M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

NTNU – Trondheim
Norwegian University of
Science and Technology

Methods for architecting software

Peter Gammelgaard Poulsen
(s093263)

Kongens Lyngby 2014

DTU
NTNU

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

This thesis covers the challenges of developing event-driven software applications. It covers the analysis of the problem in order to suggest methods for architecting software for designing and implementing a framework dealing with the challenges of developing software. The implemented framework uses concurrent components to divide the overall behavior of an application into contained parts, having their own execution context. Further a message passing implementation is suggested, that uses ports and channels to establish a decoupled messaging system between components. Lastly a statechart engine is designed and implemented providing a way of keeping track of abstract states in a component. The framework is implemented in Objective-C, allowing it to be used to develop applications for the iOS platform and for the Mac OSX platform. The prototype application shows how the framework can be used to model an iOS application, using the abstraction introduced by the framework. The framework has been tested as to establish the correctness of the implementation and the performance in order to determine its usage for large complex applications. It is concluded that the methods considered and implemented into a framework provide a way of architecting event-driven software in ways that increases the abstraction level and introduce new possibilities for debugging event-driven applications.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark under main supervision of Nicola Dragoni, co-supervision of Rolv Bræk and external supervision of Philip Bruce in fulfillment of the requirements for acquiring a M.Sc degree in Engineering, Security and Mobile Computing from Technical University of Denmark and a M.Sc degree in Security and Mobile Computing from Norwegian University of Science and Technology.

Kongens Lyngby, June 30, 2014

Peter Gammelgaard Poulsen (s093263)

Acknowledgements

I am deeply thankful to my supervisors Nicola Dragoni and Rolv Bræk for giving me the freedom to explore on my own, while still being extremely helpful whenever I had any questions.

A big thank you to Philip Bruce for sharing his visions with me and for his extraordinary mentoring and guidance throughout the project. I would like to say thanks for the time and effort he put into the project, including our meetings, helping me stay focused and pushing me forward.

I would like to thank Katharina Kluge for her patience and help throughout the whole project period.

A special thanks to all employees at *Shape A/S* for being friendly and helpful to me every day and for allowing me to win a *Towerfall* match once in a while.

Contents

CHAPTER 1

Introduction

This chapter covers the motivation and background of the thesis by looking at the challenges of developing complex event-driven software, which leads to the problem description in which the thesis focuses. Further the requirements for the proposed solution are established including a specification of the developed prototype application.

1.1 Motivation & background

Developing complex software is a challenging task that requires planning and structure in order to master. Often software systems contain many bugs that make them unstable. When developing software there are several key elements that will improve the quality of code. One of these is to build a system that is robust, meaning that it behaves well in all scenarios including unexpected situations. Further the code should be easy to maintain and get an overview of. Often software projects are developed by teams that need to cooperate. This requires the software to be readable and well structured in order for people to understand it. Another important aspect is how easy it is to test the software in order to make sure that it behaves correctly and does not contain critical bugs.

Modern software solutions for desktop and mobile devices are hard to write and maintain. This kind of software is often event-driven with a user interface. The user interface allows the user to create events by interacting with it. Furthermore events can be generated by the system. It can be in the form of hardware such as sensors, it can be from other running applications or from the operating system itself. The complexity of even-driven applications comes from both the user interface and the model layers of the application. A major problem is isolating all the different states in which an application can be in and making sure it acts correctly when events are triggered. Incoming events are asynchronous and handling those is a complex task in graphical user interface(GUI) applications and it often results in software, which has many edge cases that are hard to discover and code that is difficult to test. The end result can be buggy software that has unexpected behavior. A common technique for creating more robust software is to lower the level of complexity. Lower complexity often results in fewer edge cases and is therefore easier to test and maintain. However, lowering the complexity may result in a loss of functionality and it is merely a result of not having better ways to architecting asynchronous software.

1.1.1 Complex system

The thesis covers methods suitable in order to architect and build complex and asynchronous event-driven systems. According to Booch (1993, p. 10) five attributes should be considered in order to classify a system as complex.

- **Hierarchy** - A complex system contains subsystems that each contains their own subsystem until an elementary component is reached.
- **Components** - The various parts of the system can be divided into components where internal interactions in a component are stronger than the interaction between the components.
- **Usage** - The user of the system determines the parts of the system that are in use.
- **Reuse** - A hierarchic system is composed by a *few* subsystems that are reused in various combinations and arrangements.
- **Evolvement** - A complex system needs to be designed so it evolves from a simple working system. A complex system cannot be designed and implemented from scratch.

Thus the software systems in interest are complex in the sense that they consist of various parts whose usage is not static but determined by usage of the system. They are structured in a hierarchical way, which allows the parts of the system to be reused in various arrangements.

1.2 Problem description

Software developers are faced with many challenges when developing complex applications. As the code base of a software system increases, the complexity increases as well and it becomes more challenging keep the software robust and maintainable

This thesis analyses methods for improving the architecture of software in complex applications and proposes a framework for easily applying these methods when developing iOS applications.

1.3 Context

Developing complex software is a general problem. While the findings in this thesis cover many common aspects of software development, the focus will be on software development in a specific context, which is defined in the following sections.

1.3.1 Event-driven GUI applications

The primary context in interest is event-driven programs where the events come from a GUI. Such programs are non-deterministic as the flows are determined by the usage. In traditional sequential programming, it is the program that is in control of the flow. When needed the program will ask the user for input and wait until it has been provided. This means that the program and the user are always synchronized. However, as explained by Schmidt (2003, p. 567) the tables have turned in event-driven programs. Instead of a user reacting to a program, the program reacts to the user. This is because the user can provide input at any time. It makes the program asynchronous. The user determines the usage. The asynchronous events are a key element in the complexity of event-driven programs.

The motivation for making GUI programs asynchronous is to higher the user experience. By allowing the user to interact with different parts of the application at the same time gives a more powerful application. Tasks performed by a program often take an undefined amount of time. It may depend on the CPU, the network or the hard drive. If everything were synchronous the program would only be able to do one thing at a time. For GUI application this would mean that the user would not be able to interact with the application while for instance the application was opening a file on the disk. While performing a time consuming task, the application would not responds to the user interactions, which would result in a poor user experience. Thus an event-driven application needs to be asynchronous so, while opening a file, the application can still receive new inputs from the user.

1.3.2 Mobile phone application

Modern mobile phones, also known as smartphones, provide a base for developing complex applications and this will be the main context of the thesis, However, many of the challenges addressed are also relevant for developing desktop applications. Since the introduction of the App Store for iPhone in 2008, the need for software running on mobile phones has exploded. Furthermore, the capabilities of the devices have increased. Most smartphones today have performance that is similar to desktop computers. However, especially memory consumption must be taken into account when developing for mobile, as it is lower than normal desktop computers. Together with the various sensors and an advanced operating system, smartphones provide a base for building complex systems. One of the major operating systems for smartphones is iOS, which is built on top of a Unix-like operating system. The environment for building applications for iOS is object-oriented using `Objective-C`[1], which is a general-purpose language that is suitable for building applications consisting of a large codebase. Apple has announced a new version of the iOS platform every years since

[1] 2nd of June 2014 Apple announced that they developed a new language called Swift as an alternative language for developing native applications for iOS and OSX.

the introduction, and thus it is often changing as libraries are being deprecated and new once are being introduced.

1.3.3 User experience

Applications for iOS are driven by a user interface, which act as the layer between the user and the device. The visual impression a user gets from an application comes from the user interface. As presented in Klein (2013, p. 155) having an attractive user interface is fundamental for the user experience. The contentment the users get comes from the interactions in the application. Good interactions increase the users understanding of how to use the application. A real challenge when developing mobile software is to create something that is delightful to use, while solving a problem.

Further in order to increase the users understanding of how to use the software it should be intuitive with great interactions. The application should be kept intuitive and interactive at all times in order to provide a better user experience (Services, 2012, p. 2). The competition among mobile applications is often very high with many competing applications offering similar functionality hence a user experience is an important factor when developing applications for mobile.

1.3.4 Shape A/S

The project has been supervised externally by Philip Bruce, co-founder and senior developer at Shape A/S, which is specialized in developing applications for mobile devices. The investigated problems in this thesis are directly connected to challenges Shape A/S meets daily when developing software for iOS. The focus has been on applying theory and methods in order to solve problems when developing iOS applications.

