Assessing challenges for remotely downloaded telecom features

by

Viggo Fredriksen

Master Thesis in
Information and Communication Technology

Agder University College
Faculty of Engineering and Science

Grimstad
May 2005

Summary

The telecom and software industries are converging, providing new market and revenue potentials for both industries. Personalization of services and applications is an emerging trend, and this thesis suggests using remotely downloaded UML 2.0 submachines components as a candidate for enabling this. This thesis thus assess the challenges raised by replacing submachines at run-time.

This report gives an introduction to the core technologies of interest in this thesis work; UML 2.0 and Ericsson's service creation architectures. Ericsson's service creation architecture is a set of frameworks for development of services and applications with the new hybrid networks created by the merging of the Internet and the telecom service networks. This architecture is built upon the UML 2.0 concepts of asynchronous communication through message passing and the use of Actors.

There are several issues which must be evaluated when allowing remotely downloaded and replaceable submachines at run-time. This thesis does not try to solve all these issues, but tries to alleviate potential problems by decomposing the externally visible properties of the structured classifier in which the state machine is deployed. This allows us to assert these properties at run-time, and the thesis hence introduces a concept called the *event acceptor*. This approach is based on the UML 2.0 concepts of the port and the protocol state machine. Solutions for setting up connectors between ports, sending and replying to messages and execution of the assertion mechanism are shown.

The main conclusion of this thesis is that run-time replacement of submachines is both useful and viable. Allowing such replacement to happen enables personalization of running services and applications. At the same time services may increase their longevity by adding new and emerging technologies without affecting their availability. Although this has not been a formal study, it has been shown to work when there exists mechanisms which can detect erroneous behavior – such as the run-time event acceptor proposed by this thesis.

The main contribution of this thesis is the run-time event acceptor. Ideas on how to route, send and reply to messages is proposed and implemented. Furthermore, a solution on how to configure connectors using signaling is proposed and implemented. The implementation of an example phone book service suggests that this solution work well, and that the introduced con-

cepts should be comprehendable by designers and developers familiar with the NorARC service creation architectures.

Preface

This thesis was written as part of the Masters degree in Information and Communication Technology at Agder University College, Faculty of Engineering and Science located in Grimstad. The project is related to the *Teleservice lab* in Grimstad and *Program for Advanced Telecom Services* (PATS), and was carried out in the period between January and May 2005.

I would like to thank my supervisors, Fritjof Boger Engelhardtsen and Geir Melby, for their invaluable counseling, ideas and support throughout this thesis project.

Grimstad, May 2005

Viggo Fredriksen

Table of Contents

Summary **i**

Preface **iii**

Table of Contents **iv**

List of Figures **vii**

1 Introduction **1**

1.1 Background 1
1.2 UML 2.0 2
1.3 Ericsson NorARC's service creation architectures 2
1.4 Thesis definition 3
1.5 Work description 3
1.6 Report outline 4

2 The Unified Modeling Language 2.0 **5**

2.1 Introduction 5
2.2 Classes 5
2.2.1 Structured class 6
2.3 Port 6
2.3.1 Behavior Port 6
2.3.2 Complex Port 7
2.4 Connectors 7
2.4.1 Delegation connector 7
2.4.2 Assembly connector 8
2.5 State machines 8
2.5.1 States 8
2.5.2 Transitions 10
2.5.3 The behavioral state machine 10
2.5.4 The protocol state machine 10
2.6 Summary 11

3 Ericsson NorARC's service creation architectures **12**
3.1 Introduction . . . 12
3.2 EJBFrame . . . 13
3.2.1 State machine . . . 14
3.3 EJBActorFrame . . . 15
3.3.1 Actor . . . 15
3.3.2 The ActorFrame protocol . . . 16
3.4 Summary . . . 19

4 Allowing remotely downloaded telecom features **20**
4.1 Introduction . . . 20
4.2 An example telecom service – the phone book service . . . 20
4.3 The submachine as a remotely downloaded feature . . . 22
4.4 Encapsulating the submachine component . . . 23
4.4.1 Instance variables . . . 23
4.4.2 Timers . . . 24
4.4.3 Ports and messages . . . 24
4.5 How and when to download and replace a submachine . . . 25
4.6 Ensuring behavioral conformity . . . 26
4.7 Summary . . . 27

5 Using ports and protocol state machines as run-time event acceptors **28**
5.1 Introduction . . . 28
5.2 Goals of the event acceptor . . . 28
5.3 Ports . . . 29
5.3.1 Connectors and addressing of ports and state machines . . . 29
5.3.2 Routing signals between connectors . . . 31
5.3.3 Replying to signals . . . 32
5.3.4 Configuring connectors at creation time of a part . . . 34
5.3.5 Limitations of the approach . . . 36
5.4 Event acceptor execution . . . 37
5.5 Event acceptor and multiple clients . . . 38
5.6 UML 2.0 and message interleaving . . . 39
5.7 Structured classifiers violating protocols . . . 40
5.8 Updating the protocol specification at run-time . . . 40
5.9 Summary . . . 41

6 Run-time replacement of submachines in EJBActorFrame and EJBFrame **43**
6.1 Introduction . . . 43
6.2 What has been realized? . . . 43
6.3 Modifications overview . . . 44
6.3.1 Modified EJBFrame Java package . . . 44
6.3.2 Modified EJBActorFrame Java package . . . 45

6.4 Implementation of SubMachine using EJBFrame CompositeState 46
6.4.1 Structural modifications 46
6.4.2 Behavioral modifications 47
6.5 DynamicActorCS – actor behavior for dynamic submachine loading 48
6.5.1 Messages 49
6.5.2 Getting the reference of the SubMachine to be replaced 49
6.5.3 Searching the active state configuration 50
6.5.4 Remotely download and initialize a SubMachine 51
6.5.5 Replacing the existing SubMachine 51
6.6 Run-time event acceptor implementation 52
6.6.1 Class overview 53
6.6.2 Port addressing implementation 53
6.6.3 Port implementation 54
6.6.4 PortSM implementation 54
6.6.5 Sending and replying to messages 55
6.6.6 Message reception 55
6.6.7 Port description implementation 58
6.6.8 Port and connector creation implementation 58
6.7 Summary 60

7 Implementation of a service using remotely downloaded features 62
7.1 Introduction 62
7.2 The phone book service 62
7.3 Design of phone book service 63
7.4 Phone book service implementation 63
7.4.1 Implementation of run-time replaceable submachines 65
7.4.2 Implementation of the event acceptor 67
7.5 Summary 70

8 Discussion and conclusion 71
8.1 Introduction 71
8.2 Allowing remotely downloaded telecom features 72
8.3 Using ports and protocol state machines as run-time event acceptors 73
8.4 Run-time replacement of submachines in EJBActorFrame and EJBFrame 73
8.5 Implementation of a service using remotely downloaded telecom features 74
8.6 Usefulness of remotely downloading and replacing submachines at run-time . . . 75
8.7 Future work 75
8.8 Conclusion 77

Bibliography 77

Appendix A: CD-ROM 80

List of Figures

2.1 Class Television with an attribute and an operation. 5
2.2 Class Television with internal structure. 6
2.3 Port with required and provided interfaces. 7
2.4 Port *remote* specified as a behavior port. 7
2.5 Connectors constrained by a protocol. 8
2.6 The different types of connectors. 8
2.7 State machine with simple states and a submachine state. 9
2.8 Submachine with entry and exit points. 9
2.9 Protocol state machine. 11

3.1 ServiceFrame - A model driven service development kit [3]. 12
3.2 Class diagram for the EJBFrame Java package. 13
3.3 Class diagram for the EJBActorFrame (gray classes from other packages). 15
3.4 The ActorFrame Actor class – adopted from [5]. 15
3.5 Actor addressing [4] (Non-normative UML). 17
3.6 Sequence diagrams for RoleCreate – adopted from [3]. 18
3.7 RoleRequest protocol concept – adopted from [2] (Non-normative UML). 18
3.8 Sequence diagram for RoleRequest. 19

4.1 Class diagram for the phone book service. 20
4.2 State machine of *PhoneBookService*. 21
4.3 State machines of *UserTerminal*, *CallEdge* and *SMSEdge*. 21
4.4 Save submachine update messages until requirements are met. 26
4.5 Protocol for the provided interface of *PhoneBookService*. 27

5.1 Event acceptor and message interleaving (Non-normative UML). 29
5.2 Port addressing example. 30
5.3 Inner classifier not able to signal enclosing classifier. 31
5.4 Static setup of reply paths. 33
5.5 Sending messages with a dynamically created reply path. 34
5.6 Incomplete connection configuration. 36
5.7 Black-box versus white-box view of the structure. 36
5.8 Addressing of ports and interfaces and their context. 37
5.9 Execution of the event acceptor. 38

5.10 Two clients connected to one port. 38
5.11 UML 2.0 and message interleaving (Left diagram: Non-normative UML). 40

6.1 Modified EJBFrame Java package overview. 44
6.2 Modified EJBActorFrame Java package overview. 45
6.3 Modifications done to State and StateMachine. 47
6.4 Classes inherited from SubMachine. 47
6.5 Modified transition execution. 48
6.6 Sequence diagrams for *GetStateProperties* and *StoreStateProperties*. 49
6.7 Activity diagram showing the submachine replacement steps. 50
6.8 ActorMsg's for *DynamicActorCS*. 50
6.9 Classes CompositeState and State with partial operations and attributes. 51
6.10 Searching active state configuration (non-normative UML). 52
6.11 The event acceptor classes Port and PortSM in ActorContext. 53
6.12 Modified ActorAddress class. 54
6.13 Modified ActorMsg class. 55
6.14 Sequence diagram for message reception (simplified). 57
6.15 Sequence diagram for event acceptor execution. 58
6.16 PortTask – Execution of a protocol state machine in the event acceptor. 59
6.17 Modified PortSpec class. 59
6.18 Setting up addresses between two ports. 60
6.19 PathRequest sequence for synchronization of connectors. 61

7.1 Sequence diagram for phone book service. 63
7.2 Sequence diagram for *SearchFeatureSms* and *SearchFeatureCall*. 64
7.3 Class diagram of the phone book service. 64
7.4 Class diagram for manually modified classes of *PhoneBookService*. 65
7.5 Class diagram for manually modified classes of *SMSEdge*. 65
7.6 Implementation of behavior – class *PhoneBookServiceCS*. 66
7.7 Actor descriptor with part and connector mapping for *ActorDomain* (partial). . . 67
7.8 Protocol state machine mapping for *SMSEdgePortSM*. 68
7.9 Actor descriptor and port mapping for *PhoneBookService*. 69
7.10 Actor descriptor and port mapping for *SMSEdge*. 69
7.11 Actor descriptor with part and connector mapping for *ActorDomain* (partial). . . 70

Chapter 1

Introduction

1.1 Background

These days the computer domain and the more traditional telecom domain are starting to converge. The telecom network operators are opening up their service network through initiatives such as the Parlay/OSA API's [18]. Such initiatives enables development of applications that operate across multiple networking-platform environments. At the same time the computer domain is becoming more IP network oriented. As such, this convergence provides both the computer and telecom industry with new revenue and market possibilities.

This convergence enables third-party service providers to integrate their special purpose services with the telecom service network. At the same time new mobile devices with increased processing power is becoming available to customers – and thus the opportunity to create new and exciting services. The demand for personalization of services and applications is increasing. Customers want the opportunity of customizing applications run on their terminals or services executed by the service provider to fit their specific needs. However, the customers may find it unacceptable to stop the service while doing such customization and *feature* enrichment. It thus becomes interesting to see if a service or application could be updated by remotely downloading and deploying newly available features at run-time – without having to affect its availability. Furthermore, if such features could be designed and deployed independently, it would allow new and emerging technologies to achieve shorter time to market.

Such possibilities also imposes some major challenges. Third-party service providers and the telecom network operators will have to agree upon security requirements which they must adhere to. When the telecom networks are exposed through the IP networks, they need mechanisms which can protect them from abuse. There already exists solutions to such network level abuses, and the lack of security is thus found at application level.

This thesis will use the traditional approach by modeling the behavior of such services and applications through the use of *state machines*. More specifically, the *Unified Modeling Language*

2.0 (henceforth UML) state machine construct. The *NorARC service creation architectures* are based upon this programming technique.

1.2 UML 2.0

Telecom companies have successfully used SDL [19] in the design of telecom products. This allowed making functional models of a system, while being able to formally verify and automatically generate code.

The Object Management Group [16] (OMG) recently adopted the UML 2.0 Superstructure [8] specification. Earlier versions of UML became the *de-facto* standard modeling language in the software industry, but lacked the formalisms and concepts which made SDL successful in the telecom industry. This new version promises to leverage this lack of concepts and formalisms, and is the core language component to be used in the *Model Driven Architecture* [20] (MDA).

1.3 Ericsson NorARC's service creation architectures

Ericsson NorARC's service creation architectures is built upon three distinct frameworks; ServiceFrame [2], ActorFrame [3][4] and JavaFrame [13]. These are all tools for design, deployment and management of services. The architecture is meant to deal with the complex systems which emerges with the convergence between telecom and Internet services.

The main objectives of the ServiceFrame project is described as follows in [2]:

- Enable model driven service development using UML 2.0 to achieve both short time to market and controlled quality.
- Provide a solution that leverage and support the emerging all-IP and full service networks.
- Remove constraints known from the IN approach in order to enable horizontal, end to end, services and the widest possible range of advanced hybrid services.
- Take advantage of emerging technologies for service creation and execution.
- Provide architectural support for incremental service development and deployment.
- Support personalized services and mobility.
- Serve as a test case for UML 2.0.

1.4 Thesis definition

The thesis will evaluate the submachine component as a candidate for remotely downloaded telecommunication features introduced at runtime. The final thesis definition is thus:

> *The student will assess the usefulness of utilizing submachines as remotely downloaded and replaceable components with regards to dynamic feature adaption. This assessment will be conducted using NorARC's telecommunication service framework ActorFrame as test bed. Replaceable submachine components will impose challenges on both the underlying framework and the components themselves. These challenges will be identified and proposed solutions to them will be evaluated.*

Furthermore, the thesis will provide the following artifact:

> *A prototype demonstrating some of the proposed solutions will be implemented in the ActorFrame framework.*

The final thesis title is:

> *Assessing challenges for remotely downloaded telecom features.*

1.5 Work description

The main questions to be answered in this thesis are:

- In order to customize services, are submachines viable and useful as components for enabling third-party service providers to create new features that may be downloaded at runtime?
- Which challenges needs to be addressed when designing replaceable submachines?

To answer these questions, the thesis work will therefore start off by designing a telecommunication application. This application is modeled using UML 2.0 notation, and shall be able to adapt its behavior using remotely downloaded and dynamically replaceable submachine features introduced at runtime. This application model will provide some of the means for evaluating how useful such a possibility is with regards to rapid deployment.

Furthermore, the model will act as a basis for an analysis of which challenges submachine components impose – i.e., which mechanisms and properties are needed in the framework and components to make them viable for implementation. Finally, the following question will be answered:

- How can dynamic submachine components be realized in the NorARC service creation architecture?

This question will be answered by implementing the designed application in ActorFrame – using some of the proposed solutions from above. It is expected that changes within the framework are necessary to accommodate the new features. Furthermore, the implementation will help to conclude if remotely downloaded, replaceable submachine components are useful for telecommunication service developers.

1.6 Report outline

The UML 2.0 specification constitutes a large set of concepts. The NorARC service creation architecture is a complex framework stack involving many techniques which will not be further explored in this thesis. This report will therefore focus on parts which are important for understanding the contents of this report. Readers unfamiliar with these technologies are encouraged to read referenced literature if the given descriptions are too brief or incomplete.

Chapters 2 and 3 present the technologies UML 2.0 and NorARC service creation framework stack which are used throughout this report. Readers familiar with these technologies can skip these chapters.

Chapter 4 presents a telecom service which is used to identify different issues when allowing remotely downloaded submachine features.

Chapter 5 expands Chapter 4, and presents an approach which combines the UML 2.0 port and protocol state machine constructs as means for ensuring behavioral conformity at run-time.

Chapter 6 proposes a solution for run-time replacement of submachines in the NorARC service creation frameworks, using some of the proposed solutions from Chapter 4 and Chapter 5. Chapter 7 describes an implementation of the service described in Chapter 4 and demonstrates the use of the modified frameworks.

In Chapter 8 the proposed solutions are discussed according to the problem definition given in this chapter, followed by the thesis conclusion at its end.

Chapter 2

The Unified Modeling Language 2.0

2.1 Introduction

The Object Management Group [16] recently adopted the newest UML Superstructure specification, UML 2.0. Amongst the most notable improvements over previous versions of the specification, one finds better support for scaling large software system with its new architectural modeling capabilities. Furthermore, important improvements have been done to state machines and its encapsulation of submachines through entry and exit points.

This chapter focuses on the parts of UML 2.0 which are most relevant for this thesis, namely the architectural elements *structured classes* and the behavioral elements *state machines* and *protocol state machines*.

2.2 Classes

The class is a standard UML construct used to specify objects that share the same attributes, operations, relationships and behavior. Figure 2.1 shows such a class specifying a television.

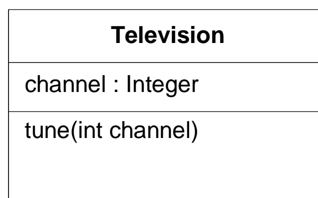

Figure 2.1: Class Television with an attribute and an operation.

2.2.1 Structured class

A structured class is a class showing its inner structure consisting of parts, ports and connectors. When an instance of a structured class is created, its parts, ports and connectors are created, and upon destruction they are destroyed. Port instances are not created or destroyed during the lifetime of the structured class.

Figure 2.2 shows the class Television with an internal structure. The Television class contains two ports that connect it with the environment. One of them receives signals from a remote control, while the other shows the video output. Connectors are drawn from these ports to the inner part Tuner. This enables the tuner to receive remote control events and send video output.

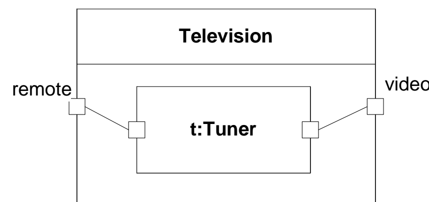

Figure 2.2: Class Television with internal structure.

2.3 Port

A port is a structural feature of a classifier that encapsulates interaction between the contents of the classifier and its environment. Each port can define a set of *required* and *provided* interfaces, drawn as socket and ball respectively. These interfaces specifies what the class offers to, and expects from, its environment. While the *required* interface specifies what is needed to interact with the class, the *provided* interface specifies what the port offers to the environment.

The port provides a mechanism for encapsulating a classifier from its environment. This makes it possible to hide the inner structure of a classifier and to design classifiers without any other knowledge then the interfaces or protocols it requires and provides. Furthermore, the use of ports permits the internal structure of a classifier to be modified without affecting external clients, provided the specified interfaces of the ports are correctly supported. UML does not explicitly specify how reception on a port with multiple defined *connectors* should be routed.

As shown in Figure 2.3, the class Television specifies a required interface *IControl*, to control the unit, and a provided interface *IStream* which gives access its video stream.

2.3.1 Behavior Port

The behavior port is a port with the flag *isBehavior* set to *true*. This port type specifies a port which delivers requests to the behavioral part – such as a state machine or procedures imple-

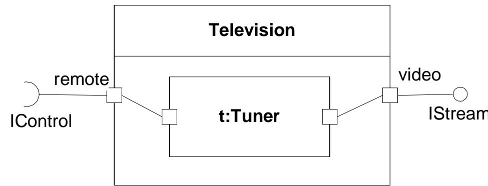

Figure 2.3: Port with required and provided interfaces.

mented by the classifier. A port which is not a behavior port must have a connector to a port on an internal part. Figure 2.4 shows the *remote* port with the *isBehavior* flag set. This is drawn as a small state symbol attached to the port.

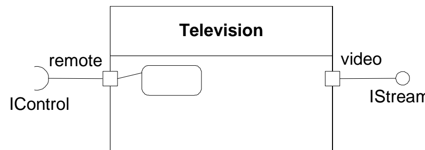

Figure 2.4: Port *remote* specified as a behavior port.

2.3.2 Complex Port

The complex port is a port which has complex behavior – meaning it does not have a single set of *required* or *provided* interfaces. In other words, the port could specify more than one provided and more than one required interface.

2.4 Connectors

A connector defines the connection between two parts within a structured part. This connection is a specification of a contextual association which applies in a certain context. The connector makes it possible for two parts to communicate. A connector can be constrained to a protocol as shown in Figure 2.5. Note that such a constraint represents an assertion, not an executable mechanism. There are two different types of connectors, the *delegation connector* and the *assembly connector*.

2.4.1 Delegation connector

A delegation connector is defined as a connector between an external port and an internal port. A signal arriving on the external port should be received by the internal port. A request sent on the internal port is sent by the external port. Such a connector is shown between the *Compiler* and the *Optimizer* in Figure 2.6.

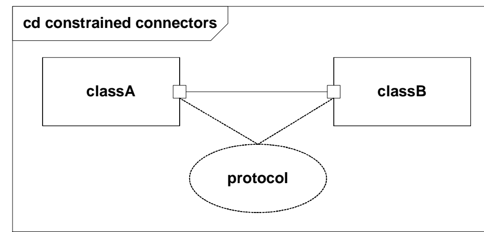

Figure 2.5: Connectors constrained by a protocol.

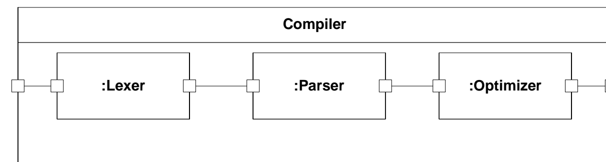

Figure 2.6: The different types of connectors.

2.4.2 Assembly connector

An assembly connector is a connector which connects a required interface or a port on one component, to a provided interface or port on another component. Such a connector is shown between the *Lexer* and the *Parser* in Figure 2.6.

2.5 State machines

The state machine is an universal and well-known formalism for specifying the state space and the state transition relations of objects. It is a convenient way to specify a sequence of *states* that an object passes through during its lifecycle in response to events sent to the state machine. UML 2.0 defines two different kinds of state machines; the generic *behavioral state machine* and the specialized *protocol state machine*.

2.5.1 States

An object has different states during its lifetime, and a state is said to be *active* when the object satisfies the conditions of the state. The *active state configuration* is the set of states which are active for an object at any point in time. A state may optionally contain internal transitions in the form of *entry* activity, *do* activity and *exit* activity. The entry activity is performed upon entering the state, the do activity is performed as long as the state is active, while the exit activity is performed upon leaving the state. UML 2.0 specifies different types of states – the *simple state*, the *composite state*, the *submachine state* and different types of *pseudo states*.

