Using Ericsson NorARC's frameworks as test bed for dynamic change of behavior based on CompositeStates

by

Arne Wiklund

Master Thesis in
Information and Communication Technology

Agder University College

Grimstad, May 26, 2003

Abstract

Frameworks are a widely used *re-use* technique in the object-oriented community today. A framework re-uses both code and design and makes it easier to develop new components. Ericsson NorARC (Norwegian Applied Research Center) has developed a set of frameworks; JavaFrame, ActorFrame and ServiceFrame to aid advance service development. JavaFrame is a modeling kit for Java with good support for state machines. ActorFrame is a framework emphasizing the actor and role notion. ServiceFrame emphasize the modeling of service functionality.

UML is today's de facto standard for visualizing, specifying, constructing and documenting software systems. Modeling software in advance of making them is just as essential as having a blueprint of any large building construction. Failing to generate unambiguous models and lack of support for component-based modeling has impelled a demand for revision of today's UML standard.

This thesis evaluates the NorARC service creation environment, and especially JavaFrame and ActorFrame, with a view to dynamic service behavior update. More specified, if and how a state machine's state space (composite state) can be altered during run-time (i.e. dynamically changing its behavior). Further, the thesis evaluates the possibility of showing this dynamic substitution and behavioral change in UML 2.0.

Based on the guidelines and patterns from JavaFrame and ActorFrame, as well as a thorough inspection of the UML 2.0 proposal, state machines and asynchronous communications, some conceptual ideas has been developed. These range from state machine equivalence to 1:1 UML 2.0 modeling. A prototype is made demonstrating some of these concepts.

Through this thesis project, we have discovered that the NorARC service creation environment is, at present state, probably not the ideal framework for run-time behavioral update trough CompositeStates. Further, we have discovered that it is possible to model run-time introduction of composite state, using interaction diagrams in UML 2.0. On the other hand, no solution to modeling such dynamic change of behavior is found, neither with UML 1.4, 1.5 nor the proposed UML 2.0 standard.

Preface

This thesis was written as a part of the Norwegian Master degree in Information and Communication Technology. The work was carried out between January 2003 and May 2003, at Agder University College. No external company was involved in this thesis, but the thesis project is related to the *Teleservice lab* in Grimstad.

I would like to thank my supervisors, Fritjof B. Engelhardtsen and Jan P. Nytun at Agder University College. I would also like to thank Stein Bergsmark (Director of Study, Master of Science Study, Agder University College) for his valuable counselling during this thesis project.

Grimstad, May 2003

Arne Wiklund

Table of Contents

Preface I

Table of Contents II

List of Figures V

1 Introduction 1
- 1.1 Background 1
- 1.2 Ericsson NorARC's Framework 2
- 1.3 UML 2.0 3
- 1.4 Thesis Definition 4
- 1.5 Work Description 4
- 1.6 Report Outline 5

2 State Machines and Asynchronous Communication 6
- 2.1 Introduction 6
- 2.2 Finite State Machines (FSM) 6
  - 2.2.1 What is a State Machine? 6
  - 2.2.2 CFSM 7
- 2.3 Concurrent Programming 8
- 2.4 The Actor Based Approach 9
- 2.5 Summary 9

3 NorARC's Service Creation Environment 10
- 3.1 Introduction 10
- 3.2 JavaFrame 10
  - 3.2.1 Active Objects and Composites 12
  - 3.2.2 State Machines and CompositeStates 12
  - 3.2.3 Mediators 14
  - 3.2.4 The Trace File 15
- 3.3 ActorFrame 15
- 3.4 ServiceFrame 17
- 3.5 Summary 18

4 UML 2.0 **19**
4.1 Introduction 19
4.2 Components and Composite Structure 19
4.3 State Machines 21
4.3.1 ConnectionPointReference 21
4.3.2 FinalState 22
4.3.3 State 22
4.3.4 State Machines 23
4.3.5 Transition 24
4.4 Interaction Diagrams 25
4.5 Summary 26

5 Patterns, Methods and Architecture **27**
5.1 Introduction 27
5.1.1 Dynamic Java 27
5.2 JavaFrame Guidelines and Patterns 28
5.3 The Architecture Provided by the Framework 30
5.4 Summary 33

6 A Conceptual Approach **34**
6.1 Introduction 34
6.2 State Machine Equivalence 34
6.3 The CompositeState 35
6.4 Modelling UML 1:1 35
6.5 Summary 37

7 The Prototype **38**
7.1 Introduction 38
7.2 The State Machine 38
7.3 Architecture and Design 39
7.3.1 The Initial State Machine 39
7.3.2 The Extended State Machine 42
7.4 Implementation 43
7.4.1 The Actor's Deployment Descriptors 43
7.4.2 CarDriver 45
7.4.3 CarControl 48
7.4.4 CarMovingCS 49
7.4.5 ExtendedCarMovingCS 50
7.4.6 CarMovingBackwardCS 51
7.5 UML 52
7.6 Summary 53

8 Results 54
8.1 Introduction 54
8.2 Logical vs Physical Execution 54
8.3 UML 55
8.4 Summary 56

9 Discussion 57
9.1 Introduction 57
9.2 Dynamic change of state machine behavior in JavaFrame and ActorFrame 57
9.3 Run-time CompositeState replacement 58
9.4 The Prototype 59
9.5 State Machines Communicating Asynchronously 59
9.6 UML 2.0 59
9.7 Future work 60
9.8 Summary 60

10 Conclusion 62

Bibliography 64

Appendix A - Java Source Code CD-ROM

Appendix B - Deployment Descriptors CD-ROM

Appendix C - Trace File CD-ROM

List of Figures

2.1 State chart of an electrical light bowl switch. 7

3.1 The NorARC Stack 10
3.2 JavaFrame classes and run-time system. 11
3.3 JavaFrame - alternative modeling notation. 11
3.4 Active object model. 12
3.5 CompositeStates. 13
3.6 Different type of mediators. 14
3.7 The use of mediators 15
3.8 The logic execution trace. 15
3.9 A play and its participating actors. 16
3.10 Class structure for ActorFrame 16

4.1 Black-box or external view notation of a component. 20
4.2 An internal or white-box view. 20
4.3 Dependencies in the CompositeStructure package. 21
4.4 The state machine class structure. 22
4.5 Composite state with entry and exit points. 22
4.6 A transition example with incoming and outgoing messages . . . 24
4.7 Sequence diagram 25
4.8 Communication diagram 26

5.1 JavaFrame framework/Modelliong Development Kit 28
5.2 Conceptual model of ActiveObject 30
5.3 Message conceptual model 30
5.4 Scheduler conceptual model 31
5.5 State conceptual model 31
5.6 The Actor base behavior 31
5.7 Non-restrict layering 32
5.8 Conceptual design - Application and ActorFrame 32
5.9 Conceptual design - Application and JavaFrame 33

6.1 Conceptual class diagram 36
6.2 Conceptual class diagram 37

6.3 Conceptual communication diagram 37

7.1 The initial state machine 39
7.2 The driver state machine 40
7.3 UML diagram 1 41
7.4 UML diagram 2 42
7.5 UML diagram 3 43
7.6 StateGraph of the initial composite state 43
7.7 The extending composite state chart 44
7.8 State graph of the extended composite states. 44
7.9 Code for the RootActor deployment descriptor 45
7.10 The deployment descriptor for the CarDriver 45
7.11 Implementation of CarDriverSM 45
7.12 Implementation of CarDriverCS 46
7.13 Implementation of CarDriverCS.sendRoleRequest 46
7.14 Implementation of CarDriverCS.treatRoleConfirm 46
7.15 Implementation the Stop state 47
7.16 Implementation of the class CarControlSM 48
7.17 State and Composite State Declaration for CarControlCS 48
7.18 Implementation of outOfInnerCompositeState 48
7.19 Implementation of treatEnterState in CarControlCS class. 49
7.20 Implementation of the CompositeState update procedure 49
7.21 Extract of the enterState method from the class CarMovingCS. . . 50
7.22 Recursively calling outofInnerCompositeState 50
7.23 ExtendedCarMovingCS constructor. 51
7.24 Recursively calling the method outofInnerCompositeState 52
7.25 The prototype update sequence diagram 52
7.26 The prototype communication diagram. 53

8.1 Parts of the trace file 55

Chapter 1

Introduction

1.1 Background

Standing with a mobile terminal in your hand, wondering if it works as you try to pay a bill, or waiting for a location update for your child's whereabouts, the mobile terminal gives people in the 21st century a whole new set of applications, which they did not have before, when *all you could do was make a call*. This gadget evolution poses new possibilities as well as new problems for software and hardware developers within tele and data communication.

Nowadays, value added services within the mobile network are widely used both as entertainment and as substitute for daily functions (e.g. ordering tickets by Short Messages Service, SMS). These services are often modelled by state based behavior. There may be several hundred thousand people using a particular service at a given time. Problems arise when updating or changing the service by altering the behavior of the service. By shutting down the service, all subscribers will lose the possibility to use the service might find that unacceptable when using the mobile to pay a bill. A way to solve this problem could be to update the service while running (i.e. changing the state machine behavior in run-time). Other problem could be in mobile terminal itself. Although today's terminals have undergone a massive increase in both processing power and memory, there are still processes that are too CPU[1] demanding for the mobile to process adequately. Even when introducing new and more powerful mobile terminals the problem with older terminals are still not solved. One could, after figuring out the processing power of a terminal, dynamically change the state machine of the service making it suitable for the terminal to process. Parts of the service might run in the service network, relieving the terminals processing unit. Vice versa, if the subscriber during the use of service, for instance a game, replaces the terminal with one more powerful, the

[1]Central Processing Unit

service could initiate a dynamic update, pushing more of the workload from the service network to the mobile terminal.

The rest of the chapter will give an overview of the technologies that are used as the foundation of the work related to this thesis. There will be an introduction to the Ericsson NorARC service creation environment and UML 2.0.

1.2 Ericsson NorARC's Framework

The word *framework* is widely used and its definition has varied a lot due to the fact that the concept of framework is difficult to define. Some definitions describe the structure of the framework and others describe the purpose. Frameworks are object oriented re-use technique in particular, and in general look like any other reuse technique [7].

As seen with several other frameworks e.g. AWT[2] and beans[3], frameworks are important when it comes to component development. *The ideal reuse technology provides components that can be easily connected to make a new system* [7]. But frameworks are more powerful than any other regular components architecture. As such, it requires more effort to understand and utilize them.

Ericsson NorARC's (Norwegian Applied Research Center) framework is a framework stack assembled by three distinct frameworks; ServiceFrame, ActorFrame and JavaFrame, which are all tools for design, verification, deployment, management, and testing services. These are the core concept of most frameworks, providing a standard way to handle different aspects of component development. The NorARC framework is meant to deal with complex systems within telecommunication and Internet services; scenarios that are described in the Introduction. ServiceFrame [1], the top layer in the framework supports the creation, deployment and execution of the service. The ActorFrame [1],[3] is like ServiceFrame implemented using JavaFrame, and ActorFrame deals with the Actor and Role concept of a service. Finally, JavaFrame [3] is the core of the framework with regards to state machines and asynchronous communication. It is a MDK[4] for large, complex real time systems, whit an emphasis on the Active Object asynchronous communication concept. Like most frameworks, the NorARC framework is currently language specific regarding component development within the framework. All three layers of the framework, Service-, Actor- and JavaFrame are implemented by using Java [5]. Hence, all components developed within the NorARC framework must be done so using Java as programming language. The use of specialized mediators that make use of communication middleware standards like CORBA[5] and RMI[6] enables NorARC services to communicate with applications made in other frameworks and languages.

[2]Advance Window Toolkit
[3]Javabeans, www.java.sun.com
[4]Modelling Development Kit
[5]Common Object Request Broker Architecture, www.corba.org
[6]Remote Method Invocation, www.java.sun.com

1.3 UML 2.0

In general, UML[7] is a visual language for specifying, constructing and documenting software systems. UML is an open standard, which has been accepted as a common modeling language throughout the software and system industry, and is treated as today's de facto standard for specifying software architecture. OMG[8] is the body responsible for the standardization of UML, which started with UML 1.1 in 1997. At the beginning of this project UML 1.4 [21] was the current standard but UML 2.0 [4] was planned to be standarized during the timescale of this thesis project. March 2003, UML 1.5 [22] was released, but had only small changes from version 1.4. The UML 2.0 upgrading is due to the fact that version 1.x has lots of shortcomings (e.g. non-standard implementation, exaggerated complexity, imprecise semantics etc [8],[23],[24]). One inadequate part of UML 1.x often mentioned is the insufficient support for component-based development. To cope with these drawbacks UML 1.x has been used in a non-standard way. Other tools like SDL[9], ROOM[10] and MSC[11] have been used together with UML 1.x to achieve the goal.

The basic improvements from version 1.x to 2.0 are:

- better support for component-based software development.
- better alignment of other widely used standards like, XML/XMI and SDL.
- improved support for composite state machines.

OMG has issued 4 RFP[12] for UML 2.0. The first is an Infrastructure RFP, which deals with restructuring the basic constructs and improving customizability. The second RFP is the Superstructure, which concerns components, activities and interactions. The third document is the OCL[13] and concerns increasing the precision and expressive power of the constraint language of UML. The last RFP is the Diagram Interchange, which discusses the model diagram interchangeability between different UML tools. When comparing these four RFP's with UML 1.x's one RFP, some claim that this may result in an awkward and big specification, and that UML 2.0 will suffer from *second-language syndrome* [8]. Hopefully, OMG and its UML consortium will use architectural moderation to prevail UML 2.0 to be a design-by-committee compromise.

It is important to notice that the term *composite state* related to this thesis is used in two different manners:

- When talking about composite states in general, and especially in relation to UML, *composite state* is used.
- Since JavaFrame's composite state notion is captured in the class *CompositeState*, this notation is used whenever talking about composite states in relation to both JavaFrame and ActorFrame.

[7] Unified Modelling Language
[8] Object Management Group
[9] Specification and Design Language
[10] Real-Time Object Oriented Modelling Language
[11] Message Sequence Chart
[12] Request For Proposal
[13] Object Constraint Language

1.4 Thesis Definition

The thesis will use the NorARC frameworks to test the feasibility of a dynamic change in the behavior of CompositeStates. The final definition of the project is:

> *The student will evaluate the UML 2.0 Run-time Enviroment/Service Creation Technology from NorARC with regards to dynamic change of behavior during run-time. CompositeStates may contain an inner state space composed of CompositeStates. The goal is to get a better understanding of how these inner CompositeStates may be candidates for on-line replacement and thereby changes in behavior (state space) during run-time.*

The thesis title is:

> *Using Ericsson NorARC's frameworks as test bed for dynamic change of behavior based on CompositeStates.*

In addition the project will consider another objective:

> *If possible, evaluate how on-line replacement of JavaFrames CompositeStates can be used to change parts of the state space during run-time and how this can be described using UML 2.0.*

and finally:

> *If possible, a prototype will be made, demonstrating some of the concepts involved.*

1.5 Work Description

The notion *three zeroes* [6] is a technique introduced to aid server developers and administrators in their struggle for 100 percent server and service availability. Three zeroes includes:

- Zero development
- Zero deployment
- Zero administration

Zero development does not mean *No* development, but to develop in a way that enables the code to be reused, preferably as components. Zero deployment is a struggle to minimize the time and work used to push out new developments. There are several ways to this, and by developing components that are deployment-friendly much is achieved. Zero administration signifies how to reduce time and effort to maintain and administer a system. The NorARC frameworks are good candidates for all of these aspects, building on well proved techniques and patterns. Some aspects within development and deployment will be investigated through the following questions, which are fundamental for this project:

- Are JavaFrame based and ActorFrame based applications, candidates for dynamic change in run-time, by introducing new classes?
- Are JavaFrame based and ActorFrame based applications, candidates for dynamic change on regards to the state machine behavior (i.e. without ending a play or introducing a new actor or child actor)?
- Are CompositeStates candidates for run-time replacement in applications based on both JavaFrame and ActorFrame (i.e. without ending a play or introducing a new actor or child actor)?

Further, some other relevant questions are also addressed:

- How can all this be modelled by using UML 2.0?
- Does run-time replacement of composite state violate good programming practice within asynchronous communicating state machines?

The answers to the above questions, apart from the two last, are based on two concepts, Java and the NorARC framework. Firstly, the possibilities for introducing and changing components run-time to a Java program will be evaluated. Java is a widely known programming language, thus it will not be described as a language, but only the parts with regards to dynamic change are discussed and described. Secondly, the NorARC framework and especially CompositeStates will be analyzed with regards to dynamic change of behavior. Thirdly, UML 2.0 will be evaluated as the candidate to formally describe the CompositeState substitution, and the dynamic change of behavior. In this thesis, *dynamic change of behavior* implies changing the static structure of a state machine, providing new behavior. This is in contrast to what is normally understood by this expression in the context of UML, and especially interaction diagrams [19],[20]. UML is language independent and shall describe concepts only, not any language features.

In this thesis a prototype, in which a dynamic change of behavior in a state machine occur, is made with the framework of NorARC, but without making any modification to either Java- or ActorFrame. Suggestions of changes and modifications to both are specified for future work.

1.6 Report Outline

The remaining part of the report, describes the technologies described above. Chapter 2 will elucidate some basic topics regarding both services and the NorARC frameworks, namely state machines and asynchronous communication. Chapter 3 describes the NorARC framework as a whole, and then each of the three distinct parts of it. In Chapter 4, the parts of the UML 2.0 proposal relevant to this project, are described, followed by patterns, methods and architecture relevant for this thesis in Chapter 5. Based on the five first Chapters, Chapter 6 presents some conceptual solutions to problems related this thesis. The prototype, with its design and implementation, inter alia, is described in Chapter 7, and finally the last three Chapters, 8, 9 and 10, presents Results, Discussion and Conclusion.