1.4 Goals & requirements

The overall goal of this thesis is look into method for building complex software for iOS. Based on the methods, the goal is provide a framework for architecting and developing complex application for iOS. The following requirements have been established for the proposed framework.

- Implemented in Objective-C for integration with the iOS platform.
- Support for integrating with the existing iOS SDK and third party frameworks.
- Ability to adapt to platform changes as new versions of iOS are released.
- Performance suitable for use in developing large complex applications.
- Convenient to work with for the developer.

The last requirement is weak in the sense that it is subjective whether a framework is convenient or not, however it requires that the framework keeps the implementation overhead required to use the framework low.

1.4.1 Prototype

In order to explain how the framework can be used to develop iOS applications and to make a practical verification of the proposed framework, a prototype application should be developed that uses the framework. The prototype should be comprehensive enough to qualify as complex. As described in Andrachek et al. (2013, p. 27) the complexity of an application comes from data types having relations. Furthermore the attributes from Section 1.1.1 put requirements on the system in order for it to be considered complex. Because of that the following properties have been established for the prototype application.

- Containing several data types.
- Data types that are related to each other.
- Containing several views showing different but dependent data.
- Having views showing the same data in different ways.
- Allow manipulation of data.

Specification

Based on the requirements the following specification describes an application for tracking expenses that should be implemented using the framework.

An expense tracker allows a user to save expenses that are shared by a group of people. This is done by creating an expense and specifying basic information such as category and amount. Each user can see all the expenses in the system and the debts between the users based on the added expenses. The application contains the following use cases:

- As a user I can login to the system.
- As a user I can add a new expense to the system by providing an amount and a category.
- As a user I can delete an expense.
- As a user I can see a list of all the expenses.
- As a user I can see the debts between users in the system.

The specified prototype qualifies as being complex since it contains the properties listed in Section 1.4.

1.5 Structure of thesis

The structure for the rest of the thesis is shown in Figure 1.1, where for each section an arrow shows the section that it built on top of. Further description of each chapter is listed below.

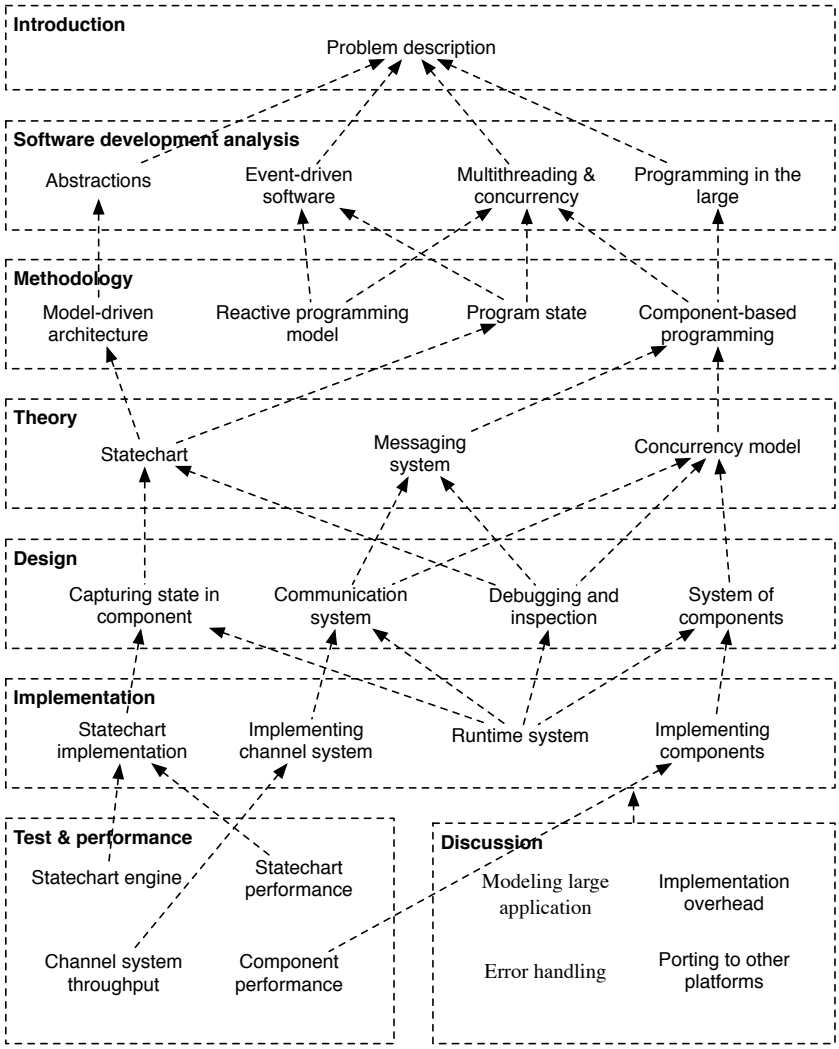

Figure 1.1: Overall structure of thesis.

Chapter 2 analyzes the context, while taking the problem description into account in order to established a set of challenges that should be considered further.

Chapter 3 analyses a set of method for dealing with the challenges of building complex event-driven software and based on the analysis selects suitable methods.

Chapter 4 explains the theory of the methods selected and establishes a foundation of knowledge about them.

Chapter 5 covers the design decisions and modeling challenges faced in the solution.

Chapter 6 explains the implementation of the implemented framework and the prototype application.

Chapter 7 provides the results of the various tests performed on the proposed solution.

Chapter 8 evaluates the initial goals and requirements against the proposed solution.

Chapter 9 covers the overall findings and suggestions for further work.

CHAPTER 2

Analysis

This chapter analyses the problem presented in the previous chapter. This is done by going deeper into the software development challenges for developing event-driven software for iOS. The findings of the analysis will in the next chapter be used as a foundation for evaluating suitable methods for architecting event-driven software.

2.1 Programming in the large

As presented by DeRemer and Kron (1975, p. 114) two overall approaches are used when writing software. *"Programming in the small"* refers to writing smaller pieces of software, which mainly solve a single purpose. On the other hand "Programming in the large" refers to writing more complex systems, which involves many developers and a large codebase. Programming in the large is the kind of software in interest in this thesis. DeRemer and Kron (1975, p. 114) introduce the *"languages for programming-in-the-small (LPS)"* as typical programming languages and the concept of a `module` by saying that

> *"the term module refer to a segment of LPS code defining one or more named resources. Each resource is a variable, constant, procedure, data structure, mode, or whatever is definable in the LPS."* (DeRemer and Kron, 1975, p. 114)

Further they state that *"structuring a large collection of modules form a system"*. In other words they define `module`s as being separate chunks of code that together form a system. As explained by Szyperski (1999, p. 35) the term `module` is weakly defined and in the literature it is often also referred to as an `object`, a `class` or a `component`. For now the term `module` is used. van Roy and Haridi (2003, p. 457) present several challenges when building a system of modules.

- **Module connections** - As a system consist of many modules that together form a whole, these modules are connected to each other. The connections can either have a static or dynamic structure. The static structure is known when the application starts and it does not change during execution. A dynamic structure however is determined during run-time and may change as the program is being executed.
- **Module communication** - Certain modules depend on each other in order for the system to work. Hence the modules need a way for communicating with

each other. Many ways for defining the communication between modules exist. A way of doing it is using procedures where the control is passed back and fourth between components.

- **Module implementation independency** - As noted by van Roy and Haridi (2003, p. 460) *"The interface of a component (module) should be independent of the computation model used to implement the component (module). The interface should depend only on the externally visible functionality of the component (module)"*. In other words the interface visible to other modules should be independent of the implementation.
- **Module execution context** - In some scenarios the execution context of a module should be independent from other modules, since otherwise the work done by one module would prevent another module from doing its work.

In short this means that a system consists of independent modules structured in either a static or dynamic way, which communicate with each other. Each module should, in some scenarios, execute independently of others.

2.2 Abstractions

Software development is built on abstractions.

