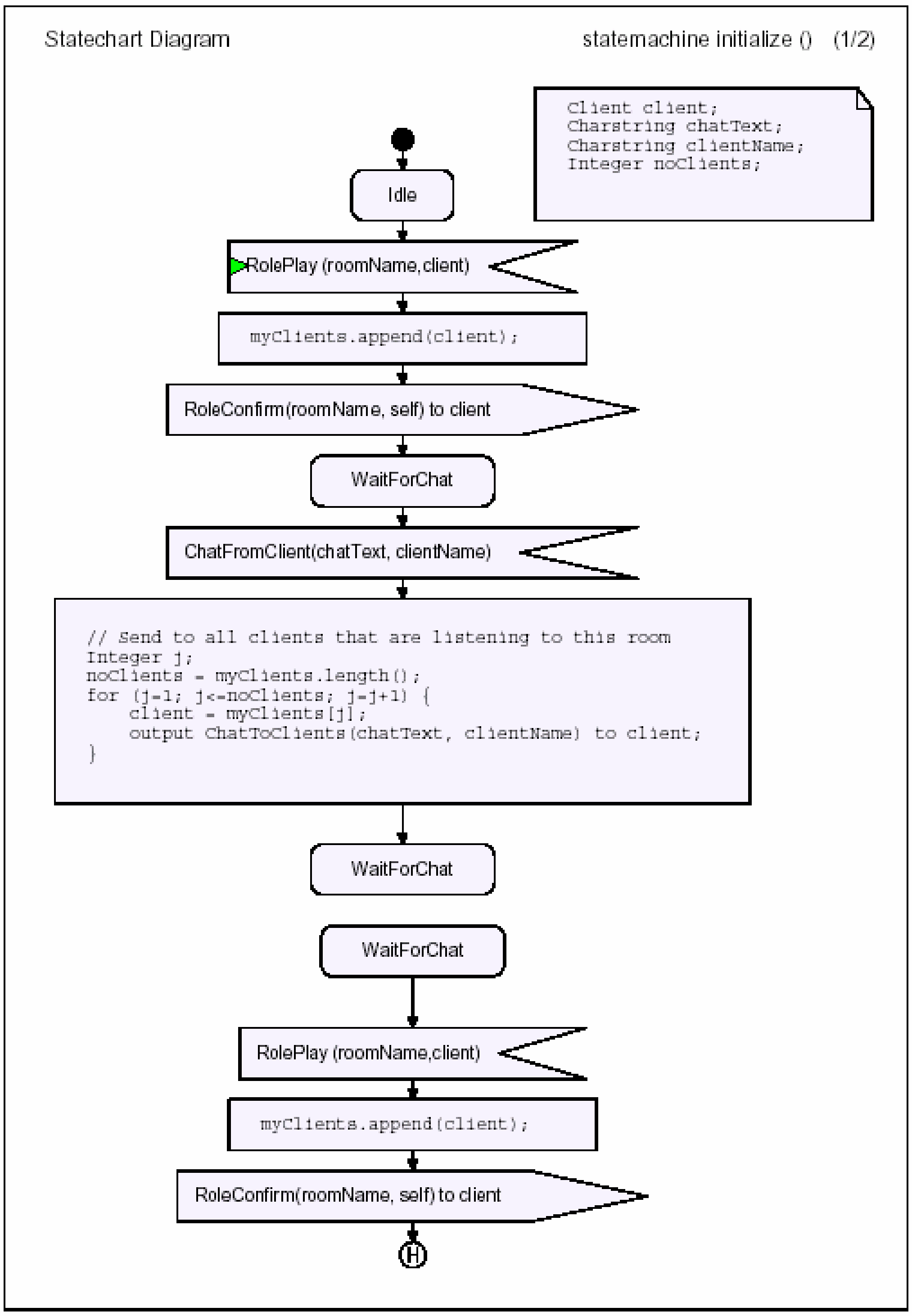
```
Charstring room;
Integer el;
```

```
Boolean foundElement;
Charstring n;
ChatRooms cc;
Integer no;
no = rooms.length();
for (Integer j = 1; j <= no ;j = j +1) {
    cc = rooms[j];
    n = cc.chatName;
    if (n == room) {
        foundElement = true;
        el = j;
    }
 }
 if (foundElement == true) {
     rooms.remove(el);
     TheChats.remove(el);
 }
```

8.5 Client


Statechart StartClient
statemachine initialize () (3/3)
Charstring chatText;
Running
StartClient(clientName, roomName, chatText)
new Client;
TheClients.append(offspring);
Init(clientName, roomName, chatText) to offspring
Running

8.6 Class Chat - RolePlay


Statechart Diagram
statemachine initialize () (1/2)
Idle
RolePlay (roomName,client)
RoleConfirm(roomName, self) to client
WaitForChat
ChatFromClient(chatText, clientName)
WaitForChat
WaitForChat
RolePlay (roomName,client)
RoleConfirm(roomName, self) to client
H