Chapter 2

State Machines and Asynchronous Communication

2.1 Introduction

State machines are important mechanisms of service creation within the NorARC service creation environment and programming in general. Shown later, both state machines and asynchronous communication are crucial parts of the actor notion. To get a better understanding of these concepts, this chapter introduces and discusses their most important aspects in relation to this thesis subject. First, finite and communicating state machines are debated, followed by some important issues regarded concurrent programming. Then, the actor based approach shows how all this can be fitted together.

2.2 Finite State Machines (FSM)

FSM is a technique that allows simple and accurate design of sequential logic and control function. Designing systems using state machines enables you to make the design more sophisticated and with more ease than normal. Also, by designing state machines with a formal notation, one can formally verify their behavior, even with mathematics.

2.2.1 What is a State Machine?

A basic concept of FSM is that a system can only be in a finite number of states (e.g. an electrical light switch can be either *on* or *off*, indicating two states). A more complex example is a microwave, which has more states than a light bowl switch and have states not visualized for the user. A state machine can only be in one state at a given point in time. Naturally one can switch back and forth between two states, like rapidly switching a light switch, but at any given point in time the state machine is at most in one state. This can be shown in a state diagram of the electrical light bowl switch in Figure 2.1. If the switch is in the state *off*, one can cause the light to switch *on* by pushing the switch, which is an input to the

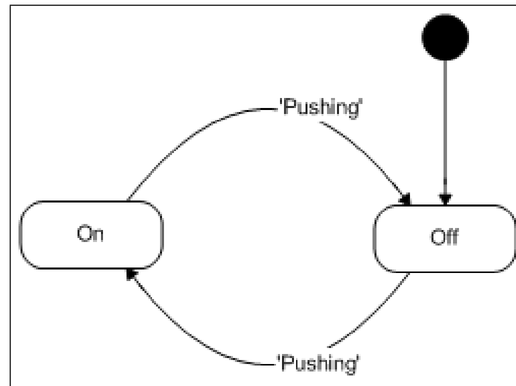

Figure 2.1: State chart of an electrical light bowl switch.

state machine, and vice versa if the current state of the state machine initially was *on*. This is very simple and also very fundamental concepts for state machines. A state machine must, based on its current state, evaluate the input to give the correct output, or move to a new state. Electrical light switches are easily understood state machines because the states *on* and *off* are visually understandable. Most state machines do not have any visual interfaces to their inner states, and most have more than two states. The input and output are also often too complex to understand by mere a visual examination. The input is often what drives the state machine to a new state, either by just 'waiting' for some external event to come along, or by actively look for external events, that are significant to its current state and previous input. Each state in a FSM has a set of one or more transition rules. That is a rule for *what state to enter next given the current state and current inputs*. If several rules are executable the, FSM is said to be non-deterministic. Some definitions of state machines are:

- a state is said to be transient if once left there is no way back to that state.
- a state is said to be a sink state if once entered then there is no way out of this state.

Each bubble in the state chart represents a state. As shown above there is a finite and rather small number of states, and all states are known in advance. This project will deal with state machines that have the possibility to change some of its states, by replacing them with sub machines or composite states. But first lets look at communicating FSM, CFSM.

2.2.2 CFSM

Communicating FSM models are extensions of FSMs. This allows communication and synchronization among FSMs. There are several synchronization paradigms, but that is a subject outside the scope of this project. CFSM is communicating by connecting the output from one machine to the input of another FSM. As a particular case an FSM can communicate with itself by a suitable feedback from its output to its input. The CFSM must execute two-step algorithm:

1. Inspect the input signal and select a transition rule.

2. Change its control state and update its output signal.

CFSMs can be either tightly or loose coupled. Tightly coupled CFSMs use synchronous communication or rendezvous channels to communicate. This means that two CFSMs must be at a given point of execution, in order to send and receive signals. For example, the sending CFSM must be at its sending point of execution, and simultaneously the receiving CFSM must be at its receiving point of execution in order for the information to be exchanged. If either of them reaches a sending or receiving point of execution, it must wait at this point until its opponent reaches the antagonism point of execution. If one CFSM waits at *send*, the other CFSM must reach *receive*, or vice versa, in order to have a communication. A less tightly coupled system is CFSMs, which use asynchronous message passing. The CFSM are then coupled via FIFO message queues. This means, that a CFSM may send a message to another CFSM and then *move on* (e.g. entering a new state or executing new statements without waiting for the opponent to read the message). This way of parallel computing makes the system more modular.

2.3 Concurrent Programming

When dealing with real world problems, like the one mentioned in Introduction, real-time software design is of utter importance, since a failure of the system may have catastrophic consequences. On the other hand, real-time systems are complex to design and verify since they are models of the physical world, which usually behaves in a non-deterministic manner. Therefore, one major concern when designing real-time systems, are management of concurrency. Concurrent activities in large systems are often managed by creating application-controlled context switching [11] at times when there is little or no context that needs to be saved. As shown in the chapter 3, active objects abstraction is an example of such application-specific concurrency management.

There are two ways of performing communication, either shared data or message passing.

Shared data Using shared data, two or more threads can communicate by accessing the same data. To prevent inconsistent updates, critical sessions and mutual exclusion are used. In order to use shared data amongst threads or processes, they must share the same memory. This way of synchronizing data is simple but may be difficult to understand and maintain, and is therefore often erroneous. There may also be times when shared memory is not acceptable, both due to complexity issues as well as performance issues.

Message Passing Message passing can be divided into synchronous and asynchronous communication. Both these forms of communication uses the encapsulation synchronization mechanism, where the shared data is encapsulated as private data, only accessible through operation invocation.

Further, when discussing concurrency there are two types of design paradigms that need to be mentioned, time-driven and event-driven [11]. The former style is widely used within the producer-consumer relationship, feedback control loop

and digital signal processing. The latter is widely used with large complex system and when unpredictable asynchronous events occur. This often involves queuing messages for future processing while working on the current task. The combination of event-driven software design and object-orientation makes the active object concurrency abstraction suitable for representing finite state machines, because the FSM specification gives a behavioral specification of the active object [11].

2.4 The Actor Based Approach

Picture all the advantages of concurrent communication mentioned above assembled into one notion, and you have the outline for an actor. One definition of an actor is *all computation in terms of various patterns and idioms of message passing among entities* [12]. Another similar definition is *autonomous objects: ... objects in that they encapsulate data, methods and an interface ... autonomous in that they encapsulate a thread of control ... interact with their external environment by sending and receiving messages* [13]. The actor notion provides us with an easy model for making communicating components. Asynchronous communication is a rudimentary form of communication, for which other communication abstraction may be implemented. Hence, asynchronous communication is an important part of the actor model and actor languages. This is embodied in the actor model kernel implementation guidelines [13]:

- messages needs to be buffered at their destination.
- the communication must be 'non-blocking'.
- provide scheduling algorithm that ensures application-controlled context switching to optimize processor utilization.

Another issue worth mentioning when talking about actor communication, is that an actor message recipients are specified using a location-independent email address, maintained in a name table. This lightens the actor migration.

By using actor languages one may build a complex system at a higher level of abstraction, because it is possible to hide unnecessary architectural details. More information of the notion of actors is presented in chapter 3.

2.5 Summary

This chapter has demonstrated which aspects are involved when dealing with concurrent communicating components, especially concurrent communicating state machines in large complex systems. It has been shown that there are paradigms suitable for such software development (e.g. actors). By leaning to the actor approach one can create application at a higher level off abstraction, leaving out architectural details. The next chapter will show how Ericsson has implemented a framework to use the actor notion to cope with such large complex software systems.

Chapter 3

NorARC's Service Creation Environment

3.1 Introduction

Ericsson NorARC service creation environment was made to cope with large, complex software systems and service creation within the telecommunication domain. This work can be tedious and expensive when done in an *old-fashioned* manner. By making a new service creation environment, Ericsson could fragment the work involved in creation of services, making each service modular. This is both direct and indirect cost effective. By reusing modules, one gets an direct cost reduce, and by making projects simpler and more surveyable, one has a indirect cost reduce. The service creation environment from NorARC consists of the frameworks, JavaFrame, ActorFrame and ServiceFrame. This chapter gives an outline of each part of the framework stack, which is shown in Figure 3.1.

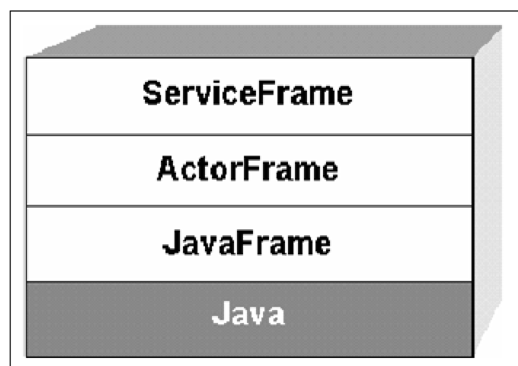

Figure 3.1: The NorARC Stack

3.2 JavaFrame

JavaFrame was introduced to fill the gap between modeling tools with good support of modeling system structure, like ROOM or SDL and modeling tools for good support for Java, like UML. Therefore, JavaFrame can be seen as a convergence

of ROOM/SDL, UML and Java, which benefits from ROOM/SDL modeling, and the support of 1:1 relationship between model and Java from UML. By this, one can model the system structure in a way that makes it possible to analyze, but still implement it using Java. JavaFrame can be viewed in two different ways [3]: *an advanced API or a specialized language with support for modelling concepts.* As an API[1], JavaFrame can be seen as an extension to the JSDK[2] package already available by SUN[3]. By using the standard Java packages, classes from JavaFrame and user-defined classes, one can make JavaFrame applications, just as any other Java extension, as shown in Figure 3.2.

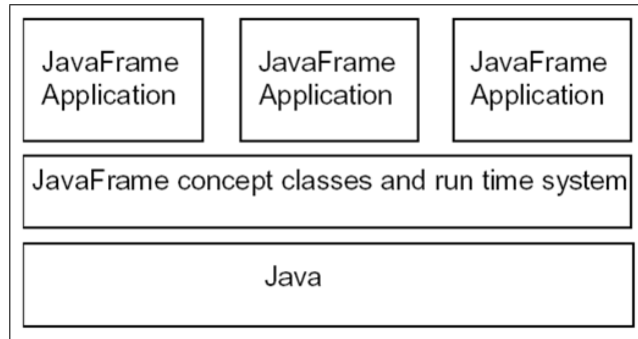

Figure 3.2: JavaFrame classes and run-time system.

When observing JavaFrame as a language with support for modeling concepts, Figure 3.3 can be a good help. Starting at the far left side, UML 1.4 is used with regards to JavaFrame, a common model approach when using java extension. The next modeling and transformation is done with a JavaFrame profile. Neither of these approaches makes a 1:1 mapping between model and code because the transformation is done in a JavaFrame specific way. To have a 1:1 mapping, one needs a UML like modeling language specific for JavaFrame. The difference between this model and the former is that the modeling language is JavaFrame specific, not the mapping. Also, when using JavaFrame for modeling purposes, classes that are introduced for implementation purposes only, are not graphically presented, making the system structure more obvious [3].

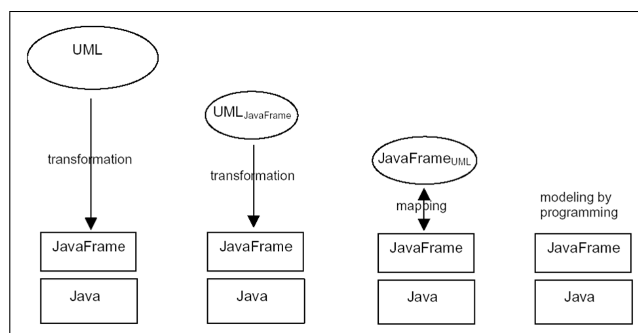

Figure 3.3: JavaFrame - alternative modeling notation.

[1]Application Programming Interface
[2]Java System Development Kit
[3]www.sun.com

3.2.1 Active Objects and Composites

In addition to state machines and mediators, active objects and composites represents the backbones of JavaFrame. Active objects are described as an *abstract class representing the notion of something that acts on its own, sending and receiving messages through its associated mediators* [3]. The interaction between active objects is done asynchronously, because this is generally known as a well-proven concept and has advantages over synchronized message passing (e.g. the parties can handle messages in individual pace). A conceptual model is given in Figure 3.4.

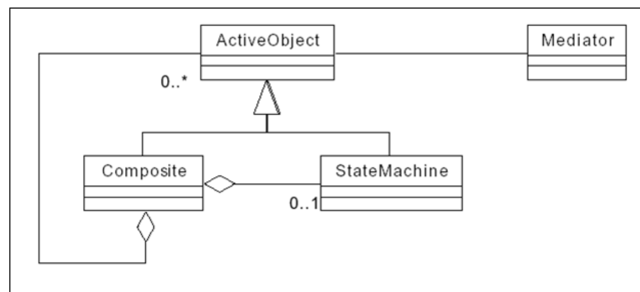

Figure 3.4: Active object model.

The model in Figure 3.4 shows two things:

- There may be substructures of interacting active objects making the model scalable.
- The communication between active objects *appears to be the same regardless of whether it is a Composite or a single StateMachine* [3].This transparent communication is possible due to the mediators, which is explained later in this chapter.

By using Java to define JavaFrame, Ericsson made it possible to make a structure of a composites superclass and for the transitions of a state machines superclass to be inherited to child classes. Also, there may be several composite objects of the same class. This introduces the possibility for a class to be re-entrant, which leads us to the next element; state machines.

3.2.2 State Machines and CompositeStates

JavaFrame specific state machines are said to be re-entrant, meaning that *many states machines refer to the same state machine structure* [3]. This means, that if the source state machine structure changes, all state machines built on this structure also changes. Each state machine has a set of State objects whose structure is separated from the state machine. It is important to grasp this concept, because in order to dynamically change the behavior of a state machine, we need to alter/reorder the set of States. If the States are to tightly coupled to the state machine, any dynamically change in the set of states would make it necessary to change the state machine, any update will effect all the state machines at the same time[4].

[4] state machine objects made from the same class

The JavaFrame complexity does not stop with the state machines being re-entrant. State machines also have CompositeStates, a state with inner or internal states as shown in Figure 3.5. A CompositeState may also contain CompositeStates making a nested structure of CompositeStates.

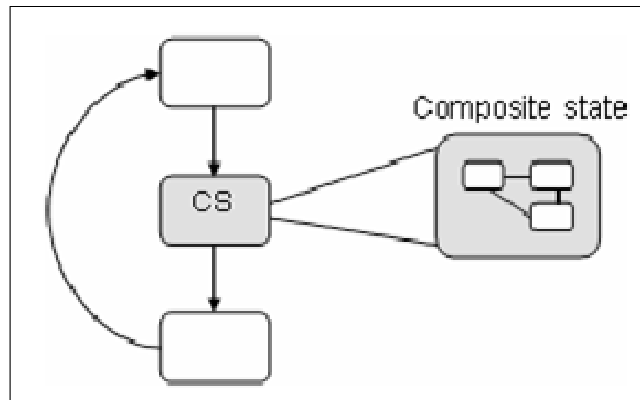

Figure 3.5: CompositeStates.

Since state machines can send and receive messages in an asynchronous fashion, a message queue is implemented for each state machine and managed by a scheduler. The scheduler serves the state machines in a round-robin fashion.

CompositeStates are states with inner states, either new CompositeStates or atomic states. Thus, a state structure can be hierarchically and arbitrary deep, ending in an atomic state. To enter and exit CompositeState, entry and exit point respectively are used. They are given an identity of the type integer. The CompositeState contains all definitions of its inner states, and all the actions related to the transitions between the inner states and their entry and exit points. To manage this, the following inherited methods can be used:

protected final void exitState(int exitNr,StateMachine curfsm) used when leaving the CompositeState through a given exit point.

protected final void sameState(StateMachine curfsm) The CompositeState stays in the current leaf state.

protected final void save(Message sig, StateMachine curfsm) The receiving message is not consumed at the moment or in this state and is saved for later execution.

protected final void output(Message sig, Mediator receiver, StateMachine curfsm) Sends a given message to a given local mediator.

public void enterState(int enterNr, StateMachine curfsm) Used to enter CompositeStates, that have more than a default entry point. If a default port is used, the integer is not a part of the method signature.

protected boolean execTrans(Message sig, State st, StateMachine curfsm) This function is called (but not by the user code) to execute a transition. If the transition is executed or saved, this method returns true.

protected void outofInnerCompositeState(CompositeState cs, int exNo, StateMachine curfsm)

This method is used if the CompositeState has an inner CompositeState, and this inner CompositeState should be exited trough one of its exit points. The index of the exit point is linked to an inner atomic or a new CompositeState, the exit point of the current CompositeState or the entry point of another inner CompositeState.

3.2.3 Mediators

Mediators are a very important concept in JavaFrame. In order for large software projects to evolve, the need for component independence is crucial. Mediators are the components that make this happened in JavaFrame. It is the glue that makes the system a whole. *Mediators are objects that represents interaction interface between Active Objects* [3]. Mediators comes in different forms, all inherited by the superclass Mediator. Figure 3.6 shows some of the mediators implemented in JavaFrame.

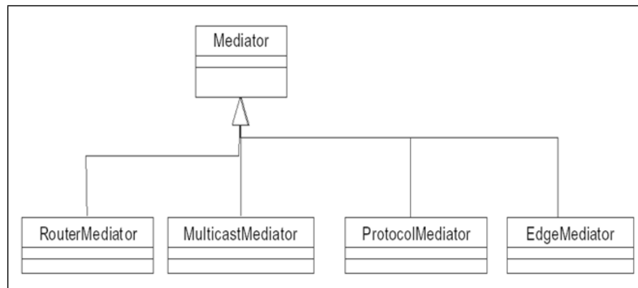

Figure 3.6: Different type of mediators.

By having different types of mediators, a developer do not need to know what is outside his scope, he only need to know what mediator to connect to, and each state machine only addresses its innermost mediators. Some Mediator description from [3]:

- RouterMediator has the possibility to route the message to the appropriate next mediator, this is made possible by using routing databases.
- MulticastMediator can replicate a message and send it to several different mediators.
- ProtocolMediator makes the application independent of communication environment by hiding the lower levels of protocols.
- EdgeMediators makes it possible for a JavaFrame architecture to communicate with exterior systems.