> *"As far as we know, the most successful system-building principle for intelligent beings with finite thinking abilities, such as human beings, is the principle of abstraction"* (van Roy and Haridi, 2003, p. 418)

Abstractions are structures that reduce ideas to simple forms. Details are omitted and thus the complexity lowered, however good abstractions capture the details as well. In software many layers of abstractions exist. An example of, how the level of abstraction has changed in the history of software development, is the move from assembler to higher level programming languages. By raising the abstraction, to a level closer to how humans think, the complexity is decreased. For programming in the large, abstractions help keeping the complexity low. When building a complex piece of software, abstractions are used to capture the idea of the system. However, going from an abstraction to the actual implementation is not necessary an easy task. The idea that you can simply design your abstractions and afterwards implement those abstractions, capturing the exact meaning of the abstraction wishful thinking (Jackson, 2005, p. 14). This means that many challenges exists when dealing with abstractions.

2.2.1 Model-driven architecture

Model-driven architecture is an approach for developing software systems by implementing a system based on models. It is a way of increasing the abstraction level of the system. In this context the word `model` means a platform independent specification of a complex system.

Code-driven & Model-driven development

Developing complex systems require structure on the code. Models describe the design of the system by abstracting away details and thus make it easier to understand the system as a whole. Models can be used to specify the required functionality in the system. The system is implemented using code, which needs to be debugged, tested and maintained. The relation between the model and the code is often weak or non-existing for software systems (Kelly and Tolvanen, 2008, p. 5). However, different approached can be used as listed below.

- **Code only** - The *code only* approach is where the code is developed without any use of models. The code is the highest level of abstraction used.
- **Separate model and code** - By having *separate model and code*, both the models and the code are created but they are not kept automatically synchronized. A change in one requires an update in the other.
- **Code visualization** - With the *code visualization* approach, the models are created based on the code and used as documentation for more easily understanding how the code is structured.

As noted in Section 1.3.2, iOS is a platform continuous development. Providing model-driven development for such is a difficult task since changes to the platform may break the models. This is in conflict with the goal from Section 1.4 of adapting to platform changes. A code-driven approach handles changes better since code written explicitly by a developer is easier than debugging code generated by a code generator. The proposed solution should be code-driven, however in order to obtain a higher abstraction of the system `code visualization` should be considered.

Generic vs. Domain specific model

The generic-purpose modeling language, known as Unified Modeling Language(UML), was designed to support creating models for all kinds of software. Using a generic model means that in order to model a system operating within a specific domain, the concepts from the domain have to be mapped into the generic concepts from the generic model. As a contrast to a generic model, it is possible to use a domain specific model instead. By doing so the mapping disappears because the domain specific model supports the constructs from the domain. A domain specific model needs to be implemented using a domain specific language, as a general-purpose language does not know about the domain specific elements in the model (Kelly and Tolvanen, 2008, p. 5). The advantage of having a specific domain model is the possibility of creating code generators that can generate code based on the models. Code generation targets a specific platform for the model to be implemented on. The advantage of generating code is that it can save development time and create code containing fewer bugs. However, using a domain specific model introduces limitations. This is because the development becomes sandboxed in the sense that only structures supported by the

domain model can be used and thus it is fragile to domain changes that requires the model to be updated. Also creating a domain specific language supporting all the necessary concepts for architecting iOS applications is a massive task to develop and maintain.

As explained in Section 1.4 the solution should adapt to changes in the platform, thus a domain specific model is unsuitable. Instead only generic models will be considered further.

2.3 Event-driven Software

As mentioned in Section 1.3.1 the iOS platform is event-driven. In an event-driven application the control-flow is determined by events occurring during the execution of an application. The control-flow determines the order in which statements and methods are executed in a program. This introduces challenges, which are discussed in the following sections.

2.3.1 Event-loop & Control-flow

In order to understand the challenges of event-driven software, it is important to look at how the system handles events. The way it is handled in iOS is using an event-loop that takes care of incoming events. The event-loop is managing a thread and the main purpose of the event-loop is to keep its thread busy when there is work to be done and otherwise allow it to sleep. The thread has work to do when events are happening in the system. Incoming events are put on an event queue and then served one by one. An important property of an event-loop is *Run-to-completion* which ensures that an event is processed completely before the next event can be taken from the queue. An event occurring very often may cause the event queue to increase in size and have other events suffer from this by forcing them to wait in the queue. For this reason event-driven programs may contain several event-loops that process different kind of events. For an event-driven GUI application, a GUI event-loop is used to process events concerning the GIU. This is done to keep the GUI interactive. The thread used by the GUI event-loop is called the *main thread* or the *GUI thread*. Figure 2.1 shows the relations between the application, the event queue and the main event-loop in an event-driven application. The event-loop is started once the application starts and runs until it is terminated. The event-loop process events as they are added to the event queue.

2.3.2 Events

The control flow of an event-driven application is dynamic since it is determined by the events that occur in the system, and since the occurrence of events is determined during run-time and depends on the environment, the user and the state of the system.

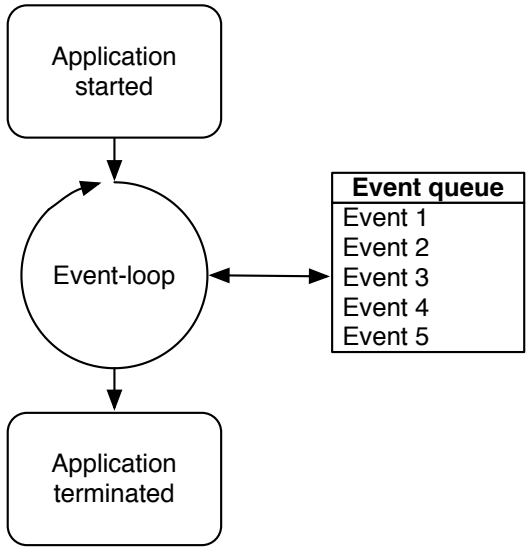

Figure 2.1: Application with event-loop

Events in the system are generated for various reasons, which are presented in more detail below.

2.3.3 User interaction events

Events coming from user interactions are the main way for the user to control the application. All iOS devices have a touch screen in order to allow user interaction. The user generates user interaction events by touching the user interface. Depending on the specific application this could result in buttons being pressed or touch gestures being performed. These events all result in events being queued by the main event-loop in order for the application to respond to these interactions.

2.3.4 Hardware events

Devices running iOS contain several hardware sensors that can be used to generate various events. This could for instance be the a gyroscopic for monitoring motion, an accelerometer for orientation changes, a magnetometer for magnetic field changes or a GPS for location changes. All of these sensors report about changes by sending events to the system. On iOS events from hardware components are typically only fired if an application specifically requests it. This is done in order to save battery and not load the application with unnecessary events.

2.3.5 System events

The last kind of event considered is events generated by the operating system. This could for instance be when the device is running low on battery or the application is using too much memory.

All of the events described above define the control-flow in an event-driven application. The fact that these events can occur at any time during the lifetime of an application in an infinite number of combinations creates challenges when developing event-driven software. First of all is it impossible to test exactly how the program behaves in all scenarios. An application must be structured in a way where the order of the events will not result in undesired behavior. However this is often difficult and many edge cases may exist in a complex application. Edge cases are scenarios where a certain combination of events is able to put the application into a state that requires special handling. As mentioned in Section 1.1 a robust piece of software behaves well even in unexpected scenarios. If the developer of an application has assumed in which order events may occur it may be possible to force an application into an unexpected scenario by the user simply by touching the display and thus triggering user interaction events. When dealing with complex applications many control-flows exist and it is difficult to make sure that none of them result in unexpected behavior such as crashing the application.

Because of the above described challenges the proposed solution should take account that events can occur at any time and it should handle this in a way where the state of the application is controlled and unexpected edge cases caused by the order in which the events occur are avoided.

2.4 Multithreading & concurrency

As mentioned in Section 1.3.3 a fundamental property when developing applications with a user interface is to keep the application responsive at all times. A non-responsive interface gives the impression that the application is frozen and crashed. As explained in Section 2.3 an event-driven GUI application contains a GUI event-loop. In order for the application to stay responsive each iteration of this event-loop must be kept as short as possible. In fact, iterations that takes longer than 1.6 milliseconds result in lost frame with a frame rate of 60 hertz. Because of the `run-to-completion` property, it is important that each event does not end up blocking the GUI thread. A thread is blocked when it is waiting for a resource to become available. Many operations cause the thread to be blocked. An example of this is when communicating with a web server or reading from the hard drive. Since GUI applications may contain blocking operations and in order to keep the user interface interactive these operations must instead run on a different thread than the main thread, thus multithreading is needed. Multithreading allows the program to run several threads simultaneously. Most modern mobile phones contain several cores in

the CPU but even smartphones with a single core in the CPU can take advantage of multithreading. A scheduler will quickly switch between the threads and thus making sure those events from the GUI event-loop is handled with a higher priority in order to keep the user interface responsive.