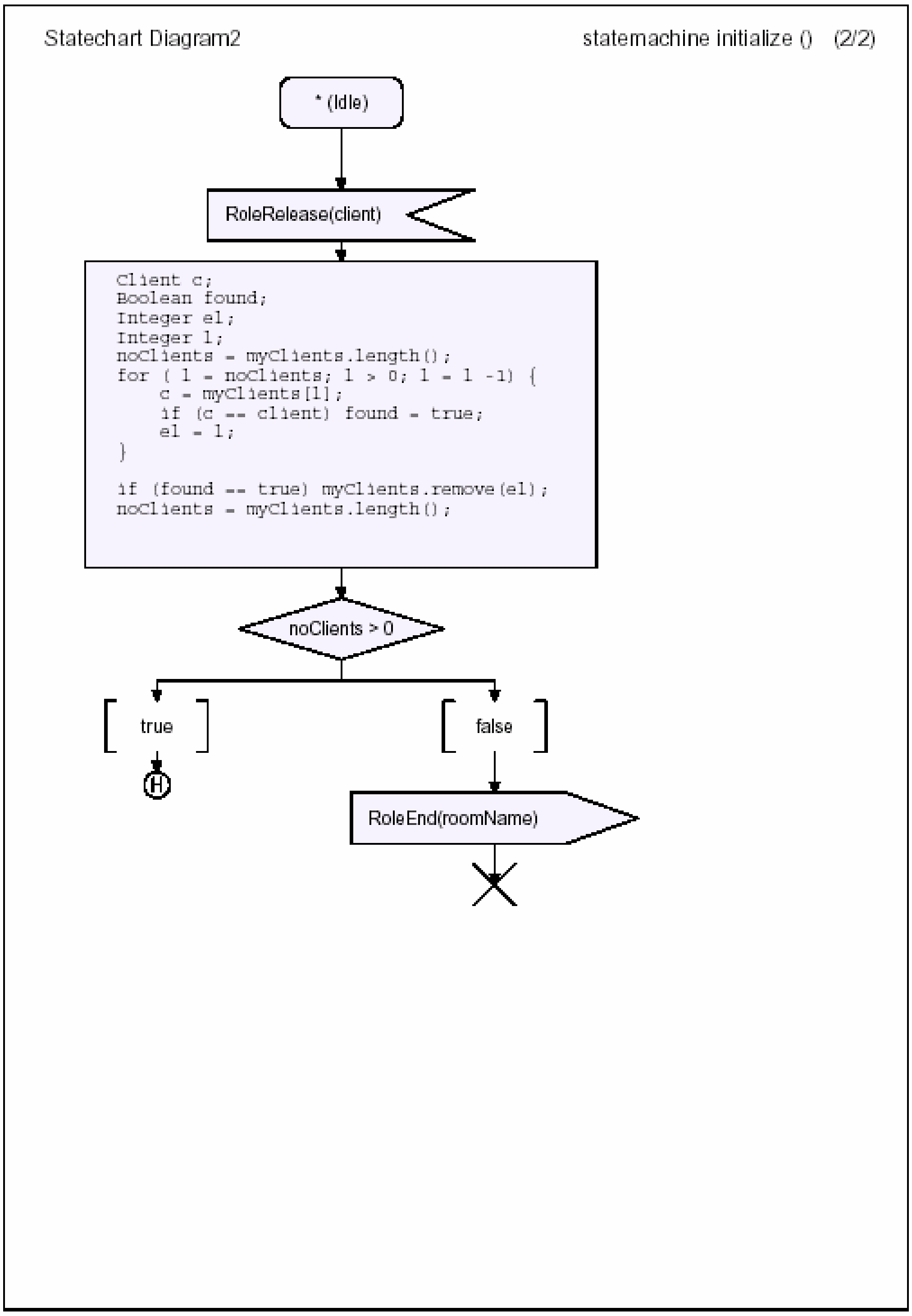
```
Client client;
Charstring chatText;
Charstring clientName;
Integer noClients;
```

```
myClients.append(client);
```

```
// Send to all clients that are listening to this room
Integer j;
noClients = myClients.length();
for (j=1; j<=noClients; j=j+1) {
    client = myClients[j];
    output ChatToClients(chatText, clientName) to client;
}
```