A StateMachine discerns between an inward and an outward mediator, so no message can be sent in to an outwards mediator and the StateMachine cannot send messages to an outward mediator and every StateMachine or Active Object knows which Mediator it communicates through. Figure 3.7 shows the use of mediators. Different color, white and light grey, discern between inward and outward mediators.

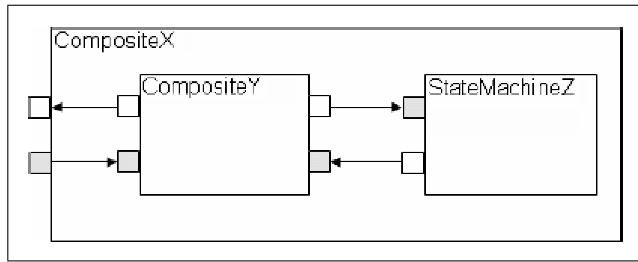

Figure 3.7: The use of mediators

3.2.4 The Trace File

JavaFrame is supported by a trace system. This makes it possible to trace the execution of various state machines, their transitions and messages that are sent and received. When the code is executed, the logic can be followed by using the supporting trace observation tool as the example in Figure 3.8 shows. This trace system is also available from both Actor- and ServiceFrame. Each framework has an extension special for that framework, like ServiceFrame has extension for tracing *roles involved in a play* [1].

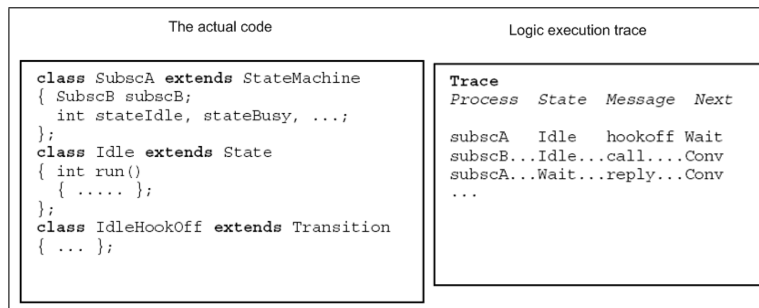

Figure 3.8: The logic execution trace.

3.3 ActorFrame

In general, actors are computational agents, which act according to a certain behavior as response to incoming communication. When considering the NorARC ActorFrame, *an Actor is a (composite) object having a state machine (ActorSM) and an optional inner structure of Actors* [1]. As shown earlier, ActorFrame resides between JavaFrame and ServiceFrame, and here the actor is the core component. Historically the actor notion was introduced based on the perception that VLSI[5] would lead to the widespread availability of computers with very large number of relatively small processors. Today's telecom network can be seen as such, with lots of interacting processors in form of mobile phones, PDA's, etc. An actor can be seen as a autonomous component, with the ability to play roles. Different roles are played by different inner actors, all managed by a management functionality. The available roles and their rule of conduct are provided by a deployment descriptor. The term play is used for a *dynamic structure of interacting actors performing ac-*

[5] Very Large Scale Integration

cording to role types [1]. This term is used both in ActorFrame and ServiceFrame and has the same signification. Figure 3.9 shows a play and the actors associated through roles.

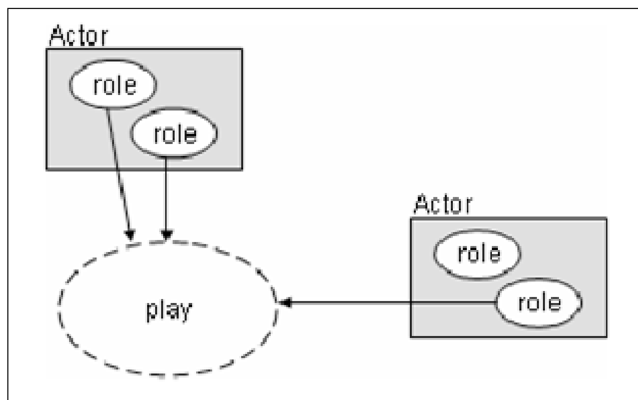

Figure 3.9: A play and its participating actors.

An actor has a recursive structure, meaning that it can contain instances of its own type, so-called inner actors. These inner actors lifecycle are handled by the outer actors default behavior. The ActorFrame is a framework that uses JavaFrame to create an environment for development and testing of actors. It supports concurrent computation processing, process movement between actors and dynamic reconfiguration [10].

The Actor concept mapped down to classes are shown in Figure 3.10, and the following description is from [14].

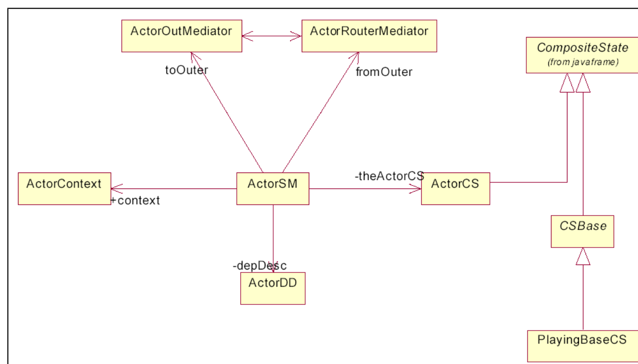

Figure 3.10: Class structure for ActorFrame

- ActorSM - is the base for all further role definition in ServiceFrame. It contains variables for managing sub-roles and lifecycle management.
- ActorContext - holds the reference to the surroundings.
- Play - detonates the association between Actors taking part in a play.
- ActorDD - holds the definition of the Actor Deployment Descriptor file.
- ActorAddress - holds the unique entity of the Actor.

- ActorOutMediator - the output port of all messages from this Actor.
- ActorRouterMediator - contains the routing algorithm. This routs the messages to its belonging Actor or one of its child's.
- CompositeState - this is the basic class for defining transitions and states. This is the core object of this project.
- ActorCS - Contains the logic and sub states for managing sub-roles and lifecycle management.
- CSPlayingBase - The basic for defining the virtual Playing state.
- CSBase - basis for all new states.

3.4 ServiceFrame

ServiceFrame is the topmost level of the NorARC framework, and it is an application of ActorFrame with generic support for actors, plays and roles [3]. ServiceFrame was introduced to relieve service designers from the problems involving creation, deployment and execution of services. ServiceFrame has 5 core elements:

1. *Conceptual abstraction* is generally to separate the functionality from the implementation. This conceptual abstraction is made by introducing communicating state machines by using JavaFrame. Furthermore, as stated in the section about JavaFrame, the state machines communicate in an asynchronous manner to further ease the burden for the service modeler or designer.
2. *Environment mirroring*. ServiceFrame is not designed for a particular set of services, but by using general patterns, it support a wide range of services to be rapidly provided.
3. *Role modeling*. A service is a set of actors playing different roles, and each actor can play several roles, and a role can be played by several actors. With ServiceFrame, this actor concept is separated from the role concepts to achieve *better modularity and reusability as opposed to traditional Actor oriented design* [3].
4. *Service centered architecture*, in contrast to network centered, is an architecture, which emphasizes the services [18], and has no specific type-of-service feature. It is also, as far as possible, independent of the underlying network architecture.
5. *Frameworks and patterns.* Both ServiceFrame and ActorFrame are built on the JavaFrame framework, which is implemented using well-known working patterns. This use of patterns in JavaFrame influences both ActorFrame and ServiceFrame.

3.5 Summary

It has through this chapter been shown that Ericssons NorARC framework is an ideal framework for service creation. By use of JavaFrame and ActorFrame, the ServiceFrame is completely independent of today's technology regarding the communication and service aspect. All frameworks are built on well proven techniques and patterns that aid application and service creation. By using non-strict architectural layering the NorARC framework may be hard to learn, but when mastering the framework, application and service development are easily done. In lack of good tools for modeling these frameworks, Ericsson has combined techniques from different modeling environment to aid their modeling. Ideally, all this should be modelled in one environment, UML 2.0.

Chapter 4

UML 2.0

4.1 Introduction

This chapter embroiders some aspects of the UML 2.0 superstructure. Since UML 2.0 is used as a modeling language, and since the NorARC system is based on expectation of what UML 2.0 will be, some basic understanding of UML 2.0 is needed, especially within the subjects' components and state machines. The information in this chapter is an extract of the superstructure released from OMG. At the time of writing, the superstructure has not yet been standardized and may not be standardized before after the end of this project.

4.2 Components and Composite Structure

A component within the UML 2.0 notion can be considered as an autonomous unit within a system, which provides and requests interfaces to communicate with its environment. UML 2.0 introduces components as an aid for defining software of arbitrary size and complexity. By using components, the software development is meant to be more modular and reusable. UML 2.0's component package supports both logical and physical components. UML 2.0 Components has two packages, the BasicComponents and the PackingComponents, the former package *focuses on defining a component as an executable element in a system* [4] and the latter *focuses on defining component as a coherent group of elements as part of the development process* [4]. Components can be seen as a type of composite structure. Interfaces play an important role within component-design, and UML 2.0 has pursued this by introducing *provided interfaces* and *required interfaces*. These former interfaces are the kind of interface known from UML 1.x, which may be implemented and realized by a classifier (e.g. component). The latter interface represents the set of interfaces required from other components, in order to fully provide its functionality or function properly. Through interfaces a component can also provide and require ports, and this way a component can specify the service provided or required by a *specifier* (e.g. component), to its surrounding environment. The encapsulation of a components inner structure makes it easier to substitute a component in both design time and run-time, as long as the replacing component offers equivalent functionality through its port definitions. These public properties are

only visualized as an external view through the ports. The internal view, or private properties, shows how the external behavior is realized internally. The external and internal view of a component is shown in Figure 4.1 and 4.2, respectively.

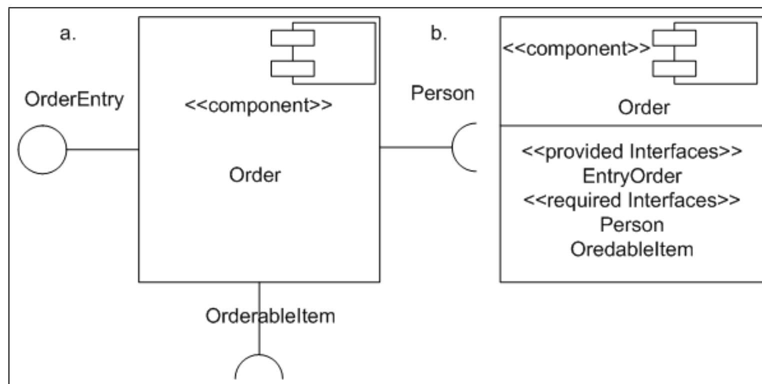

Figure 4.1: Black-box or external view notation of a component.

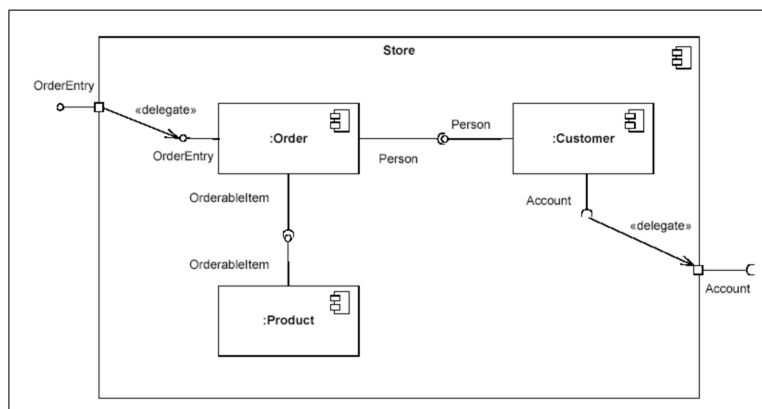

Figure 4.2: An internal or white-box view.

The connection between ports from different components are realized with *connectors*. There are two kind of connectors, *delegation connector* and *assembly connector*. Delegation connectors link the external part of the component to the internal realization of that behavior by the component's part. The link in Figure 4.2 labelled <<*delegate* >> is such a connector. Assembly connectors link the connection between two components antagonizing ports or interface (e.g. a providing and requiring port or interface). The notation *ball-and-socket* shown as link from Order to Customer labelled *Person* in Figure 4.2 is a assembly connector.

Figure 4.3 shows the subpackage dependencies of *composite structure*. Some of these subpackages are described in the rest of this section.

The Port subpackage provides mechanism for encapsulation a classifier from its environment. Through ports it is possible to hide the inner structure and functionality of the classifier, and at the same time provide an interface for public functionality, and creation of classifiers whiteout knowledge of the environment it will be embedded in.

Collaboration *allows us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific role that the instance will play* [4].

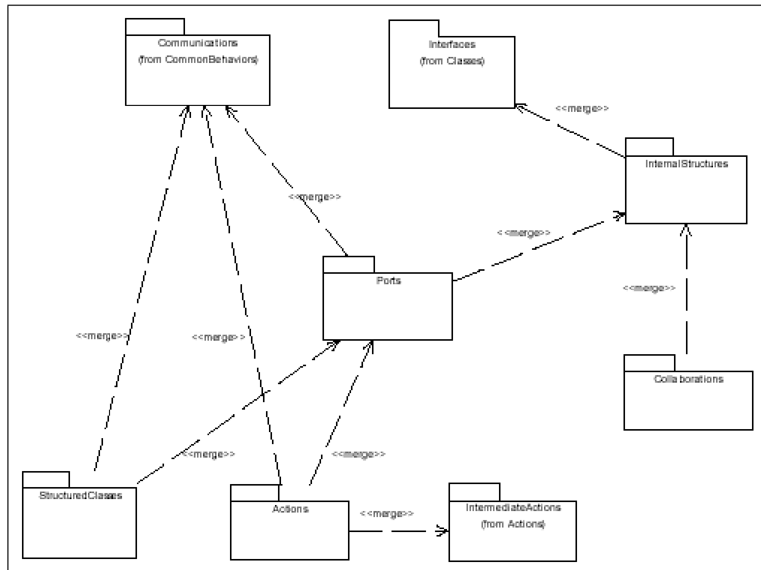

Figure 4.3: Dependencies in the CompositeStructure package.

Further, [4] states that by using interfaces one is *allowed to external observe the properties of an instance to be specified without determining the classifier that will eventually be used to specify this instance.*

The StructuredClasses subpackage supports modeling classes with an internal hierarchical structure, encapsulated via ports. Although very similar to BasicComponents from the Component package, the modularity of components is never mentioned in the context of structure classes.

4.3 State Machines

The state machine notion is a convenient way to define the lifecycle of object, and UML 2.0 defines two different kinds of state machines; *behavioral state machines* and *protocol state machines*. A behavioral state machine describes the behavior of various elements (e.g. classes), while the latter kind of state machine express usage protocols, legal transitions that a classifier[1] can trigger. Figure 4.4 shows the state machine class structure. The following will describe some of the elements from the state machine model from Figure 4.4.

4.3.1 ConnectionPointReference

A connection point reference is used to represent an entry or exit point in a sub state machine or composite state. These points are source or targets of transitions in order to leave or enter sub state machines. A connection point reference for entry points are visualized by placing a circle on the border of the composite state. For an exit point, the connection point reference is an encircled cross on the border of a composite state or submachine state. Figure 4.5 shows a composite state with an entry and exit point.

[1] A collection of instances that have something in common.

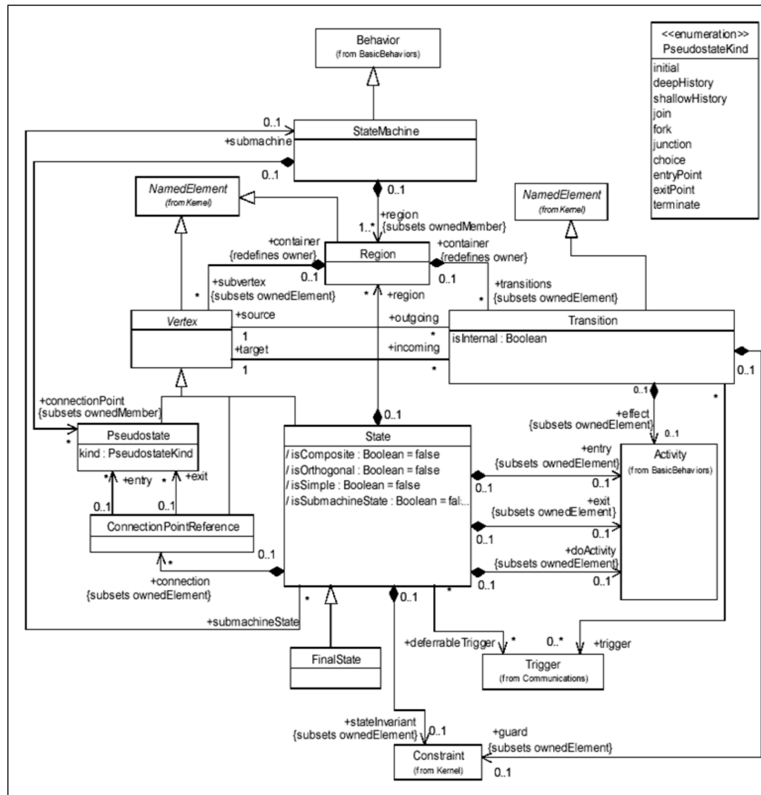

Figure 4.4: The state machine class structure.

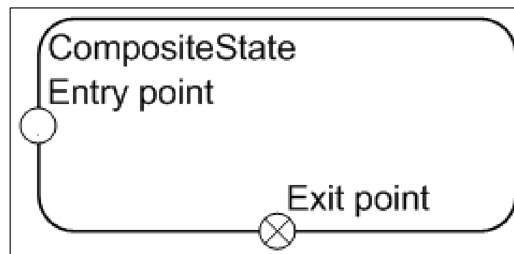

Figure 4.5: Composite state with entry and exit points.

4.3.2 FinalState

A final state is a special kind of state, which indicates that the enclosed region is completed. *If the enclosing region is directly contained in the state machine and all other regions in the state machine also are completed, than it means that the entire state machine is completed* [4].