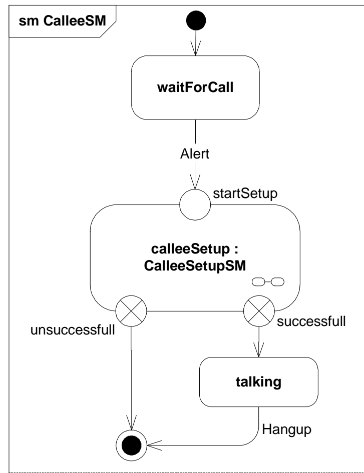

Figure 2.7: State machine with simple states and a submachine state.

Simple state

The simple state is a state without any substructure of states. Figure 2.7 shows the two simple states *waitForCall* and *talking*.

Submachine state

The submachine state is a state which references another state machine – a *submachine*. A submachine state is semantically equivalent to inserting the copy of the referenced submachine in place of the state. The submachine state itself has no substructure, as this structure comes from the referenced state machine.

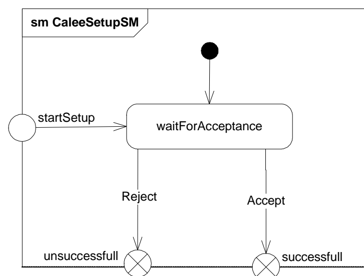

Figure 2.8: Submachine with entry and exit points.

A transition to the submachine state establishes the initial states of the submachine – if there exists no initial state, transitions to the boundary are not allowed. A transition to the state may also target a named entry point on its boundary, which is equivalent to a transition to the corresponding entry point on the submachine. Equally, transitions exiting the submachine through an

exit point targets the corresponding exit point specified in the submachine state. As the submachine which is referenced by the submachine state is defined independently from its environment, it is not allowed to cross their boundaries by entering one of its sub-states directly. Entry and exit points are thus specified *explicitly*.

A submachine state is shown in the Figure 2.7 where the submachine state *calleeSetup* references the submachine *CalleeSetupSM* from Figure 2.8.

Composite state

A composite state is a state with sub-states, or inner structure. This inner structure contains states and transitions which specifies the behavior the state machine has while it is in this state. If a composite state is part of the active state configuration graph, and contains one region, exactly *one* sub-state is active. The composite state is semantically equal to the submachine state, but with one difference – it is allowed to cross the composite state boundaries directly into one of its sub-states.

2.5.2 Transitions

A transition is a relationship between two states within a state machine. This relationship specifies conditions for which a state machine should exit one state and enter another. Such conditions may be defined by *event triggers* and *guards*. If the guarding condition is met, the event trigger is the event received by a state machine which makes the transition fire. When a transition fires, it may have an *effect* – an activity or action performed by the transition.

2.5.3 The behavioral state machine

As in contrast to the *protocol state machine*, the behavioral state machine is an executable behavior which specifies the executions of objects of a given class as triggered by the occurrence of events. A behavioral state machine is shown in Figure 2.7.

2.5.4 The protocol state machine

The protocol state machine is defined in [6] as follows:

> *A state machine used to specify the legal sequences of operation calls and signals received by an object.*

In contrast to the *behavioral state machine*, where it is stated that any legal sequence of events produces an outcome, a *protocol state machine* only specifies the legal sequences of events that may occur within the context of a classifier. A protocol state machine is thus not responsible for ensuring that the legal sequence of events occurs.

If a sequence of events leads to a valid path through the protocol state machine, the sequence

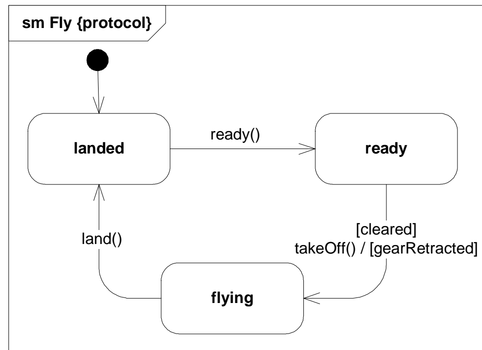

Figure 2.9: Protocol state machine.

of events is legal and shall thus be accepted by the system. If the sequence is invalid, it may not occur. The protocol state machine could thus be used for assertion of sequences – where illegal event sequences may not be handled by the receiving class.

Furthermore, the following differences exists between the *behavioral state machine* and the *protocol state machine*:

- Transitions do not have effects.
- Transitions may have preconditions.
- Transitions may have postconditions.

Figure 2.9 shows a protocol state machine. This states that a plane which wants to *take off*, shall be in state *ready* and be *cleared* for take off before allowed to do so. Furthermore, it specifies that that after such a take off, the gears shall be *retracted* and be in state *flying*.

2.6 Summary

This chapter introduced the main concepts of UML 2.0 which are used throughout the remainder of this thesis report. Architectural elements such as parts, ports and connectors are described. A brief overview of the behavioral state machine and protocol state machine has also been provided. In the next chapter I will introduce Ericsson's service creation architectures – parts of which is built upon the concepts of UML 2.0.

Chapter 3

Ericsson NorARC's service creation architectures

3.1 Introduction

ServiceFrame, ActorFrame and JavaFrame are three frameworks which are being actively developed by Ericsson NorARC. At the time of writing this thesis, there exists two different implementations of the built on the abstract ideas of ActorFrame and JavaFrame; one implementation on the J2EE [21] platform targeted for service development and one MIDLet implementation targeted for thin user terminals. This thesis will use the J2EE implementations, called EJBActorFrame and EJBFrame accordingly.

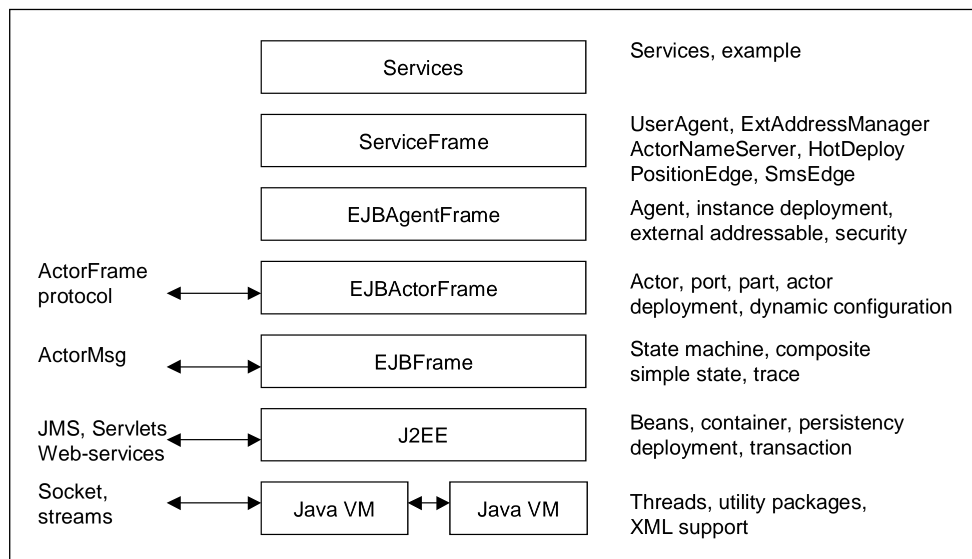

Figure 3.1: ServiceFrame - A model driven service development kit [3].

Figure 3.1 shows the layered model of the framework stack, each implementing its own set of concepts with regards to service development, deployment and description. The frameworks are implemented using Java, and each of them is contained in separate Java packages. The layering used by this model is not strict, and services are implemented by extending the classes from these packages. This thesis will concentrate on the two framework layers EJBActorFrame and EJBFrame, as they deal with structured classifiers, or *Actors*, and behavior using asynchronous communication with message passing and state machines. I will in this chapter focus on the basic concepts needed for understanding this report, for further information on these two framework layers refer to [3] and [4].

3.2 EJBFrame

EJBFrame is the bottom layer of the NorARC service creation architecture. EJBFrame is a *Modeling Development Kit* (MDK) for developing and executing state machines in the Java language. It provides a subset of the UML 2.0 state machine modeling concepts. According to [13]: *"With JavaFrame it is possible to apply modeling techniques and still work in Java. The Java source and the model have one-to-one relationship. The framework provides classes of well-proven modeling concepts, and by using these, instead of just programming in plain Java, the abstraction level is raised"*. The current version of JavaFrame is called EJBFrame and is implemented using J2EE technologies for asynchronous message passing, persistent state data and state machine addressing. Figure 3.2 shows the main classes of the EJBFrame Java package.

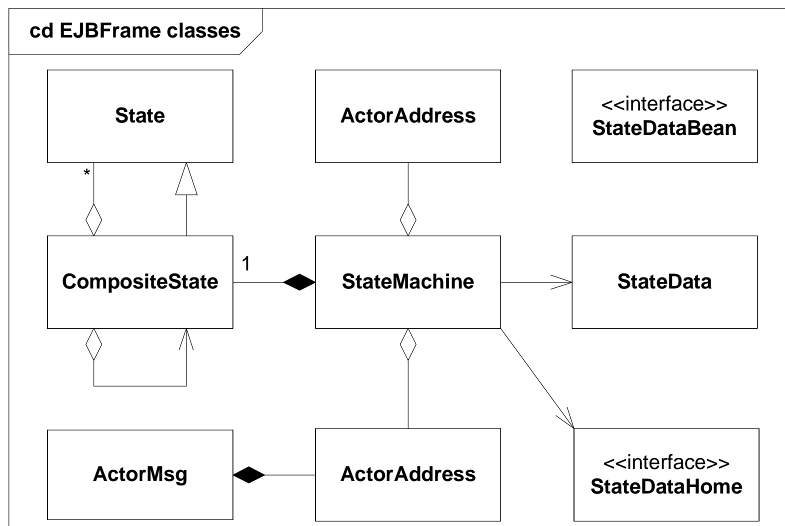

Figure 3.2: Class diagram for the EJBFrame Java package.

3.2.1 State machine

EJBFrame models behavior of systems with state machines. This is done by using the Java classes; *StateMachine*; *State* and *CompositeState*. Messages sent between *StateMachine* objects are of type *ActorMsg*. This message specifies an address to the sender and an address to the receiver of the message in addition to optional message data.

State

The *State* class represents the simple state from UML 2.0, and thus has no inner structure. The class holds a reference to its enclosing *CompositeState* object with the attribute *enclosingState*. As in UML 2.0, the *State* class can have internal transitions by specializing the operations *entry()* and *exit()*. They do not allow any *do* activities.

CompositeState

The *CompositeState* class extends the *State* class and implements a state that contains an inner structure of states and transitions. The inner state structure is contained in the *children* Hashtable attribute, using the state name as key. States contained by this *children* hash references either objects of type *State* or other *CompositeState* objects. The EJBFrame *CompositeState* does not support the use of orthogonal state regions as in UML 2.0.

As in UML 2.0, the *CompositeState* can define different entry and exit points, which are named with integer values. Entering the *CompositeState* is done by calling *enterState* with the given integer identifying the entry point. When *enterState* is called without such an integer identification, the default entry point is used. Exiting its boundaries is performed by calling *exitState* with the given integer identification, and execution continues by calling *enclosingState*'s operation *outOfInnerCompositeState*. The inner sub-structure of states is not exposed, and the EJBFrame *CompositeState* thus resembles the UML 2.0 submachine state construct, as it is not allowed to cross its boundaries without *explicitly* using entry or exit points.

StateMachine

The class *StateMachine* has one *CompositeState* which contains the state and transition structure of the state machine. In this way, the containment of states and transitions in the class *CompositeState* can be reused in the state machine without implementing this again.

The *StateMachine* class implements all the behavior for executing the state machine. This involves storing and resurrecting persistent data and handling of message reception and message sending. Furthermore, the *StateMachine* class implements all the necessary methods for programing a state machine.

StateMachine objects are addressable through the use of the *ActorAddress* class. This address

represents the message input queue of the state machine, and is used by other *StateMachine* objects which wants to send messages to it.

3.3 EJBActorFrame

EJBFrame is layered above EJBFrame in the framework stack and is ServiceFrame's service execution environment. The core concept of ActorFrame is *"actors plays roles"*. According to [2]: *"An Actor is a (composite) object having a state machine (ActorSM) and an optional inner structure of Actors"*. Figure 3.3 shows the main Java classes of the EJBActorFrame package.

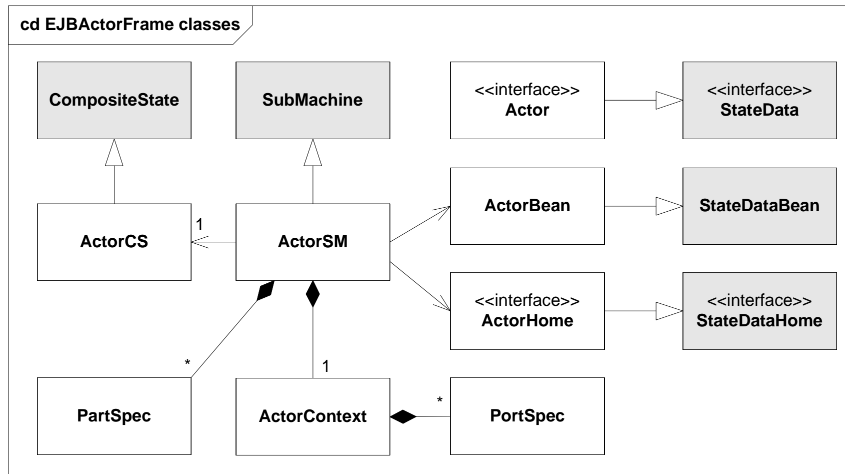

Figure 3.3: Class diagram for the EJBActorFrame (gray classes from other packages).

3.3.1 Actor

The ActorFrame Actor is a composite object with a state machine and optional inner structure. Figure 3.4 shows the ActorFrame Actor, with *in* and *out* ports and an inner part *innerActor*.

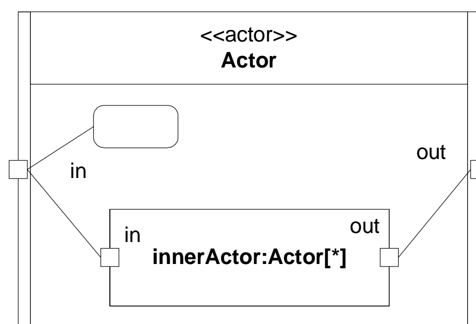

Figure 3.4: The ActorFrame Actor class – adopted from [5].

Actor behavior

The generic behavior of an Actor is implemented by the state machine combination of the *ActorSM* and the *ActorCS* classes. These classes extends the *StateMachine* and *CompositeState* classes from EJBFrame. The generic behavior implements the Actor lifecycle management protocols as described in Section 3.3.2.

Actor descriptors

The inner structure of an ActorFrame Actor and its contextual relations with ports and connectors is described in a set of XML files, called *actor descriptors*. This information is parsed from the XML files upon initiation of the Actor, and is contained in the *PartSpec* and the *PortSpec* classes. The *PartSpec* class specifies the inner structure of the Actor, while the *PortSpec* class specifies the contextual relations the Actor has to other Actors. Both these classes can have multiplicity, meaning they can have several inner Actors and relations with several other Actors.

Ports, connectors and addressing of Actors

The *PortSpec* class specifies a relation between two Actors. This relation is described as a set of *ActorAddress* instances; *inquiredRole* and *requestedRole*. The connectors are set up between two Actors by using this information in the *RoleRequest* protocol of the ActorFrame protocol.

As shown in Figure 3.4 an ActorFrame Actor has two ports, one *in* port and one *out* port. In EJBActorFrame the *in* port is mapped to a JNDI name of the input queue for the state machine, while the *out* port is mapped to the operation *output* in the class *StateMachine*. Sending a message through an *out* port is done specifying other JMS destinations representing the input queues for other Actors.

This means the Actor port does not use the UML 2.0 delegation connectors – only assembly connectors directly between two adversary Actors. As such, no hiding of inner structure through the use of ports is supported by EJBActorFrame.

Figure 3.5 shows how Actor instances are addressed using the *ActorAddress* class. The addressing is based on the instance name of the and class type of the Actor. The address also includes the contextual information of its enclosing Actors.

3.3.2 The ActorFrame protocol

ActorFrame Actors implement a set of protocols which are used for invocation, control and management of other Actors. The combinations of all these smaller protocols constitutes the ActorFrame protocol. A subset of these, named by its initiator message, is shown in Table 3.1.

The most important of these protocols, with regards to this thesis, are named *RoleCreate* and *RoleRequest*. I will thus provide a brief description of these.

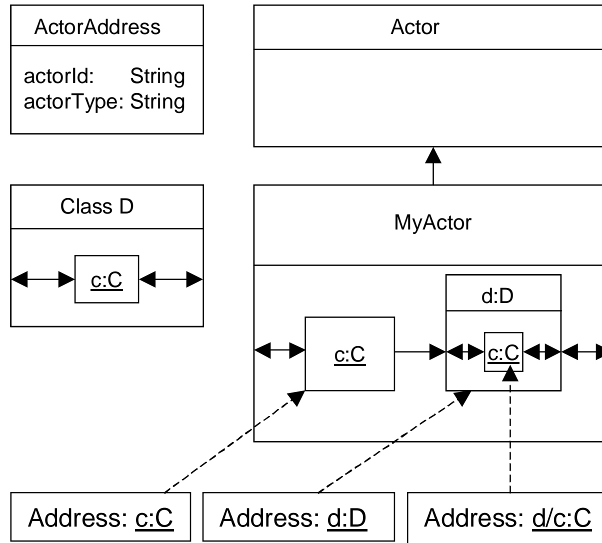

Figure 3.5: Actor addressing [4] (Non-normative UML).

Initiator:	**Description:**
RoleCreate	Creation of new actors.
RoleRequest	Request roles from other actors.
RoleRemove	Remove actor and all its associations.
RoleUpdate	Actor reconfiguration.
RoleReset	Reinitialization of actors.
RoleReport	Get status reports from actors.

Table 3.1: Protocol names in the ActorFrame protocol.

RoleCreate

The *RoleCreate* protocol is used to create new Actor instances. This protocol is utilized when a new inner Actor part is created, or upon initiation of an Actor which shall instantiate all its inner parts. Figure 3.6 shows how *d:DeltaActor* creates the new inner part *g:GammaActor*. After this inner part has been created, a message with the *PortSpec*'s of this part is sent. These *PortSpec* objects represents the Actors this actor shall *play* with, and specifies the ActorAddresses for these Actors.

The actual setup of the play – i.e., the creation of ports an connectors, is shown in Figure 3.6. The figure shows that *actualPart:Part* sends a *RoleRequest* message to *partB:Actor*, which it shall initiate a play with. The Actor *partB:Actor* sends confirmation that it can play the specified role. The inner part *g:GammaACtor* hence sends the message *RoleCreateAck* to its creator.

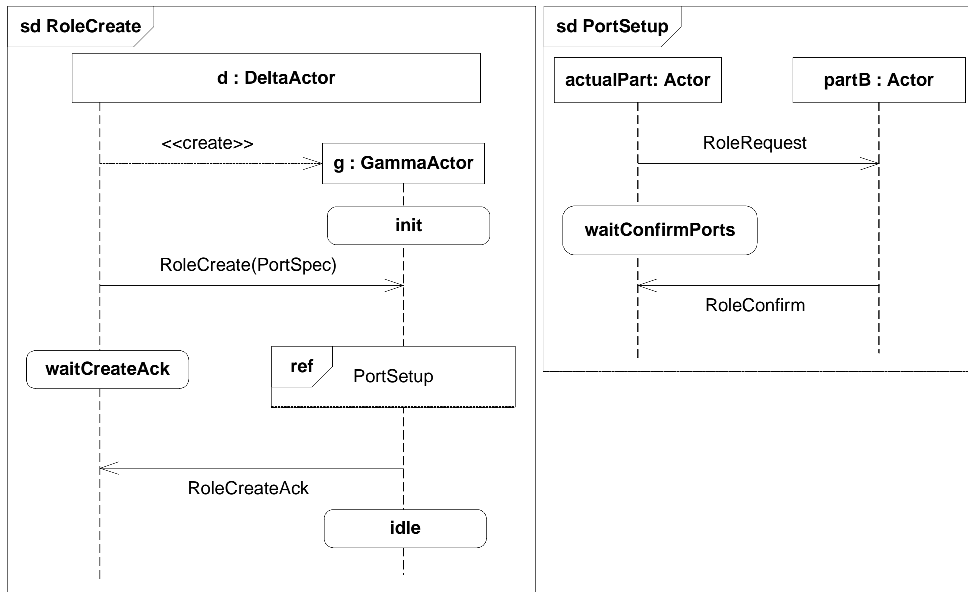

Figure 3.6: Sequence diagrams for RoleCreate – adopted from [3].

RoleRequest

The *Role Request* protocol is the protocol in ActorFrame which enables Actors to request roles from other actors. The principle workings of this protocol is shown in Figure 3.7. The figure shows that the Actor *requester:Actor* wants to initiate a play with *requested:Actor*. This is done by sending a *RoleRequest* message to an *inquired* Actor. The *inquired* Actor checks if this Actor can contain the *requested:Actor*, either by creating a new instance or by asking an existing inner Actor. Such a request is made by sending a *RolePlay* message to the Actor. The receiver of this message sends *RoleConfirm* back to the initiator which confirms the request. This has thus created a relationship between the *requester:Actor* and the *requested:Actor*, and a play can start between these. Figure 3.8 shows the full RoleRequest protocol interaction.

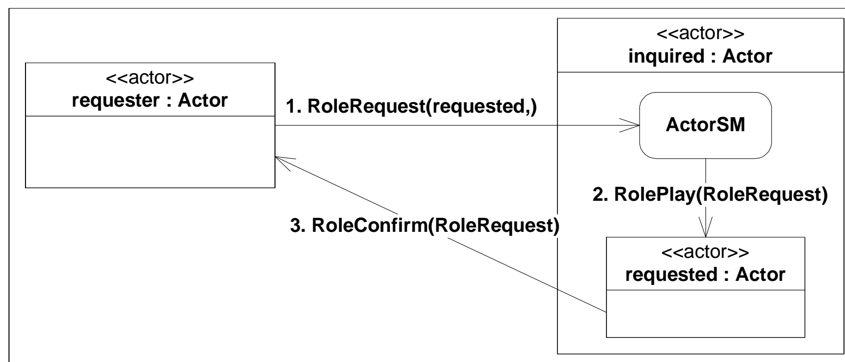

Figure 3.7: RoleRequest protocol concept – adopted from [2] (Non-normative UML).