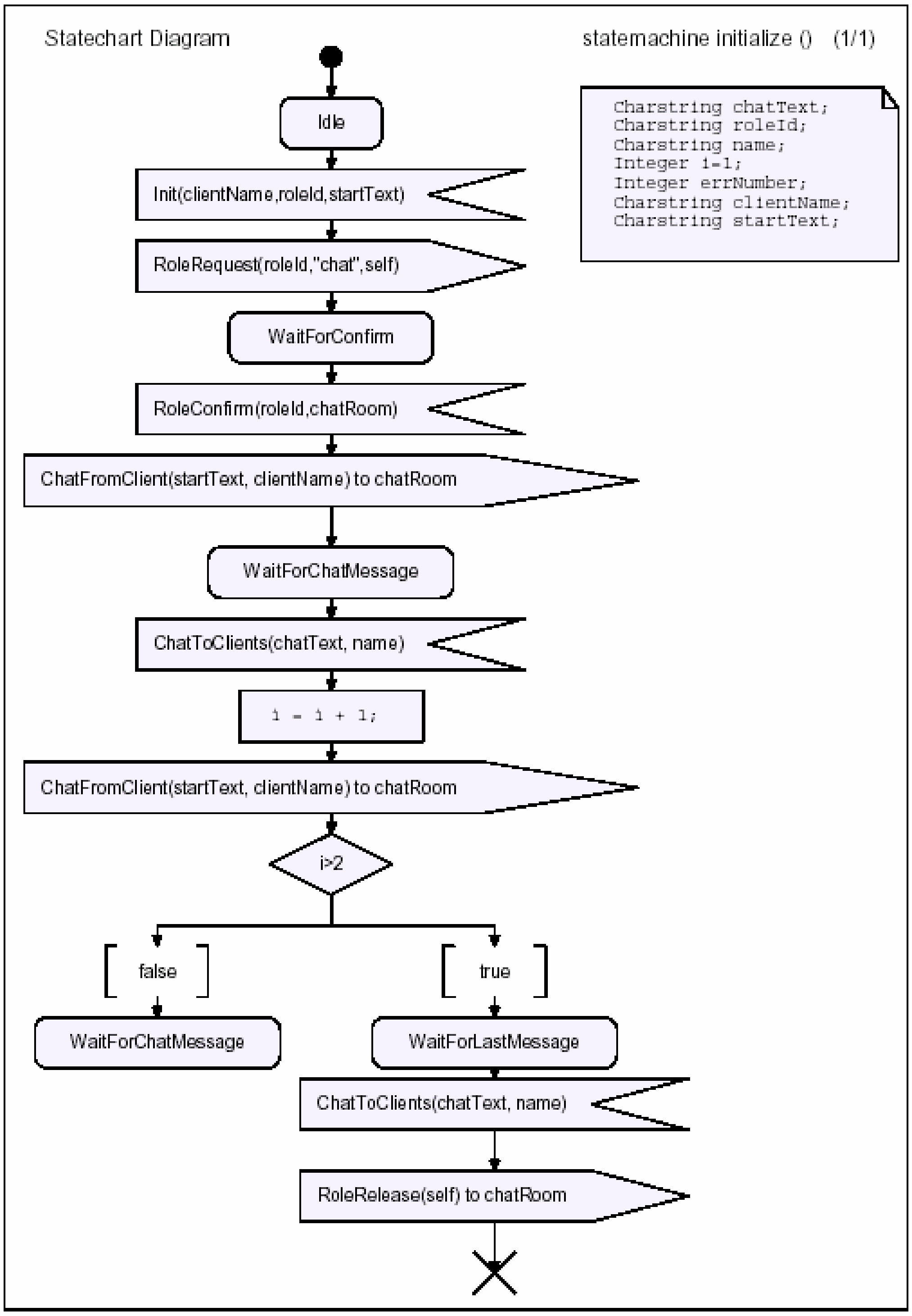
```
myClients.append(client);
```

8.7 Class Chat - RoleRelease



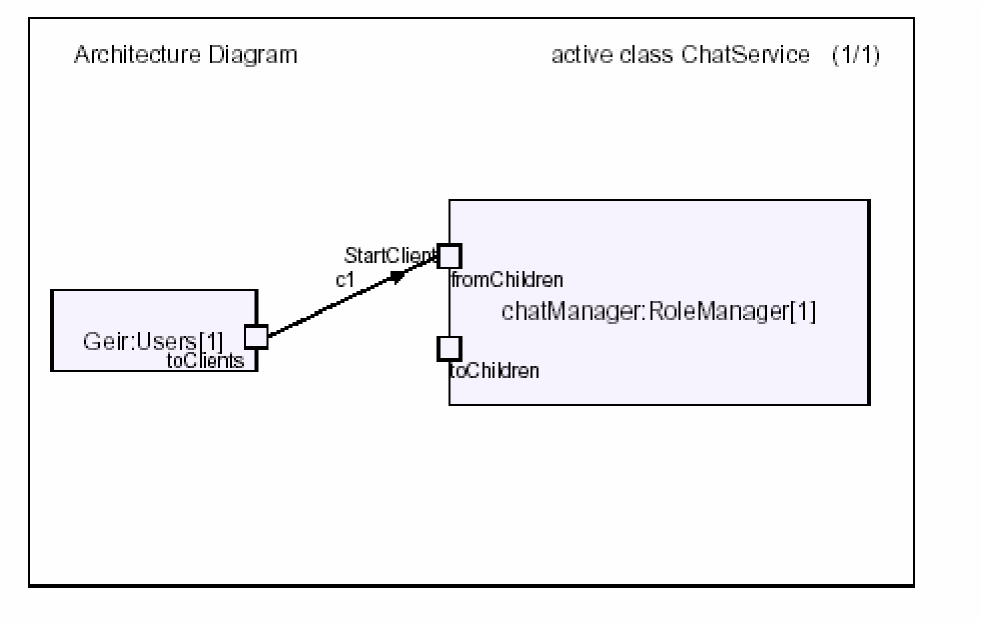
```
Client c;
Boolean found;
Integer el;
Integer l;
noClients = myClients.length();
for ( l = noClients; l > 0; l = l -1) {
    c = myClients[l];
    if (c == client) found = true;
    el = l;
}

if (found == true) myClients.remove(el);
noClients = myClients.length();
```