2.4.1 Communication between threads

As mentioned in Section 2.3.1 event-driven GUI applications have a single GUI thread, which handles all drawings. The reason for that is that often the frameworks used to draw the user interface are not thread-safe. Because of this only a single thread can manage the drawing and thus the need for a GUI thread. Other threads trying to draw would trigger unexpected behavior and may crash the application.

However working with threads introduces many challenges. A major challenge when developing in a multithreaded environment occurs when the threads need to communicate with each other, because of communication between concurrent parts should be considered as a challenge for developing GUI applications.

2.5 Summary

Based on the analysis in this chapter, the following challenges have been established.

- **Structure of applications** - As an application consists of several modules, challenges exist for how these are structured, while providing an independent implementation and the possibility for an independent execution context.
- **Communication between concurrent parts** - As the applications consist of concurrent modules, challenges exist for how the communication between these is defined.
- **Keeping track of state** - As events occur the current state of the application often determines how the reaction should be. Keeping track of this state is challenging for large systems.

Further the following requirements have been established.

- **Abstraction** - The methods should provide an abstraction level for lowering complexity of the system.
- **Code visualization** - In order to higher the abstraction and provide a detail view of the system, the possibility for providing code visualization should be considered.
- **Generic model** - The methods should follow a generic model in order to fit in with existing platform and SDK.

- **Asynchronous event** - The methods should allow asynchronous events coming from various parts of the system.

These challenges will in the next chapter be used in order to evaluate suitable methods and models for building event-driven software for iOS.

CHAPTER 3

Methodology

The analysis of the problem performed in the previous chapter resulted in several challenges and requirements concerning the development of complex event-driven software for iOS. This chapter performs an evaluation of several state of the art software development models and methods in order to select suitable approaches. The next chapter then looks deeper into the theory behind these methods.

The following example of an iOS application will be used as base for showing examples of how a given method can be used to an application.

Example 1 *Consider a simplified version of the prototype application as described in Section 1.4.1, where the user is only able to see the expenses added to the system. The expenses are loaded from a web server and the user can press a button to make the application fetch the expenses from the server.*

3.1 Structure of applications

As defined in Section 2.1 how the system is structured is an important challenge when developing complex applications.

> *"One could write the program as one big monolithic whole, but this can be confusing. A better way is to partition the program into logical units, each of which implements a set of operations that are related in some way."* (van Roy and Haridi, 2003, p. 223)

3.1.1 Object model

The object model and object-oriented (OO) programming captures the concept of a module as presented in Section 2.1 into objects. This was first introduced with the programming language *Simula* in 1967, but OO programming, as we know it today, was introduced in *Smalltalk* in 1976. OO programming is a way of increasing the abstraction level compared to only having functions. The emphasis is moved to the data instead of the algorithms. The introduction of an object that encapsulates features lowered the complexity of software development (Buyya, 2009, p. 29). A powerful property of OO programming is the introduction of inheritance, where as noted in van Roy and Haridi (2003, p.421) *"It is possible to build the system in incremental fashion,*

as a small extension or modification of another system.". This can be seen as a way of defining the connections between objects and it gives the opportunity to make the implementation less redundant. However this also introduces a strong dependency in the system, as an object depend on the object they inherit from. Because of this van Roy and Haridi (2003, p.421) states that *"We recommend to use it whenever possible and to use inheritance only when composition is insufficient".* Using object composition the system is structured into relations between objects, which gives a more decoupled design.

As noted in Section 2.1 a challenge for modelling modules is having an independent implementation. This can be achieved in OO programming by defining a public interface for other objects to call, which does not reveal the implementation of the object. However, as further discussed in Section 2.1 having an independent execution is not supported by the object model as objects have a dependency to each other if they are executing on the same thread as discussed in Section 2.4.

3.1.2 Model-View-Controller

A way of providing a structure of a GUI application is to apply the *Model-View-Controller* pattern. This abstraction divides responsibilities and structure of an application into three overall interconnected layers. The idea of the abstraction is to separate code that is independent of each other in order to make a more decoupled and reusable codebase. Each layer communicate through an interface and the idea is that a layer can be changed without having an effect on other layers, as long as the interfaces are kept intact. Decoupling the layers from each other gives flexibility. The cleaner the interfaces are between the layers, the easier it is to change or modify a layer without affecting other layers.

The *model* layer keeps the state and all the logic of the application. The model provides an interface so other layers can use the data and services provided by the model. The *view* layer represents what is visible for the user. The view is independent of the exact data that it needs to show and it provides an interface for specifying the data. Further the view generates events interactions from the user. This could for instance by a view acting as a button and generating events when the user clicks the view. The last layer is called the *controller*. It is responsible for connecting the model and the view layer by making sure that data from the model is shown in the view and events in the view triggers updates to the model. An example of the interaction between these three layers is shown in Figure 3.1

Example 1 could be modeled using Model-View-Controller as illustrated in Figure 3.2. The view layer contains a button and a list, which is populated with the expenses. The model layer is responsible of fetching the expenses from the web server and the

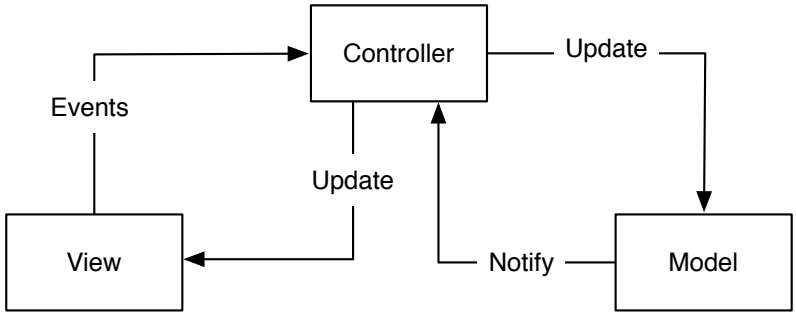

Figure 3.1: Component interaction in Model-View-Controller

controller layer is in charge of responding to the touch event when the button is clicked and asks the model to update the data from the server. Further it is in charge of updating the view once the model has fetched the expenses.

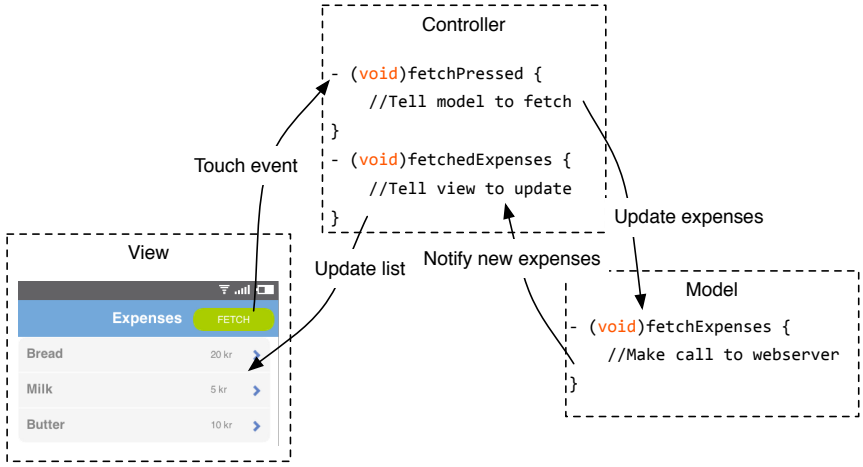

Figure 3.2: Modeling example 1 using Model-View-Controller

While the *Model-View-Controller* pattern separates the responsibility of the code into three overall layers, the codebase can still end up being monolithic for a complex application. As each layer increases its complexity the interfaces between the layers

becomes floated with connections. This means that there is a big interface between each layer as illustrated in Figure 3.3. A property of the *Model-View-Controller* pattern is that the *view* and *model* never interface directly with each other, however as the complexity grows the dependencies between the *view* and the *controller* grow as well as the dependencies between the *model* and the *controller*.