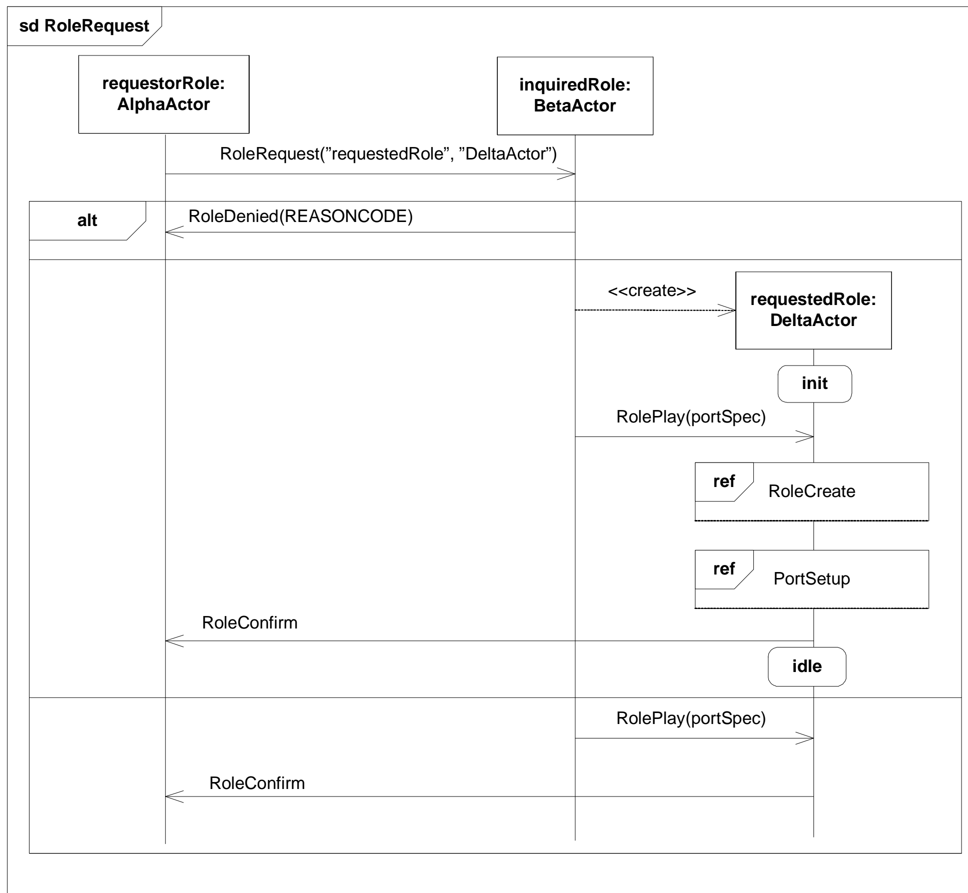

Figure 3.8: Sequence diagram for RoleRequest.

3.4 Summary

This chapter provided a brief overview of the relevant frameworks in the NorARC service creation architectures. The focus of the chapter has been the ActorFrame Actor structure and the ActorFrame protocol in EJBActorFrame. Furthermore, the state machine structure in EJBFrame was described.

I will in the next chapter introduce the concept of allowing remotely downloaded telecom features, and the issues which arise when using the submachine component as a candidate for runtime feature adaptation.

Chapter 4

Allowing remotely downloaded telecom features

4.1 Introduction

In the previous chapters the technological background with regards to UML 2.0 and the NorARC service creation architecture was established. With this foundation, I will in this chapter show an example telecom service and how this service could benefit from using remotely downloaded telecom features. Furthermore, this chapter establishes this thesis' conceptual approach to using the UML 2.0 submachine construct as a feature which enables dynamic behavior adaptation in services. The approach taken in this chapter is partially based on the findings in [11].

4.2 An example telecom service – the phone book service

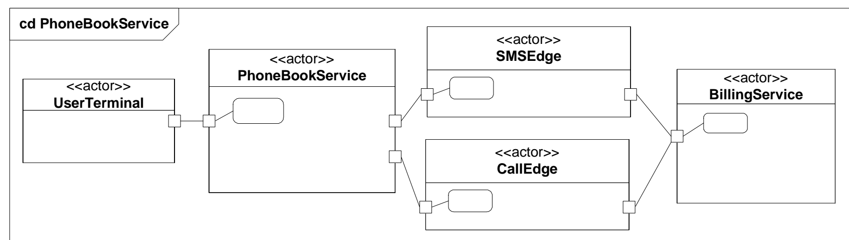

Figure 4.1: Class diagram for the phone book service.

The phone book service is a telecom service which provides a simple phone book lookup service. Figure 4.1 shows the structure of the actors involved with this service. The *PhoneBookService* is the actor which provides the phone book service to a user. The *UserTerminal* is the actor which represents the users terminal, such as a PDA or mobile phone. The *SMSEdge* is the actor which makes it possible to send SMS messages to mobile phone subscribers using the Parlay-X SMS

API. The *CallEdge* is an actor which enables clients to set up conversations between two arbitrary phone numbers using the Parlay-X Call API. As shown in the Figure 4.1, these two actors requires the interface of *BillingService* to be able to perform their service. This *BillingService* is an actor which enables charging of end-users for the provided services.

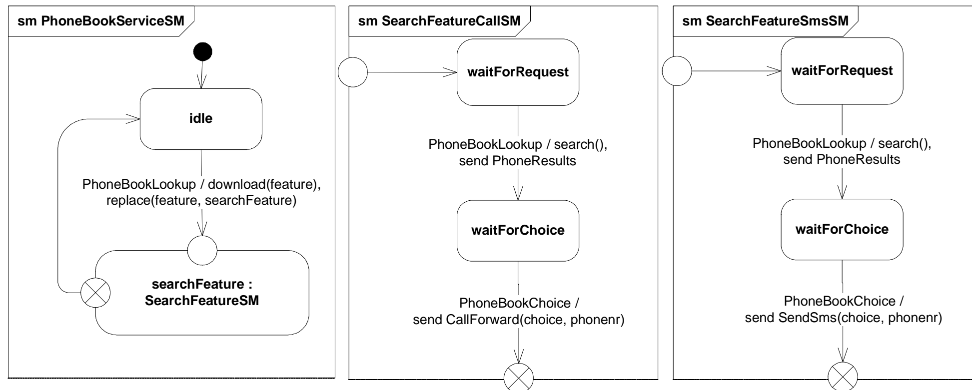

Figure 4.2: State machine of *PhoneBookService*.

Figure 4.2 shows the principle behavior of the *PhoneBookService* with its state machine *PhoneBookServiceSM*. It accepts a message which enables it to download and bind an appropriate submachine to the submachine state *searchFeature*. The figure also shows two different submachines which this state can contain, *SearchFeatureCallSM* and *SearchFeatureSmsSM* – each yielding different behavior of the *PhoneBookServiceSM*.

Figure 4.3 shows the state machines which *PhoneBookServiceSM* interacts with. The state machine *UserTerminalSM* is the state machine of the *UserTerminal* actor, *SMSEdgeSM* is the state machine of the *SMSEdge* actor, while *CallEdgeSM* is the state machine of the *CallEdge* actor.

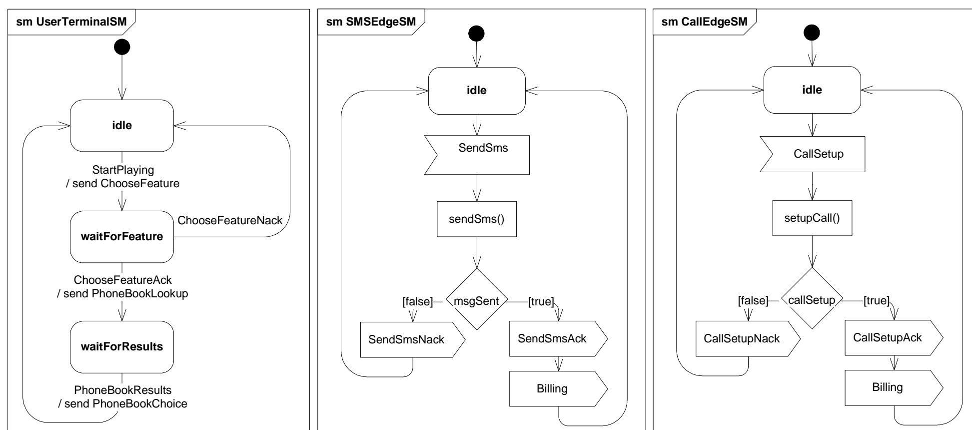

Figure 4.3: State machines of *UserTerminal*, *CallEdge* and *SMSEdge*.

4.3 The submachine as a remotely downloaded feature

Figure 4.2 shows the behavior of the *PhoneBookService* actor. As seen, this simple state machine specifies the submachine state *searchFeature*, describing parts of its behavior. The submachine state also specifies one entry and one exit point, encapsulating the state and transition substructure of the referenced submachine. Furthermore, the two submachines *SearchFeatureCallSM* and *SearchFeatureSmsSM*, shows two different behaviors for this submachine. The difference in behavior between these two components is that one sets up a call between two phones, while the other sends an SMS, based on the results of the phone book lookup.

Now, imagine that these components could be changed at run-time – meaning one could choose between these behaviors while the service is still running. Furthermore, imagine that more such *features* could be developed, remotely downloaded and chosen long time after this service was initially deployed. New and emerging technologies could thus continue to add value to the service after the inception and deployment stages. Such submachine components could even be developed and deployed by *third-party* service providers, and as such expand availability of different features. Furthermore, allowing this feature enrichment to be conducted at run-time gives it the advantage of not affecting the availability of the service.

However, such a possibility also adds additional complexity. Several problems needs to be addressed before allowing such replacements:

- How to ensure that the remotely downloaded submachine component originated from the correct source?
- How to minimize the interdependencies between the state machine and the submachine to allow the view of a submachine as a truly replaceable component?
- How to ensure that a new component does not introduce deadlocks, livelocks or non-deterministic behavior in the system?
- How and when should a submachine be replaced?

When downloading a feature which is meant to be deployed and executed in a service network, one needs to ensure that this feature indeed is safe to execute – meaning the origin of the code is verifiable. Features could be downloaded via *http*, *ftp* or other internet protocols viable for service attacks such as *IP spoofing* [17]. A mechanism which ensures the executed feature indeed originates from the correct source is thus needed. This could be done by signing the features with digital signatures, and check these before executing the code. Such an approach is possible in Java by signing classes.

Minimizing the interdependencies between the state machine and the submachine is done through proper encapsulation of the component. This is addressed in Section 4.4. Furthermore, to ensure a new component does not introduce unwanted behavior into the system, one needs to decide

how and when a replacement is to be done and how the behavior should be verified. This is addressed in Section 4.5 and Section 4.6 respectively.

4.4 Encapsulating the submachine component

Entry and exit points serves the role of encapsulating the submachines state substructure and makes it a reusable component. As described in Section 2.5.1, state machines are not allowed to perform direct boundary crossing from the submachine state into a submachine. This way, the entry and exit points explicitly states how a submachine is to be entered and exited by its environment.

However, exit and entry points alone does not ensure behavioral conformity when replacing a submachine at run-time. The new submachine may indeed turn out to be incompatible as it may breach the protocol adherence as specified in the environment in which it is deployed. The submachine may very well be found to introduce deadlocks, livelocks or non-deterministic behavior to the encapsulating state machine or to state machines with which it communicates. I will in the remainder of this section establish some guidelines for how to deal with the following encapsulation issues:

- How to encapsulate instance variables?
- How to handle timers created by submachines?
- How to deal with ports and messages?

4.4.1 Instance variables

UML allows modeling of submachines and composite states which are capable of reading and writing to instance variables defined by the encapsulating state machine. A newly introduced submachine may make the wrong assumptions about the instance variables of the enclosing state machine or have undesired effects upon these. Ensuring that such erroneous assumptions are not being made when designing the components thus requires a thorough investigation of these instance data interdependencies. When remotely downloading and binding submachines, allowing these components to freely access the instance data of the encapsulating state machine may not be wanted. [11] suggests assessing at which level the new submachine can be trusted to behave correctly, and select a security policy specifying to what extent the submachine can access its environments instance variables.

An encapsulating state machine is designed with a fixed set of instance variables. It is very likely that a newly and independently designed submachine needs to specify additional variables which was not specified by its encapsulating state machine. Allowing submachines to encapsulate its own set of instance variables should thus be allowed.

One could require that submachines which are to be replaced at run-time shall extend an already defined submachine. This enables the encapsulating state machine to know which instance variables to expect in this submachine, and possibly even access this data. However, the submachine is an independently designed component, and to minimize the interdependencies which exists between these two components, it would thus make sense to not allow such access. This becomes even more evident when we allow replacing the submachine at run-time. This would require mechanisms which are able to copy the instance variables expected by the enclosing state machine from the old submachine into the new. If a previously replaced submachine at some time made the wrong assessments of the state of these variables, the error would propagate to submachines which replaces this at a later time. *I therefore propose deleting the instance variables defined by an encapsulated submachine upon replacement, and to not allow the encapsulating state machine to access data of its submachines.*

4.4.2 Timers

Timers are often used as a technique to leverage problems where actions might be expected to cause a deadlock in a state machine – e.g., not getting a response from a request in a timely manner. A timer can thus be of vital importance when designing a new submachine. Timers are usually specified in the enclosing state machine environment, and allowing submachines to create and start such new timers thus raises a problem; if the submachine starts a timer and exits before this timer expires, the enclosing state machine would receive an unspecified timer event. Such timer events could be ignored, as the enclosing state machine does not know how to handle it. However, it would be a good design guideline to stop all timers which are not stopped upon exiting the submachine.

4.4.3 Ports and messages

A state machine sends and receives messages on the ports defined by its encapsulating classifier. When dealing with dynamic pluggable submachines, it is expected that these components may need additional ports to communicate with other classifiers which was not thought of at the time of creation of the classifier. However, [8] states that *"A port cannot be created or destroyed except as part of the creation or destruction of the owning classifier"*. If the state machine realizing the behavior of the classifier adapts new functionality at run-time, there may be situations where this would require new ports.

This can be shown in Figure 4.1 where the classifier *PhoneBookService* requires the provided ports of *SMSEdge* and *CallEdge*. Imagine the port which connects to *CallEdge* was not specified at the creation time of the *PhoneBookService*. This would make it impossible to insert the *SearchFeatureCallSM* submachine, as it is not allowed to communicate with *CallEdge*. It may therefore appear to be reasonable to allow loadable submachines to define and create new ports in addition to the already existing ones.

[11] proposes differentiating between dynamically and statically created ports, where submachines could be allowed to create ports upon instantiation and destroy them upon destruction. Such an approach would require that these ports are able to create connectors to other ports dynamically. This raises the issue that messages might be sent over these ports while the submachine is not part of the active state configuration – i.e., the enclosing state machine would receive an unspecified event which might lead to undesired effects. This might be alleviated by ignoring messages sent through a dynamic port while the submachine that created them is not part of the active state configuration. Furthermore, [11] suggests the use of security policies to specify which messages submachines are allowed to send and to what extent they are allowed to create new ports.

4.5 How and when to download and replace a submachine

There needs to be a mechanism which is able to download a submachine component and to decide whether the current state is appropriate to replace the existing submachine. The most flexible approach of doing this is to implement the functionality for downloading and binding submachines within the state machine based on reception of an *update* message. Such an update message should specify which submachine the new submachine component shall be replacing, and could also have additional attributes which instructs the state machine on how and where to find the new component. This would give us the possibility of doing the correct assessments on whether the current state of the state machine allows such behavior. However, this adds the additional concern of deciding which of the communicating peers should be aware of such updates.

When receiving an update message, a state machine may be in a non-suitable state for updates. Lets consider the case where the adversary state machines are aware of the update, and maybe even depending on it. In such cases it might be of utter importance to reach a state safe for updating before the update can take place. This may be done by leaving the current state abruptly, saving the current state, execute the update and maybe returning to the previous state. As these adversaries are aware of the update, they may expect the change of behavior by the state machine. On the other hand, peers not aware of such updates should not be able to observe any externally visible change of behavior by the state machine – i.e., updates needs to be atomic. These two requirements are met by not allowing a submachine to be replaced while part of the active state configuration. *I therefore propose to defer update messages until this condition is met and send an acknowledgment to the update-aware state machine when the replacement has been done*. An example of how this could be done is shown in Figure 4.4. The figure shows a state machine which accepts an update message, *SubMachineUpdate*, in any state and defers this message until the requirements are met.

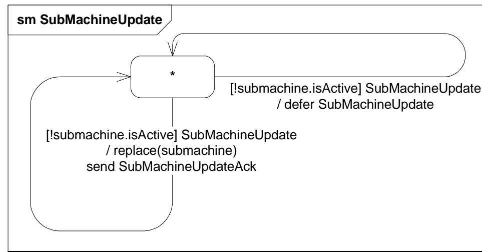

Figure 4.4: Save submachine update messages until requirements are met.

4.6 Ensuring behavioral conformity

When replacing a submachine at run-time, one wants to make sure that this does not introduce ill effects upon the system in which it shall be deployed. There exists several approaches to achieve this. Anomalous behavior between communicating state machines can be detected using the *projection* approach described in [7]. Such a projection is an abstraction technique, as described by [7]: *"A projection is a simplified system description or viewpoint that emphasises some of the system properties while hiding others."*. The validation is done on these projections rather than the whole system, and thus simplifies the designers work. However, the practical use of this method is yet to be established in any real examples.

Furthermore, [11] suggests the use of security policies which specifies what the submachine is allowed to do in the classifier it is deployed. One approach to enforce such guidelines could be by *sandboxing* the submachine. The submachine component would thus have its own environment which explicitly states the effects the submachine is allowed to have upon the system. Although such an approach may very well turn out useful, it is hard to formally verify the effects such a *sandbox* shall have.

One of the most important properties of using component based development is the possibility of being able to understand the relevant parts of a state machines behavior by examining its required and provided interfaces. When using UML 2.0 classifiers, this would involve observing the signal interleaving over the ports which are defined at its edges. One can thus decompose the external visible behavior into several different protocols which the state machine must adhere to. This does not conflict with the use of replaceable submachines, as the submachine shall still adhere to the protocols defined for its communicating adversaries. However, this approach can not cover all the failure scenarios. A submachine may still have internal failures – e.g., it may use inappropriate operations on its own structured classifier. On the other hand, it does alleviate the problem of a failing part having ill effects upon other parts the system.

Figure 4.5 shows the decomposed protocol which exists between *PhoneBookService* and *UserTerminal*. This protocol could be asserted by both *PhoneBookService* and *UserTerminal* at run-time, thus both parties makes sure the other state machine adheres to the protocol. *I therefore propose using* ports *and* protocol state machines*, an* event acceptor*, as a method of asserting*

behavioral conformity *when allowing remotely downloadable and replaceable submachine components.*

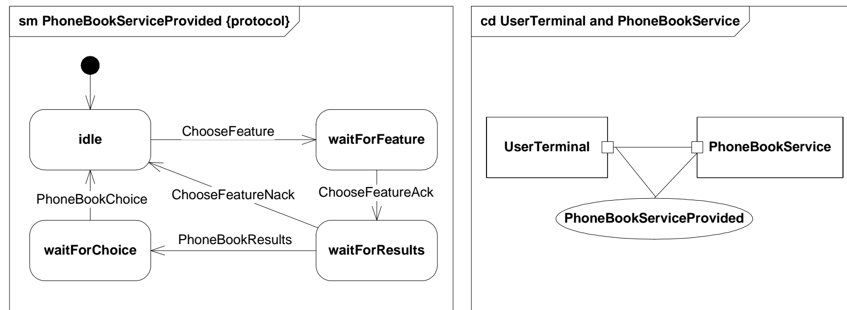

Figure 4.5: Protocol for the provided interface of *PhoneBookService*.

4.7 Summary

This chapter first introduced an example telecom service and established how this service could benefit from dynamic loading of remotely downloaded telecom features. Furthermore, the chapter found the following for remotely downloaded submachines:

- Exit and entry points is not enough to ensure behavioral compatibility.
- Instance variables should be handled carefully.
- Timers started by a submachine should be stopped upon exit.
- It may be beneficial to allow dynamic creation of ports.
- The external visible properties may be decomposed and asserted through the use of ports and protocol state machines.

In the next chapter I will present conceptual approaches to using ports and protocol state machines as an *event acceptor*.

Chapter 5

Using ports and protocol state machines as run-time event acceptors

5.1 Introduction

Section 4.6 introduces the idea of ports and protocol state machines as an appropriate approach to assert whether state machines which are able to adapt behavior during run-time adheres to the protocols as defined by its provided and required interfaces. Such an approach would also have the additional benefit of being able to protect critical resources in the system, and thus resembles a sort of statefull application level firewall. This is of great advantage when allowing third-party service providers and developers to access a system. Architectural flaws can be discovered in the running components of the system, as could misuse of critical system components. This chapter will hence show a conceptual approach to enabling such a run-time event acceptor.

> *"In component-based software engineering, a basis for reasoning on behavioral compliance is highly desirable in order to validate software architectures and to reason on component compatibility."* [10]

5.2 Goals of the event acceptor

The main goal of the *event acceptor* is to capture the message interleaving between two adversary state machines and assert its legality at run-time. By decomposing the behavior of each state machine into a set of provided and required interfaces, one can thus assert the behavior of the state machine in a given context. A state machine may have many different adversary state machines, each communicating over different ports and connectors. By asserting the message interleaving in all these specified contexts, one is thus able to verify whether the externally relevant properties of the state machine is correct.

Figure 5.1 shows the interleaving of messages over the assembly connector between *portA* and *portB*. This interleaving is constrained by *protocolAB* – a protocol state machine. Doing such

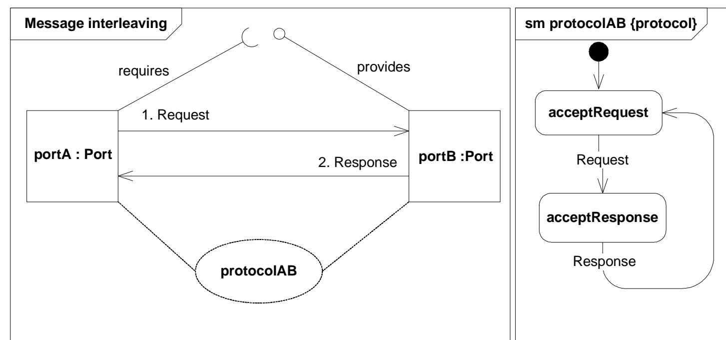

Figure 5.1: Event acceptor and message interleaving (Non-normative UML).

assertion at run-time, one wants to make sure that both state machines involved with this context adheres to the given protocol. As the two state machines involved utilizes ports to communicate, the message interleaving shall thus be asserted at these interaction points.

5.3 Ports

When ports shall be created as part of the structured classifier which owns it, there needs to be mechanisms which are able to do the following:

- Unique and global addressing of ports and state machines.
- Routing, sending and replying to messages between the connectors.
- Configure the ports with connectors upon creation.

These issues will be discussed in the remainder of this section.

5.3.1 Connectors and addressing of ports and state machines

Naming is of critical importance when it comes to being able to address a part's port and state machine uniquely throughout the system. A behavior port must be able to address the state machine to which it is connected. In addition, ports should be able to address other ports through the use of delegation and assembly connectors. One thus needs an addressing scheme which can handle addressing both state machines and ports.