8.8 Class Client

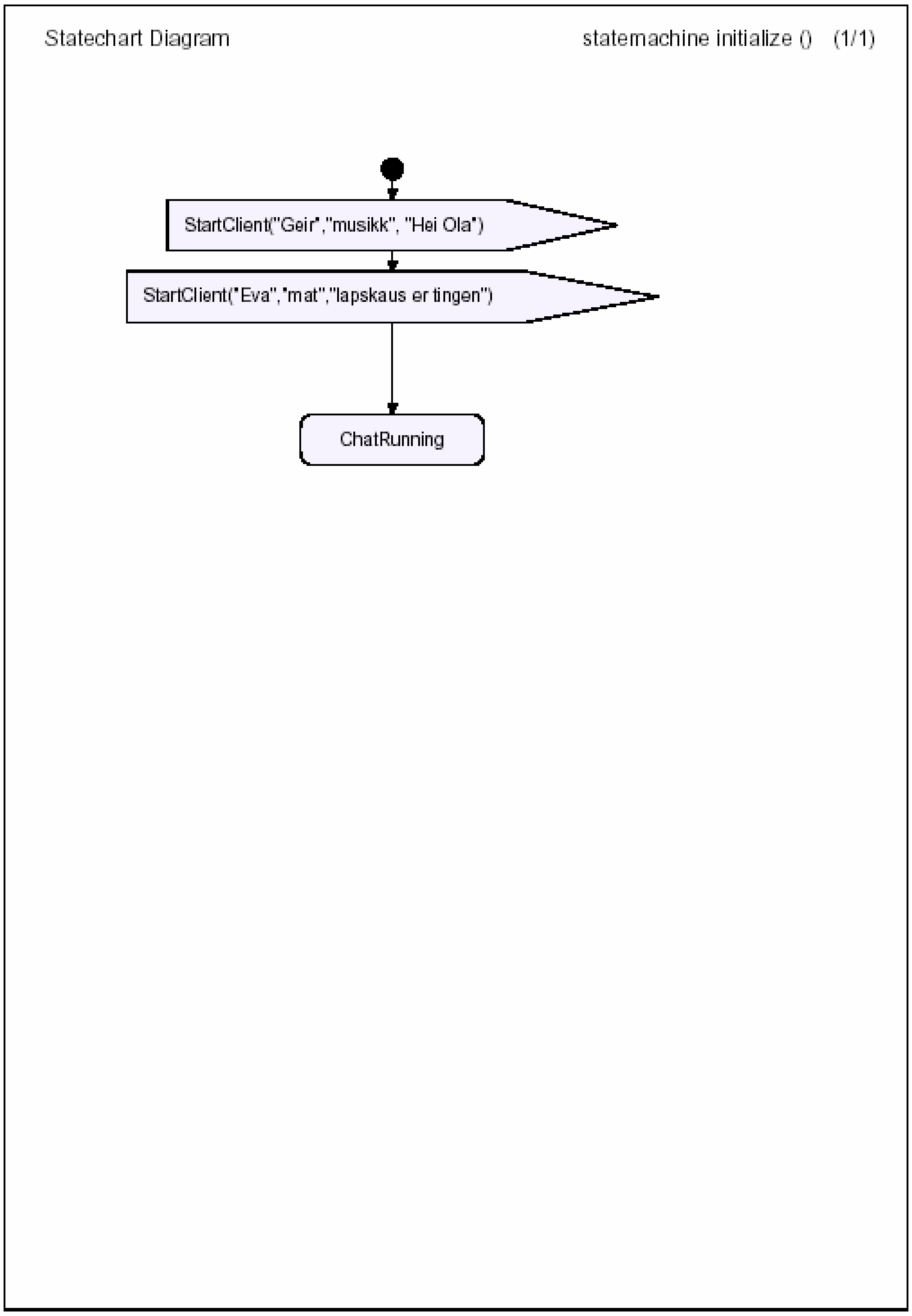

Statechart Diagram
statemachine initialize () (1/1)
Idle
Init(clientName,roleId,startText)
RoleRequest(roleId,"chat",self)
WaitForConfirm
RoleConfirm(roleId,chatRoom)
ChatFromClient(startText, clientName) to chatRoom
WaitForChatMessage
ChatToClients(chatText, name)
i = i + 1;
ChatFromClient(startText, clientName) to chatRoom
i>2
false
true
WaitForChatMessage
WaitForLastMessage
ChatToClients(chatText, name)
RoleRelease(self) to chatRoom

```
Charstring chatText;
Charstring roleId;
Charstring name;
Integer i=1;
Integer errNumber;
Charstring clientName;
Charstring startText;
```

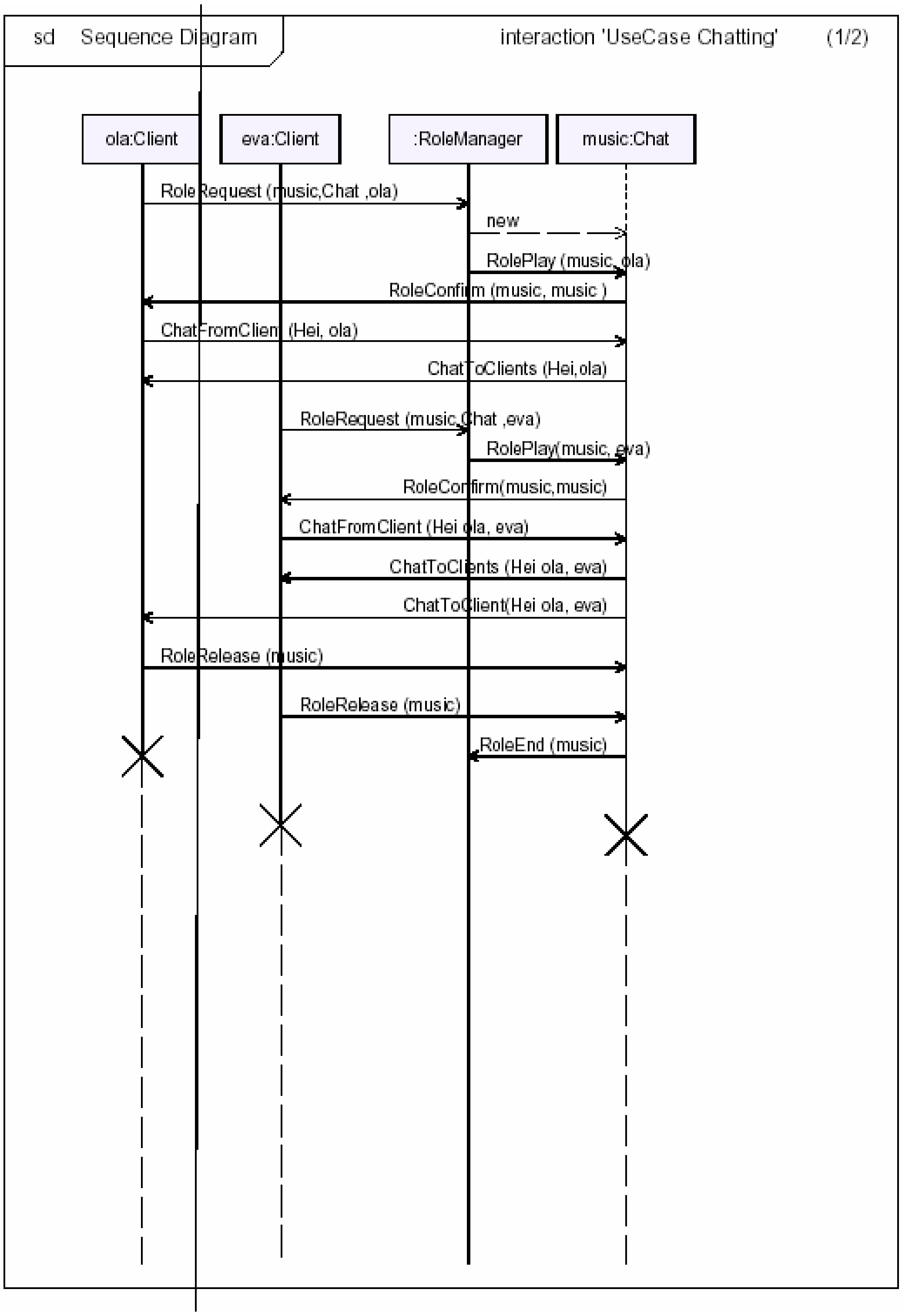
8.9 Class ChatService – Internal structure