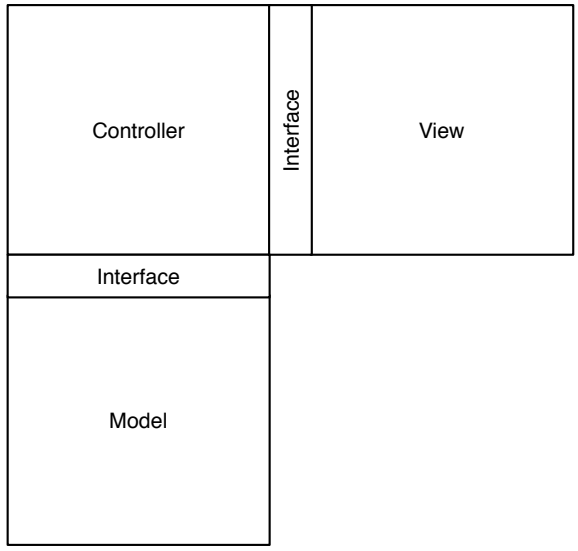

Figure 3.3: Using Model-View-Controller for complex applications.

Example 1 contains a blocking operating for fetching expenses from a web server, however the *Model-View-Controller* pattern does not define how this should be handled in a way that keeps the view layers interactive. As with the object model, defining a unique execution context for different layers is not built in. This means that a model for composing the system into smaller parts is needed.

3.1.3 Component-based

An abstraction on top of object-oriented programming is the use of components. As defined by Szyperski (1999, p. 36) a component has the following properties.

- A component is independent of its deployments.
- A component is self-contained.

- A component does not contain externally observable state.

This definition shares many ideas from the object-model. According to Szyperski (1999, p. 38) *"a component is likely to act through objects"*. However, instead of being bound to a single class, a component can consist of a collection of classes. Further this gives the possibility of defining a unique execution context for certain components. Consider Figure 3.4 that uses the *Model-View-Controller* pattern together with components. The `controller` and `view` part is combined in the GUI component. Further the model consists of two components, namely `Expense Model Component` and `Webservice component`. Since the `Webservice component` contains a blocking operation, it is running in its own context. In the example this is archived by having a dedicated execution thread.

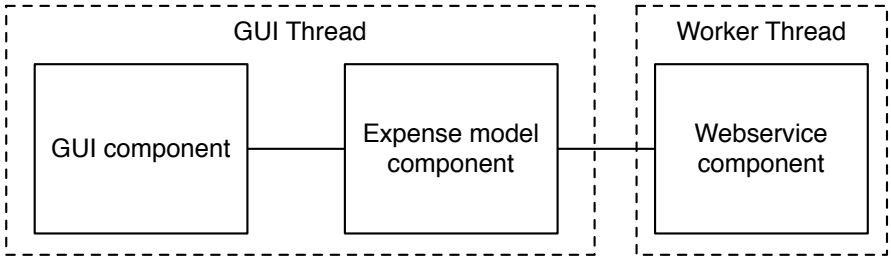

Figure 3.4: Using components for Example 1

Component-based programming provides a flexible model for structuring an application, and used together with the *Model-View-Controller* pattern a good separation of the responsibilities of the code is obtained.

3.2 Communication between concurrent parts

As explained in Section 2.4 communication between threads is needed. Methods for doing so are considered below.

3.2.1 Invoking methods

A simple way of communicating between threads is simply for one thread to cause the invocation of a method on another thread. A way of giving a thread the possibility to communicate back is using a callback (Scheifler and Gettys, 1987). When invoking a method on another thread a callback is provided. The callback represents what

will happen once the task has been completed. This is illustrated in Figure 3.5 for Example 1 where the blocking operation of fetching the expenses from the web service is running on a separate thread and the GUI thread is kept responsive. A callback is provided once the expenses has been fetched in order to update the view, as it must be done from the GUI thread as explained in Section 2.4.

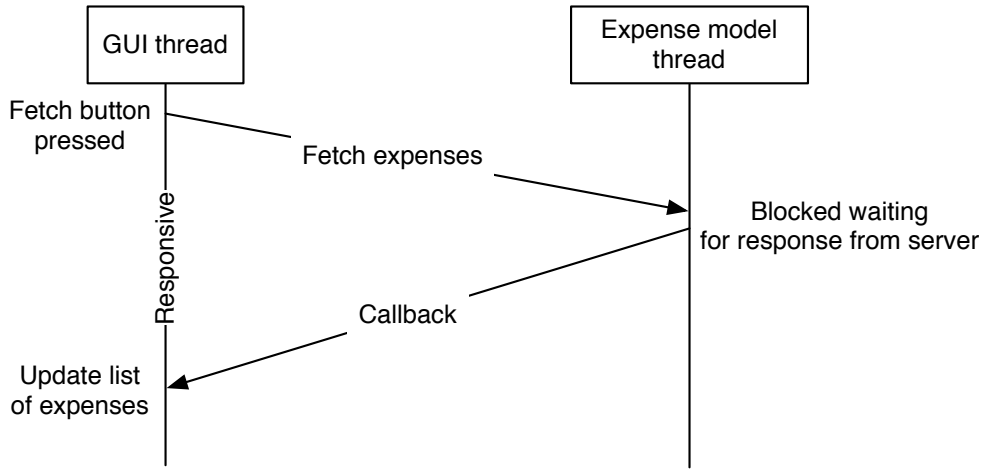

Figure 3.5: Using multithreading and callback

Using simple method invocation and callbacks provide a simple mechanism for communicating between threads, however the simple nature brings limitations. When a thread invokes a method on another thread, this method is invoked as soon as possible without taking the current state into account. If the resources needed to perform the task were unavailable at the time of the invocation the system might end up in a unexpected situation. The problem comes from the fact that, the time the message is delivered, is not managed by the receiver by instead the sender.

3.2.2 Message passing

A more general solution to communication between concurrent parts is using message passing. Instead of directly invoking behavior a message is being sent. It is the job of the receiver to act on the message. This can either be by performing a task or simple by ignoring the message. The sender can decide to whom a message should be sent to, and the receivers are not necessarily treated equally. The sender can also decide to

broadcast a message to all receivers in the system. The communication can either be synchronous where the caller blocks until a reply is received or asynchronous where the caller continues execution and the reply comes in a callback from the callee. Message passing can even be used together with the concept of callbacks, where information about how to reply is included in the message. This is done in order for the sender to determine the conversation when receiving replies. The use of message passing allows for a much more flexible system. Messages send can be queued up by the receiver and handled one by one, when ready.

3.3 Keeping track of state

In order to determine the behavior of an application, the current state must be taken into account. However keeping track of this state for a complex application is a difficult challenge. Methods for doing so will be investigated further below.

3.3.1 Program state

This context is known as the state of the system. The state can be defined as:

> *"A state is a sequence of values in time that contains the intermediate results of a desired computation."* van Roy and Haridi (2003, p. 416)

Having a state allows to keep a history of what has happened so far in the system. For an application the state is given by the memory or more precisely the current objects and their values. The state space of a system defines the number of different states it can be in. As noted by van Roy and Haridi (2003, p. 413) *"State adds a potentially infinite branch to a finitely running program"*. This means that most applications have an infinite large state space. As events occur in the system the state changes. Handling state changes is not an easy task and thus several techniques exist.

3.3.2 Global state

A simple way of keeping the state of a program is by defining it globally. A global state is achieved by having variables that are accessible from all parts of the program. Since it is global there is no need to propagate changes around in the system. This is because each part always has access to the current state directly. A way of doing so is using the `Singleton` pattern as presented in Gamme et al. (1994). The pattern allows creating one instance of an object that is accessible by all parts of the source code. In Figure 3.6 it is illustrated how a global state can be used for Example 1 to store new expenses.