4.3.3 State

A state in a behavioral state machine is a situation during which some invariant condition holds. One distinguishes between three kinds of states.

- Simple state
- Composite state

- Submachine state

A simple state does not have sub states or submachine state machine. During execution, a state can be either active or inactive, and in a non-hierarchical state machine only one state can be active at the same time. An inactive state may become active if one of the transition rules applies, and it may become inactive by a new or the same transition. As long as a state has the Boolean value of *isFinal* set to false, the state may be redefined by extending the state. A state can be extended by adding composite states and a composite state can be extended by adding new state or new inner composite states.

Composite states contains sub states, either atomic sub states called direct sub states, or an indirect sub states, sub states with inner sub states. Composite states may also contain one or more regions. If it contains more than one region, the regions must be orthogonal in relation to each other.

Stated earlier, non-hierarchical state machines can only have one state active at a time. Hierarchical state machines can have several active states at a time, and also several atomic states active at a time. First, if a sub state is active, then its parent state is also active. Second, if an orthogonal region is entered, each of its orthogonal regions are also entered, subsequently one atomic state in each orthogonal region can be active at the same time.

A semantically equivalent to composite state is sub state machine. The main difference is that a there are defined entry and exit points for entering and exiting the sub state composite.

4.3.4 State Machines

UML defines state machines as objects that can be used to express the behavior of a system or parts of a system. It can be seen as a graph with each state as a node, and with the possibility to move from one node to another by transitions that can be triggered according to transitions rules. When traversing from one node to another the state machine executes a set of activities specified within each state. As an example, consider the soda machine that accepts coins, and after collecting a given amount, releases a soda. When inserting the last dime, the state machine moves from a given state to the state *release soda*. Activities associated with this transition are in the simplest form, the action of releasing a soda to the customer. Each instance of a state machine has its own event queue, which holds instance events relevant for the state machine. The order of handling events in the queue is not defined, leaving open the possibility to use different scheduling schemes.

Event processing is based on the *run-to-completion* assumption, which means that *an event can only be dequeued or dispatched if the processing of the previous current event is fully completed* [4]. Like the NorARC framework shows, one way to implement run-to-completion, is to let an event-loop run in its own thread and read events from the queue. This is exactly what the scheduler class in the NorARC framework does, handling the event queues for every instance of the state machine by running its own thread.

Like states, composite states and sub states, state machines can be extended or redefined by adding new states, composite states, regions, transitions may be have

its target replaced etc.

4.3.5 Transition

A direct relationship between a two vertexes[2], source vertex and destination vertex is called a transition. Transitions are split into four groups:

- high-level (group) transitions, orients from a the composite state themselves reflecting the exiting of the composite state.
- compound transitions, made up by one or more transitions and orients from several states with one or more states as target. Both the source and the target may be the same state or set of states, called self-transition.
- internal transitions, executes without exiting or re-entering the state in which it's defined.
- completion transition, a transition where the source is a composite state, sub state machine state or an exit point and without an explicit trigger. This transition will generate a *completion event* which is an implicit trigger for completion trigger.

Transitions are visualized by a solid line connecting two vertexes and separate the source and the target vertex, by placing an arrow at the target end of the line. For the furtherance of understanding the model, it is possible for the line to be labelled with a trigger-signature, guard-constraint and activity-expression as shown in figure 4.6.

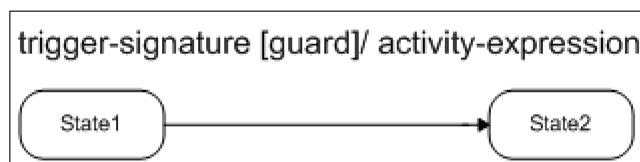

Figure 4.6: A transition example with incoming and outgoing messages

The trigger-signature is a list of one or more names of triggering events, of the form, *attribute-name* or *attribute-name : type-name*.

A guard-constraint is a Boolean expression based on a formal constraint language like OCL[3] written in terms of parameters of the triggering event and attributes and link of the context object.

Activity-expression is used to describe actions that executes when the transition fires. There may be a sequence of distinct actions and also actions that generates events, like sending signals or invoking operations.

[2]A vertex is an abstraction of a node in a statechart graph.
[3]Object Constraint Language, www.ibm.com

4.4 Interaction Diagrams

Interaction diagrams comes in different variants, like sequence diagrams, communication diagrams and activity diagrams. The most common diagram is the sequence diagram. Sequence diagrams are used to show associated messages, arranged in a time sequence. Figure 4.7 shows that while a sequence diagram look

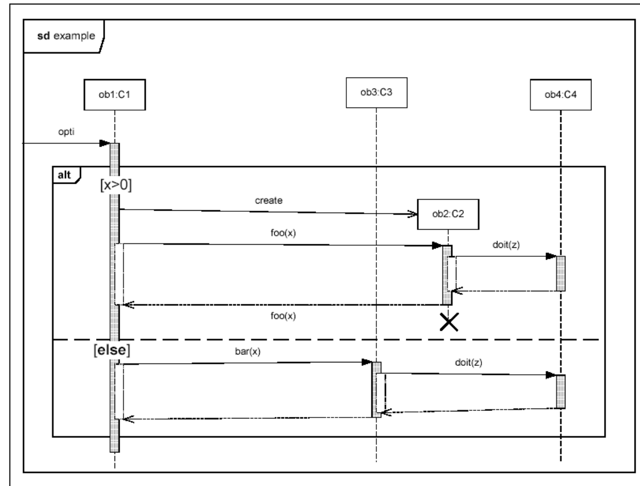

Figure 4.7: Sequence diagram

much like the same as in UML 1.x, there are some apparent differences. The outermost frame with the compartment labelled *SD example* is the operator for the interaction fragment. The operator *SD* names the fragment or sequence diagram. Listed comes all alternative operators:

sd - named sequence diagram or fragment.

ref - reference to interaction fragment.

loop - repeat interaction fragment.

alt - selection.

par - concurrent section.

seq - partial ordering.

strict - strict ordering.

assert - required.

opt - optional.

neg - negative specification.

break - breaking scenario

consider - consider messages within this fragment, example, *sd M consider*{*t,r*}, which means that messages t and r should be considered in the fragment M.

ignore - messages are ignored within this fragment.

region - critical region.

Figure 4.8 shows a guard in the lifeline of *C3*, denoted *[x >0]*. It this guard stands then the upper part of the *alt* fragment is executed. Otherwise, the lower part, separated by the dashed horizontal line, the *operand separator* is executed. This new notation for sequence diagrams improves the diagrams scalability and ability to specify behavior.

Communication Diagrams focus on the interaction between Lifelines where the architecture of the internal structure and how this corresponds with the message passing is central [4]. These diagrams corresponds to simple sequence diagrams that do not use structuring mechanisms. Non-arrowed lines are used as notation of messages while the sequence of messages is shown by a sequence numbering scheme.

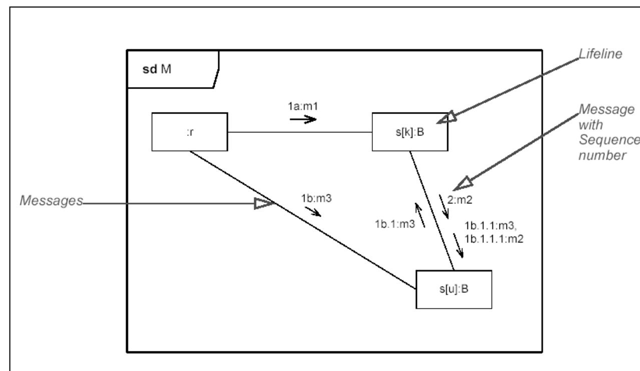

Figure 4.8: Communication diagram

4.5 Summary

This chapter has focused on some important aspects of the proposed UML 2.0 standard. The structure of both components and composites are described along with state machines and how they are modelled in UML 2.0. Changes in the diagrams needed for this thesis, along with introduction of a new diagram, communication diagram were also provided. The final UML 2.0 specification may diverse from the proposal as the work of the specification progresses.

Chapter 5

Patterns, Methods and Architecture

5.1 Introduction

Through Chapter 3 and 4, the manner of operation for both the NorARC's framework and UML 2.0 is shown. This chapter will show mainly what techniques in general can be used to accomplish the goal of the thesis definition. Since the NorARC framework entirely is based on Java, the first part of this chapter will be dedicated to Java and how Java as a programming language in general can be used to accomplish the goal. Further, JavaFrame and ActorFrame are discussed on regards of a conceptual solution. Finally, the methods of UML on regards of dynamic changes during runtime are discussed, but first, dynamic Java.

5.1.1 Dynamic Java

Java is a well known programming language, hence only a small part of Java is discussed here. Java is a dynamically linked system. This means that instead of linking a complete program before execution, the JVM[1] links classes and interfaces on demand during execution if they are required. The JVM links the required components by using the bootstrap classloader to load the binary represented classes and/or interfaces. When loaded into memory, classes and interfaces will be verified and prepared in order to be used in the program. These parts consist of checking the byte code for inconsistency and making appurtenant tables and variables respectively.

As an extension to dynamic linking comes run-time loading. Unlike traditional Java programming, run-time loading take the advantages from dynamic linking to load a class while the program is running, using the standard or a custom class loader. By this approach, a Java program can be changed or extended during runtime, with classes that did not exist when the program first was executed. This is a technique often used in Java, especially when loading drivers for databases.

The latter method to introduce new classes or interface to a Java program already

[1] Java Virtual Machine

running has a drawback. The class or interface introduced at run-time must be pre compiled in order to load it. A class loader, which the name implies loads classes with the extension *.class*. A regular Java file has the extension *.java and only after compilation to byte code there will be made a *.class file. To make a Java program fully generic it should have the possibility to load files not yet compiled to byte code. In other words, let the program itself compile the code before loading it. It is possible to make CompilerClassLoaders [6] that has the possibility to both compile and load classes and interfaces. This could make it possible to introduce *.java files not yet compiled to a Java program currently running. This approach is not recommended because first it is a violation [6] of the JDK's [2] licensing scheme, so an approval from Sun is needed. Second, by making it possible for a Java program to compile a **.java* file run-time, one forces the target system running the Java program to have parts of the JDK in the CLASSPATH, or else a run-time error will occur. Regularly, only JRE [3] is required for the target system to run a Java program. Therefore, the methods used in this project will not enforce the target system to implement further parts of Java than normally used to run Java programs. In other words, only run-time loading of pre-compiled classes will be used.

Since the NorARC framework is assembled by three distinct frameworks, which can be viewed as levels of abstractions, methods from both Java- and ActorFrame are here described. Shown from the NorARC stack in chapter 2, JavaFrame is, next to Java, the lowest level of abstraction. Methods from that part will be describes in the following section.

5.2 JavaFrame Guidelines and Patterns

Stated earlier, JavaFrame can be seen as a Modelling Development Kit or MDK. Technically this is just a framework with appurtenant tools and guidelines, which can be exemplified by Figure 5.1. A framework is a form of design reuse or domain-specific architecture [7]. The goal is to reuse high level design described by both code and design or components and patterns.

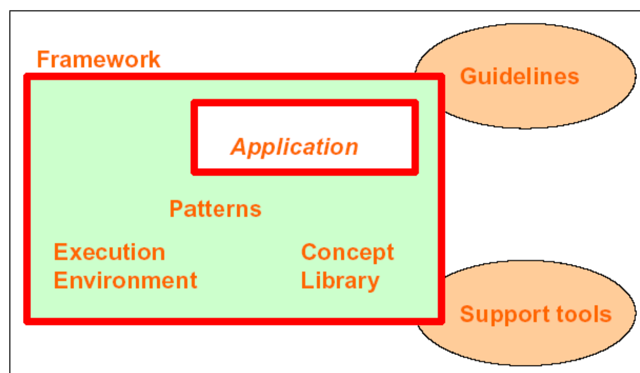

Figure 5.1: JavaFrame framework/Modelliong Development Kit

As a further consequence from the statements above, [9] states that *a JavaFrame*

[2] Java Development Kit
[3] Java Run-time Environment

system should be described graphically as well as textually. Textually this is done in Java and graphically notation is done in UML 2.0 with a 1:1 relation ship between the textual and graphical notation.

Since JavaFrame is ultimately implemented by Java, one could make a program in any manner according to Java guidelines. However, in order to reap maximum benefit from JavaFrame, there are guidelines for JavaFrame both as a MDK and as a framework. A violation of these principles may make analysis difficult, and dependability can not be guaranteed. Some of the modeling guidelines from [9] are presented below:

- Every active object will communicate only through its own mediators.
- Every state machine will only address its own innermost mediators.
- A composite will contain other active objects, and the containment is defined through the communication structure of the mediator connections.
- Whenever an active object is going to be created, it is assumed that mediators already exist. They are parameters to the constructor.

Further, for state machines the following invariants must hold:

- All state machines of a specific extended class are reentrant in that they refer to the same state structure.
- All dynamic variables are local to the state machine.
- Every transition should end in an operation that defines the next state.
- At any point in time a state machine is associated with either zero or one scheduler.

The above guidelines are added to the framework as seen in Figure 5.1. When modeling in the framework itself the following set of templates of patterns that should[4] be used.

Figure 5.2, the extended active object model, is one of the patterns that ensure a correct use of the JavaFrame framework. It shows, that an active object can be a composite or a state machine, further it shows that a composite can contain active objects. As the atomic part, a composite sooner or later contains a state machine. The possibility of nested composites is an important feature when evaluating if the composite is a candidate for real time replacement or not.

Shown in Figure 5.3 is the conceptual model of messages. Messages are the input stimuli to a state machine, which behaves in a reactive manner. To each state machine, there is a message queue, not shown here, which enables state machines to react in an asynchronous manner. The only message predefined, is the TimerMsg, which trigger a transition after a time period. Each state machine can be connected to zero or one scheduler, and each scheduler schedules a set of state machines. Figure 5.4 is a conceptual model of such a scheduler and its associated state machine. The JavaFrame framework has hidden a lot of the underlying actions around the

[4] read must

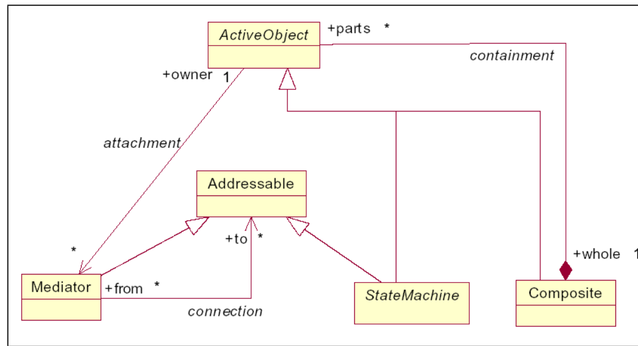

Figure 5.2: Conceptual model of ActiveObject

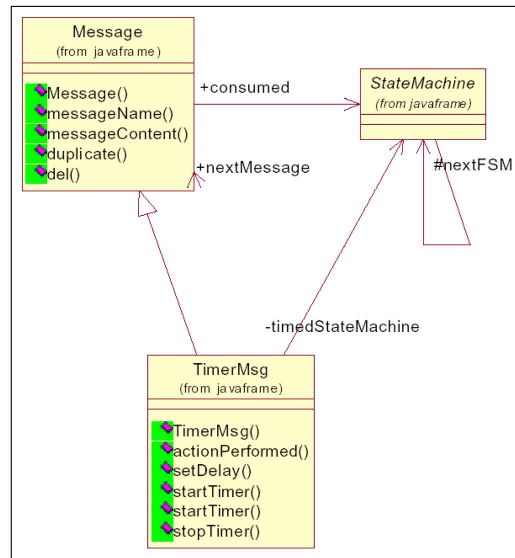

Figure 5.3: Message conceptual model

scheduler, amongst it the synchronization, thread and trace scheme. When talking about threads [9], also states that introducing user-specified threads should be avoided. JavaFrame also have patterns for the use of states contra composite state and state machines, shown in Figure 5.5. This shows that a state machine has a current state which is associated with a state space or composite state. The main issue in this project is to evaluate if this composite state is a candidate for runtime replacement. Based on the forgoing, the big outline methods of developing in JavaFrame is to first model the JavaFrame application system, second to model transformation to Java and last fill in the details before tuning the system.

5.3 The Architecture Provided by the Framework

The architecture of component-based systems is significantly more demanding than that of traditional monolithic solutions [15]. Therefore, this section will describe the architecture of the framework used in this project, the Java- and ActorFrame framework. Since these frameworks are used in the prototype, and therefore the same architecture, no architecturally solution is presented later on.

By mere looking at Figure 3.1 one could easily misinterpret that this is the architec-

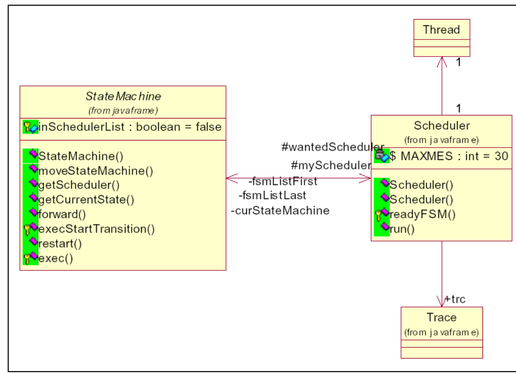

Figure 5.4: Scheduler conceptual model

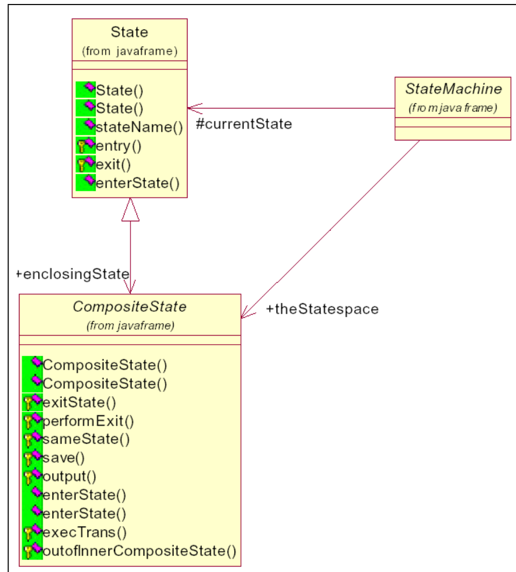

Figure 5.5: State conceptual model

tural structure, but that is not the case. It is true that the framework has a stratified layering architecture but it is a little bit more complicated.