In UML 2.0 a port can be given a name and a structured classifier can be given a type. Furthermore, when a structured classifier shall be an inner part of another structured classifier, it has an instance name. It is thus possible to create a naming scheme which qualifies addresses with this information. This naming scheme is based on the existing ActorAddress scheme from ActorFrame, and the introduced addition of port names makes it possible to address both state

Port	Direction	In address	Out address
bOut	out	–	/a:aOut@A
aOut	out	–	/d:dInOut@D
cIn	in	/c@C	–
dInOut	in-out	/d/e:eIn@E	/c:/cIn@C
eIn	in	/d/e@E	–

Table 5.1: Connector addresses described in the ports.

machines and ports. *I thus propose adding the port name as an addressing qualifier to the existing ActorAddress scheme. Furtermore, I propose the following textual representation of this addressing scheme*:

$$\lfloor /contextinstance \rfloor_{multipleopt} /instance \lfloor : port \rfloor_{opt} @Type$$

Summarized, the following invariants apply to the addressing scheme:

- Each port name must be unique within the structured classifier.
- If there exists more than one part of the same type within a structured classifier, the instance name must be unique.

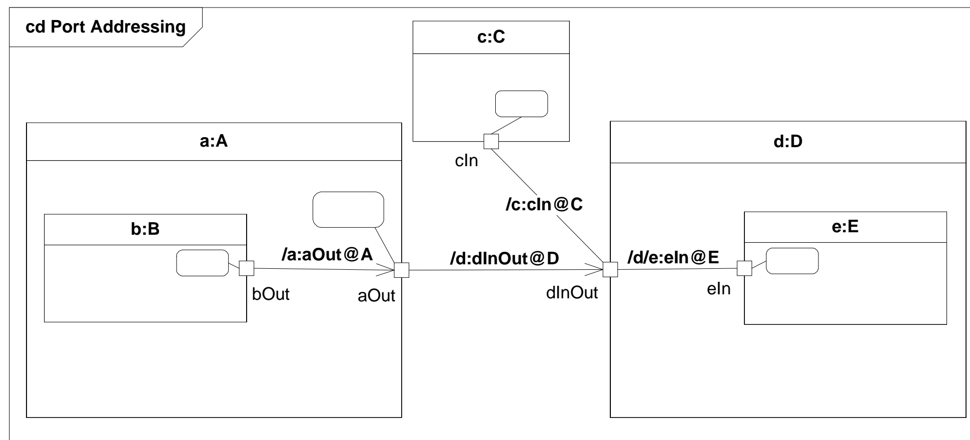

Figure 5.2: Port addressing example.

Figure 5.2 shows an example class diagram involving five different parts. As seen in this figure, it is necessary for each port to distinguish between *in* and *out* addresses. This is due to the fact that we allow ports to have connectors both in and out of the part for which it is defined. Following Figure 5.2, a configured port shall end up having the connector addresses as described in Table 5.1.

5.3.2 Routing signals between connectors

When a signals is received by a port, one needs to define how this signals is to be handled – i.e., should it be delivered to the state machine or be forwarded through one of its configured connectors. UML 2.0 does not explicitly define how this should be implemented.

Using the above connector addressing approach, such a task has a trivial solution. As each port is limited to have one connector for each direction, I thus propose using the *direction* in which the signal was sent to decide where the port should forward the signal. Signals arriving on a port which was sent by the behavior of the classifier, or through a connector from an inner part, should forward this signal to the *out* address. Likewise, signals which were sent by a connector owned by the enclosing structured part are forwarded to the behavior or through a connector to an inner part.

Determining the direction in which a signal was sent amounts to comparing the address of the sender with the address of the port. As the proposed addressing scheme has context information embedded in its structure, it is thus possible to evaluate whether an actor is an inner part of a structured part. *I thus propose evaluating the direction of a sent signal by comparing the context part of the senders address with the address of the part which owns the port.*

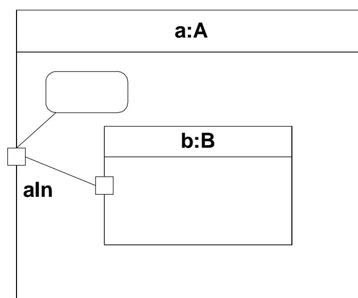

Figure 5.3: Inner classifier not able to signal enclosing classifier.

Using this approach it is thus possible to determine over which connector the signal should be conveyed. However, the approach raises another problem. Figure 5.3 shows a scenario where the inner part *b:B* provides an interface to the enclosing part's *a:A* behavior port *aIn*. With the above described propagation technique, signals across such connectors can not be sent in a proper manner. The above signal forwarding mechanism would decide that the signal should be sent on the *out* address – which was clearly not the intention.

A partial solution to this problem is thus to explicitly define whether the port has connectors *in* to the part, *out* of the part or *both*. In the scenario shown in Figure 5.3, it would thus be possible to specify that the port *aIn* only connects in to the part. With this information it is thus possible to forward the signal to the *in address* regardless of the direction which the signal was sent. Section 5.3.3 describes the complete algorithm for signal routing solving this problem.

5.3.3 Replying to signals

A part providing a service, such as the *SMSEdge* described in Section 4.2, often has many clients connected to its port. Such types of services often handle request signals without caring who actually sent it. When the request has been processed, it is often needed to convey a response which describes the successfulness of the request. There should thus be a mechanism which is able to route the signal back to its originator in such protocol scenarios. There exists two solutions to this:

- Send reply directly to the requester.
- Send the reply through the path of connectors which the request arrived through.

By sending the reply directly to the requester, the signal is guaranteed to arrive correctly. However, this solution has the disadvantage of making it harder for the event acceptor to assert whether the reply adheres to its protocol. An erroneous reply could be sent which violates the protocol of the requester. Furthermore, this violates the use of assembly connectors.

A better solution is thus to send reply signals through the same path of connectors which the request signal arrived on. This makes it tangible to assert whether the reply adheres to the protocol between the two communicating state machines. I propose two different solutions for doing this:

- *Statically* define reply paths.
- *Dynamically* define the reply path.

A *statically* defined reply path involves having the routing information embedded in the ports – thus resembling IPv4 routing [12]. This is in contrast with the *dynamically* defined reply path, which resembles the IPv6 routing [12] in that it has the possibility to embed routing information in the header of a package.

Static reply path

One approach to ensure replies are being sent through the necessary ports, is to create a protocol which sets up such a reply path at creation time of the port.

A port which requests a connector to another port sends a request along the configured path of connectors. While this request is being passed from port to port along the path, the ports which are interested in asserting the communication between the two classifiers adds its address to a list in the request message. This list specifies which port addresses a reply shall be sent through.

This is shown in Figure 5.4, where the only port interested in asserting the communications between *portA* and *portD* in a reply situation is *portB*. When portB receives the *PathRequest* it adds the address of the port to a stack in the message. When *portD* receives the message, and

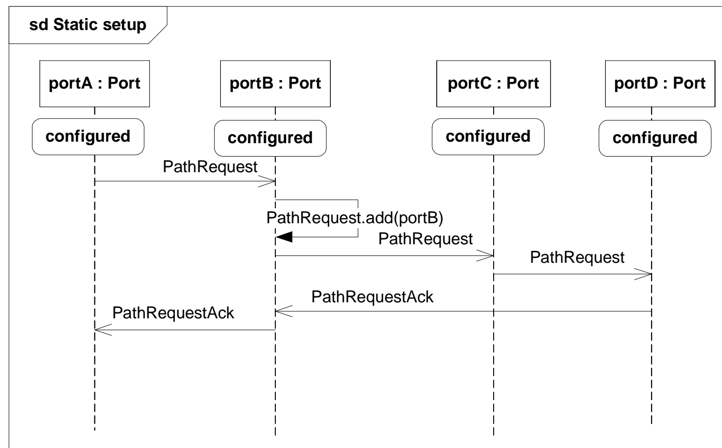

Figure 5.4: Static setup of reply paths.

verifies that it is a behavior port, an acknowledgment is sent back to *portB*. This would thus provide *portB* with sufficient information to create a static map which specifies that messages sent from *portD* shall be forwarded to *portA*. However, this approach is probably not fail safe, as such static maps might be conflicting if there exists other clients which uses the same path of connectors.

Dynamic reply path

The *static reply path* approach described above is unnecessary, and furthermore adds complexity to the protocols which shall be used for setting up the connectors between ports. Another approach to doing this, is thus to tag messages with the port addresses which it passed on the way to its final destination. This tagging could be implemented as a stack, and an algorithm could be created which supports replying through the use of this stack.

Figure 5.5 describes how the message is handled by a port upon reception. If the last element in the reply stack is not equal to the ports address, it is not in reply to a previous message. The port should thus add its address to the reply stack in the message. The receiver of this message is hence set according to the direction in which the message was sent. If the message was sent by an inner classifier, the port should forward the message to the *out address* connector. If the message was sent by an outer classifier, the port should send the message to the *in address* connector or to the behavior of the part.

Furthermore, messages which are received by a port first checks if the last element in the reply stack is equal to the address of the port. If this is so, the last sender of this message was this port – meaning it is in reply to an earlier sent message. This last element is removed from the stack, and the new last element is thus the address of the port which should receive the message.

A state machine which wants to send reply to a message would thus only need to copy the

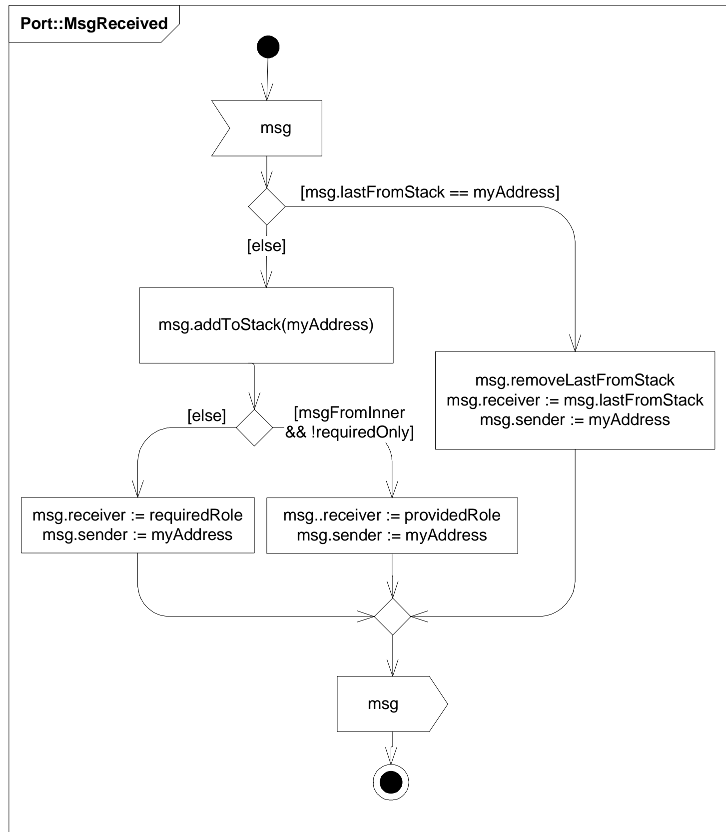

Figure 5.5: Sending messages with a dynamically created reply path.

reply stack specified by the request message into the response message. *I thus propose using the routing algorithm as shown in Listing 5.1.*

5.3.4 Configuring connectors at creation time of a part

We now have the necessary routing information for addressing and sending messages between ports. However, these are all useless unless there exists mechanisms which are able to specify and configure these properties at instantiation time of the part.

Upon instantiation of a structured classifier, the ports associated with it should be created. This structured classifier is a part of another structured classifier, and in this encapsulating context the connectors of a part's ports are specified. As a port in some situations may require a connector to another non-local port across a distributed system, it thus makes sense to use signaling to set up connectors.

This approach requires that the ports must implement a protocol for such signaling. I propose the following principal behavior of this protocol:

Listing 5.1: Message routing algorithm described using Python.

```
def route(sig):
  if sig.replystack.last == myAddress:
    # last sender on stack is me, reply
    sig.replystack.remove(sig.replystack.last)
    sig.receiver = sig.replystack.last
    sig.sender = myAddress
    send(sig)
  else:
    # not a reply message
    if myAddress.isInnerActor(sig.sender) and !isInPort:
       sig.receiver = outRole
    else:
       sig.receiver = inRole

    sig.replystack.add(myAddress)
    sig.sender = myAddress
    send(sig)
```

> *A port which wants to set up a connector to another port shall send a request to this port. A port receiving such a request shall send a confirmation to this request back to the requester. Upon reception of a confirmation, the connector address is set according to direction of the received signal and address of the sender.*

A path of assembly and delegation connectors should always end with a port which is a behavior port. The approach taken here is that each part is responsible for setting up the connectors for its ports – i.e., they have no further knowledge of what connectors the port they connect to specifies. It is thus necessary to have a mechanism which ensures that the path of connectors are properly set up before signals can be conveyed across them.

This scenario is shown in Figure 5.6, where *a* has not yet set-up its connectors to *c*. As *b* is already connected, it might send a message which was destined for *c*. However, as part of the connector path is not configured at this time, the message would not arrive to the correct destination.

A solution to this problem is thus to make sure ports which define out connectors shall not send messages until the whole path has been acknowledged. This is done by signaling a request through the path upon confirmation of the connector which it requested. This request propagates through the path until it is received by a behavior port. The behavior port sends a confirmation back to the requester, which hence can allow messages to be sent.

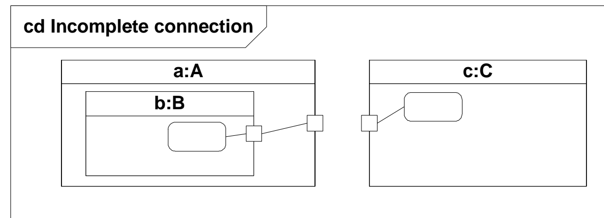

Figure 5.6: Incomplete connection configuration.

5.3.5 Limitations of the approach

Ports can have connectors to other ports and be a behavior port. Supporting hiding of the inner structures of other classifiers presents some problems which needs to be addressed. Such hiding of structure prevents connecting classifiers from seeing the *whole* picture, and the ports are thus to be viewed as *black-boxes* to these classifiers. This difference is shown in Figure 5.7.

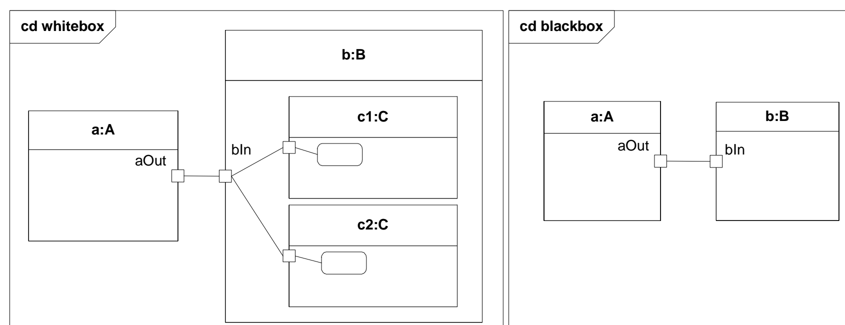

Figure 5.7: Black-box versus white-box view of the structure.

As Figure 5.7 shows, it is impossible for the port *aOut* to know whether it is connected to the behavior of classifier *b:B*, *c1:C* or *c2:C*. This causes trouble when receiving messages at the port *bIn*, as it does not have sufficient information for deciding which instance *a:A* wants to address. UML 2.0 specifies that such multiplicity could be handled by either copying the signal which arrived to the port to all these connectors, or by selecting one of the connectors. In Figure 5.7 we have shown two inner parts of the same type, which thus has the same state space in their state machines. By duplicating the signal and sending it to both these parts, we thus have the risk that both will react to it – which in most cases would be regarded as an unwanted effect. It should therefore be possible to select one connector. However, with the lack of information it is hard to choose the correct route.

The above described approach to solving this problem is to explicitly limit the number of allowed connectors on each port to one in each direction of the part. However, this approach is not very flexible, as it would require one port to be specified for each inner part of the structured classifier. Furthermore, this effectively prevents creating new parts after the structured classifier has been created. This is because it would require creating new ports in the enclosing part after

its instantiation – which is not allowed by the UML 2.0 specification.

Extending the addressing scheme with interface names

The shortcoming of the previous addressing scheme with regards to multiplicity of delegation connectors is thus in need for some modifications. As UML 2.0 ports is allowed to specify named interfaces, a solution is thus to extend the previous scheme with the following added invariant:

- Interface names specified by a port shall be unique for the port.

This scheme thus makes it possible to route messages based on interface names. Figure 5.8 shows the context which qualifies the unique addressing of the required interface *bReq* which is specified for *bPort* in part *b:B*. Creating a new inner part of *a* would thus involve adding a new provided interface on *aPort* which is to be used by clients communicating with the new inner part.

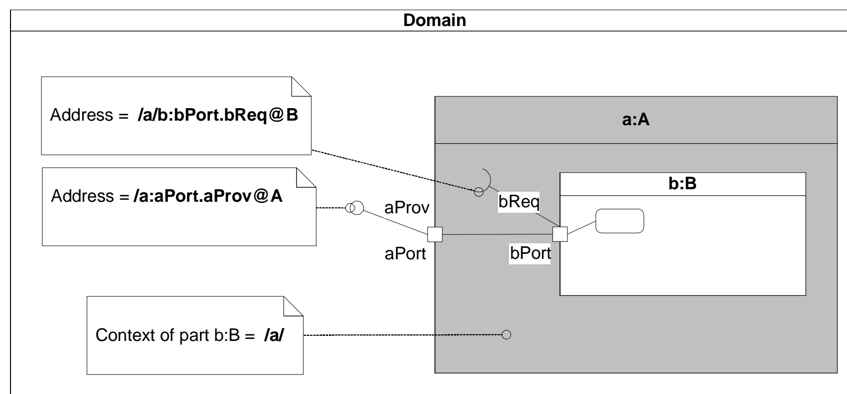

Figure 5.8: Addressing of ports and interfaces and their context.

This approach has not been investigated any further in this thesis, but it is clear that a solution based on this approach would be more appropriate and flexible in use.

5.4 Event acceptor execution

I propose the structure for the event acceptor as shown in Figure 5.9. When a signal is received by or sent through a port *Port*, the protocol state machine *PortSM* is executed with *execTrans()*. The result of this execution yields two possible outcomes; either the message is regarded legal or illegal by the protocol state machine. A message regarded as legal would thus be propagated to the correct connector by the algorithm described in Section 5.3.3.

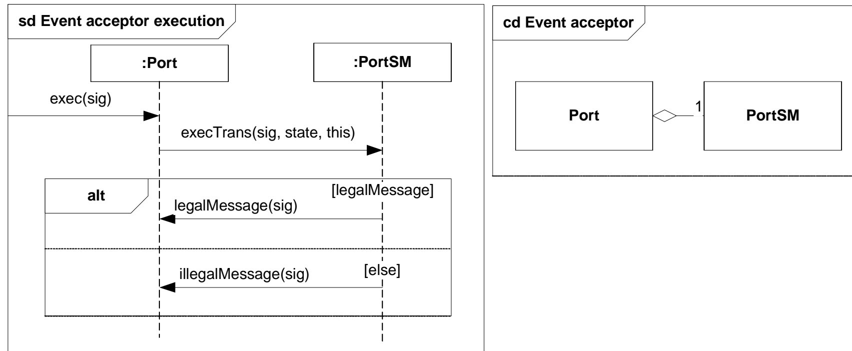

Figure 5.9: Execution of the event acceptor.

5.5 Event acceptor and multiple clients

It is necessary for protocol state machines to have state data to operate correctly. One single port instance could have many clients connected directly and indirectly to it. Figure 5.10 shows an example where two different clients are connected to the port *aPort*. The port *aPort* is a port with a delegation connector to the inner part *c*'s behavior port.

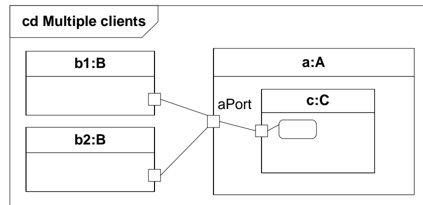

Figure 5.10: Two clients connected to one port.

At design time of the protocol state machine for the *aPort*, one could in this scenario probably foresee this, and thus make it understand that these are two different clients which needs to be taken care of accordingly. However, this is not very flexible if we were to extend this model to include new clients at run-time.

There is thus a need to differentiate between the clients which could send signals on a port. This would ease the design of such protocol state machines substantially, as each port is able to handle signals according to the identification of the client which sent it.

Enabling such port sessions in a framework supporting the port and protocol state machine concepts could be done in several ways. One method would be to create sessions on a *first message* basis. This requires less work in the port implementation, as one would only have to associate each message with a protocol state machine. If a message with unknown sender has been received, one would then initiate a new protocol state machine for this sender. Since we by using

Sender assoc. key	**Protocol state**
b1:B	@stateref1
b2:B	@stateref2

Table 5.2: Hash lookup table for state data.

this approach can not know if the actor has stopped existing, there is a need to have timers for how long each session is to live in the port. However, this method has several pitfalls. Even though it is easy to create such a session, deleting and keeping them alive is hard. It would solely rely on well defined timers which do not delete the associations before an actor has stopped sending on the port. Deleting too early would in effect reset the protocol state of this client, and if the protocol does not allow the new message, it is not allowed to be sent through the port.

Another way of doing this altogether, is to create a protocol between the ports which sets up these association along the path as they are created. Doing this would reserve a protocol state machine on each port along the path of connectors. This session approach would also make it possible to tear down the associations upon a clients destruction. Table 5.2 shows how this information is gathered in the *aPort*.

I thus propose that the port shall perform a hash lookup with the sender of the message as identifier. This lookup shall return the state of the protocol state machine associated with this session. The sender is found using the first element of the reply stack described in Section 5.3.3.

5.6 UML 2.0 and message interleaving

The event acceptor as proposed in this chapter violates the current UML 2.0 specification. This is because the protocol state machine shall only verify events received by a classifier. Translated to the example shown in Section 5.2, Figure 5.11 shows the protocols which matches this specification.

It should prove very useful to be able to assert the bi-directionality of events. To set this into perspective, assume a remotely downloaded submachine has made the wrong assumptions upon its environment. The encapsulating state machine has no control over what it is actually doing. The failing submachine is entered and is part of the active state configuration. A message is received which is expected to have a given effect. However, since assuming this component is ill behaving, it sends the wrong response using the reply mechanism described above. Figure 5.11 clearly shows that this message interleaving assertion would not be possible in this case, and the message should thus be allowed by the system.

This worst-case scenario shows that the current constraint is too restrictive. I thus suggest allowing UML 2.0 protocol state machines to both check the received event and the sent effect

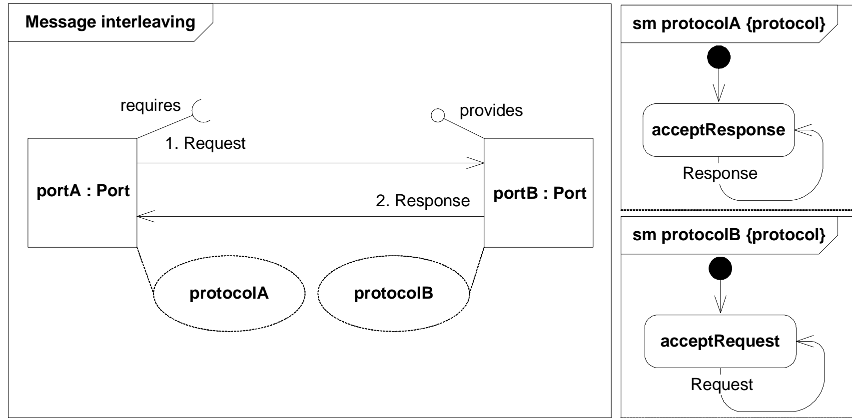

Figure 5.11: UML 2.0 and message interleaving (Left diagram: Non-normative UML).

of the fired transition in the state machine. However, the formalities and reasoning behind this current restriction has not been investigated any further in this thesis.