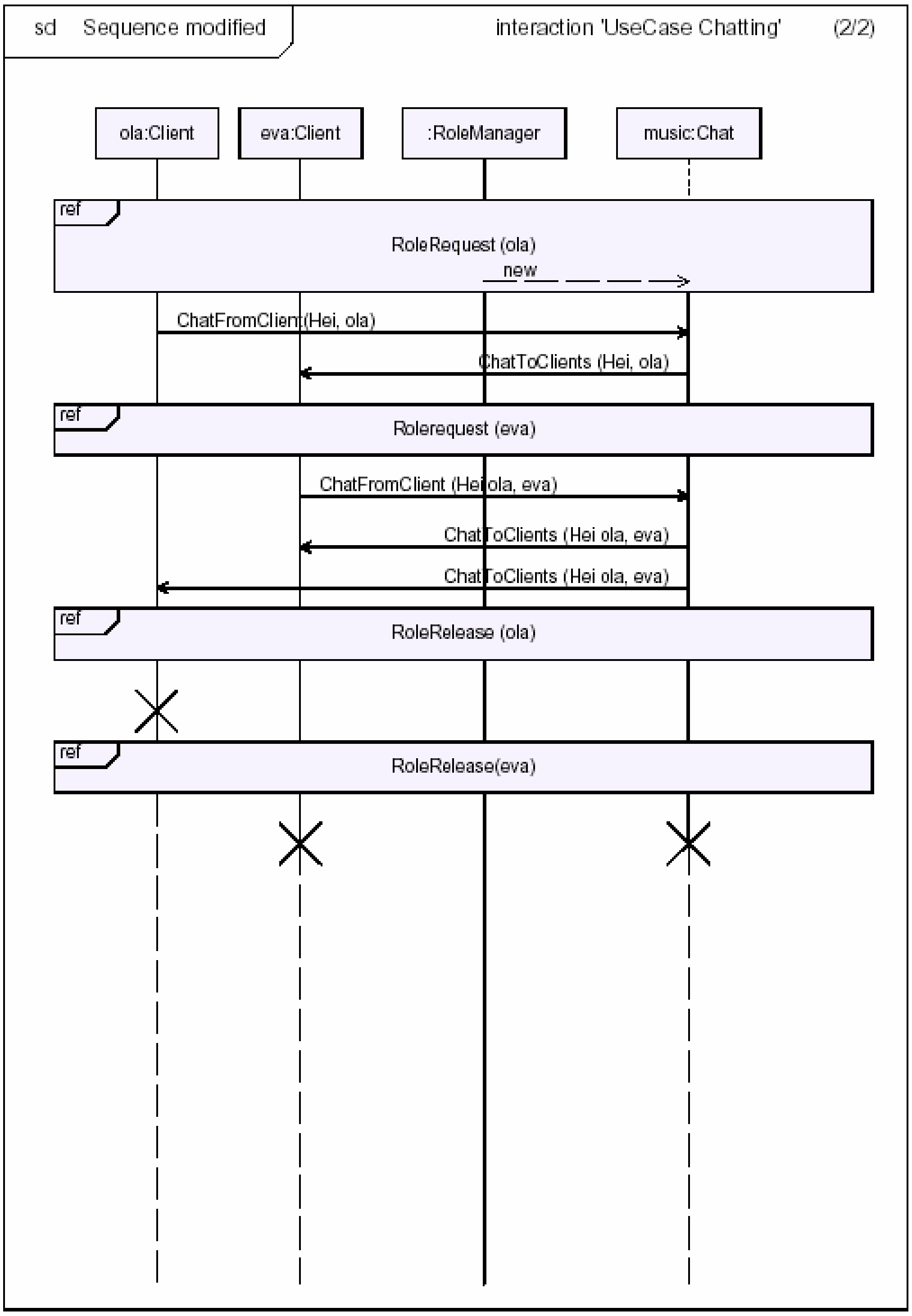
8.10 Class Users


Statechart Diagram
statemachine initialize () (1/1)
StartClient("Geir","musikk", "Hei Ola")
StartClient("Eva","mat","lapskaus er tingen")
ChatRunning