First of all, the NorARC framework has a white-box architecture, meaning that using it requires knowledge of the superclass implementation, due to class and inheritance concentration. As to contrast to inheritance and the white-box approach, is the black-box approach, where interfaces are more common. To exemplify the white-box approach lets look at the following; to take advantage or make use of the actor layer, the application classes must inherit from some base actor classes shown in Figure 5.6. To do this, the programmer must acquire knowledge of the superclasses implementation prior to using them. This approach is worth mentioning since it affects the architecture of the framework. Another fundamental impact

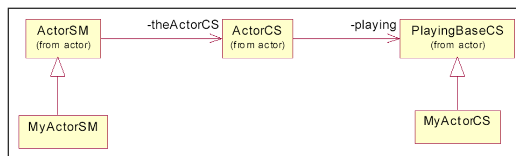

Figure 5.6: The Actor base behavior

on the architecture is, if the layering is strict or non-strict.

Strict layering imposes that the implementation of one layer should only be based on the operation of the layer immediately below. Figure 3.1 shows an example of this approach, although that is not the intent of the figure. This is called the onion model [15], emphasizing the distinct layer approach. The upside of this approach is that each layer can be understood incrementally, learning it layer by layer. The downside is that this architecture is hard to extend, and may suffer performance penalties.

The framework provided by NorARC use the non-restrict architectural approach. This means, that one layer can access methods from any of the lower layers.

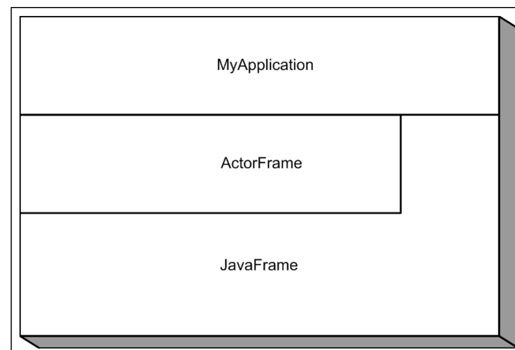

Figure 5.7: Non-restrict layering

There are of course both upsides and downsides with this architectural approach, like loosing the benefit of slicing layers in obvious ways done in strict-layering. The gain however, is that it solves extensibility and performance problems by eliminating intermediate layers where useful.

The system architecture, built on several micro architectures patterns, is a mean to capture an overall generic approach, which makes it more likely that a concrete system following that architecture will be more understandable. The overall architecture of the ActorFrame and JavaFrame visualized by Figure 5.7 can be mapped to the following conceptual design approach. Given the framework, the archi-

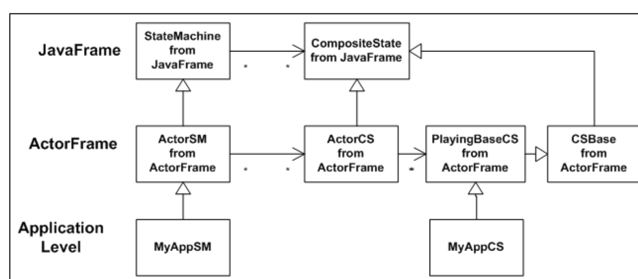

Figure 5.8: Conceptual design - Application and ActorFrame

tecture and overall design are also given for an application based on that framework. Therefore, the above architecture and conceptual design are the base for the prototype described in chapter 7. Figure 5.8 shows an application based on ActorFrame, which uses JavaFrame. In Figure 5.9, the application is based solely on JavaFrame. Either way, the application has the same base architecture as the given framework(s).

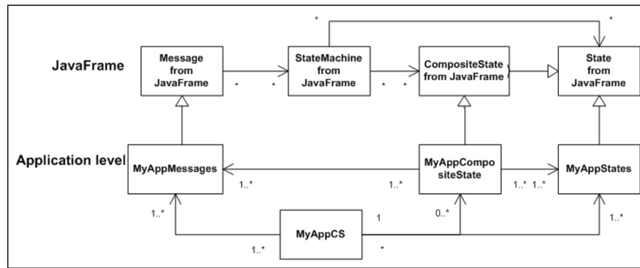

Figure 5.9: Conceptual design - Application and JavaFrame

5.4 Summary

When using the Java- and ActorFrame frameworks to develop applications, it is important learn and understand how these frameworks work. Violating their guidelines and patterns may cause the application to crash the target system. Although the NorARC frameworks has non-strict layering, and therefore are harder to learn, this approach has its advantages, like faster execution due to layering bypass.

An adequate Java approach also was presented in this chapter. Even though runtime compilation are dissuaded, introduction of pre-compiled Java class files still is possible.

Chapter 6

A Conceptual Approach

6.1 Introduction

This chapter will present some conceptual solutions related to the problems of this thesis. The solutions are based on the information given in the previous 5 chapters. Some of these solutions are demonstrated in the prototype.

6.2 State Machine Equivalence

Generally, in order to replace parts of state machines, one must ensure that the new state space neither introduces deadlocks or nondeterministic behavior to the given state machine, nor, when communicating with other state machines, causes the other state machine enter a deadlock or behave in nondeterministic manner. To address the latter problem first, let us consider two asynchronous communicating state machines A, and B. Lets assume that these state machines have formally been verified not to enter deadlocks. If reconfiguring state machine A, by altering its behavior, the formal verification has no value, unless the sequences of messages expected and provided remains the same. This consents to the *observer equivalence* notion of [16]. Using τ as a notation for internal actions does not represent communication, one can by [16] verify that a state machine is observer equivalent. Further, Robin Milner in [16] uses the notion *bisimulation* when an actor can not distinguish between two state machines by mere interacting with them. State machine or actor B, in the example, can not distinguish between the *old* and the *new* state machine by just interacting with it. Hence, no deadlock or nondeterministic behavior will occur as a result of the substitution. Bisimulation can be divided into strong and weak types of bisimulation, where strong is the hardest to prove and verify. Observer equivalent state machines are at least weak bisimulated but may prove to be strong bisimulated. Due to the time scope of this thesis, observer equivalence and weak bisimulation are an adequate approach.

One could argue that one of the state machines could be programmed to handle several different sets of message sequences, and therefore the other state machine also can alter its sequence of messages without introducing erroneous behavior. That is correct, but it is still bismulation. If an actor can handle several sets of message sequences, then a new behavior of the communicating actor must provide

one of the expected sequences, or else a deadlock may occur. If one wants to change one of the communicating state machines in a manner, which causes it to produce and expect messages in a sequence that the other state machine has not foreseen, the other state machine needs to be updated too.

The first of the problems introduced in this section was how to avoid altering a state machine that causes it to enter a deadlock or causing nondeterministic behavior, which may happen because a guard never becomes true, or entering a state not accounted for. Using Spin[1] or some other formal tools, one can verify that situations like these do not happen.

6.3 The CompositeState

When altering the behavior of a state machine by substituting its inner composite state(s), this will in principal work as long as the composite states are equivalent. When using the NorARC creation environment, this is not the whole truth. Since entering and exiting a CompositeState happens through its entry and exit points, these points must be considered also. The best approach is if the substituting CompositeState has equivalent exit and entry points as the original. Only if the parent CompositeState also is substituted, changes to the entry and exit points are recommended.

When receiving an update message, a state machine may be in a non-suitable state for updates. In these cases there are some rules of thumb. First, let us consider the cases when the adversary state machine is aware of the update, and maybe counting on it. In such cases it is of utterly importance to reach a state safe for updating as soon as possible. This may be by leaving the current state abruptly, saving the status quo and executing the update. If possible, maybe entering the previous state after the update and continue from that point on. A state safe for updating may be several different states, but it may require a specified state (e.g. update), that is particularly constructed for such a task. It may include initializing some data or variables local to a state machine or the composite state. If the opponent state machine is not aware of updates, has no information about it, and will not receive any kind of information when updates takes place or finishes, another way of updating will be more appropriate. By saving the update message, the state machine can wait until it transits to a state safe for updating. By this approach, the expected sequence of messages can be maintained. In both cases, messages received while updating the state machine should be saved for later execution or at least for later evaluation.

6.4 Modelling UML 1:1

By using UML, one tries to achieve 1:1 relation between the model and the code. This section explains how the conceptual solutions regarding this thesis may be in UML 2.0.

Through the vast of papers and books on the subject of UML, there has not been

[1]http://www.spinroot.com

found[2] any good solution to neither of the UML objectives; modeling the CompositeState run-time replacement and dynamic change of behavior. The rest of this section presents a possible solution to the first UML objective, and then dwell upon the second.

To model any run-time replacement, let us first consider a common class model, which the name implies, models the classes involved. The purpose is to show that two classes may be present in the same class diagram, but not associated with the run-time environment at the same time. Let us consider an example: Figure 6.1

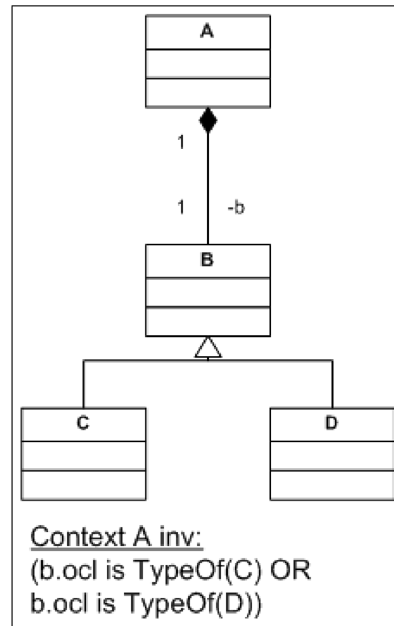

Figure 6.1: Conceptual class diagram

shows a class diagram with four classes, A, B, C and D. Class A has an association to B named b, which is private. The classes C and D inherit from B, meaning that b may also point to C or D. To make sure that b is only associated with one of the classes C and D an OCL expression is made. This states that b can only be associated with either C or D at a given time.

By using sequence diagrams one can further explain this. The sequence diagram, in Figure 6.2, shows object a as an instance of class A. It has an association to object b, which is an instance of class C. At some point in time, object *a* creates a new object based on class D. The object reference is the same as the one associated with the instance of class C, therefore that object cease to exist, as the object reference is now associated with the object of class D.

The final diagram used to explain of the modeling, is the communication diagram. Communication diagram shows the use of the *become* flow. Figure 6.3 shows the transformation from one value of an object to another, in this case by replacing the variable b. The *become* flow relationship indicates, that the instance of A has, at a different point in time, a new value for the association b (i.e. object b as an instance of D).

Regarding the second UML objective, modeling dynamic change of behavior, there

[2]This does not mean that it do not exist

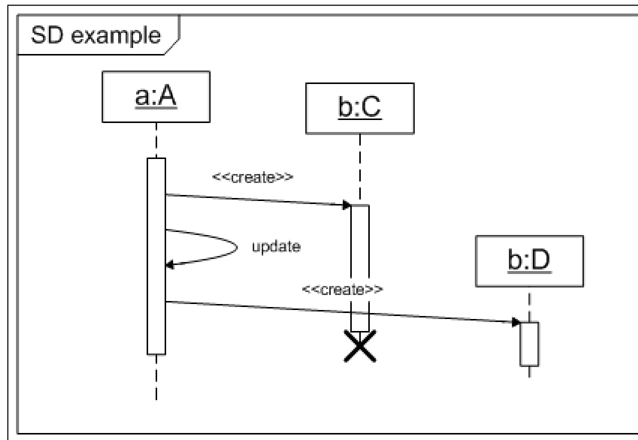

Figure 6.2: Conceptual class diagram

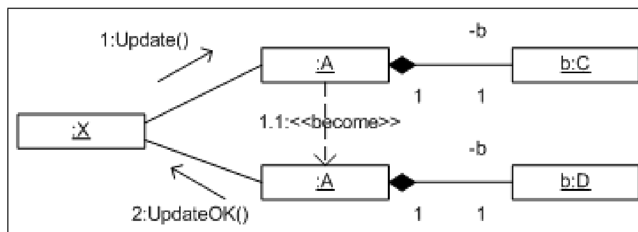

Figure 6.3: Conceptual communication diagram

is not found any solution. It may even be impossible to model such behavior in a 1:1 relation in UML at all. The reason for this may be that UML is a modeling language, optimized to visualize static structured system design and requirement definitions. This is a great contrast to the dynamic change of structure that is caused by run-time CompositeState substitution. It is possible to extend UML to be applicable at certain domains by using profiles or UML extensions, but this may at best help in simplified situations.

6.5 Summary

This chapter has presented some conceptual solutions to problems related to this thesis. The most crucial aspect is avoiding errors when substituting the composite state. This can be done by applying the rules of bisimulation and equivalence to the composite states. Further, the guidelines concerning states safe for updating are provided. Finally, it is shown how to use UML to create a 1:1 relation between the code and the model. Although this is not possible when modeling change of behavior over a time period, the actual run-time introduction of new classes are shown. Some of the conceptual solutions are demonstrated in the prototype, described in the following chapter.

Chapter 7

The Prototype

7.1 Introduction

This chapter will describe the prototype which is a product of this thesis project. The prototype demonstrates some of the concepts involved with regards to using NorARC frameworks for dynamic change of behavior during run-time (i.e. to show how replacement of inner CompositesState is possible using Actor- and JavaFrame). This will provide the state machine owning the CompositeState with new behavior. Shown in chapter 6, the architecture is given based on the framework, and so is the conceptual design, and methods used to achieve the generic goal through Java. Section 7.2 describes the initial state machine that was the foundation for the work around change of behavior. The rest of the chapter will describe the design and implementation of a new composite state that will alter the behavior of the state machine.

7.2 The State Machine

In order to have a state machine, which behavior should be changed, an initial state machine must be made. The initial state machine then must be extended by new CompositeStates providing new behavior.

The outline for the state machine is the following; the state machine shall illustrate a car that has the possibility to drive and stop, based on some inputs. The environment of the input with regards to a *play* is not addresses here, since it can vary a lot from another state machine to a graphical user interface. Nevertheless, the principal of dynamic change of behavior of a state machine is the same whatever the stimuli source, although different stimuli sources may introduce different ways to inform the source, assuming that there has been a change.

Further, the notion drive can be divided into driving forward and driving backward, indicating that the drive state should be a CompositeState with forward and backward as inner states. This behavior will be changed, providing new behavior with regards to driving backwards. This new behavior is represented by internal state changing, and the CarDriver actor shall not be affected by the change.

7.3 Architecture and Design

Given the framework, the architecture and the design for the overall application are given, on the other hand, JavaFrame guidelines restricts the architecture and design of the state machine. The architecture of the prototype is fairly simple. It is a deterministic state machine and it has some atomic states and at least one composite state. Further, for the simplicity there should be no sink or transient states.

7.3.1 The Initial State Machine

Based on the information given in the previous section, one can already picture in mind the design of the state machine. Chapter 6 explained that there may be states safe for updating, either explicit states or implicit states. The prototype has an explicit update state, for which the state machine has to enter to update its CompositeState. The following figure is a visualization of the state machine described by the English-language.

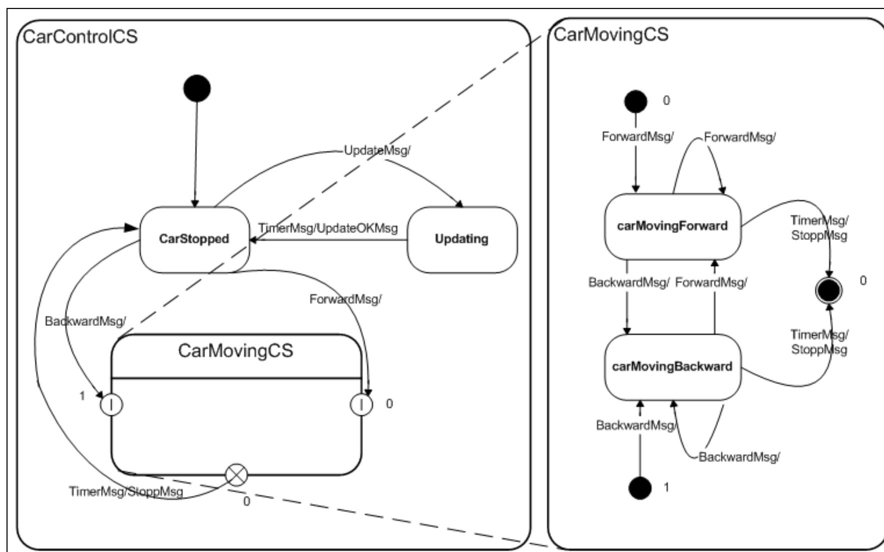

Figure 7.1: The initial state machine

Figure 7.1 shows a simple car represented by a state diagram. This diagram shows the basic outline of the car; it can be in the state move, stopped, or updating. The CompositeState, carMovingCS has, some inner states, indicating that the car can move forward or backward by the states, carMovingForward and carMovingBackward. The messages attached to the lines, connecting the states together, shows what input message that triggers the transition, and what message, if any, is sent when entering a new state. All this can be used to create the state encoding or transition table.

Current State	In	Out	Next State
Idle	-	-	CarStopped
CarStopped	UpdateMsg	-	Updating
CarStopped	ForwardMsg	-	carMovingForward
CarStopped	BackwardMsg	-	carMovingBackward
CarStopped	StopMsg	-	CarStopped
Updating	TimerMsg	UpdateOKMsg	CarStopped
carMovingForward	ForwardMsg	-	carMovingForward
carMovingForward	BackwardMsg	-	carMovingBack
carMovingForward	TimerMsg	StopMsg	CarStopped
carMovingBackward	BackwardMsg	-	carMovingBackward
carMovingBackward	ForwardMsg	-	carMovingBack
carMovingBackward	TimerMsg	StopMsg	CarStopped

Further, the state diagram also shows the entry and exit points with their respective indexes for the CompositeState. Shown in chapter 4, these indexes are important with regards to indicating which entry or exit point to use, because the entry/exit point leads directly to a new state. A failure to use the right index may cause the state machine to enter the wrong state, causing nondeterministic behavior.