The use of global state makes it easy to share state across a system. Parts of the system that have no direct connection can influence each other by changing the global state. However, the program state quickly becomes unpredictable, since the state of

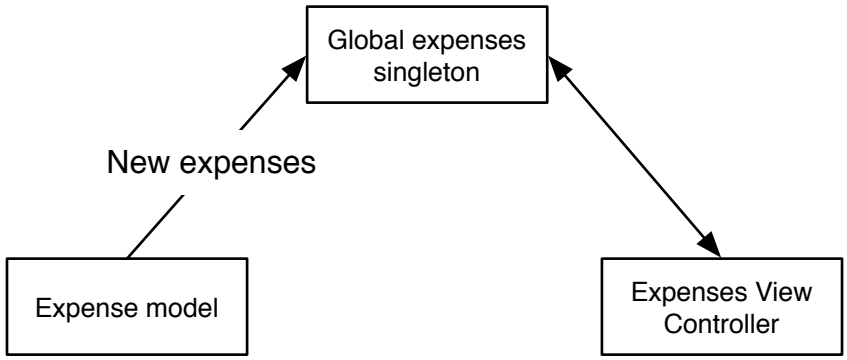

Figure 3.6: Using global state to store expenses.

the application can be changed by any part of the code base, which introduces hidden dependencies. Further the system becomes very tightly coupled as parts of the program rely on a global state. Changes to one part of the system might have influence on other parts, which makes refactoring and testing difficult. Generally global states should be avoided when building complex system.

3.3.3 Observer pattern

Instead of having a shared state across the system as with global state, the state can to be communicated across the application. More specifically the change of state can be communicated. A common pattern for notifying objects in an application about state changes is the *observer pattern*. An object, known as the subject, has a list of objects, known as observers. The observers are notified automatically when the subject changes state. This usually happens by a method being called in each of the observers to notify about the state change. Applied on Example 1 and as illustrated in Figure 3.7 the *controller* can observe the *model* for changes to the expenses, and get notified if they change.

With the *observer pattern* objects get less tightly coupled as the subject does not need to know anything about its observers. The only thing the subject knows about its observers is how to notify them upon changes. The *observer pattern* is suitable for being used in object-oriented languages. A property of the *observer pattern* is that all observers should be treated equally. Trying to put priority on the observers would introduce a tight coupling between the subject and its observers. This limitation makes it very difficult to use the *observer pattern* in certain scenarios. For

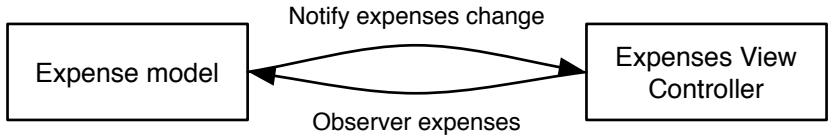

Figure 3.7: Observer pattern between model and controller.

instance to implement an order in which observers are notified or conditions for when a specific observer should be notified. Trying to use the *observer pattern* as a way of communicating state changes in a complex event-driven application with many events resulting in state changes can result in unintended behavior or even memory leaks as discussed in Eales (2005, p. 165). Another challenge with the *observer pattern* is the possibility of creating an infinite recursion. This is illustrated in Figure 3.8. The change of the subject causes a notification to the observer. The observer reacts on the notification by making a state change in the application, which again results in a change of the subject. Since observers are added and removed during run-time it can be difficult to test all cases of the application in order to ensure no recursions exist.

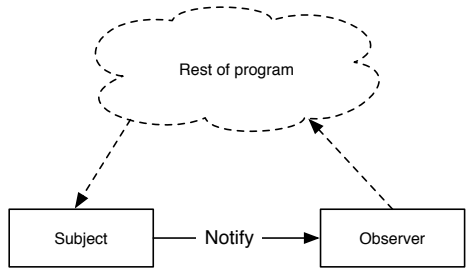

Figure 3.8: Infinite recursion with observer pattern.

3.3.4 Bindings

A pattern related to the *observer pattern* is *bindings*. Instead of an object simply notifying a list of objects as it is done with the *observer pattern*, the objects are bound together. The *bindings* ensure that once an object changes state it will automatically result in a state change in another object. A binding between two objects can either one way or two-ways. For instance *bindings* can be used in a GUI application to automatically update the view when data in the model changes. The use of *bindings* does

not require extensive work in order to be achieved in an object-oriented environment as it can be implemented using the *observer pattern*.

Using *bindings* provides a way for easily propagating changes from one object to another by simply changing the value of a variable. However when having many bindings in the system it becomes unclear what side effects a change to one object might have on other objects. This means that the same disadvantages as with the *observer pattern* exist.

3.3.5 Reactive programming model

Instead of trying to keep track of the state of an application, another approach is to try avoiding it. This is one of the features of the *reactive programming model*. The reactive programming model uses dataflow constraints as a way of propagating change in a system using reactive behavior. A simple spreadsheet is an example of reactive model, where each cell is a part of the system and a change to the value in one cell can cause a reaction in the system with the result of changing other cells.

The idea behind the reactive programming model is to create functions for the behavior of the system. For instance consider the function $y = f(x_1, ..., x_n)$ where the value of y is reevaluated every time one of the parameters x_1 to x_n changes. As explained in Demetrescu et al. (2011, p. 1) the use of dataflow constrains make the execution of a program data-driven rather than control-driven in contrast to more imperative approaches. Since the reactions happen automatically upon change of data instead. This means that a change can cause a chain of reactions across the system. A single change can cause many reactions in different parts of the system. This is done by transmitting signals around. Using the reactive programming model in an object-oriented environment makes it possible to send signals between objects. An object can then decide to react on the signal. The reactive programming model is most often used in functional languages it is also possible to model it in an imperative environment by having reactive data structures that can propagate change automatically. A key element of the reactive programming model is how the focus is on what is happening and how to react on it. The idea is to avoid keeping track of state in the program and instead let the system automatically react on events occurring. For instance when data becomes available the change can be signaled automatically to the interested parts of the system by setting up functions as a reaction to the change. For instance consider Example 1 a reaction can be set to update the expense model once the fetch button has been pressed. Similarly the list of expenses can react on the fact that new expenses have been become available. This is illustrated in Figure 3.9. The reactions can be written as functions. For instance `fetchExpenses(refreshClicked)`, where fetching new expenses is a function of the refresh button being clicked. The reactive programming model makes it possible to let data flow from one part of the system to another as events occur.

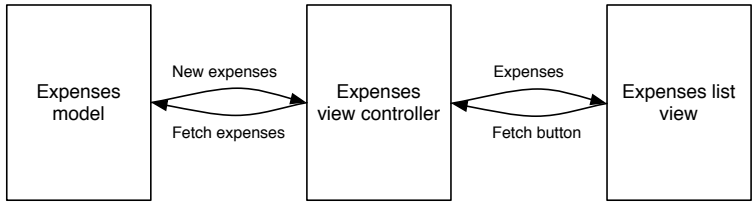

Figure 3.9: Using reactive programming to model Example 1.

The Reactive programming model provides a powerful way of defining the dataflow in system having asynchronous events. The model fulfills many of the requirements set in Section 2.5. The introduction of signals and reactive behavior introduces new abstractions. Further it provides a mechanism for having control of the events occurring in an application by setting up reactions as functions. It works in a multithreaded environment as signals can flow from different threads in the system. It tries to avoid the problem of keeping track of state in the application by avoiding state and instead use reactive behavior. Applying a reactive programming model in an imperative context requires reactive data structures and introduces many challenges. As noted in Demetrescu et al. (2011, p. 2) *"A natural question is whether the [reactive] dataflow model can be made to work with general-purpose, imperative languages, such as C, without adding syntactic extensions and ad hoc data types".*

3.3.6 Finite-state machine

The methods considered so far for dealing with state changes has been of a very dynamic nature. Another approach would be to have a more static setup where the system has a structure at compile time that determines which effect a change of the state may have. In order to introduce such structure a more controlled way of handing state and state changes is needed. A finite-state machine gives such a structure by defining a machine that captures the current state of a system.

Abstract state

A challenge of trying to enforce a structure on states is the fact that the state space of a program is often infinitely high. This is due to the unlimited number of values the memory of the program can take. By instead defining the states of a program in an abstract manner it is possible to limit it. This can be done by encapsulating the state in a state variable. The possible values for a state variable are defined by dividing the system into chunks of behavior, each represented by their own abstract

state. In the context of a state machine the word *state* refers to a predefined abstract state. As the states are predefined they have a finite number.

State machine engine

The state machine changes the state variable during execution. This happens as a reaction to events. The state machine engine determines how a program should react on event by identifying the type of the event and looking at the current state variable of the system. If the current state responds to the event, the reaction can lead to a state transition. During a state transition the current state of the system changes.