5.7 Structured classifiers violating protocols

We now have the conceptual tools to assert protocols between classifiers ports at run-time. It is thus necessary to decide what to do when one behavioral component *breaches* such a protocol. The UML 2.0 protocol state machine construct clearly specifies that it is not allowed to have effects. However, it becomes clear that such a restriction may be too strict when doing run-time assertion. A classifier which fails to adhere to a protocol may introduce deadlocks in adversary state machines. Although the wrongfully sent signal were not allowed through, adversary state machines may wait for other signals from this classifier – thus entering a deadlock which could potentially propagate throughout the system.

Such deadlock situations might be remedied with careful use of timers, but solely relying on this mechanism could lead to unreliable services. Thus it seems reasonable to allow protocol state machines to have effects when performing run-time assertions, even though this violates the UML 2.0 specification. As an example, TAPAS' [15] Plug-and-Play [9] system uses the countermeasures as shown in Table 5.3.

5.8 Updating the protocol specification at run-time

Allowing submachines to be replaced at run-time raises another dilemma. It is probable that the replacement of one submachine for another invalidates one or more of the currently specified protocols. For interfaces requiring the update to be invisible this would probably mean the submachine had unwanted effects upon the system – and thus is failing. However, there may exist cases where communicating state machines *agrees* upon the change of behavior – i.e., some state

No action	It is decided not to take any action.
Actor initialization	It is decided to install a new actor; this includes instantiation of a new actor, installation of the manuscript defining the behavior of the actor, and execution of the actor.
Actor termination	It is decided to terminate the actor freeing all resources allocated to and consumed by that actor.
Actor reinitialization	A sort of combination of *actor termination* and *actor initialization* that terminates the actor and reinitializes it on the same node
Actor relocation	It is decided to move the actor to a new node.
Play reconfiguration	It is decided to reconfigure the whole play. All actors involved in the play are influenced by this action. The best node for executing each role is computed, and the actors will be relocated to these positions.
Default	Relocation of all actors involved in the problem.

Table 5.3: TAPAS' Plug and Play countermeasures for failing actors.

machines are aware of the update taking place, and the change of behavior is expected by these adversary state machines. In such cases it may thus sound reasonable to allow updates to the protocol specified by a protocol state machine at run-time. Indeed, allowing such updates would make the system even more adaptable to behavioral changes.

A conceptual approach to do this may be as simple as sending signals to the port which should be updated – specifying where the new protocol can be found and which state this protocol state machine shall be in after installing the new protocol. Since all the state machines communicating over this port are aware of the update, signaling could continue as normal after the update has taken place.

5.9 Summary

I have in this chapter proposed solutions to addressing ports and state machines as well as sending and replying to signals between these. A solution to identify a sender's protocol state using sessions and the execution behavior of the event acceptor was proposed.

Furthermore, concerns regarding UML 2.0's restrictions on the protocol state machines were raised. The protocol state machine is neither allowed to verify signals sent by the classifier nor to have effects. I proposed that this clearly limits the usability of run-time event assertion, and the reasoning and formalities behind these restrictions should be further explored.

I will in the next chapter describe the implementation work done as part of this thesis. This implementation is based on the proposed solutions and approaches presented in this chapter and

Chapter 4.

Chapter 6

Run-time replacement of submachines in EJBActorFrame and EJBFrame

6.1 Introduction

Chapters 4 and 5 presents several issues and solutions to be considered when dealing with remotely downloaded and replaceable submachines. In this chapter I present the prototype modification done to the NorARC frameworks EJBActorFrame and EJBFrame to enable some of the proposed solutions.

6.2 What has been realized?

Modifying existing code-bases takes a lot of time, and the ideas presented in the previous chapter requires core changes within the NorARC frameworks. Not all the issues in the two previous chapters have thus been addressed.

With regards to replacing submachines at run-time, the following have been implemented:

- New SubMachine class which can encapsulate its own instance variables.
- Actor behavior for actors which wants to enable remotely downloadable and replaceable submachines.
- SubMachine replacement while not part of the active state configuration.

Furthermore, the run-time *event acceptor* has been implemented, with most ideas from Chapter 5 implemented:

- Port construct support for EJBActorFrame.
- Protocol state machine support for EJBActorFrame.

- Updated addressing scheme including support for Ports.
- Connector description and setup between Ports.
- Sending and replying to messages through Ports.

6.3 Modifications overview

Modifications have been made to both the EJBFrame and EJBActorFrame Java packages. An overview of the changes done to each of the packages is described below.

6.3.1 Modified EJBFrame Java package

Figure 6.1 shows the modified Java package of EJBFrame. As seen in this figure, changes have been made to almost all classes in the structure. No structural changes have been done to the package, but the new class *SubMachine* has been added. The new *SubMachine* class extends the existing *CompositeState* class, and implements a submachine which is able to encapsulate its own instance variables. The modifications done to these classes spans the following:

- Inception point for reception of *ActorMsg* to *Port*.
- Storing instance data for *SubMachine*.
- Support for new *ActorAddress* addressing scheme.
- Reply stack for the *ActorMsg*.

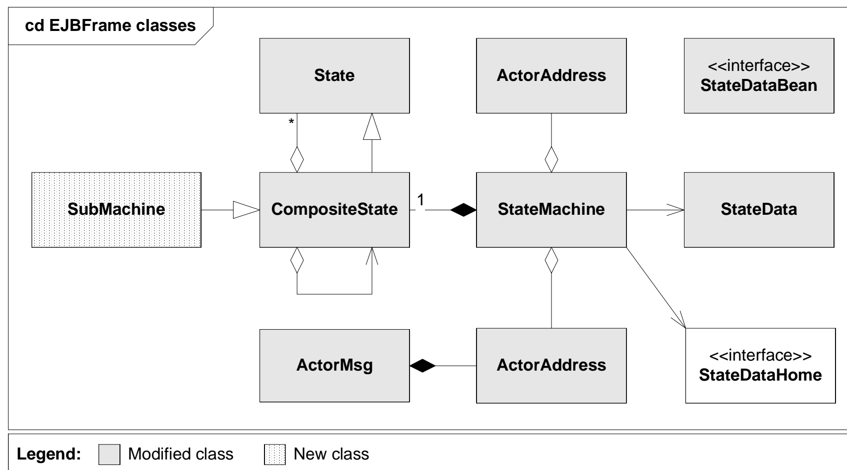

Figure 6.1: Modified EJBFrame Java package overview.

6.3.2 Modified EJBActorFrame Java package

Several changes have been done to the EJBActorFrame Java package. This is shown in Figure 6.2. Some structural changes in the package have taken place. Most notable is the relocation of the class *PortSpec* to be contained by the *ActorSM* class. This was done to accommodate creation of the new *Port* class.

Furthermore, three new classes has been added to the package. The *Port* class is the new port construct for EJBActorFrame, and is referenced in a Java Hashtable in the *ActorContext* class. The *Port* class have the new *PortSM* class attached to it, representing the protocol state machine implementation for the *Port*. The *PortSM* class extends the existing *RoleCS* class from Agent-Frame, which contains much of the needed structure for implementing a protocol state machine.

The new class *DynamicActorCS* class implements new behavior for downloading and replacing *SubMachine* classes at run-time. The actual downloading and Java classloading is performed by the utility class *FileClassLoader*.

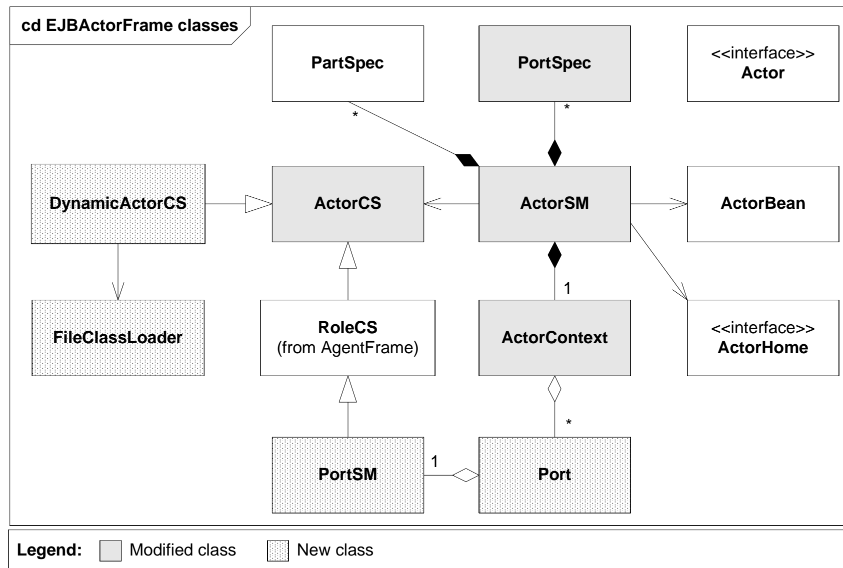

Figure 6.2: Modified EJBActorFrame Java package overview.

The following have been implemented in EJBActorFrame:

- Sending and replying to messages in *ActorSM*.
- Creation of *Port* instances using the *PortSpec*.
- Modified *ActorCS* for creation of *Port* and accommodated ActorFrame protocol to match.
- Run-time event acceptor with the *Port* and *PortSM* classes.

- Behavior for downloading and replacing *SubMachines* with *DynamicActorCS*.

I will in the remainder of this chapter show in greater detail the structural and behavioral changes done to the Java packages.

6.4 Implementation of SubMachine using EJBFrame CompositeState

EJBFrame lacked the UML 2.0 submachine construct as described in Section 2.5.1. The first step was thus to map this construct to EJBFrame.

A submachine is a state machine referenced by another state machine. The natural way of mapping this would thus be to reference a *StateMachine* object to create submachines. The *StateMachine* class in EJBFrame contains functionality such as message queues, state machine execution and J2EE specific details, while *CompositeState* is used to specify transitions and arbitrarily deep state configurations. As the submachine shall be a part of a state machine, one can thus use the already existing queue and execution handling as implemented by the *StateMachine* class. As the *CompositeState* already implements most of the necessary properties and behavior of a submachine, it thus makes sense to extend this class when creating the *SubMachine*. In fact, the EJBFrame *CompositeState* is very close to the submachine construct – it is not allowed to cross its boundaries without using entry and exit points. The main difference between the two constructs is that we shall allow additional instance variables to be stored as part of the new *SubMachine* class.

6.4.1 Structural modifications

Figure 6.3 shows attribute and operation additions to the EJBFrame classes *State*, *StateMachine* *StateDataBean* and *StateData*. These additions are done to make storage, initialization, saving and resurrection of state data possible. The *StateMachine* class is modified by adding a new Java Hashtable. This Hashtable is marked as persistent, and is thus saved and restored as part of the *StateMachine*. To uniquely identify what data belongs to which *State* object, the full state name is inserted as the key.

The implementation of the specified operations by the class *State* are left empty by default. This is done as it is most likely that classes in the class-hierarchy above the *SubMachine* do not need this feature. The addition of these methods would thus not create any significant computational overhead.

Classes which extends *State* and wants to save persistent data in the *StateMachine* should override the specified operations. As the *SubMachine* class is such an inherited class and wants to use this feature, we thus override these operations. Classes extending the *SubMachine* class must call

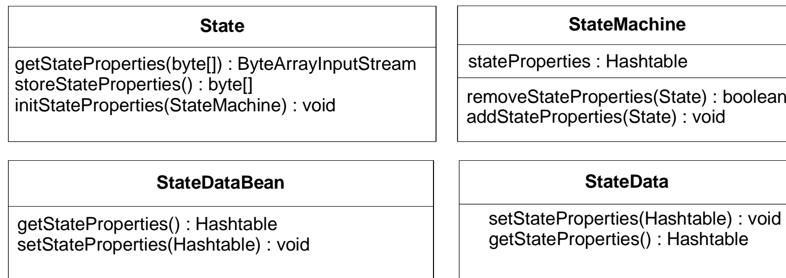

Figure 6.3: Modifications done to State and StateMachine.

the super implementation of the method to ensure that variables specified by the inherited *SubMachine* is stored properly. This is shown in Figure 6.4, where *GPSPositioning* adds an attribute *gpsPosition*. Using the described model, it has thus been assured that both the *gsmPosition* and *gpsPosition* instance data is saved for *GPSPositioning* objects.

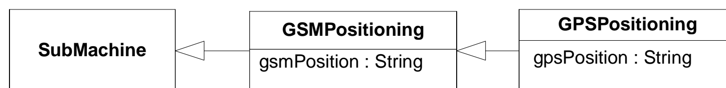

Figure 6.4: Classes inherited from SubMachine.

6.4.2 Behavioral modifications

The structural requirements of being able to save data for classes inheriting *State* are now met. Storing and resurrecting this data needs to be done inside the *StateMachine*, before and after the StateMachine executes a transition. Figure 6.5 shows the sequence diagram specifying the with transition execution of a StateMachine, with the added functionality of resurrecting and storing data with the reference to the *GetStateProperties* and *StoreStateProperties* sequence diagrams.

Figure 6.6 shows how the instance data is restored to the *SubMachine* before a transition on it is fired. As seen, this is done by fetching the byte array from the *stateProperties* Hashtable with the key of the current enclosing state's fully qualified state name. If this lookup turns out empty, it has not been entered before, and the initial values is thus created by calling *initStateProperties*. Otherwise a call to *getStateProperties* is performed, which is responsible for resurrection of the instance data.

Figure 6.6 shows how the instance data is stored after transition has ended in the *StoreStateProperties* sequence diagram. This is done by calling *storeStateProperties* of the enclosing state. If there exists data in the byte array returned, it is stored in the *stateProperties* Hashtable.

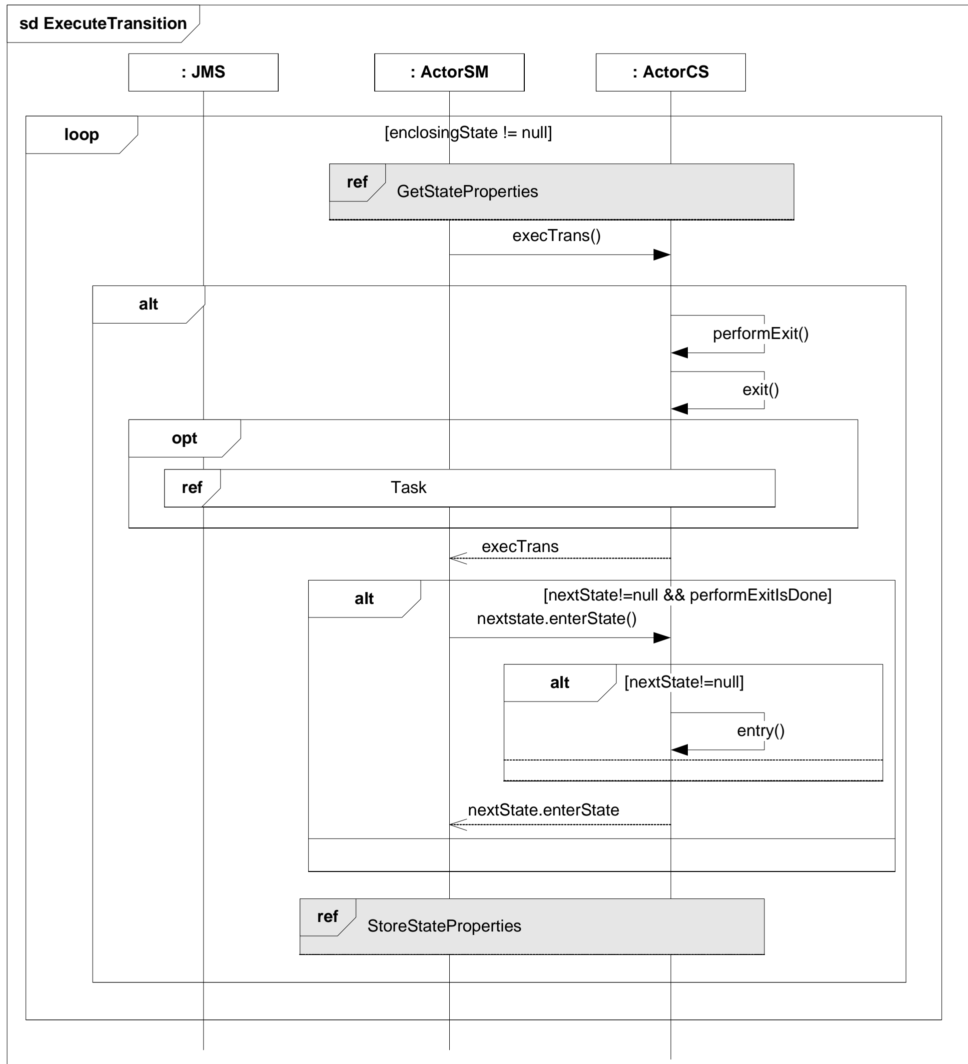

Figure 6.5: Modified transition execution.

6.5 DynamicActorCS – actor behavior for dynamic submachine loading

As shown in Figure 6.2, a new class *DynamicActorCS* was implemented by extending the existing *ActorCS* class. This class is intended to be used by actors that wants to enable remotely downloadable and possibly replaceable *SubMachines* at run-time.

Figure 6.7 shows the steps taken upon reception of *SubMachineUpdateMsg*. The first step involves searching through the list of *State* children contained by the *ActorCS*. If the *SubMachine* state name is found in this list, processing continues with getting the existing reference to this *State*. A search is then performed to check whether the *SubMachine* which should be replaced is

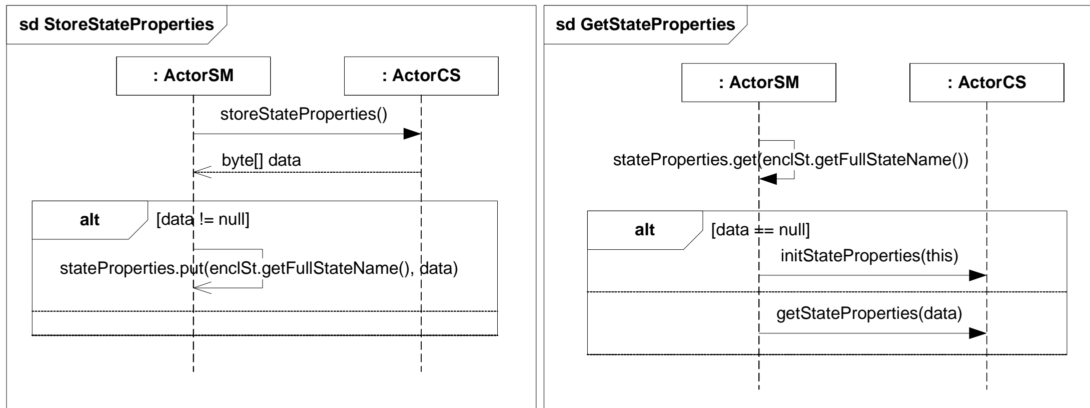

Figure 6.6: Sequence diagrams for *GetStateProperties* and *StoreStateProperties*.

part of the active state configuration. Processing then continues by creating a new *SubMachine* with the configuration specified by the *SubMachineUpdateMsg*. If the class loading is successful, the submachine state reference is replaced with new a *SubMachine* object. As seen in the figure, this implementation saves the signal if the current active state is unappropriate for updating the *SubMachine*. If any of the steps fails, *DynamicActorCS* sends a *SubMachineUpdateNackMsg* message to the requester of the update. These steps are described more thoroughly in the next sections.

6.5.1 Messages

The messages used to communicate with *DynamicActorCS* are all messages which inherits the EJBFrame *ActorMsg* class, as shown in Figure 6.8. Table 6.1 describes what each of these messages are used for. The *SubMachineUpdateMsg* is the message which is used to invoke replacement of the remotely downloaded or locally defined Java class of *SubMachine*.

The most important properties of the *SubMachineUpdateMsg* message are:

- *stateName*: State name of the SubMachine to be replaced.
- *className*: Java class name of the SubMachine which shall replace the existing.
- *classRoot*: Where the Java class is located, either locally or remotely.

6.5.2 Getting the reference of the SubMachine to be replaced

The *CompositeState* class contains a Hashtable with state names and references to *State* instances specified within the structure of a composite state. By sending the state name of the state which shall be replaced with the *SubMachineUpdateMsg*, one can thus perform a hash-lookup on the *children* list and get this reference by its state name.

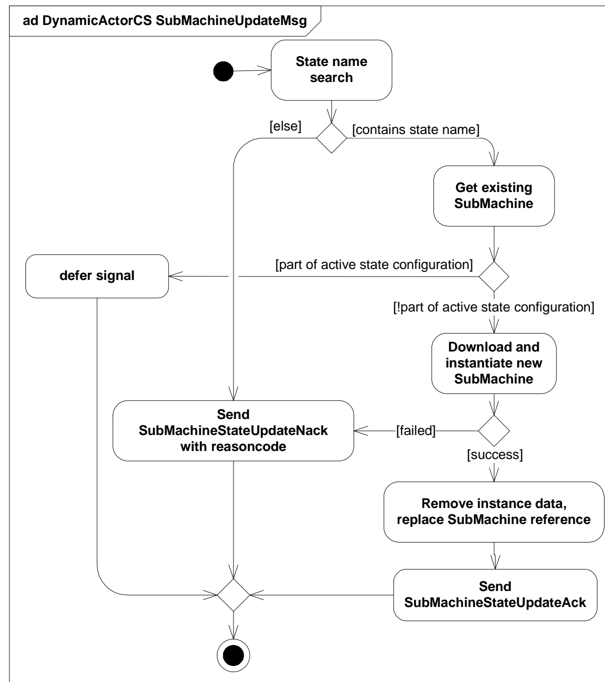

Figure 6.7: Activity diagram showing the submachine replacement steps.

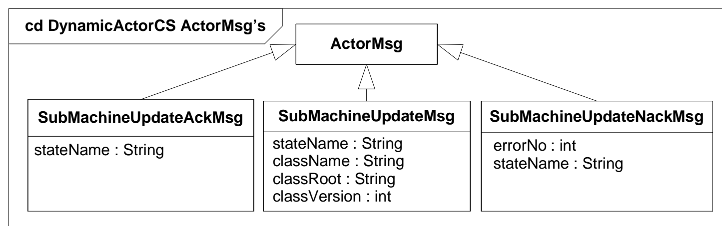

Figure 6.8: ActorMsg's for *DynamicActorCS*.