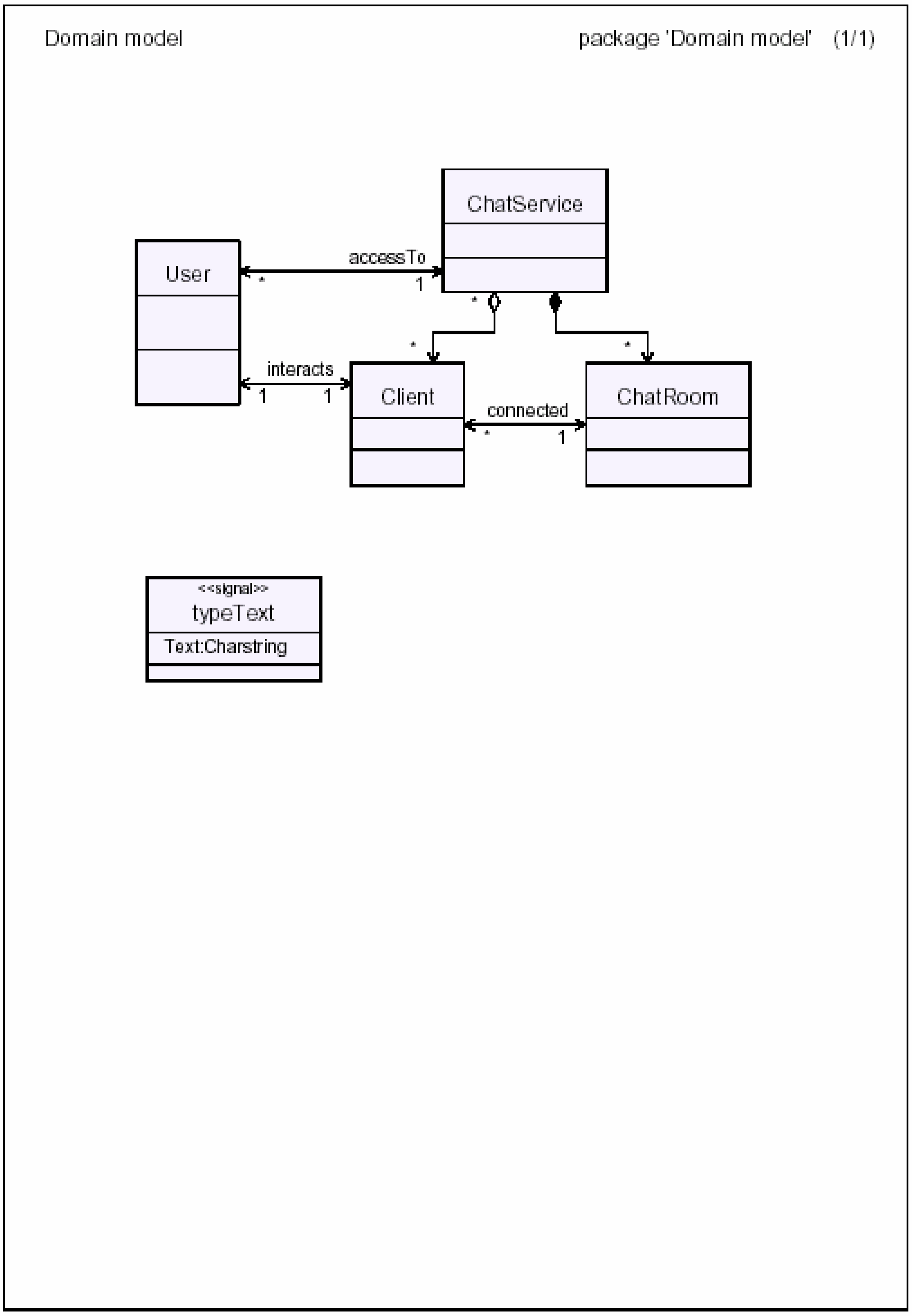
8.11 Sequence diagram Chatting


sd Sequence Diagram
interaction 'UseCase Chatting'
(1/2)
ola:Client
eva:Client
:RoleManager
music:Chat
RoleRequest (music,Chat ,ola)
new
RolePlay (music, ola)
RoleConfirm (music, music)
ChatFromClient (Hei, ola)
ChatToClients (Hei,ola)
RoleRequest (music,Chat ,eva)
RolePlay(music, eva)
RoleConfirm(music,music)
ChatFromClient (Hei ola, eva)
ChatToClients (Hei ola, eva)
ChatToClient(Hei ola, eva)
RoleRelease (music)
RoleRelease (music)
RoleEnd (music)

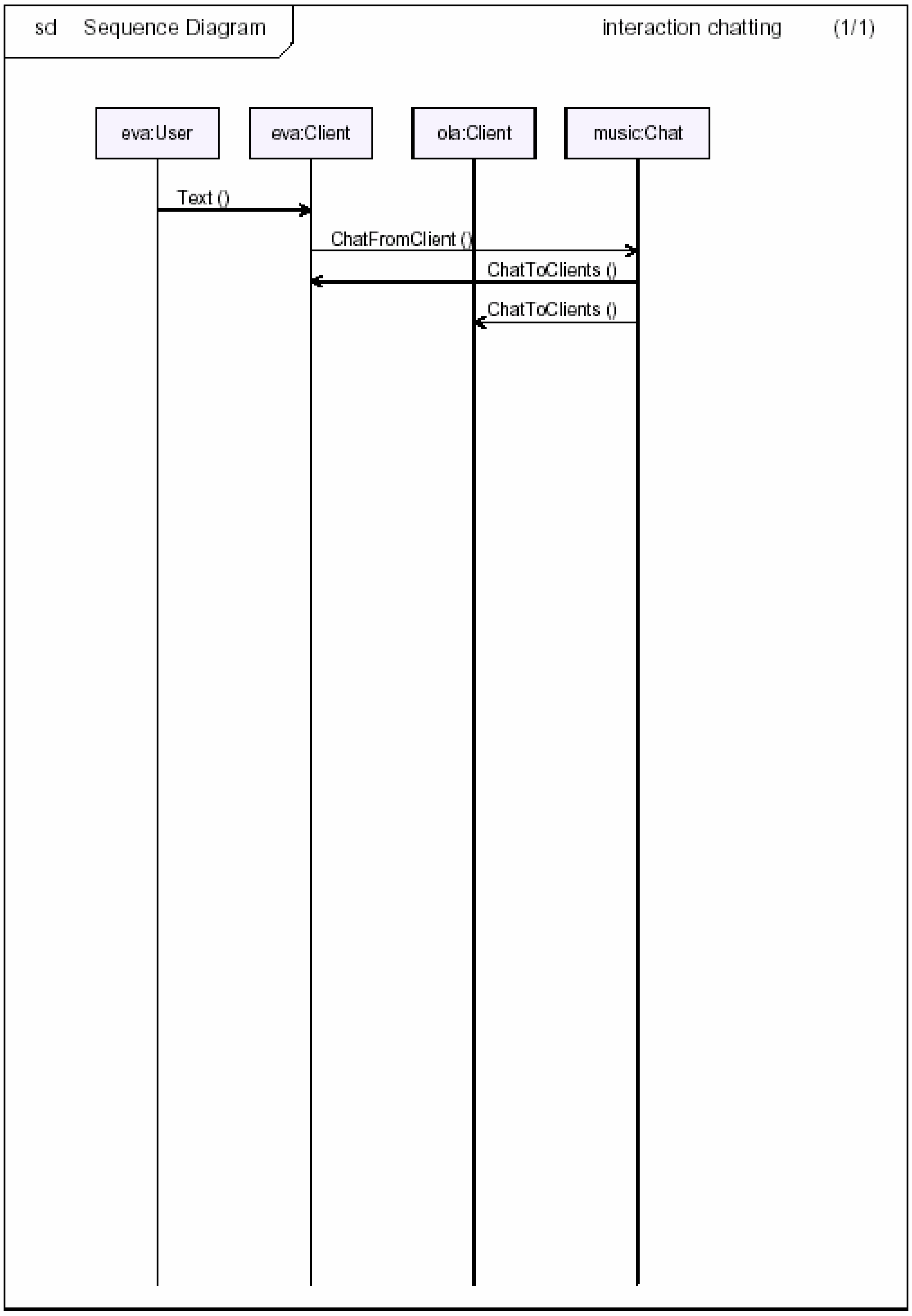
8.12 Sequence diagram Chatting with references