Now, this state machine has to communicate with another state machine, the CarDriver state machine. This state machine shall represent the opposite party to the CarControl state machine in the *play*. The CarDriver state machine, herby called carDriverCS, serves as a GUI proxy, simulates an actual user. In the case of the prototype, the carDriverCS initializes the update, therefore entering an update state. Hence, to get out of this state, it needs to get a message indicating that the update went well, shown by the *updateOKMsg* in Figure 7.2. When updating the car state machine, the carDriverCS can still use it without any modifications, cause the old and the new state machine for the CarControl are *weak bisimulated* and *observer equivalent*. The carDriverCS represents the basic states of the car only, stopped,

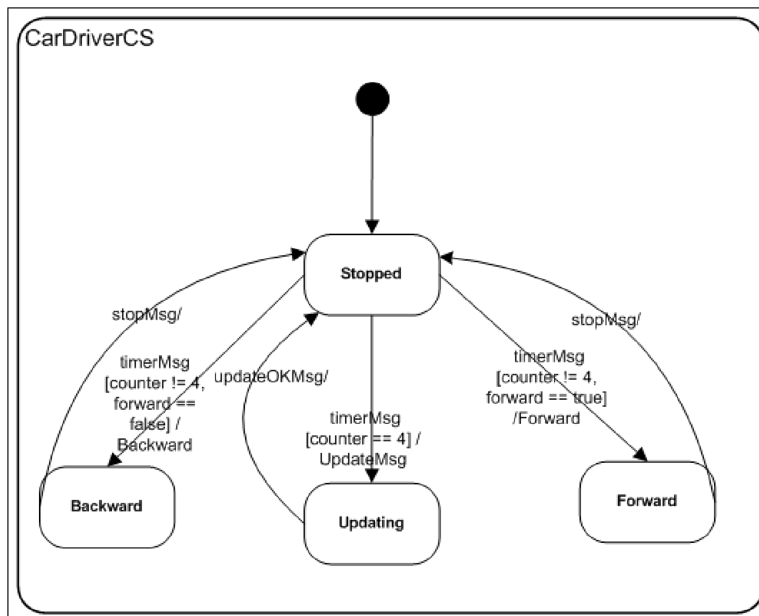

Figure 7.2: The driver state machine

updating and driving forward and backward. The basic idea is to update the car state machine without any modification to the carDriverCS, and that it shall function properly after the update (i.e. without introducing deadlock or malfunction behavior).

Putting it all togheter in a UML class diagram it looks like this:

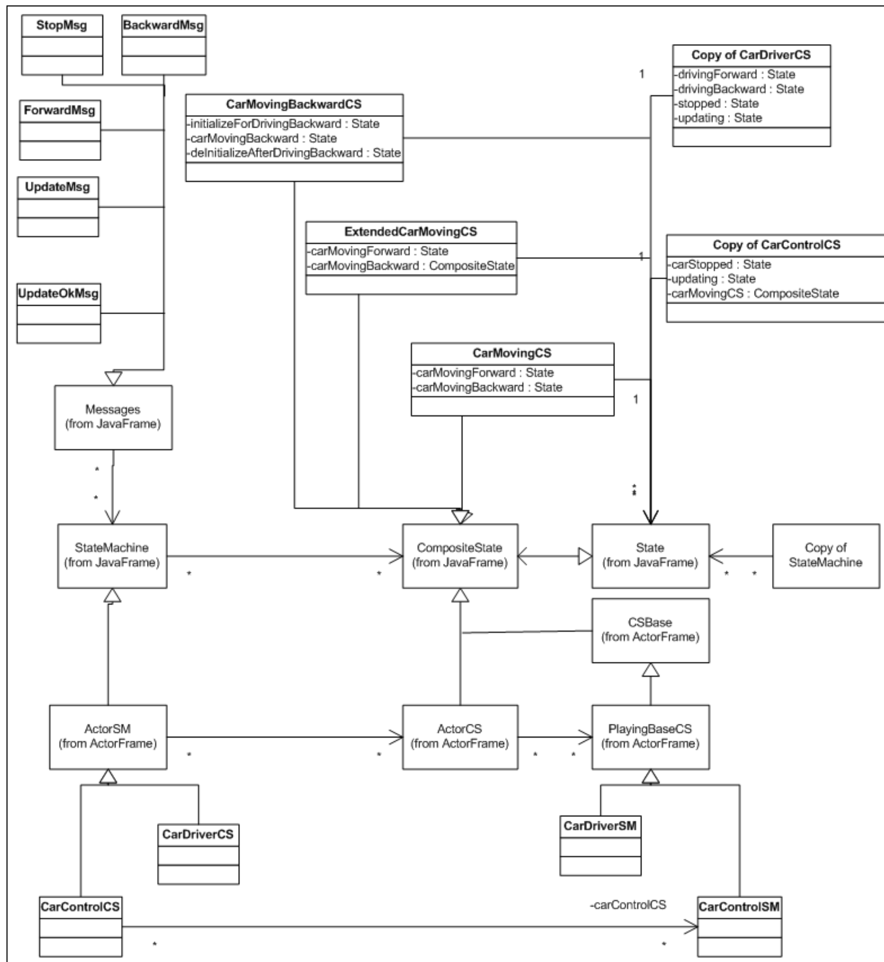

Figure 7.3: UML diagram 1

As seen in the diagram 1, Figure 7.3, when using a framework one can make a relative large and complex application with only a few classes. Further, when dominating the framework, an application is made in a short time.

The basic classes for each actor in the play are the state machine and outermost CompositeState. These are for the car CarControlSM and CarControlCS and for the driver CarDriverSM and CarDriverCS. Further, the CarControlCS has an inner composite CarMovingCS. Each composite has a set of messages available and the states are shown in earlier diagrams. Usually, the connection between the state machine and the outermost CompositeState is handled by the framework but when having updating the state machine, in order to have inner updated CompositeState refereing the outermost composite there must be an explicit connection between the state machine and outermost composite.

UML diagram 2 in Figure 7.4 shows the relation between the CompositeState Car-

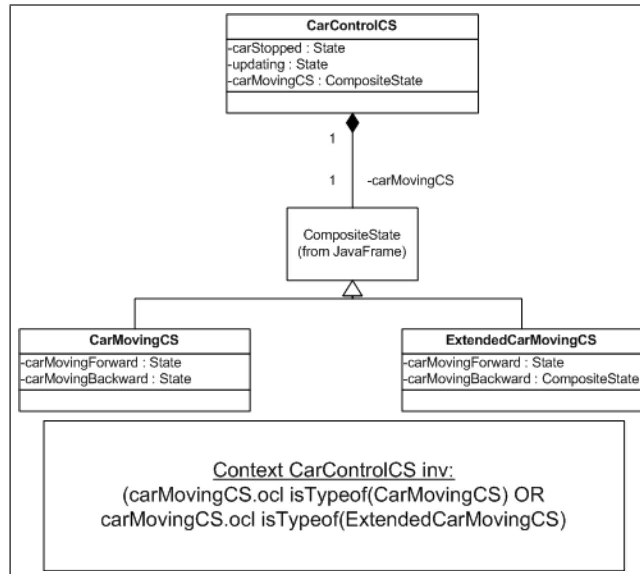

Figure 7.4: UML diagram 2

ControlCS and its inner states. The use of OCL shows that only one of the two inner CompositeStates can be associated with the CarControlCS at a given time. UML diagram 3, shown in Figure 7.5, show the same relation, save the OCL expression, between the ExtendedCarControlCS and its inner CompositeState.

7.3.2 The Extended State Machine

Introducing the extended state machine shall not lead to any necessary changes to the driver state machine. This means, that the input and output messages must remain the same, and that no new entry or exit points can be introduced to the CompositeState that is not present in the substituting CompositeState. Surely, the extended CompositeState may send internal messages and have internal transition that the driver state machine is not affected by. Further, if a CompositeState, with inner CompositeStates, are replaced, one may alter the inner CompositeStates exit and entry points, since their parent CompositeState is being replaced too.

The purpose of the extended state machine is to replace a CompositeCtate from the initial state machine. Further, if the replacing CompositeState has inner CompositeStates, this will be a manifestation that one may introduce an arbitrary number of recursively inner CompositeStates. Therefore, the CompositeState replacing the CarMoving CompositeState from the initial state machine has an inner CompositeState. This state will replace the driving backwards state from the initial state machine. Figure 7.7 shows the state chart of the new composite state.

By comparing the CarMovingCS CompositeState of the initial state machine with the ExtendingCarMovingCS CompositeState, and by viewing the state graph from Figure 7.6 and Figure 7.8, it is clear that the CompositeStates are *equivalent*. Therefore, the CarMovingCS may be substituted by ExtendedCarMovingCS without the CarDriver actor being affected. Instead of using the formal approach from [16] to verify the principals and foundation of an application, some may take the shortcut and just test the application to see if it works. This is a bad approach, and

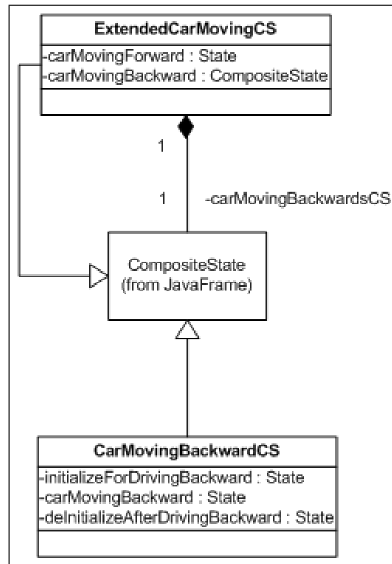

Figure 7.5: UML diagram 3

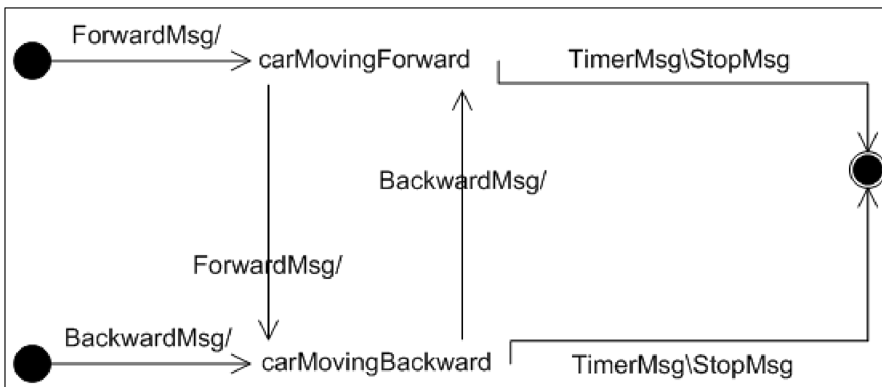

Figure 7.6: StateGraph of the initial composite state

may at best lead a developer to the same conclusion as Dijkstra, who said: *Program testing can be used to show presence of bugs, but never show their absence* [17].

7.4 Implementation

This section shows the important implementation issues, but not all the code necessary, to implement the whole application. The complete code is to be found in the appendix.

7.4.1 The Actor's Deployment Descriptors

First, let us look at the associated deployment descriptors. The framework has an actor that is controlled completely by the framework, and that is called the RootActor. This actor servers as the main actor, and all other actors implemented by the programmer are actually children if this actor. The RootActor must know the name of the play, the roles associated with the play, and how many instances there

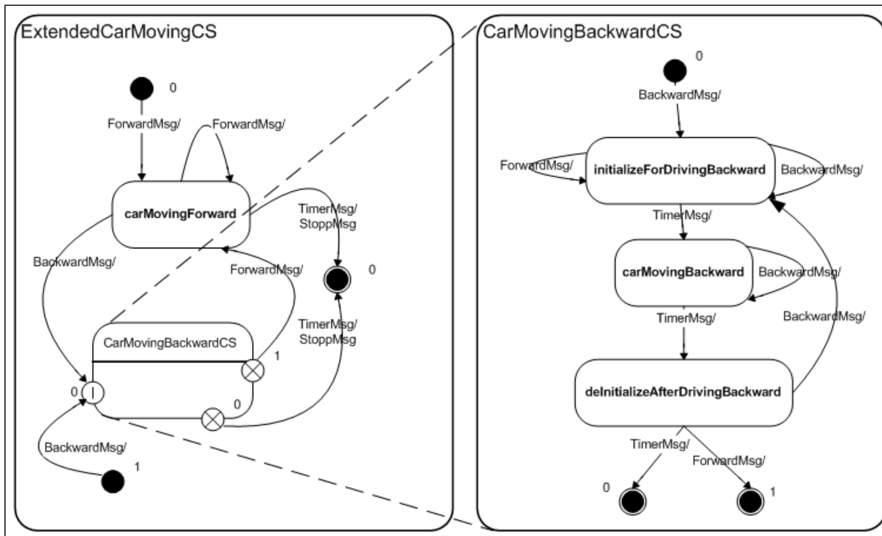

Figure 7.7: The extending composite state chart

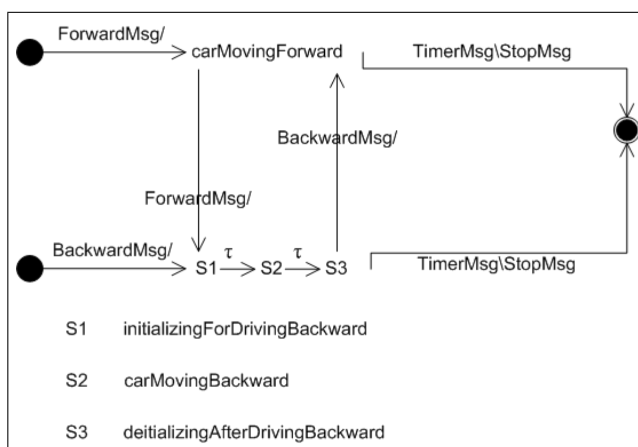

Figure 7.8: State graph of the extended composite states.

may be of each role. Further, the RootActor must know which child actor will serve as the initializing actor, plus its identification. Finally, the deployment descriptor must hold information about which class serves as the RootActor, and where to find its state machine and outermost composite state. This is the basic information the RootActor needs to know, although more information may be added optionally.

Before looking at the actual code for the implementation let us look at the other appurtenant deployment descriptors. As indicating by the RootActor deployment descriptor, there are two other actors, the CarControlCS and the CarDriver, which is the initializing actor. Like the RootActor deployment descriptor, the deployment descriptor for the CarControlCS and CarDriver, must tell the framework where to find their state machine and their outermost CompositeState. Besides, the only information an actor's descriptor needs to know, is which actors, and how many of that kind, are associated with the given play. Figure 7.9 and 7.10 shows parts of the deployment descriptor for the RootActor and the CarDriver actor.

This can informally be described by the following: The RootActor, fully controlled

```
<ActorTypeName>RootActor</ActorTypeName>
<ActorClassName>se.ericsson.eto.norarc.actorframe.actor.ActorSM
    </ActorClassName>
<ActorBehaviorClassName>se.ericsson.eto.norarc.actorframe.actor.
    PlayingBaseCS</ActorBehaviorClassName>
<Role>
    <ActorTypeName>CarDriver</ActorTypeName>
    <maxCardinality>1</maxCardinality>
</Role>
<Role>
    <ActorTypeName>CarControlCS</ActorTypeName>
    <maxCardinality>1</maxCardinality>
</Role>
<InitialRole>
    <identification>Hansen</identification>
    <ActorTypeName>CarDriver</ActorTypeName>
</InitialRole>
```

Figure 7.9: Code for the RootActor deployment descriptor

```
<Association>
    <ActorTypeName>CarControlCS</ActorTypeName>
    <maxCardinality>1</maxCardinality>
</Association>
```

Figure 7.10: The deployment descriptor for the CarDriver

by the framework, has some child actors, the CarControlCS and CarDriver. The RootActor uses one of the child actors to initialize a play not yet named. The initializing actor, CarDriver, is given an identification, Hansen. Behind the scene, the RootActor *asks* the CarDriver actor if it can play the CarDriver role. When confirmed, the CarDriver must response by *asking* the CarDriverCS if it can play the role CarDriverCS. This is done by sending a RoleRequest message to the CarControlCS actor. Doing this, the programmer gives the CarDriverCS an identification, like *Hansen* was given to CarDriver.