6.5.3 Searching the active state configuration

The EJBFrame class *State*, and thus also extended classes, contains an attribute *enclosingState* which is a reference to the enclosing state of a *State* object – as shown in Figure 6.9. An operation *isEnclosing* was added to perform a reversed, recursive search of the active state configuration – checking if each of these enclosing states is a reference to the *SubMachine* object which shall be replaced.

The *isEnclosing* operation works as shown in Figure 6.10. The currently active state in EJBFrame is always a simple *State* object, denoted *currentState* in the figure. By checking whether the reference to *enclosingState* is equal to the *replaceSM*, we can thus check if the *SubMachine* object to be replaced is part of the active state configuration. As the figure shows, the *SubMachine* object to be replaced in this example is not part of the active state configuration, and the operation

SubMachineUpdateMsg	Message specifying where to find the new SubMachine component and which state it shall replace. Initiates the replacement.
SubMachineUpdateAckMsg	Message sent by actor if the update was successful.
SubMachineUpdateNackMsg	Message sent by actor if an error occurred. Appropriate error code set.

Table 6.1: Messages used by *DynamicActorCS*.

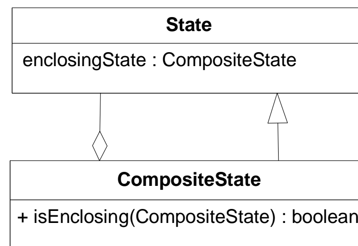

Figure 6.9: Classes CompositeState and State with partial operations and attributes.

thus returns false.

6.5.4 Remotely download and initialize a SubMachine

The initialization of a new *SubMachine* object is done by using standard Java classloading features. A new class *FileClassLoader* was implemented which tries to find Java classes in the following order:

1. Get class from local package.
2. Get class from remote using Java URL.
3. Get class from local filesystem.

The Java Class returned by this utility class is hence used to create a Java Object using the default constructor. This Object is thereafter casted into the *SubMachine* object which is to be used. The code for this is shown in Listing 6.1.

6.5.5 Replacing the existing SubMachine

Replacing the existing *SubMachine* object is done in three steps:

1. Remove the instance variables used by the old *SubMachine* object.
2. Update the environment of the new *SubMachine* object.
3. Update the references in the *children* Hashtable.

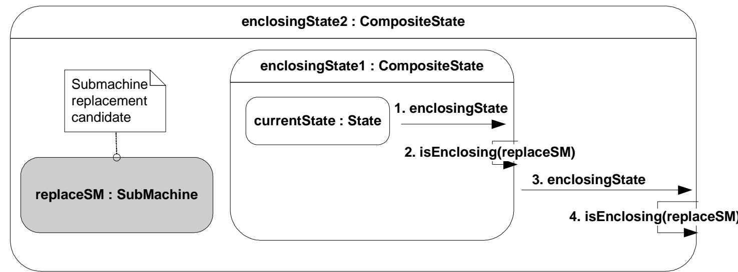

Figure 6.10: Searching active state configuration (non-normative UML).

Listing 6.1: Remotely download and initialize a new *SubMachine*.

```
FileClassLoader cLoader = new FileClassLoader();
Class cls = cLoader.loadClass(amsg.getClassRoot(),
                               className,
                               false);
Object obj = cls.newInstance();
SubMachine cs = (SubMachine)obj;
```

With the creation of the *SubMachine* component in Section 6.4, we have already defined behavior such that if the instance variables is not saved anywhere for the submachine, it will be initialized and stored properly when entering and leaving the submachine. All that needs to be done is thus to remove the instance variables as defined by the *SubMachine* object which is to be replaced. This is done with a call to the *StateMachine* object's *removeStateData* with the fully qualified state name of the *SubMachine*.

The new *SubMachine* object must be placed in the correct state structure, and must thus define its enclosing state environment. This is needed when the submachine shall exit through any of its submachine state's exit points. This is done by setting the objects *enclosingState* to reference the *DynamicActorCS* object.

All that remains for replacing a *SubMachine* is hence to update the *children* Hashtable with the new *SubMachine* instance. However, this limits the possibility of replacing *SubMachine* objects which are part of deeper nested *SubMachine* or *CompositeState* structures. As the *children* Hashtable has no knowledge of these substructures, this would require a different approach all together. The code for this, and the above steps, is shown in Listing 6.2.

6.6 Run-time event acceptor implementation

Implementation of the run-time *event acceptor* is based on solutions proposed in Chapter 5. Smaller changes have been done in the NorARC frameworks, but the overall operation of the event acceptor is as described in this section.

Listing 6.2: Replacing a SubMachine.

```
// remove instance variables
curfsm.removeStateProperties(updateCS);

// set enclosing environment
subMachine.enclosingState = this;

// update the references
this.children.remove(subMachineStateName);
this.children.put(subMachineStateName, subMachine);
```

6.6.1 Class overview

Figure 6.11 shows the event acceptor classes with an extract of the attributes and operations. As seen in the figure, the event acceptor is stored as part of an Actors ActorContext. An Actor can have multiple defined ports, and this is stored in a Hashtable in the ActorContext. One *Port* instance contains exactly one *PortSM* instance. The PortSM class represents the protocol state machine, and is responsible for allowing or disallowing messages. The Port class represents the port construct. The Port implements the ActorFrame protocol, routes and sends of signals and executes the protocol state machine *PortSM*.

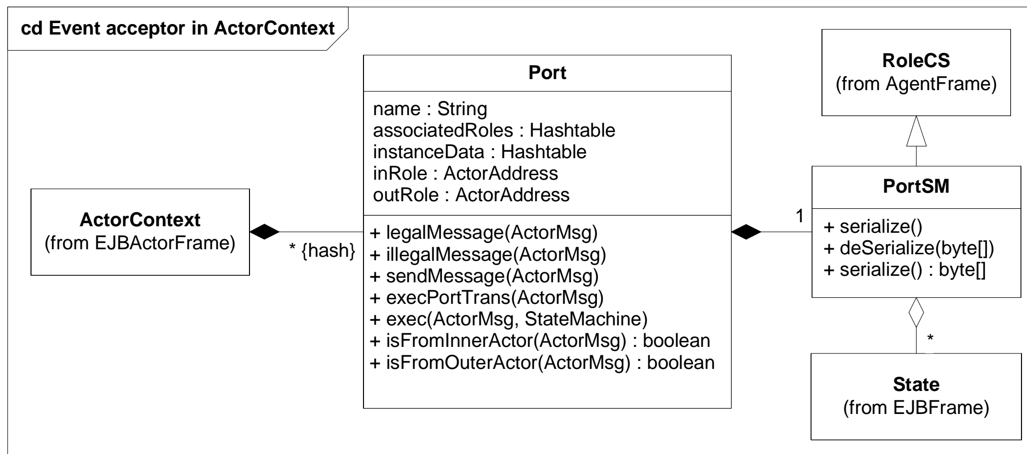

Figure 6.11: The event acceptor classes Port and PortSM in ActorContext.

6.6.2 Port addressing implementation

Addressing of Port instances is done according to the approach proposed in Section 5.3.1. As EJBFrame already implements most of this scheme, it was thus extended to add support for the Port addressing. This was done by adding the field *actorPort* to the *ActorAddress* class as shown

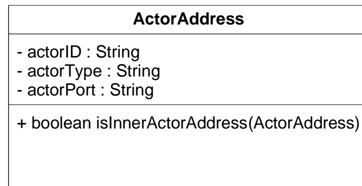

Figure 6.12: Modified ActorAddress class.

in Figure 6.12.

All utility operations specified by this class was updated with the additional information of *actorPort*. This approach does not change the default ActorFrame behavior of addressing StateMachines. In addition, a new utility operation, *isInnerActor*, was added to check if a one ActorAddress specifies an Actor which is an inner part of another Actor.

6.6.3 Port implementation

The Port class implements the necessary behavior and properties as described in Section 5.3. The class is shown in Figure 6.11.

The *legalMessage* operation routes messages to the correct sender. This operation uses the algorithm as proposed in Section 5.3.3. The *illegalMessage* operation is used to handle illegal messages. Currently this implementation has *no action*, and is only tracing the errors which occur. However, countermeasures for failing Actors as proposed in Section 5.7 could be implemented here. The *exec* method executes the event acceptor. This operation is called both upon reception and sending of messages. The *execPortTrans* operation implements the ActorFrame protocol which enables configuring the *inRole* and *outRole* connectors of the Port at creation time. The two operations *isFromInnerActor* and *isFromOuterActor* are utility methods which makes it possible to decide the direction of a message.

6.6.4 PortSM implementation

The PortSM class implements the protocol state machine, and is responsible for accepting or rejecting the signals sent to the port. The class extends the *RoleCS* class from AgentFrame, and thus has much of the structure needed to create the protocol state machine. The exception is that the arguments used in the operation implementations accepts Port objects instead of StateMachine objects.

The PortSM class should thus be used like any other CompositeState class from EJBFrame, but does not use the *sendMessage* operation to send ActorMsg's. Although *sendMessage* was implemented, it is tagged with a deprecation warning such that developers are aware that the use of it should be limited. Propagating messages is instead done by calling the Port operations

illegalMessage or *illegalMessage* – according to the legality of the message which was received or sent.

As the PortSM class gets called with both messages sent and received, it should thus take direction into consideration when asserting legality of the message. Such a check can be done by calling the *isFromInnerActor* and *isFromOuterActor* operations of Port.

6.6.5 Sending and replying to messages

Section 5.3.3 showed how signals shall be replied to. The algorithm which does the actual routing and delivery of the messages are implemented by the operation *legalMessage* in the *Port* class. However, additional methods are necessary to be able to deliver the message to the Port. The *ActorSM* class already implements this through a *sendMessage* operation which takes the ActorMsg and a port name as parameters.

ActorMsg
replyStack : Vector
+ remoteLastFromReplyStack() : void + firstElementFromReplyStack() : ActorAddress + lastElementFromReplyStack() : ActorAddress

Figure 6.13: Modified ActorMsg class.

The existing *sendMessage* method was thus updated to call *exec* of the port name described by the parameter. Furthermore, a new operation *sendMessageReply* was added. This operation takes a new ActorMsg and the ActorMsg which this message is in reply to as parameters. The method copies the reply stack from the request message to the reply message, updates the sender and receiver roles of the message and hence sends the message as normal. The modified ActorMsg class with its new *replyStack* is shown in Figure 6.13. The Java implementation of the ActorSM operations *sendMessage* and *sendMessageReply* is shown in Listing 6.3.

6.6.6 Message reception

Figure 6.14 shows an overview of what happens upon message reception by an Actor. The figure shown is simplified to increase readability. The main difference between the previous behavior and the inclusion of ports, is the addition of the method *shouldProcessMessage*.

The *shouldProcessMessage* method is declared abstract in the StateMachine class, and is implemented in *ActorSM*. The method checks if the attribute *actorPort* is set in its ActorMsg parameter. If this attribute is set and a Port with this name exists in the Hashtable of the Actors ActorContext, the Port is executed and the method returns false. Otherwise the method returns true, and the processing continues with execution of the state machine.

Listing 6.3: Java code for ActorSM operations *sendMessage* and *sendMessageReply*.

```
public void sendMessage(ActorMsg am, String portName) {
  if(context.ports != null && context.ports.containsKey(portName)) {
    Port port = (Port)context.ports.get(portName);
    am.setSenderRole(getMyActorAddress());
    am.addToReplyStack(getMyActorAddress());
    port.exec(am, this);
  }
  ..
}

public void sendMessageReply(ActorMsg replyMsg, ActorMsg requestMsg) {
  Vector replyStack = requestMsg.getReplyStack();
  if (replyStack == null) {
    ..
    return;
  }
  replyMsg.setReplyStack(requestMsg.getReplyStack());
  ActorAddress rcvRole = requestMsg.getLastElementFromReplyStack();

  if (rcvRole == null) {
    ..
    return;
  }
  replyMsg.setSenderRole(getMyActorAddress());
  replyMsg.setReceiverRole(rcvRole);
  sendMessage(replyMsg);
}
```

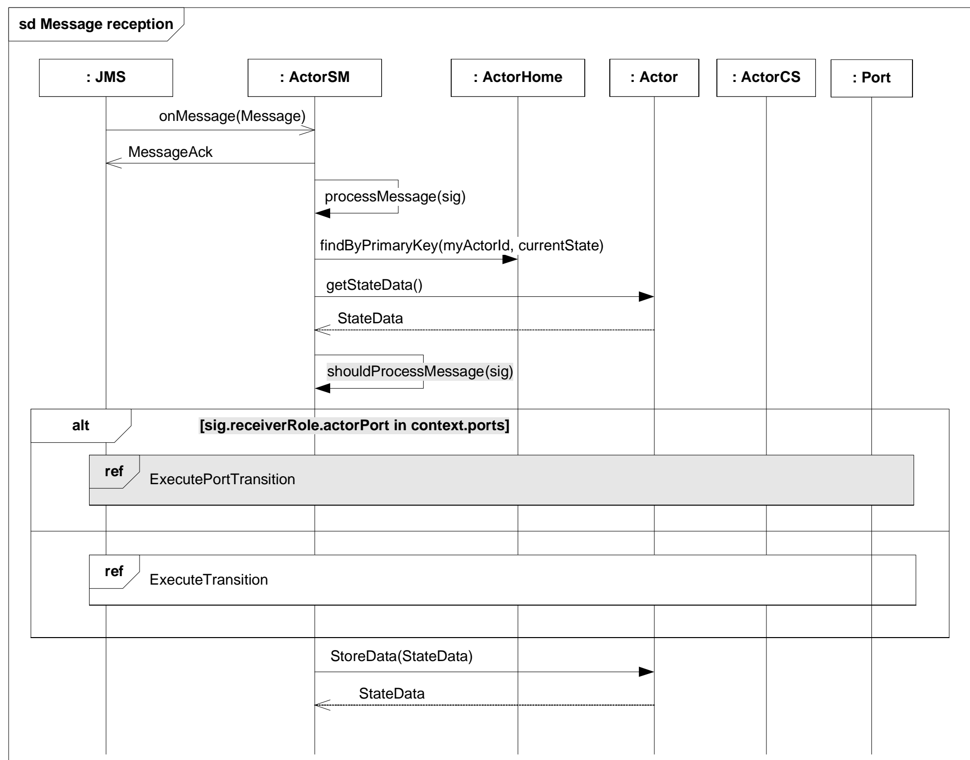

Figure 6.14: Sequence diagram for message reception (simplified).

As seen in Figure 6.14, state data is extracted and stored even though the state machine is not executed. This is done because the Port objects are stored as part of this state data. This has significant computational overhead when done on this platform, and steps should be taken to change this behavior for later implementations.

Figure 6.15 shows how the event acceptor is executed. There are two possible branches of execution for the event acceptor. Framework messages are treated specially, as these are used to implement the ActorFrame protocol for the ports. This implementation is stateless, which is not an optimal solution. The other possibility is to execute the protocol state machine. This is done by first finding the state data associated with the sender of the message, as described in Section 5.5. The PortSM instance is fed with the state data associated with the sender and the PortSM execution is performed. After the execution is performed, the updated state data is stored.

Figure 6.16 shows the possible outcomes of the event acceptor execution. In accordance with Section 5.4, there are two possible outcomes of such execution – either the message is regarded as legal or illegal. Furthermore, the protocol state machine, PortSM, is allowed to change state when it is executed.

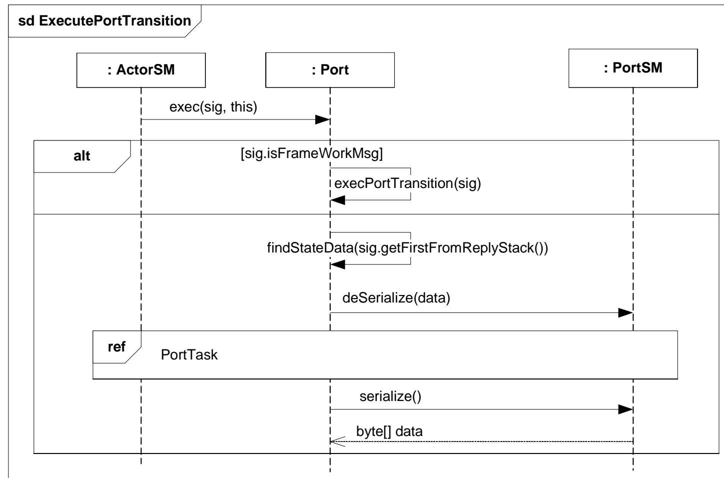

Figure 6.15: Sequence diagram for event acceptor execution.

portName	Unique name of the Port.
portType	Class name implementing the PortSM.
portDirection	Can restrict the directionality of the Port.
requestedRole	Requests connector to this ActorAddress.
inquiredRole	Inquires *ActorAddress* for the *requestedRole*.

Table 6.2: *PortSpec* attribute description.

6.6.7 Port description implementation

To be able to describe the ports and connectors between ports, the structure of actor descriptor XML-file was changed. This was done by adding additional fields to the descriptor which is read into the class *PortSpec* upon initialization of an Actor.

The modified *PortSpec* class is as shown in Figure 6.17. To support the new ActorAddress scheme described above, the actor descriptor reader method has the added support of reading port names from the *requested* and *inquired* roles. A description of the attributes of *PortSpec* is shown in Table 6.2.

6.6.8 Port and connector creation implementation

To be able to create Ports and connectors for an Actor part upon initialization, changes had to be done to the existing ActorFrame protocols RoleCreate and RoleRequest. This is done by extending the existing RoleRequest ActorMsg with the possibility of describing an *actorPort*. Figure 6.18 shows a somewhat simplified sequence diagram which sets up the *inRole* and *outRole* connectors between two Ports.

The sequence as shown in Figure 6.18 is initiated by the method *createPorts* implemented in

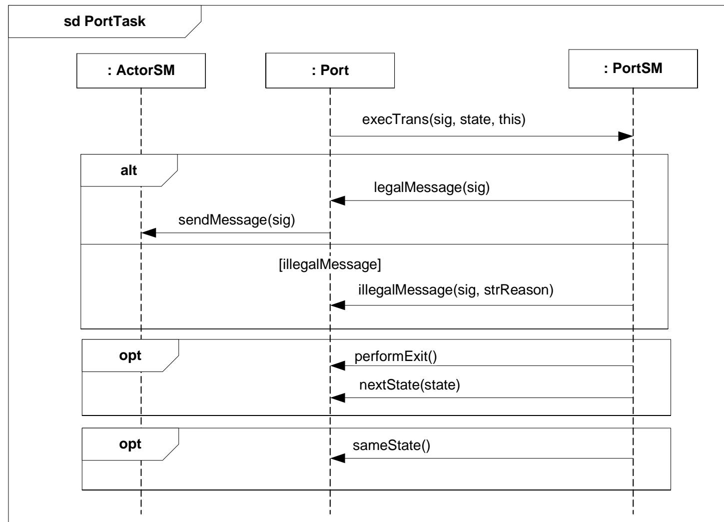

Figure 6.16: PortTask – Execution of a protocol state machine in the event acceptor.

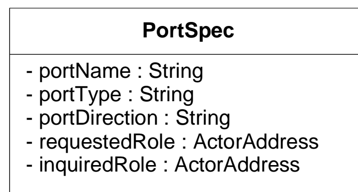

Figure 6.17: Modified PortSpec class.

ActorSM. This method is responsible for creating all the Port instances as defined by its actor descriptor. For each of the Port instances created, a RoleCreate message is sent to the Port with the PortSpec as argument. This PortSpec defines the connector this Port should try to set up – if any. When the port receives the RoleCreate, it requests the role as defined by sending a RoleRequest message. This protocol pattern is equivalent to the RoleRequest pattern in ActorFrame, with the addition that Ports may be addressed.

When a connector has been confirmed by a RoleConfirm message, the path synchronization mechanism is started, as described in Section 5.3.4. This is shown by the sequence diagram in Figure 6.19. The PathRequest protocol works by sending the PathRequest message when the *outRole* is set in the Port. This message is propagated by the *legalMessage* in the Port class. Upon reception of a PathRequest, the Port calls *addAssociatedRoles*. This call creates new instance and state data for this sender, and enables the event acceptor to keep unique state data for each potential sender.

PathRequestAck is sent back to the originator if the receiver of a PathRequest message is a behavior port. Upon reception of a PathRequestAck, the Port hence sends a RoleCreateAck to the Actor. When the Actor has received RoleCreateAck from all its specified ports, the Actor can

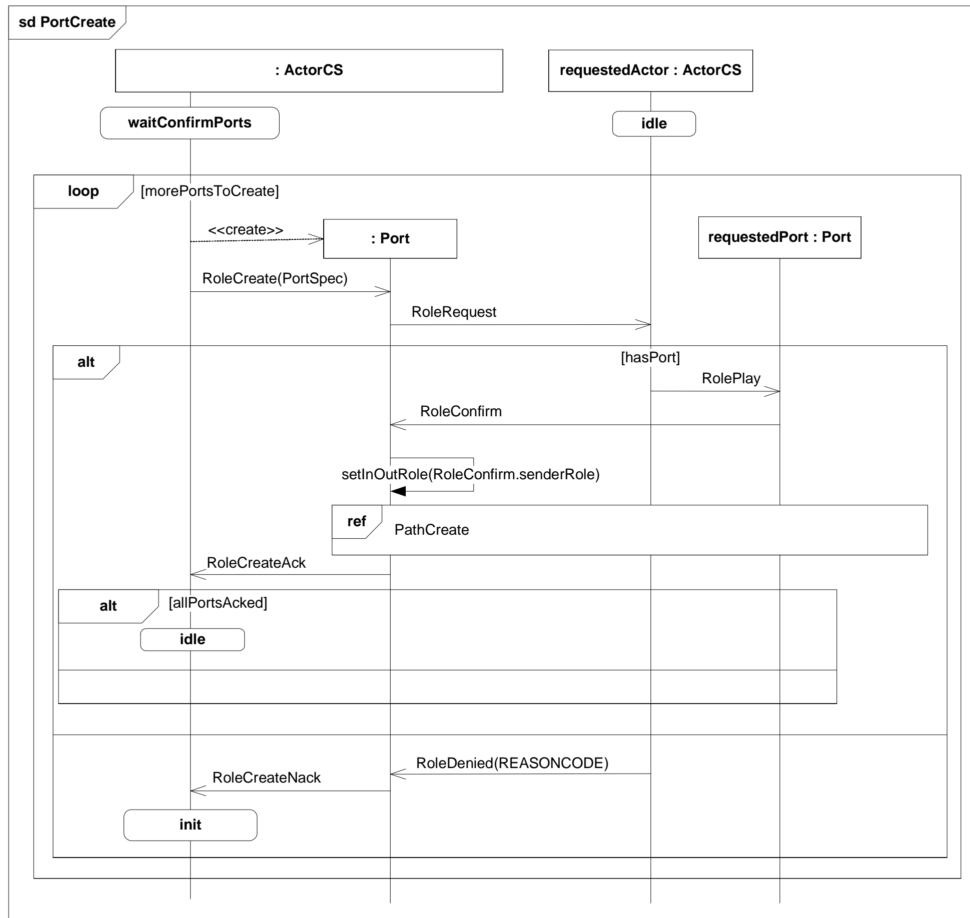

Figure 6.18: Setting up addresses between two ports.

start sending messages as normal.