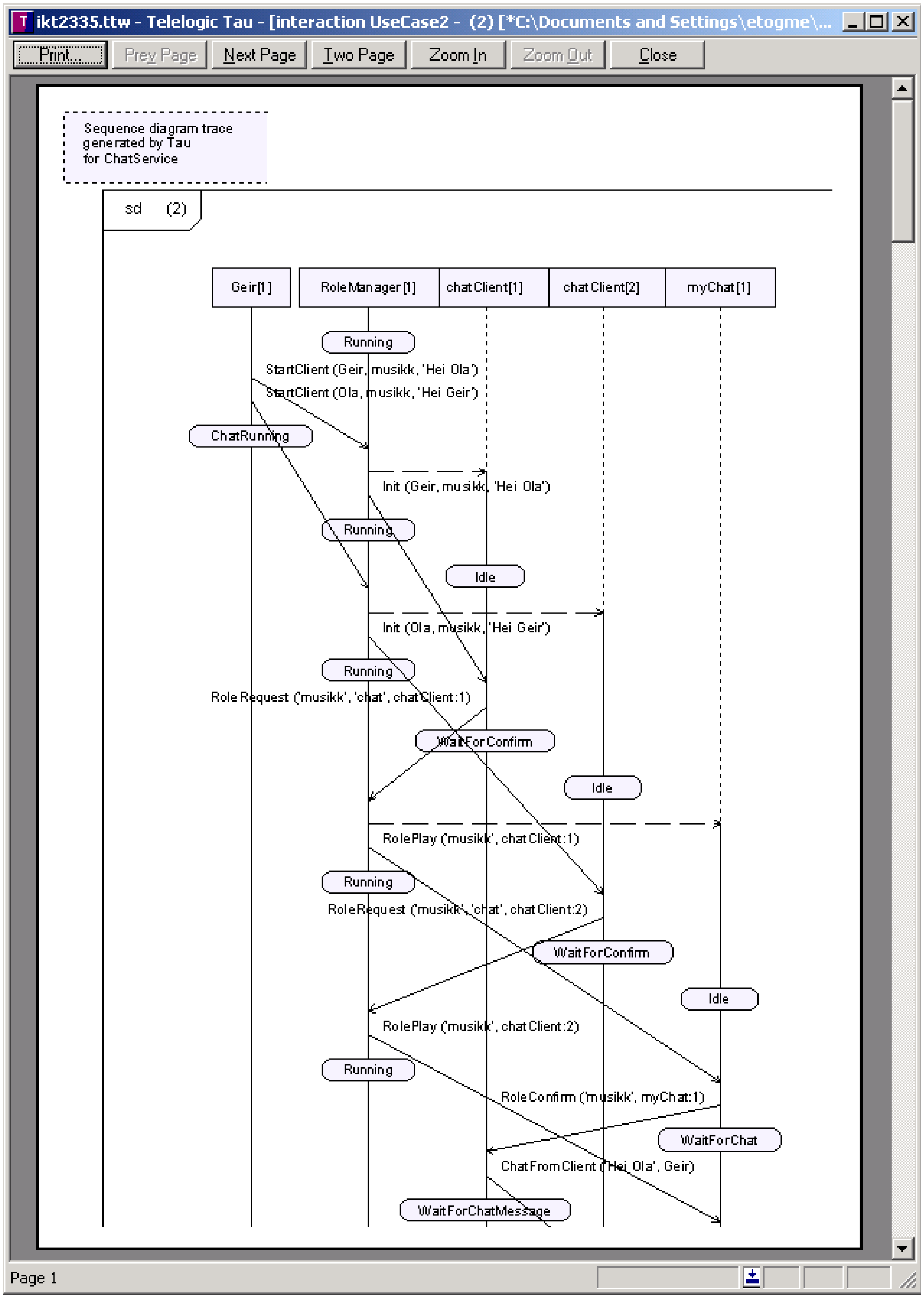
8.13 Domain model

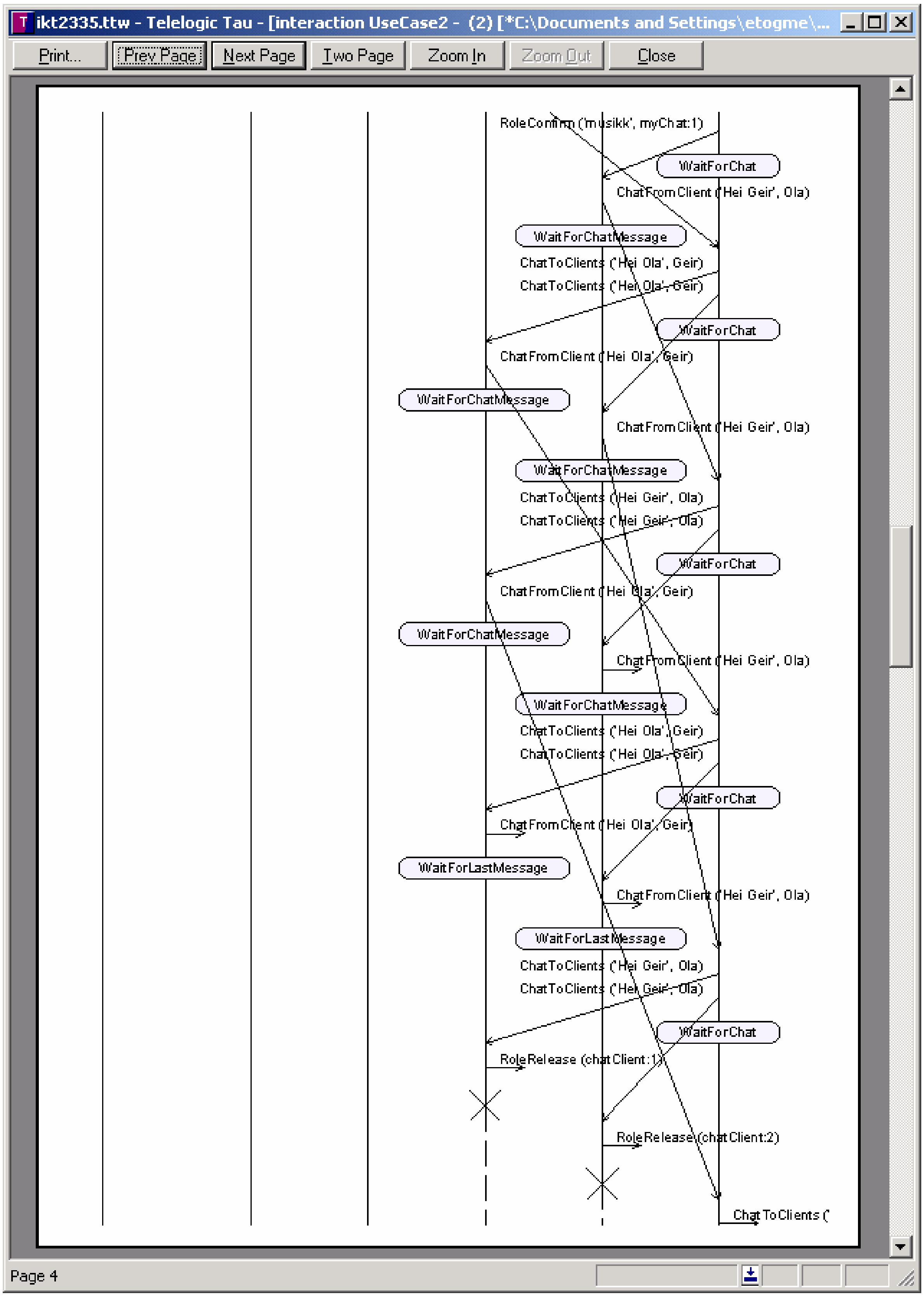


8.14 Sequence diagram chatting



Appendix B – Trace of chatting


ikt2335.ttw - Telelogic Tau - [interaction UseCase2 - (2) [*C:\Documents and Settings\etogme\...
Print...
Prev Page
Next Page
Two Page
Zoom In
Zoom Out
Close
Sequence diagram trace generated by Tau for ChatService
sd (2)
Geir[1]
RoleManager[1]
chatClient[1]
chatClient[2]
myChat[1]
Running
StartClient (Geir, musikk, 'Hei Ola')
StartClient (Ola, musikk, 'Hei Geir')
ChatRunning
Init (Geir, musikk, 'Hei Ola')
Running
Idle
Init (Ola, musikk, 'Hei Geir')
Running
RoleRequest ('musikk', 'chat', chatClient:1)
WaitForConfirm
Idle
RolePlay ('musikk', chatClient:1)
Running
RoleRequest ('musikk', 'chat', chatClient:2)
WaitForConfirm
Idle
RolePlay ('musikk', chatClient:2)
Running
RoleConfirm ('musikk', myChat:1)
WaitForChat
ChatFromClient ('Hei Ola', Geir)
WaitForChatMessage
Page 1