7.4.2 CarDriver

Given the actor's deployment descriptors, let us look at some code implementation. The class file of CarDriverSM is shown in figure 7.11, and it is a rather small class. This is due to the inheritance from ActorSM and that there are no local variables, specific for the actor, present. Much of the functionality also are hidden by the framework of ActorFrame and partially JavaFrame (e.g. the scheduling, the mediator binding, the role creation, association and confirmation). Also, some is handled by the appurtenant outermost CompositeState, CarDriverCS.

```
import se.ericsson.eto.norarc.actorframe.actor.ActorSM;
public class CarDriverSM extends ActorSM{
}
```

Figure 7.11: Implementation of CarDriverSM

The first method called at the CarDriverCS, save the constructor, is the treatEnterState.

```
protected void treatEnterState(ActorSM asm) {
sendRoleRequest("Car", "CarControlCS", new Play("theDrive"),
        asm, new Hashtable(), "carCon");
idle.enterState(asm);
}
```

Figure 7.12: Implementation of CarDriverCS

The treatEnterState method, showed in Figure 7.12, gives the programmer opportunity to redefine the actors behavior when entering the first state, which is idle. In this prototype, this method used to send the request for a play and a role to the participating actor, the CarControl. This is where the play is created, and where the CarDriver actor *asks* the CarControlCS to play the role CarControlCS in the play *theDrive*. The CarControlCS is also given an identification, *Car* like the CarDriver was given the identification *Hansen*. The treatEnterState also tells the participating actor which state machine to use, and what the connection will be labelled (i.e. carCon). There are several possible signatures to use, but the signature used is based on the one shown in figure 7.13.

```
public boolean sendRoleRequest(ActorAddress actorID,
        String roleType, Play play, ActorSM curfsm,
        Hashtable requestCredentials, String connectionID)
```

Figure 7.13: Implementation of CarDriverCS.sendRoleRequest

After executing the method treatEnterState, the state machine waits for a confirmation on the request message. The actions with regards to the actor and play, and messages sent back and forth to initialize a play, happen all behind the scene. Nevertheless, the programmer must understand what is happening and when, to fully utilize the framework, and make an application that works in accordance with the framework's specification. The possibility to redefine the actor's behavior when receiving the RoleConfirmMsg from the corresponding actor is treated in the method treatRoleConfirm in figure 7.14.

```
public boolean treatRoleConfirm(RoleConfirmMsg sig, State st,
    ActorSM asm){
    CarDriverSM psm = (CarDriverSM)asm;
    TimerMsg t = new TimerMsg(2000,psm);
    t.startTimer();
    stopped.enterState(psm);
    return true;
}
```

Figure 7.14: Implementation of CarDriverCS.treatRoleConfirm

The last method of the CarDriverCS class is the execTrans, which is the state machines main[1] method. This method handles all the transitions and is never called by the user but is always executed by the framework. It is, however the developer who decides what will happen in this method and what to be executed for each state and within each transition. Since the framework handles many internal messages, execTrans must first check if the messages can be handled by the framework. This

[1]not to be confused with the Java main method

is done by calling the super's[2] execTrans method. If the superclass can handel the incoming message the execTrans of the current class return *true*, and the method returns. If not, the message must be handled in the method execTrans. The code for the implementation is too long to include here, but the code from the appendix will show that each state in the composite state is represented with *if* statements. Further, signatures that shall trigger in each state, are represented with new inner *if* statements. As an example, let us consider the stop state of the CarDriverSM, shown in figure 7.15.

```
}else if(st == stopped){
    if(sig instanceof TimerMsg){
        counter ++;
        if(counter != 4){
            if (forward) {
                forward = false;
                    sendMessage(psm.context.getConnection("carCon"),
                        new ForwardMsg(), psm);
                    drivingForward.enterState(psm);
            }
            else {
                forward = true;
                sendMessage(psm.context.getConnection("carCon"),
                        new BackwardMsg(),psm);
                drivingBackward.enterState(psm);
            }
        }else{
            sendMessage(psm.context.getConnection("carCon"),
                new UpdateMsg(),psm);
            updating.enterState(psm);
        }
    return true;
}
```

Figure 7.15: Implementation the Stop state

This is the state that simulates a user that has three options; to drive forward, backward or update. When in the state stopped, the only message that may trigger a transition to a new state is the message TimerMsg. When receiving the TimerMsg messages, a simple check is implemented to make the car drive in the opposite direction or to update the CarControlSM. When transiting from one state to another, a message is sent to the other actor by the method sendMessage. To ensure that the right actor receives the message, each connection has a name, and in this case the connection name is *carCon* set in the treatEnterState method.

One could argue that the Boolean guard, used in the state chart, represents a state and should be treated as one instead of a Boolean guard, but this implementation represents a user that has a kind of random behavior. Further, the state machine of the CarDriverSM shall represent the basic state machine of the CarControlSM, so by introducing new states to the CarDriverSM just to represent a Boolean guard, not present at the CarControlSM, would make the state machines look very different. This can also be applied to the CarControlCS's explicit update state. In the case of the prototype, the update can be seen as an atomic action and therefore does not need an explicit state. But since this thesis is concerned with asynchronous communication, which is not the actual case of the prototype, explicit states are used to show how this would be implemented in an actual asynchronous environment, and where the update state must be more than an atomic action.

[2] the class, that this class is inherited from

7.4.3 CarControl

Like the state machine for the CarDriver, the CarControl state machine implementation, in Figure 7.16, is very small. As shown in the UML diagram from figure 7.3, it has only one association. The rest of the functionality is inherited from ActorFrame's ActorSM.

```
public class CarControlSM extends ActorSM{
public CarControlCS carControlCS;
}
```

Figure 7.16: Implementation of the class CarControlSM

The CarControlCS is the CarControlSM's outermost CompositeState. All though very similar to the CarDriverCS with regards to configuration there are, due to the composites, inner CompositeState, some additional functions. Also, the code for replacing a CompositeState is implemented.

Figure 7.17 shows that in addition to the normal state, CarControlCS also has a composite state, carMovingCS, which includes the states for driving forward and backward.

```
State carStopped = new State ("carStopped");
State update = new State ("update");
public CompositeState carMovingCS = new CarMovingCS();
```

Figure 7.17: State and Composite State Declaration for CarControlCS

One of the methods needed in the CarControlCS is outOfInnerCompositeState, it ensures that the inner composite is exited the right way.

```
public void outofInnerCompositeState(CompositeState sc, int exitNr,
        StateMachine curfsm) {
    CarControlSM ccsm = (CarControlSM)curfsm;
    if(sc instanceof CompositeState && exitNr ==0){
        sendMessage(ccsm.context.getConnection("carCon"),new StoppMsg(),
            ccsm);
        carStopped.enterState(curfsm);
        System.out.println("Car Stopped");
    }
}
```

Figure 7.18: Implementation of outOfInnerCompositeState

As seen form line four in Figure 7.18, the right exit point must be verified and as shown in the state chart in figure 7.1. There only shall exist one exit point, labelled *0*. Then, we need to cast the incoming state machine to the right kind, a CarControlSM, shown in line five in the figure. Last, the proper action for this transition shall be executed. In this case, sending a StoppMsg message to the associated actor and then enter the carStopped state.

The treatEnterState method implemented in the CarControlCS has an important statement with regards to the CompositeState replacement, but it only makes sense if the new CompositeState has an inner CompositeState.

```
public void treatEnterState(ActorSM asm){
    CarControlSM  tmp = (CarControlSM)asm;
    tmp.carControlCS = this;
    .
    .
}
```

Figure 7.19: Implementation of treatEnterState in CarControlCS class.

Figure 7.19 show an explicit assignment of the state machines outermost CompositeState. This gives in relation to the CarControlSM declaration in the CarControlSM the possibility to recursively refer a CompositeState by the state machine. The framework does not explicit make this possible, when in run-time introducing a CompositeState with an inner CompositeState. This will additionally be described later, when the implementation of the extended CompositeState is introduced.

The last significant thing about the implementation of the CarControlCS class is the possibility to update the state machine at run-time. This update is triggered by an UpdateMsg message.

```
else if(sig instanceof UpdateMsg){
    try{
        Class cls = Class.forName("example.CarControl.CarControlCS.
            ExtendedCarMovingCS");
        Object obj = cls.newInstance();
        carMovingCS = (CompositeState)obj;
        .
        .
        update.enterState(psm);
        .
        .
        }
}
```

Figure 7.20: Implementation of the CompositeState update procedure

By using Java, figure 7.20 shows the general way of introducing a pre-compiled class to a program already running. First the class is loaded, and then an object is created based on that class. Third, the object must be cast to the right type, which in this case is the CompositeState class that the CarMovingCS class is based on. One cannot cast the object through a CarMovingCS because that would lead to a compile time error in Java. To make this work one must cast it through a class that is common for both the CarMovingCS and ExtendedCarMovingCS, which is the CompositeState class.

In this prototype the path of the compiled class is not dynamic due to the fact that its path is written in the code. There are however possible to bundle this information in the update message by introducing a variable. By this approach, the application is more dynamic and may handle changes in a better fashion.

7.4.4 CarMovingCS

The first of the three inner composite state used by the prototype is the CarMovingCS composite state. It holds the states for driving forward and backward. Since it does not contain any inner composite sates, there are only two methods imple-

mented in the class; the execTrans, which must be implemented by all composite states, and enterState, which must be implemented in all inner CompositeStates. Shown by the state chart, and Figure 7.21, the enterState can handle two entry points, the *0* and *1*, which leads to the states carMovingForward and carMoving-Backward, respectively.

```
public void enterState(int nr, StateMachine curfsm){
    CarControlSM ccsm = (CarControlSM) curfsm;
    if (nr == 0){
        TimerMsg t = new TimerMsg(2000,ccsm);
        t.startTimer();
        System.out.println("Car is moving forward");
        carMovingForward.enterState(curfsm);
    .
    .
```

Figure 7.21: Extract of the enterState method from the class CarMovingCS.

The execTrans method, that handles the transitions and incoming messages for the CarMovingCS, has only two states to handle. However, an exceptional feature with inner CompositeStates, is that they must call the state machines outermost CompositeState through the state machine, and then recursively call all inner CompositeStates until it reaches its parent CompositeState, to call its outofInner-CompositeState. Only by doing this, it can exit the current CompositeState, and transit to its parent CompositeState. Although the prototype initially has two levels in its CompositState hierarchy, the procedure must still be executed. This shows the way an inner CompositeState refers its parent, by calling the state machine and moving down the hierarchy. This is the reason for the declaration of the outermost CompositeState in the state machine CarControlSM. The Java code that makes this happen is shown in figure 7.22.

```
protected boolean execTrans(Message message, State state,
       StateMachine stateMachine){
   CarControlSM asm = (CarControlSM)stateMachine;
   if(state == carMovingForward){
       if(message instanceof TimerMsg){
           asm.carControlCS.outofInnerCompositeState(this, 0, asm);
           return true;
   .
   .
```

Figure 7.22: Recursively calling outofInnerCompositeState

7.4.5 ExtendedCarMovingCS

The ExtendedCarMovingCS class is the CompositeState, which shall substitute the original CarMovingCS CompositeState. This class can be seen as a mixture of Car-ControlCS and CarMovingCS, since it is both an inner and parent CompositeState. Therefore, it has the method enterState like CarMovingCS and outofInnerCompositeState method like CarControlCS class. Also, like the CompositeState CarMovingCS, the ExtendedCarMovingCS have two enter points and one exit point. Further, the ExtendedCarMovingCS's inner CompositeState classes must be loaded at run-time, by the class ExtendedCarMovingCS, since this class is introduced at

run-time. In the prototype this is done in the constructor, as shown in Figure 7.23.

```
public ExtendedCarMovingCS(){
    try{
        Class cls = Class.forName("example.CarControl.CarControlCS.
            CarMovingBackwardCS");
        Object obj = cls.newInstance();
        carMovingBackwardsCS = (CarMovingBackwardCS)obj;
        .
        .
```

Figure 7.23: ExtendedCarMovingCS constructor.

Finally, like all CompositeStates, the ExtendedCarmovingCS class has the execTrans method implemented. This method has only one state to handle, the carmovingForward state. This may seem a bit odd, but it is actually right. Any other state will be handled by its parent or child composite state. When entering or exiting a CompositeState by the wrong reference point identification, the application will terminate or transiting to a wrong state, causing the program to crash or behave in a nondeterministic way. Therefore, when programming one must check if the identification used, when entering or exiting a CompositeState is valid and the right index for action we want, since a valid but wrong id may lead to faulty or nondeterministic behavior.

7.4.6 CarMovingBackwardCS

CarMovingBackwardCS is the final of the classes implemented in the prototype. It holds the states for all actions concerning the car moving backwards. It has the execTrans method, since it is a CompositeState, and the enterState, since it is an inner CompositeState. All these methods have been thoroughly explained through the other classes and there is noting new concerning the implementation of these methods in the CarMovingBackwardCS class, save one. When replacing the CarMovingCS class with ExtendedCarMovingCS, some object casting is necessary for some function invocation to work. When calling the parent composite state's outofInnerCompositeState method, this method must be called based on the appurtenant state machine described earlier. But now the CarMovingCS class, as an object, is no longer present, since the pointer now refers to an object of the class ExtendedCarMovingCS. However, to make this work, one must refer the object based on the carMovingCS pointer and cast it through an ExtendedCarMovingCS class, which actually is the class, that the current object is built upon. Only by doing this, one can invoke the CarMovingBackwardCS object's parent method outofInnerCompositeState. The actual code for this is shown in Figure 7.24.

It is worth mentioning, that in each custom state, in both the CarDriverCS and CarControlCS, print line commands are added to inform the programmer of the current state. This is, in addition to the trace file, a mechanism for a simple verification of the programs execution. This ends the implementation of the prototype.

```
protected boolean execTrans(Message message, State state,
        StateMachine stateMachine){
      CarControlSM asm = (CarControlSM)stateMachine;
      .
      .
      ((ExtendedCarMovingCS)asm.carControlCS.carMovingCS).
         outofInnerCompositeState(this,0,asm);
      .
      .
```

Figure 7.24: Recursively calling the method outofInnerCompositeState

7.5 UML

The remaining UML diagram of the prototype is shown in this section. These are coherent with the suggested approach from chapter 6. The sequence diagram, presented in Figure 7.25, shows the replacement of a composite state. This is triggered by the UpdateMsg message from the driver actor, Hansen. The actual termination of the *old* CompositeState is undetermined, because this is controlled by the garbage collection of the JVM. But as far as the program, this object ceases to exist, when the carControlCS has altered its association to the new composite state. The sequence diagram hides all the underlying classes related to both JavaFrame and ActorFrame (e.g scheduler). Hence, some of the communications lines in Figure 7.25 shows the logical and not the actual path of communication.

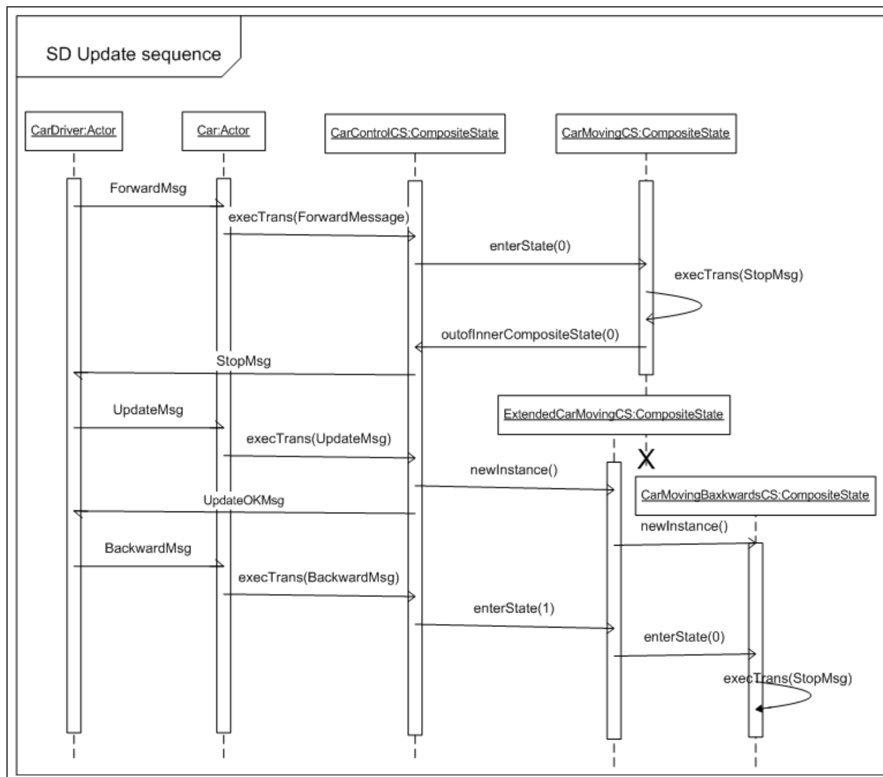

Figure 7.25: The prototype update sequence diagram

When observing the communication diagram, in Figure 7.26, one notices the same model structure as in chapter 6. The *become* flow used in an interaction shows the target object, in the case CarControlCS, representing a new version of the source

object and thereafter replaces it. The new version has its carMovingCS association sat to the composite state extendedCarMovingCS instead of the previous one.

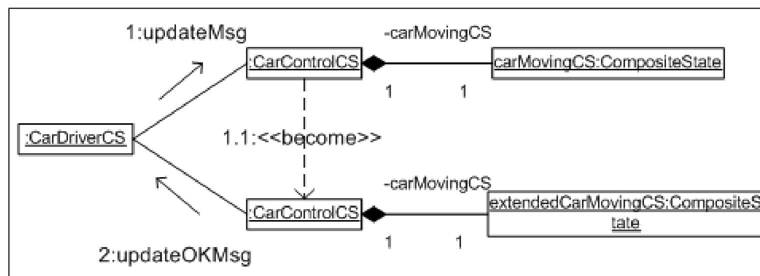

Figure 7.26: The prototype communication diagram.

7.6 Summary

This chapter has in many details shown the outline of the prototype, the architecture and design and the crucial parts of the implementation. In lieu of capturing a full formal specification of the prototype, as an application, important properties have been verified by abstracting away details of the program. Also, the most important UML models are included and explained to further substantiate the prototype. Chapter 8 deals with some of the results related to the prototype.

Chapter 8

Results

8.1 Introduction

This chapter is concerned with the results of the prototype made by using JavaFrame and ActorFrame frameworks. These results are important when evaluating if the prototype behaved in a manner that consents to the intentions and guidelines of the frameworks. The application trace will be evaluated to verify that the logical execution is correct. Further, some results of the UML modeling are addressed.

8.2 Logical vs Physical Execution

The first thing to notice of the prototype is that it runs with no deadlocks or execution errors. This gives the first indication that it, as a Java application it works fine. Further, one can, by examining the results of the trace file in Figure 8.1, confirm that there are no state deadlocks in the program, neither before nor after the composite state substitution. Earlier in the thesis these was mentioned a trace observation tool that interprets the actual trace file. This tool, JFTrace, is implemented using JSDK 1.3 in contrast to JSDK 1.4, which is the current Java development standard. By running JFTrace with the JVM appurtenant to JSDK 1.4, JFTrace made the JVM crash. Hence, the trace file needs to be examined manually, which is a bit more cumbersome than using a trace tool. The print line commands and trace file further shows that the replacement went well.