6.7 Summary

This chapter presented the most important implementation details on how the NorARC frameworks EJBActorFrame and EJBFrame were modified to support remotely downloaded and run-time replaceable submachines. The chapter also shows how the run-time *event acceptor*, as described in Chapter 5, was implemented in the NorARC frameworks.

Summarized, the changes done to the frameworks are as following:

- SubMachine construct added to EJBFrame.
- New Actor behavior which supports remotely downloadable and run-time replaceable SubMachines.

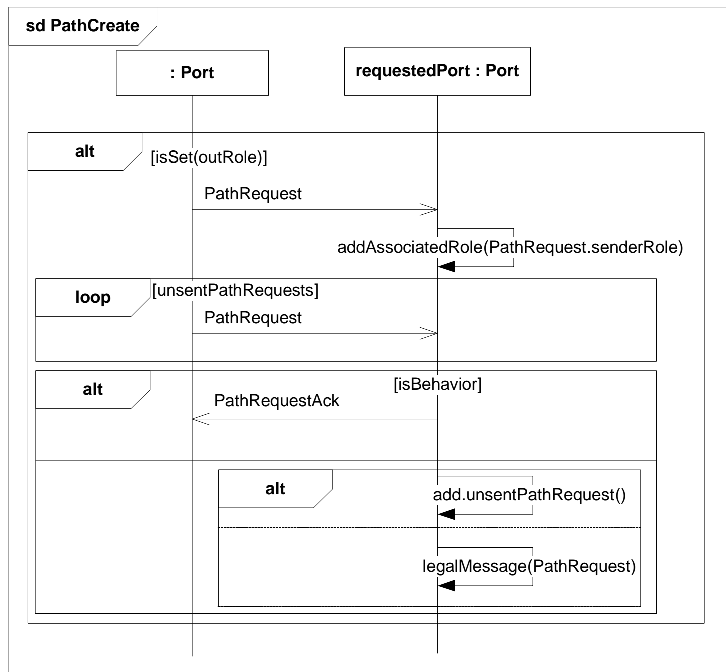

Figure 6.19: PathRequest sequence for synchronization of connectors.

- Updated the ActorAddress addressing scheme to include ports.
- New Port construct for EJBActorFrame.
- Protocol state machine which enables the run-time event acceptor.

In the next chapter I will revisit the case telecom service from Chapter 4. The service is implemented to demonstrate the usage and usefulness of the modified frameworks.

Chapter 7

Implementation of a service using remotely downloaded features

7.1 Introduction

In order to demonstrate how the proposed modifications to the EJBFrame and EJBActorFrame works, and to show a service which can replace submachines at run-time, I implemented the *phone book service*, previously introduced in Section 4.2. Furthermore, the service utilizes the *event acceptor* to protect some of the actors involved in the service.

The descriptions made in this chapter does not reflect all details of implementing services using the NorARC frameworks. I will hence focus on the parts which are relevant for the proposed changes – i.e., run-time replacement of submachines and use of the *event acceptor*.

7.2 The phone book service

The *phone book service* is a simplified service which enables users of the service to perform phone book lookups based on search keywords. The service implementation shown in this chapter does not fully implement such searching, and the focus is thus the messages sent between the actors. ServiceFrame has support for the Parlay-X API's for sending SMS and setting up calls through the use of predefined actors. However, the implementation shown in this chapter does not use these actors, and instead uses the simple actors *SMSEdge* and *CallEdge*. This was done to reduce the implementation work.

The behavior of the *PhoneBookService* actor and its environment is shown in the sequence diagram in Figure 7.1. The default *PhoneBookService* behavior is to make it possible for *UserTerminal* to choose the behavior of the actor through run-time submachine replacement. As shown in the figure, the *PhoneBookService* defines two different interaction features; *SearchFeatureCall* and *SearchFeatureSms*. The difference between these two interactions is that the *SearchFeatureCall* sets up a phone call between the phone of the phone book lookup requester and the result

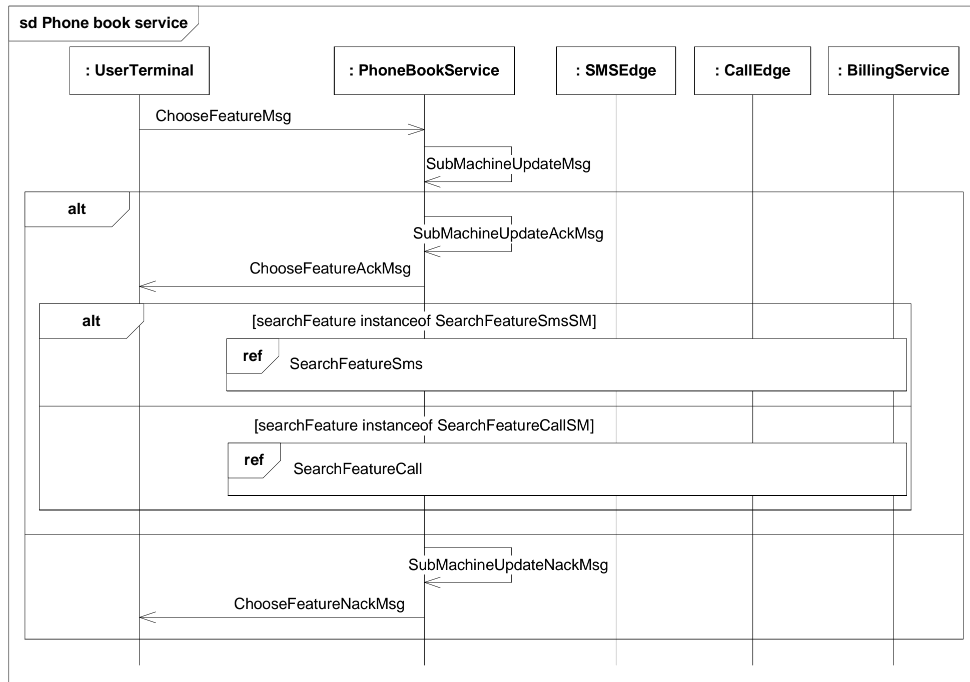

Figure 7.1: Sequence diagram for phone book service.

of the lookup, while the *SearchFeatureSms* sends an SMS with the result to the mobile phone of the lookup requester. The interactions for the *SearchFeatureCall* and *SearchFeatureSms* features are shown in the sequence diagram in Figure 7.2.

7.3 Design of phone book service

Figure 7.3 shows how the phone book service is designed. All the involved actors are ActorFrame Actors, and thus has state machines specifying their behavior and an *ActorAddress* for their state machine. The new addition as shown in the figure is thus the named ports.

The root actor of the service is the *ActorDomain* – the actor which encloses all the other actors involved. I will in the remainder of the chapter show the two actors *PhoneBookService* and *SMSEdge* as examples on how to implement such a service.

7.4 Phone book service implementation

The phone book service is implemented by extending the classes defined in the EJBActorFrame Java package. Figure 7.4 shows the classes which is manually modified for the actor *PhoneBookService* in gray. All other classes are extended in accordance to the EJBActorFrame profile, and is thus automatically generated using code-generation. The *PhoneBookService* actor does

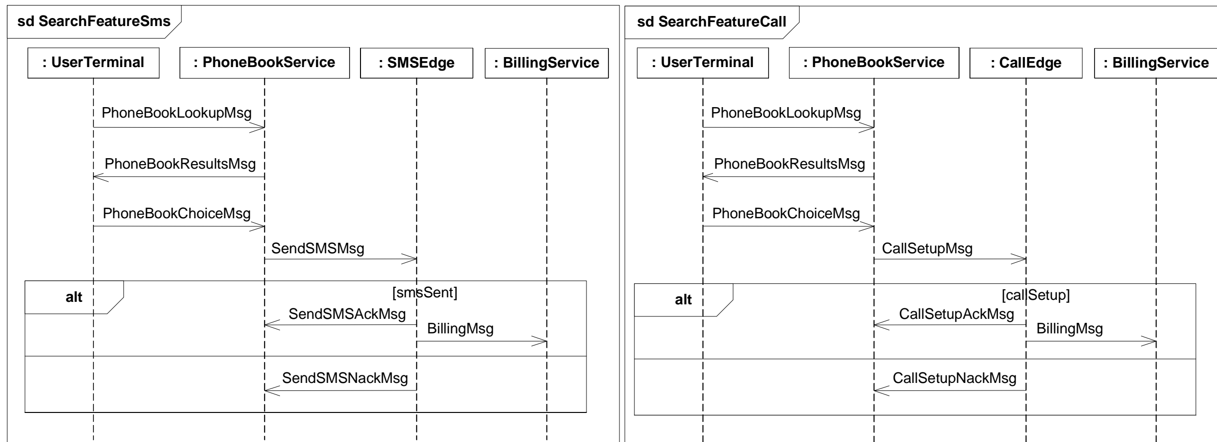

Figure 7.2: Sequence diagram for *SearchFeatureSms* and *SearchFeatureCall*.

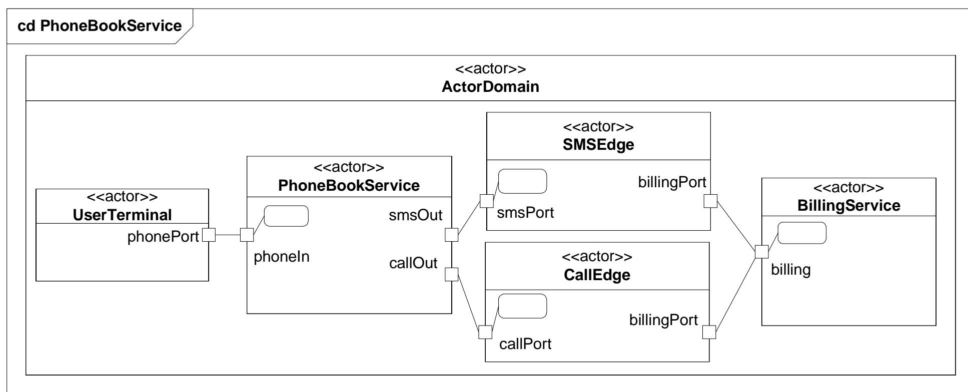

Figure 7.3: Class diagram of the phone book service.

not define any protocols for its ports, and thus uses the default *PortSM* class attached to the ports. This allows all messages by default. Furthermore, the actor allows replacement of the two SubMachines *SearchFeatureCallSM* and *SearchFeatureSmsSM*. As such, the behavior of *PhoneBookService*, *PhoneBookServiceCS*, thus extends the *DynamicActorCS* class instead of the default *ActorCS* class.

Figure 7.5 shows the manually implemented classes of the *SMSEdge* actor. This actor provides a service which could potentially accesses the Parlay-X SMS API, and should as such be protected against other failing actors. This is done by implementing a protocol state machine for the port which other actor clients access. As seen in Figure 7.5 this is done by creating a new class *SMSEdgePortSM* which extends the default *PortSM* class of EJBActorFrame.

I will in the next sections show how these two actors are mapped from UML, and how the class structure from Figure 7.3 is achieved.

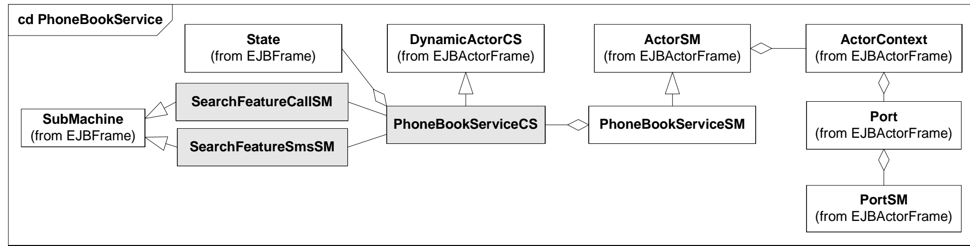

Figure 7.4: Class diagram for manually modified classes of *PhoneBookService*.

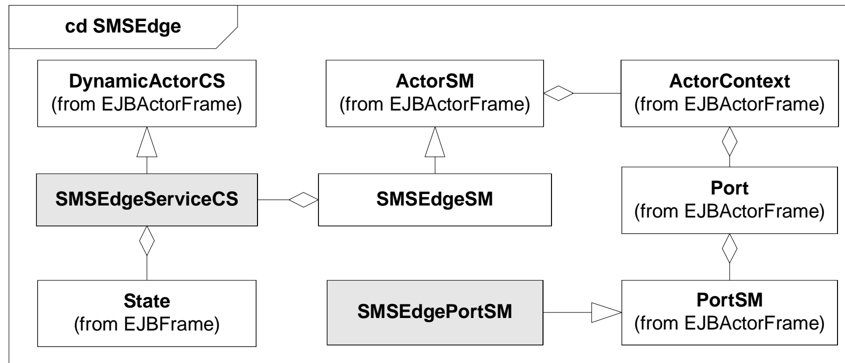

Figure 7.5: Class diagram for manually modified classes of *SMSEdge*.

7.4.1 Implementation of run-time replaceable submachines

The mapping from UML to Java of the state machine *PhoneBookServiceCS* is shown Figure 7.6. The figure shows how the SubMachine *searchFeature* is targeted for replacement by sending a *SubMachineUpdateMsg* to itself. The super implementation *DynamicActorCS* would in this case try to load the *SearchFeatureCallSM* submachine and bind this to the *searchFeature* submachine state.

Implementing run-time replaceable submachines are divided into two steps:

- Implement the enclosing state machine.
- Implement the submachine feature.

Implementation of the enclosing state machine

As seen in this example, *PhoneBookServiceCS* chooses the *SubMachine* to load based on a parameter in the *ChooseFeatureMsg*. This is not the only way to do this, as one could put the logic of selecting the SubMachine to be loaded in the *UserTerminal* actor instead. Such an approach would be more flexible, as a user of the service thus could choose a feature based upon a lookup in some kind of feature database. The implementation shown here is thus not very dynamic, as *PhoneBookServiceCS* needs to have information about all the different features it could load at

design time.

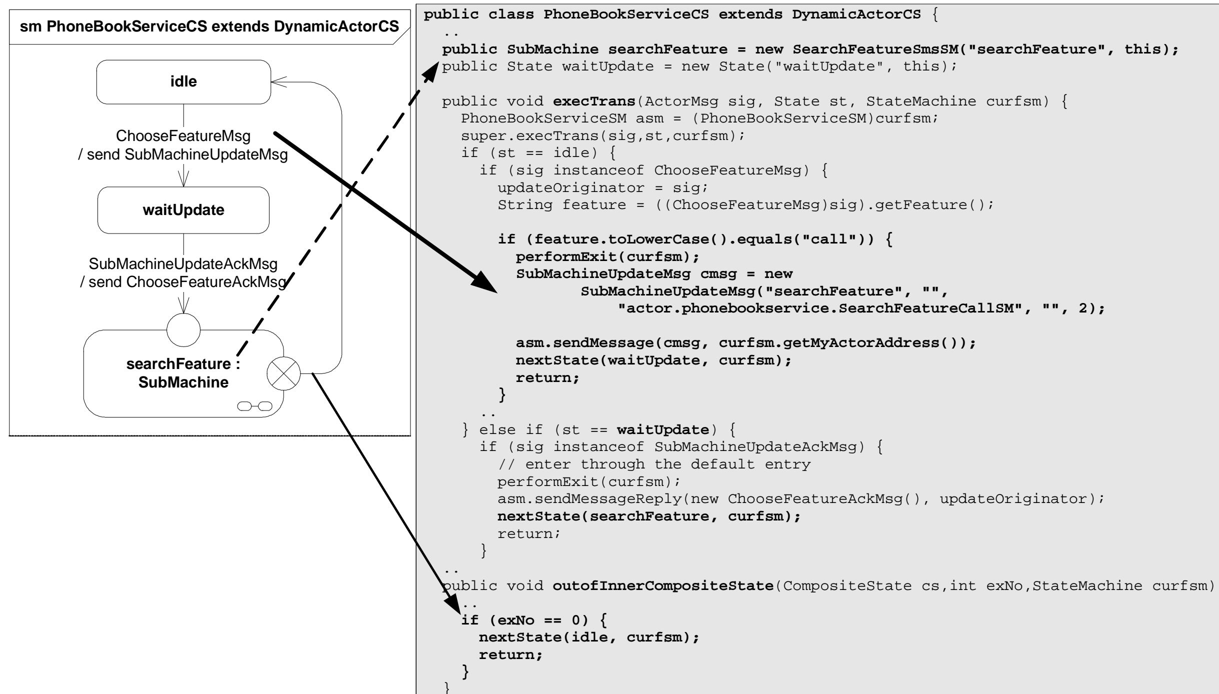

Figure 7.6: Implementation of behavior – class *PhoneBookServiceCS*.

Furthermore, Figure 7.6 shows how the submachine state *searchFeature* is entered and exited through the defined entry and exit points by using of the *nextState* and *outofInnerCompositeState* operations respectively. The implementation shown here does not specify an entry point, and it should thus be entered through the default entry point. This equivalent to the *CompositeState* usage in the original EJBActorFrame.

Implementation of the feature submachine

Figure 7.7 shows the mapping from UML to Java of the feature submachine *SearchFeatureSmsSM*. This class extends the EJBFrame class *SubMachine*, and it is thus possible to replace this at run-time. Both *SearchFeatureSmsSM* and *SearchFeatureCallSM* are mapped using this approach, and contains the same entry and exit points. As shown in this figure, the default entry point and the numeric entry point ”*0*” enters the same state; *waitForRequest*.

Furthermore, the figure shows how messages are replied to using the *sendMessageReply* operation and how messages are sent through a specific named port using the *sendMessage* operation. It also shows how the submachine is exited using the numeric exit point ”*0*” when the SMS message has been acknowledged.

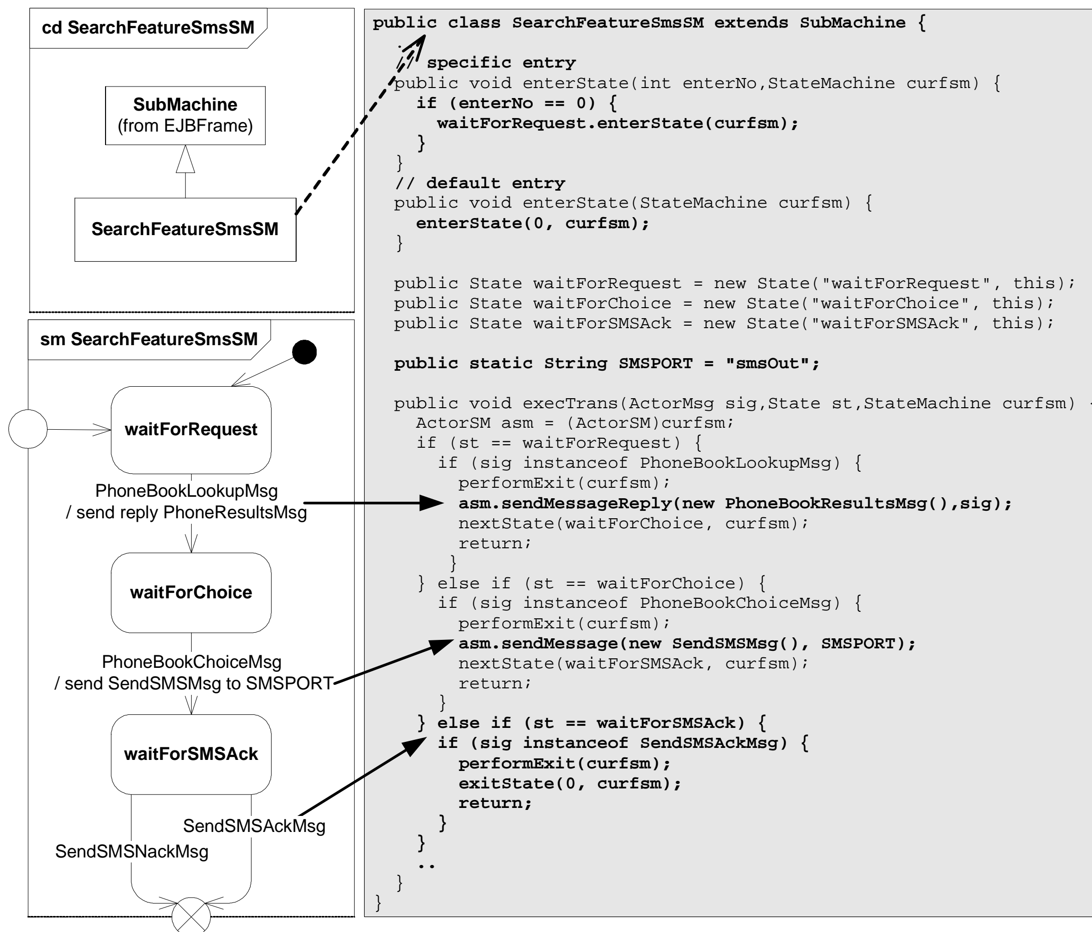

Figure 7.7: Actor descriptor with part and connector mapping for *ActorDomain* (partial).

The two replaceable submachine candidates in this example do not have any instance data. However, as described earlier this might be needed by a submachine. Implementing this for a *SubMachine* would amount to overriding the *initSubMachineInstance* operation for instantiation of data, *getStateProperties* to resurrect data and *storeStateProperties* to store data.

7.4.2 Implementation of the event acceptor

To verify the behavior of *PhoneBookServiceCS* it is decided that event acceptor should be implemented on all the ports which communicates with the *PhoneBookService* actor. This means the ports *smsPort*, *callPort* and *phonePort* from Figure 7.3. Additionally, one could also add extra protection by implementing the event acceptor in the ports *smsOut*, *callOut* and *phoneIn* of *PhoneBookService*. However, I will in this example show how to enable the event acceptor between the *PhoneBookService* and *SMSEdge* actors. Implementation of the event acceptor is done in two stages:

- Implement the protocol state machine.

- Specify the event acceptor with the actor-descriptors.

Implementation of the protocol state machine

As shown in Figure 7.5 we have already identified that the *SMSEdge* actor should protect one port with a protocol state machine, *SMSEdgePortSM*. The message interleaving which should be allowed by using *smsPort* thus needs to be established and implemented in the the class *SMSEdgePortSM*.