ikt2335.ttw - Telelogic Tau - [interaction UseCase2 - (2) [*C:\Documents and Settings\etogme\...
Print...
Prev Page
Next Page
Two Page
Zoom In
Zoom Out
Close
RoleConfirm ('musikk', myChat:1)
WaitForChat
ChatFromClient ('Hei Geir', Ola)
WaitForChatMessage
ChatToClients ('Hei Ola', Geir)
ChatToClients ('Hei Ola', Geir)
WaitForChat
ChatFromClient ('Hei Ola', Geir)
WaitForChatMessage
ChatFromClient ('Hei Geir', Ola)
WaitForChatMessage
ChatToClients ('Hei Geir', Ola)
ChatToClients ('Hei Geir', Ola)
WaitForChat
ChatFromClient ('Hei Ola', Geir)
WaitForChatMessage
ChatFromClient ('Hei Geir', Ola)
WaitForChatMessage
ChatToClients ('Hei Ola', Geir)
ChatToClients ('Hei Ola', Geir)
WaitForChat
ChatFromClient ('Hei Ola', Geir)
WaitForLastMessage
ChatFromClient ('Hei Geir', Ola)
WaitForLastMessage
ChatToClients ('Hei Geir', Ola)
ChatToClients ('Hei Geir', Ola)
WaitForChat
RoleRelease (chatClient:1)
RoleRelease (chatClient:2)
ChatToClients ('
Page 4