The included part of the trace file shows that the car, before the update transits between the states carMovingForward, carMowingBackward, and carStopped. After the update, the composite state carDrivingBackwardCS also includes the states initialize and deInitialize before and after the actual drivingBackward state. This shows, that the CompositeState replacement went well, and that the carDriver actor is not affected by the update. This is the logical resemblance of the sequence model presented in Chapter 7.

First of all, this shows that it is quite possible for an application, seen as a mere Java application built on the Java- and ActorFrame, to have run-time updates or replacements. Second, by examining the whole trace file, one can see that the dynamic substitution of the composite state is possible without violating any of the

frameworks guidelines or patterns, indicating that this run-time update is possible when observing the application from a JavaFrame or ActorFrame viewpoint.

```
<StateMachine>Car</StateMachine>
<State>carMovingBackward^example.CarControl.CarControlCS
    .CarMovingCS@181edf4^playing</State>
<Trigger>TimerMsg@d19bc8:</Trigger>

<StateMachine>Car</StateMachine>
<State>carStopped^playing</State>
<Trigger>ForwardMsg@1551f60:recieverRole :Car:CarControlCS,
    senderRole :RootActor/Hansen:CarDriver, </Trigger>

<StateMachine>Car</StateMachine>
<State>carMovingForward^example.CarControl.CarControlCS
    .CarMovingCS@181edf4^playing</State>
<Trigger>TimerMsg@17ee8b8:</Trigger>

<StateMachine>Car</StateMachine>
<State>carStopped^playing</State>
<Trigger>UpdateMsg@1995d80:recieverRole :Car:CarControlCS,
    senderRole :RootActor/Hansen:CarDriver, </Trigger>

<StateMachine>Car</StateMachine>
<State>update^playing</State>
<Trigger>TimerMsg@ce5b1c:</Trigger>

<StateMachine>Car</StateMachine>
<State>carStopped^playing</State>
<Trigger>BackwardMsg@6e293a:recieverRole :Car:CarControlCS,
    senderRole :RootActor/Hansen:CarDriver, </Trigger>

<StateMachine>Car</StateMachine>
<State>initializeForDrivingBackwards^example.CarControl
    .CarControlCS.CarMovingBackwardCS@105738</State>
<Trigger>TimerMsg@54a328:</Trigger>

<StateMachine>Car</StateMachine>
<State>deInitializeAfterDrivingBackward^example.CarControl
    .CarControlCS.CarMovingBackwardCS@105738</State>
<Trigger>TimerMsg@1e8a1f6:</Trigger>

<StateMachine>Car</StateMachine>
<State>carStopped^playing</State>
<Trigger>ForwardMsg@6025e7:recieverRole :Car:CarControlCS,
    senderRole :RootActor/Hansen:CarDriver, </Trigger>

<StateMachine>Car</StateMachine>
<State>carMovingForward</State>
<Trigger>TimerMsg@587c94:</Trigger>
```

Figure 8.1: Parts of the trace file

8.3 UML

Although UML 2.0 is not finally specified, there are through the proposal strong indications of how it will end up. Through the proposal, some models and diagrams are made, to show some aspects involved in this thesis. By using class diagrams in relation to both sequence and communication diagrams, it is possible to show how run-rime substitutions of objects are possible. The class diagram shows how several classes are related to a composite state, but only one of them are present at a given time, described by OCL. When using sequence diagram, the lifespan

of each object are shown. Hence, one can see that the two inner composite state involved with the update never exist as associated objects at the same time[1]. The communication diagram in chapter 7 shows how carControlCS flows from a state, where it is associated with carMovingCS, to a state where it is associated with extendedCarControlCS. The use of *becomes* specifies how carControlCS at a later point in time is the same as the source but has new associations. On the other hand, there are found no way to unambiguously model the change of behavior over a time.

8.4 Summary

This chapter has focused on the result of the prototype, demonstrating some of the proposed solutions. Section 8.2 shows, through the trace file, that the logical and physical execution of the prototype is the same, even when substituting the state machines inner CompositeState. This aids the conceptual solution, which states that by employing the bisimulation, contemplate the use of reference points, the run-time substitution of composite states are possible.

The results of the UML modeling regarding the prototype also have given some results. The UML diagrams shows that it is possible to model the run-time CompositeState class substitution, but no solution of modeling change of behavior over timer, in a 1:1 manner, is made.

[1]Due to garbage collection they may reside in memory at the same time

Chapter 9

Discussion

9.1 Introduction

This thesis started by presenting the three frameworks of NorARC's compound service creation framework. Further, the focus was adjusted to only include the JavaFrame and ActorFrame layer of the framework. Since some fundamental aspects of both of these frameworks are state machines and asynchronous communication, Chapter 2 presented some basic understanding of these. Finally, relevant parts of the proposed UML 2.0 specification were introduced and discussed.

The purpose of the work has been to find out if and how well the NorARC frameworks are suited for run-time behavioral change of state machines, and how inner composite states may be candidates for run-time substitution. All this should also be adequately modelled by using the new UML2.0 proposal.

The following issues will be discussed in this chapter:

- Dynamic change of state machine behavior in JavaFrame and ActorFrame.
- Composite states being a candidate for run-time replacement.
- Using UML2.0 to describe such replacement.
- The composite state replacements impact on state machines asynchronous communication and message sequence.

9.2 Dynamic change of state machine behavior in JavaFrame and ActorFrame

JavaFrame and ActorFrame are both ultimately implemented by Java, which, as shown in Chapter 5.1, handles dynamic changes. Hence only extraordinary implementation of the frameworks would prevent any dynamic changes. Since the implementation does not prevent this, the focus is concentrated on the frameworks guidelines and patterns.

An application depicts the architecture and design of the framework, on which the application is built. As shown in Chapter 5.2 and 5.3, neither JavaFrame nor

ActorFrame have a kind of architecture, design or even guidelines that prevents dynamic changes of their state machines. On the other hand, there are no properties in the frameworks that actively support that either. One way for the framework to active support state machine behavior changes, was to let the framework handle the update-message, like it handles the messages associated with a play. This way the framework itself could handle the update, making a standard way of introducing new CompositeStates at run-time.

9.3 Run-time CompositeState replacement

One of the main purposes of this project was to get a better understanding of how to introduce or replace a CompositeState during run-time. The prototype proves that this is possible. However, to introduce a CompositeState at run-time, many things need to be accounted for.

One main thing, when substituting a CompositeState, is whether the new CompositeState is equal to the old. This was presented in chapter 6 as bisimulation. Bisimulation was used to see if two state machines were equal when viewed as a black-box, or if an actor, by merely interacting with it, could tell the difference of two state machines. In relation to bisimulation, composite states in general, can be seen as a state machine. Hence it is possible to employ the properties of bisimulation to composite states. Any two CompositeStates, that are bisimulated, even weak bisimulated or observer equivalent, can in the case of Javaframe and Actorframe be substituted with one another. This is because they expect and produce the same sequence of messages and have the same communication.

When dealing with bisimulation, exit and entry points are not included. Therefore, reference points must be treated separately. Exit and entry points are important parts of a CompositeState, and must be handled the right way in order for a CompositeState substitution to work properly. There are some basic rules considering these aspects when substituting a CompositeState. Obeying these will make the substitution easier.

Do not introduce new entry points By introducing new entry points the implementation of the original composite running the update will be awkward and will not follow good programming practice. Though it is possible it is not recommended.

Do not introduce new exit points It is actually easier to implement new exit points then entry points due to the method outofInnerComposteState, but there is a great potential to ruin the possibility of bisimulation and equivalence by doing this. Further, since new exit points must be implementation in the outofInnerCompositeState method and this is done at the implementation time of the original application, new exit points must be accounted for at this time, complicating the design of a new CompositeState.

Deleting entry and exit points It is possible to remove entry and exit points in substituting CompositeStates. These always indicates the new state of either the inner or parent CompositeState, this may alter the message sequence and thereby ruin the bisimulation of the CompositeStates.

According to the items regarding introduction of new exit and entry points, substituting an atomic state with a CompositeState in run-time is not advisable, unless one also replaces the CompositeState containing the atomic state, which is done in the prototype.

9.4 The Prototype

Making any application in any framework, assumes some knowledge of the functionality of the framework and how to deal with this functionality. Pushing the envelope and making an application on the edge of what a framework was intended for, requires a firm understanding of the framework and its design. The prototype of this thesis is such an application, introducing new components at run-time, dynamically changing the behavior of a state machine. The prototype run-time CompositeState replacement shows some important aspects of JavaFrame and ActorFrame:

- The prototype shows first of all that it is possible to introduce new CompositeStates during run-time and by doing that gives the state machine possible new behavior.
- The recursive introduction of CompositeStates, shows that it is possible to introduce an arbitrary number of recursive inner CompositeStates, even in a hierarchical manner.
- By applying the rules of weak bisimulation and observer equivalence, no action need to be taken on behalf of participating actors if a state machine is dynamically updated.

9.5 State Machines Communicating Asynchronously

One of the most difficult issues regarding asynchronous communication and state machines is that the state machines may be in different states when sending and receiving messages. Some of these issues are treated by the framework, like the message queue, but the programmer still must ensure that the state machines do not enter a deadlock, due to both waiting for the opponent state machine to send a message. Timer messages may in some cases be used to inform the system of such behavior, forcing the application to take some form of action to recover. A good way to prevent this, is by verify the safety and liveliness properties of the communicating state machine designs in a formal tool like Spin. This can be cumbersome and time consuming, and therefore timer messages where mostly used in the prototype.

9.6 UML 2.0

Due to the fact that UML 2.0 specification was not done either at the start or at the end of this project, the UML modeling part of the thesis project brought a lot of

intricate problems. This was further implicated by the fact that the NorARC frameworks are modelled based on a mixture several different techniques, like UML 1.4, ROOM, UML-RT[2] and an assumption of what UML 2.0 would bring. Based on the assumption that UML2.0 final specification would not deviate from the proposed specification, it would be possible to model applications and run-time replacement made based on the NorARC frameworks, especially when the goal of UML 2.0 is to cope with component based development.

One part that proved difficult was to model the change of behavior over time. A provisional solution that may be used in some cases is to use UML extensions or profiles. This has not been proven or tried in the thesis and is just a supposition.

9.7 Future work

Several issues in this thesis could be subject to further work, some of which are outlined below:

- Some of the messages, especially those regarding initialization and ending a play, are handled by the framework itself. One direction of future work could be to investigate how update messages also could be handled by the framework, making a way for a programmer to implement any updates. This could also include the construction of the update message and inner structure for holding crucial information regarding the update (i.e. which class that should be introduced, the location of the class file etc). This should also include how this can be done in a more dynamically way than in the prototype.
- Future work may be to see if any changes in the underlying framework could ease the introduction of new entry or exit points of a composite state, and also removing such points from a composite state. A successful approach could also make it easier to replace an atomic with a composite state.
- When the final UML 2.0 specification is released, one could revisit both this thesis and the UML specification for all the sub layers of the NorARC framework to make the models consistence to the UML 2.0 specification.

9.8 Summary

The previous discussion is summarized below:

- Neither JavaFrame nor ActorFrame, either prevents or actively, support dynamic change of the state machines behavior.
- When substituting a CompositeState, one must ensure that the substituting CompositeState is at least weak bisimulated and observer equivalent to the old CompositeState.
- Even though the CompositeStates are both bisimulated and observer equivalent, all exit and entry points must be handled in the right way for the new state machine to work properly.

- It is important to verify the new message sequence of the state machine after the CompositeState replacement. This is tightly coupled to the bisimulation aspect.
- UML may be used to model run-time class replacement, but cannot at this point model dynamic change of behavior, in the sense described in Chapter 1.

Chapter 10

Conclusion

This thesis has evaluated the NorARC service creation environment and especially the frameworks JavaFrame and ActorFrame with respect to dynamic change of behavior during run-time. A good approach for dynamic update of state machines within the service creation environment may reduce the downtime and ease the development of services in the mobile network.

The thesis started out by explaining some of the core aspects regarding the framework (i.e. state machines and asynchronous communication). Thereafter, JavaFrame and ActorFrame were described in detail, revealing the inner functionality of the frameworks, their patterns of design and development guidelines. These patterns are modelled based on an anticipation of the UML2.0 final specification not yet published. Therefore, vital aspects related to the thesis and frameworks from the UML2.0 proposal were introduced in Chapter 4. All this culminates in a prototype, which shows the principals and usefulness of the thesis ideas.

Although the prototype proved that run-time changes of state machine behavior are possible, it also showed that there are several pitfalls that must be avoided. These can be divided into three groups.

JavaFrame issues When altering a state machines behavior by substituting its CompositeState using JavaFrame, there may be some problems. The issues are not related to pattern or design flaws, but rather how JavaFrame is implemented. Enforcing a substitution that has an impact on how JavaFrame is implemented, violates *good* programming practice. Introduction and removal of reference points are in fact a situations where both the implementation and good programming practice are affected. The JavaFrame implementation makes it difficult to introduce or remove new reference points. Eluding these implementation issues often violates good programming practice.

ActorFrame issues There are not many problems related to the ActorFrame framework. At this level of abstraction problems tend to be less related to programming. Who shall initialize the update? How, if at all, does one inform the affected actors about updates? These questions must be answered on a case to case basis, because they are tightly coupled to the type of service, type of update and type of actors involved.

General State machine and asynchronous issues Asynchronous communication

and communicating state machines have, by nature, a lot of pitfalls. Although some of these aspects are hidden by the framework (e.g. message queuing), some crucial aspects are still left for the application designer to work out. These are problems like substitution of non-equal state machines deadlocks and nondeterministic behavior, which always are present when dealing with state machines. These may even seem irrelevant for developers well versed in state machines and asynchronous communication, and familiar with formal tools for verifying such.

We have, through this thesis project, shown that the use of NorARC service creation environment *as is*, makes it possible to replace some CompositeStates within the state machine. In such cases, the composite states are ideal candidates for changing state machine's behavior. The frameworks, and mostly the JavaFrame framework, may need some changes concerning the implementation, especially regarding reference points, before being an overall ideal candidate for such run-time CompositeState replacement.

Further, we have shown that by the proposed UML 2.0 specification it is possible, with class, sequence, and communication diagrams, to describe the introduction of new classes in run-time (e.g. CompositeState classes). On the other hand, we have not found any good approach to model the dynamic change of behavior over time. The introduction of new classes in run-time may indicate that the behavior may changed, but not in a 1:1 relation.

Bibliography

[1] Bræk, R., Husa, K.E., Melbye, G.
ServiceFrame Whitepaper
URL: http://www.item.ntnu.no/lab/nettint1/ServiceFrame/ServiceFrame.html(visited February 2003)
Ericsson, Asker NORWAY

[2] Loyns, A.
UML for Real-Time Overview
ObjectTime white paper
http://www.rational.com/media/whitepapers/umlrt_overview.pdf (visited January 2002) April 1998.

[3] Haugen, Ø., Møller-Perdersen, B.
JavaFrame:Framework for Java Enabled Modelling
Ericsson Research NorARC - Applied Research Center, Ericsson Norway

[4] U2 Partners, Proposals for UML 2.0
Superstructure and Infrastructure, Available
from http://www.u2-partners.org

[5] Java™2
http://java.sun.com/

[6] Nedward, T.
Server-based Java Programming
Manning, 2000

[7] Johnson, R.
How Framworks compare to other object-oriented reuse techniques.
Communication of the ACM
October 1997 /Vol.40, No.10
Pages:39-42

[8] Kobryn, C.
Will UML 2.0 Be Agile of Awkward?
Communication of the ACM
January 2002
Vol. 45, No. 1

[9] Haugen, B.
JavaFrame 2.5 Modeling Guidelines
March, 2002
Rev FJ2.5

[10] Agha, G.
An overview of Actor Languages
MIT, June 1986

[11] Saksena, M., Selic, B.
Real-Time Software Design - State of the Art and Furute Challenges
IEEE Canadian Review - Summer
t 1999

[12] Tomlinson, C., Scheevel, M.
Object-Oriented Concepts, Databases and Applications
Concurrent Object-Oriented Programming Languages
Pages 79-120
Addison-Wesley and ACM Press Frontier Series, September 1989

[13] Agha, G., Kim, W.,
Actors: a Unifying Model for Paralell and Distributed Computing
URL: http://yangtze.cs.uiuc.edu/Papers/Overview.html (visited February 2003)

[14] Husa, K.E.
Serviceframe software architecture document
URL:http://www.item.ntnu.no/lab/nettint1/ServiceFrame/ServiceFrame.html
(visited February 2003)
NorARC - Applied Reasearch Center, Ericsson, Norway

[15] Szyperski, C., Gruntz, D., Murer, D.
Component Software, Beyond Object-Oriented Programming, sec.ed.
Addison-Wesley

[16] Milner, R.
Communication and Concurrency
Prentice Hall International
1989

[17] Dahl, O.J., Dijkstra, E.W., Hoare, C.A
"Notes on Structured Programming", Structured Programming
Academic Press, London (1972)
pp. 182

[18] Sanders, R.T.
Service-Centered Approach to Telecom Service Development
Norwegian Univercity of Science and Technology (NTNU)
Proceedings of EUNICE2002

[19] Douglass, B.P.
Modelling Behavior with UML Interaction and Statecharts
URL: http://www.omg.org/news/meetings/workshops/RT_2002_Workshop_Presentations/(visited Mars 2003)

[20] Martin, R.C.
UML Toturial: Finite State Machines
Enginering Notebook Column
C++ Report, June 1998

[21] OMG Unified Modellig Language Specification
Version 1.4
September 2001
URL: http://www.omg.org/cgi-bin/doc?formal/01-09-67 (vistited February 2003)

[22] OMG Unified Modellig Language Specification
Version 1.5
Mars 2003
URL: http://www.omg.org/technology/documents/formal/uml.htm (visited April 2003)

[23] Björkander, M., Kobryn, C.
Taking the Next Step with UML 2.0
Elektronik i Norden
July 2002

[24] Björkander, M., Kobryn, C.
Unified Modeling Language is taking the next step
Embedded Control Europe (ECE)
May 2002