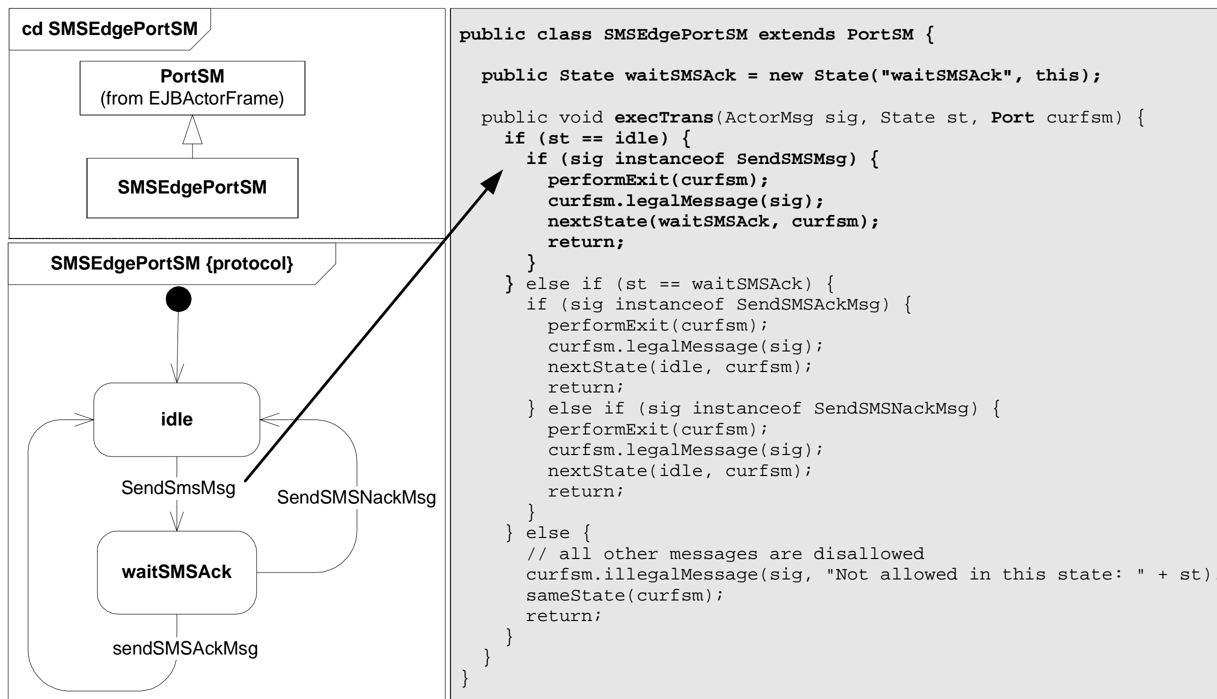

Figure 7.8: Protocol state machine mapping for *SMSEdgePortSM*.

The protocol state machine for *smsPort* is shown in Figure 7.8. Furthermore, it shows how the protocol state machine is mapped UML to the Java implementation. Mapping follows the normal pattern for *CompositeState* in EJBFrame, with the exception that it does not have any effects. Instead messages are allowed or disallowed by the use of *legalMessage* and *illegalMessage* operations. In Figure 7.8 the implementation shows the legal sequences of messages and that if a message does not fulfill these conditions, it is regarded as illegal by the use of *illegalMessage*.

A possible improvement to the *SMSEdgePortSM* would be to evaluate the frequency of messages received – i.e., to have a table with timestamps for the messages received. This would prevent clients of the service to send too many requests within a short period of time. Although the sequences are valid, its frequency may be considered unrealistic. Implementing this for classes

extending *PortSM* would amount to overriding the *deSerialize* and *serialize* methods for storing and resurrecting such data.

Specifying the event acceptor with actor-descriptors

The second part of implementing the event acceptor is to specify which ports should be created with an actor, and what protocol state machine should be used within a port. Such specification in EJBActorFrame is done with the use of *actor-descriptors* – XML-files which specifies an actors with inner parts and ports.

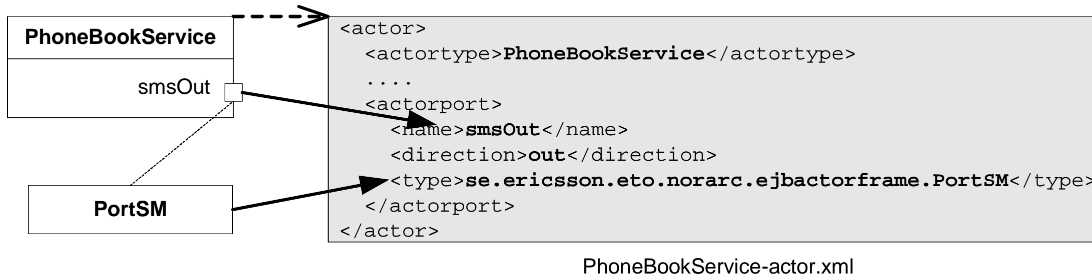

Figure 7.9: Actor descriptor and port mapping for *PhoneBookService*.

The actor *PhoneBookService* is shown in Figure 7.9. As shown in the mapping between UML and XML, the port *smsOut* is specified by the *actorport* tag. The port is given the *name smsOut*, *direction out* and uses the protocol state machine *type PortSM*.

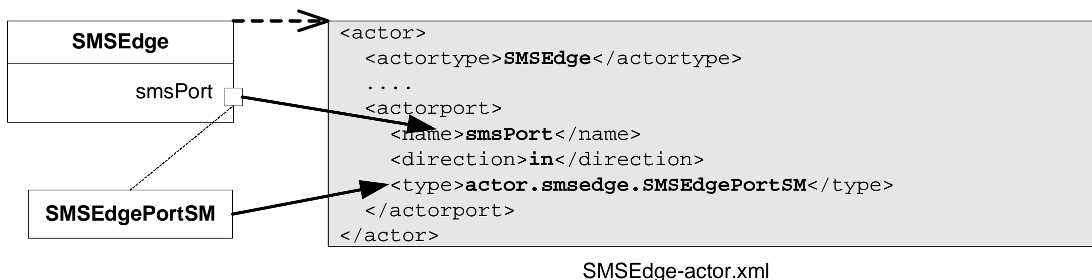

Figure 7.10: Actor descriptor and port mapping for *SMSEdge*.

The actor *SMSEdge* is specified in the same manner. The main differences are that the port specifies a port with *direction in*, and furthermore should use the protocol state machine *type SMSEdgePortSM*. The other ports specified by the actors *SMSEdge* and *PhoneBookService* are mapped to the actor-descriptors accordingly.

When all actors are described in their respective actor-descriptors, the parts involved in the service is described in the actor-descriptor of the *ActorDomain*. This is the root actor which contains

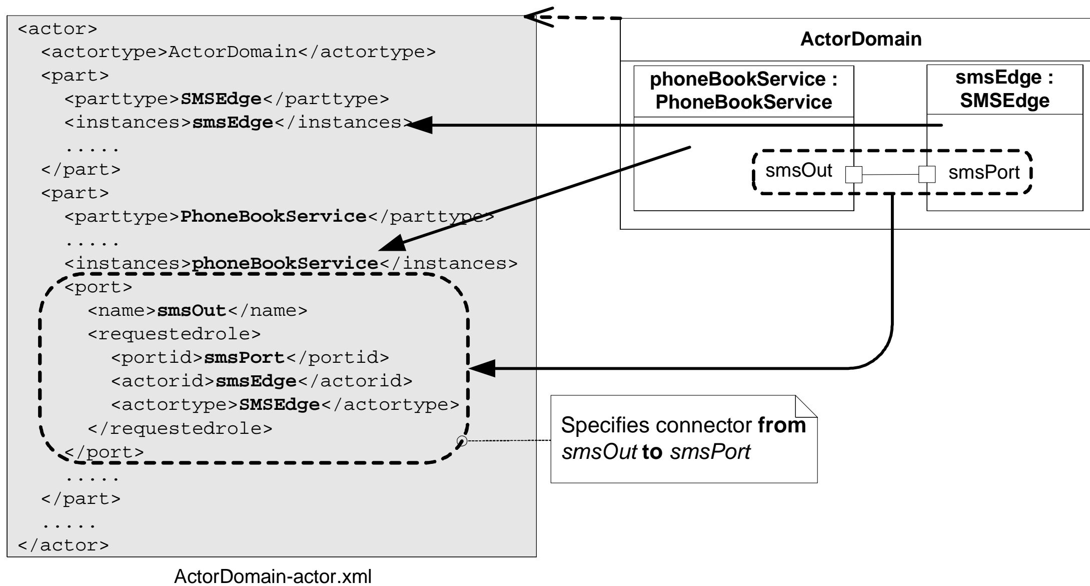

Figure 7.11: Actor descriptor with part and connector mapping for *ActorDomain* (partial).

all the parts of the phone book service.

Figure 7.11 shows a partial actor descriptor of *ActorDomain*, and how the inner parts are mapped from UML to XML. Furthermore, the figure shows how the connector from the port *smsOut* of the part *phoneBookService* to the port *smsPort* of the part *smsEdge* is specified. Upon instantiation of the part *phoneBookService* this shall thus invoke the RoleCreate and RoleRequest protocols for the port *smsOut*. When these protocols have finished, the *outRole* of *smsOut* thus specifies a connector to *smsPort* – i.e., it enables the *phoneBookService* part to send messages to the *smsEdge* part.

7.5 Summary

The modified EJBActorFrame and EJBFrame Java packages have been implemented as part of this thesis work. A trace of the run of the example service is supplied in the appendix of this report. All state machines were in this example run simulated, and was thus not tested using an application server such as JBoss [14].

The implementation of this example has shown that the proposed modification of EJBFrame and EJBActorFrame can be used to implement services which are able to remotely download an replace submachines features at run-time. It has also been shown how to implement the *run-time* event acceptor by extending the *PortSM* class from EJBActorFrame and specifying the structure in the actor-descriptors. Furthermore, it has been shown that the proposed modifications closely resembles the existing design and development patterns of EJBActorFrame and EJBFrame.

Chapter 8

Discussion and conclusion

8.1 Introduction

This thesis started off by giving brief introductions to the relevant concepts of UML 2.0 and the NorARC service creation architectures. I then introduced the issues and concepts behind using remotely downloaded and replaceable submachines as a method for customizing and adding value to services after deployment. This raised the question whether such change of behavior could be verified during run-time. I thus proposed a solution using the run-time event acceptor – based on the UML 2.0 concepts of ports and protocol state machines. Using these approaches I hence implemented this in the NorARC frameworks, and implemented a service demonstrating the solution.

One of the main purposes of this project was to get a better understanding on how to remotely download and replace submachines during run-time, and which issues must be considered when allowing this. This discussion is structured as the main parts of this report, and the issues discussed are as follows:

- Allowing remotely downloaded telecom features.
- Using ports and protocol state machines as run-time event acceptors.
- Run-time replacement of submachines in EJBActorFrame and EJBFrame.
- Implementation of a service using remotely downloaded features.
- Usefulness of remotely downloading and replacing submachines at run-time.

This chapter also proposes additional research topics in Section 8.7, before the main conclusion is drawn in Section 8.8.

8.2 Allowing remotely downloaded telecom features

In Section 1.5 I raised the following question: *"Which challenges needs to be addressed when designing replaceable submachines?"*. To answer this question I will briefly sum up the challenges found in Chapter 4.

The effects of allowing replacement of remotely downloaded telecom features is hard to properly quantify. The submachine is in UML 2.0 based on reasoning of component *re-use*, not as a method of adapting the behavior of a state machine at run-time. As such, some challenges were identified in Chapter 4:

- Instance variables must be handled carefully.
- Exit and entry points is not enough to ensure behavioral compatibility.
- Timers started by a submachine should be stopped upon exit.
- It may be beneficial to allow dynamic creation of ports.
- The external visible properties may be decomposed and asserted through the use of ports and protocol state machines.

Most notable of these issues is the interdependency which exist between a submachine an its encapsulating state machine and their instance variables. If either the submachine or the state machine makes the wrong assumptions upon the state of these, it may lead to erroneous behavior. Verifying this instance data relationship is complicated, and a solution to solve this problem was not found.

Furthermore, it was stated that a newly introduced submachine might introduce deadlocks, livelocks or non-deterministic behavior to the state machine or other adversary state machines in the system. Such behavioral anomalousness may be formally verified at design time of the component using *projections* [7], but this approach was not further explored in this thesis.

To combatant the above described problems, I proposed the *event-acceptor* as a mechanism for ensuring run-time behavior conformity. This is done by decomposing the external relevant properties of the structured part by using ports and protocol state machines. However, this method can not detect all errors, as the internal properties of the state machine is invisible to the protocol state machine.

When allowing dynamic behavior adaptation of structured parts, it was identified that one might need to create new ports to create new features. This is not allowed by UML 2.0, and I thus suggested that this restriction should be further investigated, as it puts a serious constraint on the possibilities of adapting behavior at run-time.

8.3 Using ports and protocol state machines as run-time event acceptors

In Chapter 5 I presented an approach to using ports and protocol state machines as a solution to ensure behavioral conformity at run-time. This solution represents the main contribution of this thesis. The approach taken in the chapter was based on the architectural shortcomings of EJBActorFrame and EJBFrame to enable this.

The goal of the event acceptor was to capture the message interleaving between two ports, and thus to assert whether two adversary state machines adheres to a specified protocol at run-time. Some of the advantages of using the event acceptor are as follows:

- Capture erroneous behaviors as an effect of dynamic feature adaptation.
- Proper run-time protection of critical system components.
- Ease debugging of architectural and protocol flaws.

The UML 2.0 protocol state machine construct only allows assertion of incoming events, the proposed event acceptor thus violates its formal semantics. The implications of this violation was not further explored in this thesis. However, the event acceptor has shown that having such a mechanism in UML 2.0 would be of great value.

The chosen naming scheme for the connectors between ports was shown to have its limitations. UML 2.0 specifies that signals received by a port with multiple delegation connectors attached to it, should either copy and forward the signal to *all* these connectors or choose *one* of them. The proposed naming scheme does not allow seeing the inner structure of a structured part. I thus argued that copying and forwarding signals to all inner parts could lead to unwanted effects. I therefore chose to only allow one connector in each direction of the port. However, this approach disallows creation of new inner parts at run-time, as new ports must be created at the enclosing structured part. Creating new ports after a part is created is not allowed by UML 2.0. I thus proposed a new solution using interfaces as qualifiers in the naming scheme, but unfortunately I did not have enough time for any further investigations based on this approach.

8.4 Run-time replacement of submachines in EJBActorFrame and EJBFrame

In Section 1.5 I raised the following question: *"How can dynamic submachine components be realized in the NorARC service creation architecture?"*. This question was answered with the proposed solutions to EJBFrame and EJBActorFrame in Chapter 6. I will in this section discuss some of the implementation issues which arose from this.

The new *Port* class in EJBActorFrame was developed to be able to define explicit interception

points between ActorFrame Actors. This was done by hooking in to operations which receives and sends messages. To ease the development of these interception points, it was chosen to put the Port instances in the ActorContext of the Actor. This is not a good approach, as this requires all the persistent state data of an Actor to be resurrected and stored even if a message only shall be forwarded to an inner actor. This implies there is a serious performance penalty of the current Port implementation. A better approach would thus be to let the Port objects be contained in its own state data environment – such as the current StateMachine implementation. With such a solution, only the persistent data of the Port would have to be resurrected upon reception of a message.

The current implementation does not enforce the use of the new Port class to communicate with an ActorFrame Actor. With the current performance overhead of using the Ports, this was deemed unnecessary. However, when this issue is resolved, it should be considered that all communication should be conducted by the using the new Port class.

As stated in the previous section, the implemented addressing scheme has its limitation in that it can not create new inner actors, as this would require new ports to be created in the enclosing part. It should thus be considered to implement a different naming scheme which can specify interface names which can be added to the Port while running, and thus alleviate the routing mechanisms.

8.5 Implementation of a service using remotely downloaded telecom features

By implementing the designed telecom service designed in Section 4.2, it has been shown that run-time replacement of submachines is possible using the proposed modifications to EJBActorFrame and EJBFrame. Furthermore, the use of the event acceptor was demonstrated.

The phone book service is a simple service, and does thus not reflect all the modified parts of the frameworks. It was not properly shown how *PortSM* classes could benefit from its having instance variables. Furthermore, it was not shown how the *SubMachine* can encapsulate it's own instance variables. These concepts were not any further demonstrated because of the limited time available in this thesis work.

The implementation description shows that the modifications done to the NorARC frameworks does not affect the existing design and implementation patterns in any major way. Developers familiar with the existing guidelines should thus be able to comprehend the newly introduced patterns for replacing submachines and setting up the run-time event acceptor quite easily.

The phone book service was executed using simulation, and was thus not tested using an application server. This implies that neither the modifications to the frameworks nor the phone

book service has been thoroughly tested on an application server. However, considerations have been taken such that these should work even when the execution is not simulated.

8.6 Usefulness of remotely downloading and replacing submachines at run-time

In Section 1.5 i raised the following question: *"In order to customize services, are submachines viable and useful as components for enabling third-party service providers to create new features that may be downloaded at run-time?"*. This is not an easy question to answer, as the usefulness and viability of the replaceable submachine component is hard to quantify and measure. However, I will express my subjective conclusions on this matter. First my conclusion of its viability:

> *Remotely downloaded and replaceable submachines have been shown to be viable when using a combatant against ill behaving components at run-time, such as the event acceptor proposed in this thesis. However, further analysis of the instance variable relationship, and security concerns when allowing downloading and execution of code from a remote location, is needed.*

With regards to the usefulness I draw the following conclusion:

> *The usefulness of the replaceable submachine with regards to feature adaptation and personalization of services is suggested by the implementation of the phone book service. Furthermore, I believe run-time replacement of submachines to be a reasonable approach for rapid deployment of new and emerging service features.*

8.7 Future work

Several issues identified in this thesis work could be subject for future research efforts:

- The simple state could be viable for replacement with a submachine at run-time. This would involve dynamically adding and removing entry and exit points at run-time. Such an approach would increase the run-time feature adaptation capability of the state machine.
- The implications of downloading and executing code from a remote location has not been investigated further in this thesis. Allowing code to execute when its origins is not verified could have devastating effects. Processes and mechanisms for accommodating this *must* be further explored before allowing this in a *live* system.
- The naming scheme for ports proposed in this thesis has been shown to have limitations. The use of interface names as addressing qualifiers could prove to strengthen the scheme, and should thus be further investigated.

- The restriction of UML 2.0 not allowing ports to be created during the lifetime of a part should be further investigated. Allowing ports to be created would increase a state machine's run-time feature adaptation capabilities.

- The current implementation for replacing *SubMachine* objects by updating the *children* references of *CompositeState* prevents replacement of arbitrary deeply nested *SubMachines*. A different approach to doing this should be further investigated.

- EJBActorFrame and EJBFrame enables developers to send messages to other actors by addressing them directly with the use of ActorAddress, thus having direct addressing of adversary state machines. It may be more intuitive and safe hide these methods, and thus enforce the use of the port mechanisms proposed in this thesis.

8.8 Conclusion

In this thesis issues regarding remotely downloaded telecom features has been discussed. This thesis argues for replacing submachines at run-time as a method for adapting the behavior of services. Some design guidelines have been discussed with regards to allowing such replacement. Furthermore, the event acceptor has been proposed as a mechanism for run-time behavior assertion of components. Following the guidelines, the submachine component was implemented in EJBFrame, and generic behavior for replacing this was implemented in EJBActorFrame. The event acceptor was implemented by introducing ports and protocol state machines concepts to EJBActorFrame. Furthermore, a concrete service example was implemented demonstrating the proposed modifications to the frameworks.

The main contribution of this thesis was the run-time event acceptor. Ideas on how to route, send and reply to messages was proposed and finally implemented. Furthermore, a solution on how to configure connectors using signaling was proposed and implemented. The phone book service demonstrator suggests that these solutions work well, and that the introduced concepts should be comprehendable by designers and developers familiar with the NorARC service creation architectures.

The main conclusion of this thesis is that run-time replacement of submachines is both useful and viable. Allowing such replacement to happen enables personalization of running services and applications. At the same time services may increase their longevity by adding new and emerging technologies without affecting their availability. Although this has not been a formal study, it has been shown to work when there exists mechanisms can detect erroneous behavior – such as the run-time event acceptor proposed by this thesis.

This study has also shown that the current restriction of UML 2.0 not allowing to create ports after a structured part has been created, may be found too strict in practical use. As new features are introduced at run-time, it is expected that these will at some time require the use of new ports. This UML 2.0 restriction thus decreases the feature adaptation capability of the approach taken in this thesis.

Bibliography

[1] Coad P., Lefebvre E. and De Luca J. *Java Modeling in Color with UML*, Prentice Hall, 1999 - ISBN 013011510X

[2] Bræk, R, Husa, K. E., Melby, G. *ServiceFrame WhitePaper*, Ericsson NorARC, Draft 22.04.2002.

[3] Husa, Knut Eilif and Melby, Geir. *ActorFrame Architectural Guide*, Ericsson NorARC, 2003.

[4] Melby, Geir. *ActorFrame Developers guide Guide*, Ericsson NorARC, August 2004.

[5] Melby, Geir. *Using J2EE Technologies for Implementation of ActorFrame Based UML2.0 Models*, Master Thesis AUC, August 2004.

[6] Grady Booch, Ivar Jacobsen and James Rumbaugh. *The Unified Modeling Language Reference Manual, Second Edition*, Addison-Wesley, 2004 - ISBN 0-321-24562-8.

[7] Floch j., Bræk R., *Using Projections for the Detection of Anamalous Behaviors*, Proceedings of the 11th SDL Forum, Stuttgart, Germany, 1st-4th July 2003, Springer.

[8] Object Management Group, OMG. *UML 2.0 Superstructure Specification, August 2003.* OMG Adopted Specification.

[9] Aagesen, F. A., Anatariya, C., Shiaa, M., M., Helvik, B. E. *Dynamic Configuration of Plug-and-Play Systems*, 2003.

[10] Mencl, V. *Enhancing Component Behavior Specifications with Port State Machines*, Technical report, Dept. of SW Engineering, Charles University, Prague, September 2003.

[11] Engelhardtsen, F. B, *Adapting telecom services on-the-fly using run-time pluggable submachines* Paper, Agder University College, NTNU, 2005.

[12] Keshav, S. *An Engineering Approach to Computer Networking*, Addison-Wesley, 1997 - ISBN 0201634422.

[13] Haugen Ø., B. Møller-Pedersen. *JavaFrame - Framework for Java-enabled modeling*, in ECSE2000 Stockholm, 2000.

[14] JBOSS; J2EE Application Server; http://jboss.org/

[15] TAPAS; Telematics Architecture for Play-based Adaptable Systems; http://tapas.item.ntnu.no/

[16] OMG; Object Management Group; http://www.omg.org/

[17] CERT Advisory TCP SYN Flooding and IP Spoofing Attacks; http://www.cert.org/advisories/CA-1996-21.html [last accessed: 23.mai 2005]

[18] The PARLAY Groups; http://www.parlay.org/

[19] SDL - Specification and Description Language, CCITT recommendation Z100

[20] MDA - Model Driven Architecture, OMG; http://www.omg.org/mda/

[21] J2EE - Java 2 Platform, Enterprise Edition, Sun; http://java.sun.com/j2ee/index.jsp

Appendix A: CD-ROM

If the report you are reading is a hard copy, there should be a CD-ROM supplied in the sleeve. This contains the code of the modified NorARC frameworks, and a copy of the phone book service.