Example 1 can be modeled as a finite-state machine. This is illustrated in Figure 3.10 where the circles are states, the lines between them are state transitions and the text above the lines is the events that trigger transitions. The `No expenses` state is the starting state as indicated with arrow pointing to the state. Once the `Fetch expenses` event is fired, a state transition will be performed from state `No expenses` to state `Fetching expenses`, and similarly when the `Fetched expenses` event is triggered a transition to state `showing expenses` is performed.

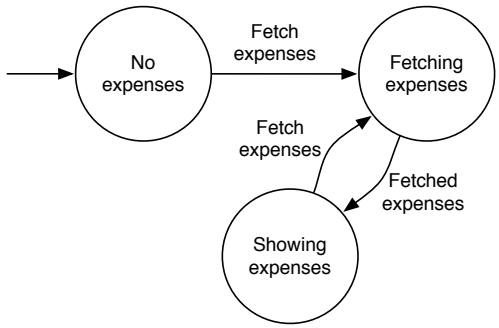

Figure 3.10: Example 1 as finite-state machine.

Actions & events

As the state machine responds to events by performing state transitions, actions can be performed as well. For instance these actions could be used in Example 1 to execute code that would change the user interface of the application reflecting the state of the state machine.
When talking about events in a state machine it is important to distinguish between the type of an event and an instance of an event. For instance the events illustrated in

Figure 3.10 represents different types of event. During execution instances of events would occur. In order to limit the number of events, an event may contain information about the event. For instance the `Fetched expenses` event could contain the array of expenses, that had been fetched.

A great feature of the finite-state machine is its graphical notation as it provides a way of doing `code visualization` as presented in Section 2.2.1. By drawing a system a better understanding of how it is structured can be obtained since code details can be neglected.

Even though finite-state machines provide a great way of introducing a structure for the state of a program they have limitations. A problem with the finite-state machine, is that even simple systems might end up suffering from state explosion. Consider the following extension to Example 1.

Example 2 *When the expenses have been fetched, the number of expenses should be shown in a label, according to the following rule. If more than 10 expenses exists then the label should say '>10', otherwise it should contain the number of expenses.*

With the extension in Example 2 the number of states required to store this information would be 12, since it would require one for every number from 0 to 10 and one for more than 10. Imagine as more requirements are put on the system, how the number of states quickly increases. Thus for complex systems the finite-state machine is not feasible.

3.3.7 Extended finite-state machine

In order to deal with the problem of state explosion in the finite-state machine the possibility of adding variables to a state is included in the *extended finite-state machine*. Each state is supplemented with the possibility to use variables. A variable can contain information about the state. This allows the number of states to can be lowered dramatically. A state having variables is known as an *extended state*. For instance consider Example 2. The number of states could simple be stored in a variable as extended state. This would result in the same number of states as in Figure 3.10.

The extended finite-state machine makes it possible to model larger applications, however it still has some fundamental limitations. For instance it does not support reusability. There is no way for states to share common behavior. As explained in Section 1.1.1 complex application consists of components that have subsystems, however the extended finite-state machine is not suitable for modeling this. Further consider the following extension to Example 2

Example 3 *The application is extended with a debts section, where the user can see the debts. While fetching expenses it is possible for the user to navigate to the debts section and fetch debts.*

With the extension in Example 3 the application all of a sudden has become much more complex. While fetching expenses, the system can also perform other actions, such as fetching debts. Since a *finite-state machine* does not allow the system to be in more than one state at a time, it is necessary to create states for all possible scenarios. This is partially illustrated in Figure 3.11. However as the figure shows the state machine becomes much more complex and unreadable. This is because the extended finite-state machines it is not suitable for modeling different parts of a system that are active at the same time. Thus a more complex model is needed.

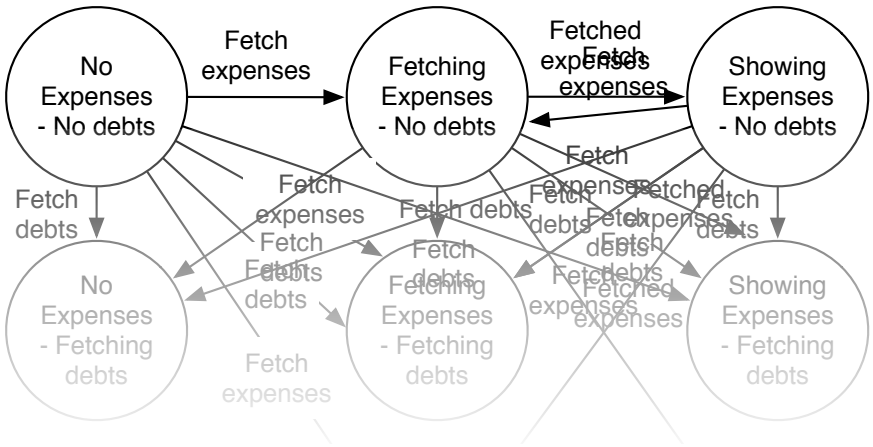

Figure 3.11: Example 3 partially shown as finite-state machine.

3.3.8 Petri net

A more complex way of modeling a system is using *Petri nets*. Petri net is a modeling language that can be used to describe a state transition system, similarly to the extended finite-state machine, but with much more expressive and flexible possibilities, which allow modeling much more complex applications. It is often used for specifying distributed systems or for modeling workflows because of the support for parallelism.

Petri nets can be made into a directed bipartite graph consisting of two kinds of nodes, the *places* and the *transitions*. Directed arcs connect the nodes in the graph. Petri nets can be used to capture the state of a system by defining the *places* as states.

When petri nets are used to model an event-driven system it can be referred to as an *events net* (Bobbio, 2010). Tokens are used to mark the currently active states. Opposite the finite-state machine, a petri net can be in several states at once, giving the possibility to model orthogonality.

Consider for instance Example 3 as illustrated in Figure 3.12. The circles represent the states of the application and the black bars are the transitions. The black token represents the currently active states. In the figure two states are active.

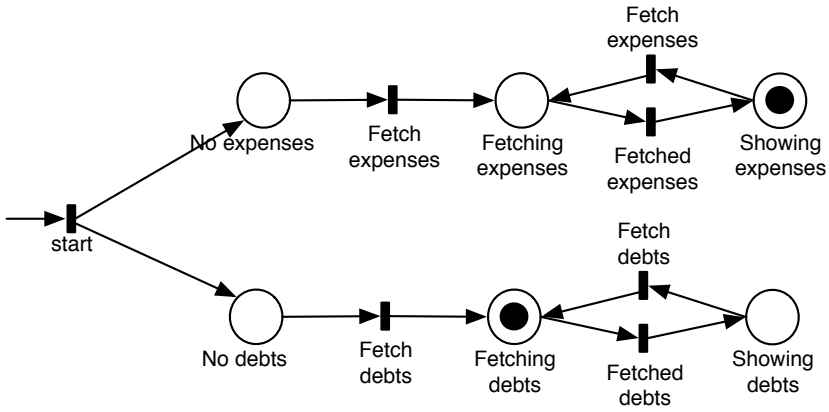

Figure 3.12: Example 3 modeled using Petri net

Petri nets support many advanced modeling features that makes it possible to model a complex application and keep track of the state in an application. Further the visual notation provides a good abstraction for understanding of the internal behavior of an application.

3.3.9 Statecharts

Another model that extends the traditional finite state machine as explained in Section 3.3.6 is statecharts. It extends the finite-state machine with many capabilities, however the two most fundamental ones are hierarchically nested states and orthogonal regions. These constructs allow capturing the state for much more complex applications. For instance Example 3 is being modeled as a statechart in Figure 3.13. Using orthogonal regions it is possible model having many active states at once.

As explained by Horrocks (1999, p. 204) “Statecharts are easier to understand than

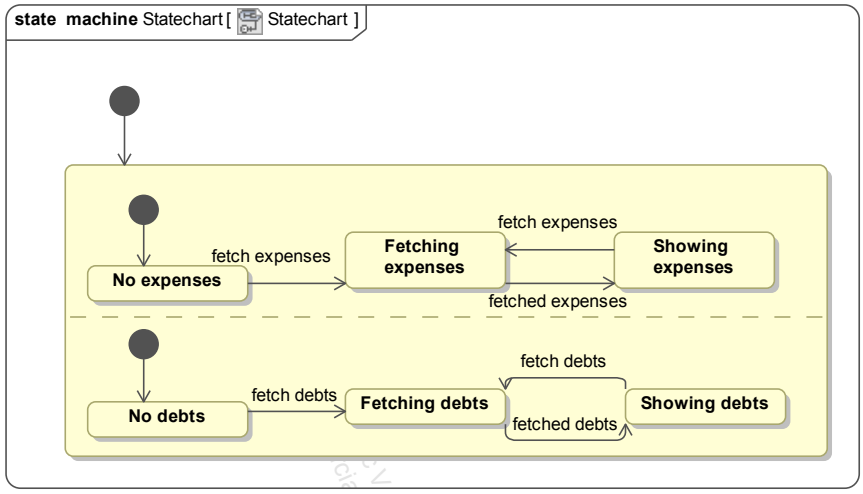

Figure 3.13: Example 3 modeled using a statechart

code alone". The graphical notation provides a great way for understanding how an application is structured. The graphical notation of statechart provides a really powerful way of making code virtualization. The notation and feature set offered by statechart are very much similar to Petri nets. According to Eshuis (2009) a statechart can in most cased be translated directly to a petri net and vice versa.

3.4 Summary

Several methods for dealing with the overall challenges of developing complex event-driven software have now been evaluated and analyzed. The suitable method for further investigation should therefore be selected.

- **Structure of applications** - To deal with the challenges of structuring an application, the use of component-based programming, gives much flexibility and provides a strong foundation. The theory behind this is further explained in Section 4.1.
- **Communication between concurrent parts** - Message passing provides a flexible model, which fits well into the component-based way of structuring an application. The theory behind this is further explained in Section 4.2.
- **Keeping track of state** - Keeping track of the state in an application is a general problem which many methods try to make easier. Most solutions are only dealing with a part of the problem. For instance the *Observer pattern* for sending state changes around. However, Petri nets and statecharts delivers a

complete method for managing state by introducing the concept of an abstract state. These concepts give the flexibility and expressiveness to model a complex applications. While both approach introduce powerful concepts, statechart has been selected as the solution for keeping track of the overall state in a application because of its intuitive graphical notation, and because the notation contains constructs useable for modeling GUI applications. The theory behind this is further explained in Section 4.3.

CHAPTER 4

Theory

The previous chapter concluded with a set of methods suitable for dealing with the problem of developing complex event-driven software for iOS. It was analyzed that components could solve the problem of dividing behavior in the application, pass messages for communication between the components and use statecharts to keep track of the state in a component. This chapter covers the theory behind these methods, before the next chapter uses them to design models for developing software.

4.1 Concurrent components

Operating in a multithreaded environment as explained in Section 2.4 enables the use of concurrency, where tasks can be executed simultaneously.

> *"In practice, if two, or more, tasks are concurrent, then they can run in parallel. This implies that there are no data dependencies among them."* (Vasileios, 2011, p. 1)

In other words, concurrent tasks cannot share any data between them directly. Combining the concept of concurrent tasks with the concepts of components as presented in Section 3.1.3, introduces the concept of a concurrent component.

> *"[A] concurrent component is a procedure with inputs and outputs. When invoked, the procedure creates a component instance."* (van Roy and Haridi, 2003, p. 371)

The concept of a concurrent component defines behavior running independently of other components. A concurrent component can be defined as a component having its own context to execute within. Data can flow into the component, using its input and flow out of the component, using its output. This forms the interface of the component. The interface is the public visible set of features offered by the component. This means that a component represents a part of a system that has an inside and an outside. Only the outside is visible to other components.

The concept of concurrent components is implemented in the language Erlang in the form of Erlang processes. As explained by van Roy and Haridi (2003, p. 395) a process in Erlang can run on the same machine or a different one. Each process in Erlang is given a unique thread in which execution takes place. This means that the

processes can run independently of each other.

Defining a system of concurrent components, executing independently of each other requires each component to have its own context to execute within.

4.2 Message passing

Message passing allows parties to communicate with each other by sending messages. When using message passing it is possible to model a message system. A message system allows for such communication as illustrated in Figure 4.1.

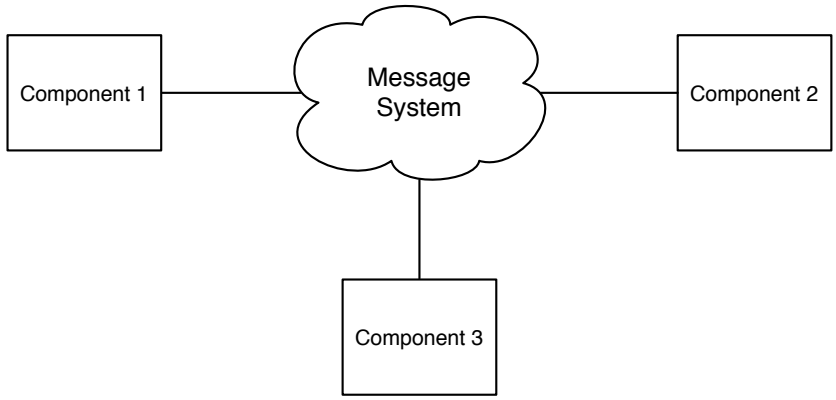

Figure 4.1: Messaging system

4.2.1 Synchronous

A synchronous message blocks the context of the caller until the message delivery has been completed. The definition of when delivery has been completed depends on the message system. It could refer to the fact that the message has been given to the receiver and it has been processed, resulting in a possible return value being provided. Note that a synchronous message implies that the sender is blocked until the message has been delivered.

Advantages Synchronous messaging provides a natural way of communication in the sense that the receiver is getting the messages as they are being sent, just like with a face-to-face conversation. This makes it simpler to provide error handling and having return values, since the receiver can reply to a message synchronously.

Disadvantages Sending synchronous messages is not flexible, since both the sender and the receiver need to be ready before the communication can begin.

4.2.2 Asynchronous

A messaging system can be asynchronous. In such a system the caller continues to run after sending the message without waiting for it to be delivered. This means that the sender of messages can still precede execution while the receiver has not processed the message yet.

Advantages As the sender does not need to wait for a message to be delivered, it is possible to overlap computation and communication. This lowers the latency in the communication, since the sender is not being blocked and thus other senders do not have to wait for it to become ready before sending messages to it.

Disadvantages A main disadvantage with asynchronous programming is that the programming model becomes more complex. This is especially the case when the sender expects a reply from the receiver, and the two parts end up having a conversation. This introduces the problem of keeping track of the conversations in order to understand incoming messages.

4.2.3 Actors

The actors of a message system are the parts that can send messages to each other. When using message passing as presented in Section 3.2.2, the actors do not share state, which means that there is no global memory in the system that can be accessed by the actors. When no memory is shared, the actors need another way of communicating with each other. A way of doing so is using the *actor model* as explained below. For the rest of this chapter the words *actor* and *component* are used interchangeable.

Actor model

The *actor model* was proposed in 1973 as a way of dealing with asynchronous messaging, using message passing. The idea is that an actor creates a message in response to an event occurred or some data that became available. Information is bundled into an object that is sent to another actor in the system. Each actor has a queue called a mailbox where messages are queued until the receiving actor is ready to process them. On top of the messaging system in the actor model, an actor also has the possibility of spawning new actors, as they are needed. When an actor wants to send a message to another actor in the system, it will address the actor directly, which means that an actor must know the address of any actor it wants to communicate with. This construct of the *actor model* is illustrated in Figure 4.2.

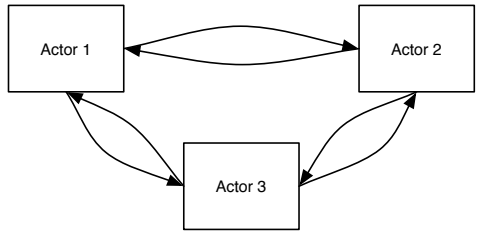

Figure 4.2: *Actor model.*

The main properties of the *actor model* are listed below.

Sending messages - An actor can send messages to the mailbox of another actor.

Receiving messages - An actor incapsulates behavior, which it executes when it receives a message.

Creating new actors - An actor can create new actors.

Asynchronous - Since all communication is asynchronous, an actor is never blocked waiting for a reply.

No order guarantee - The order in which messages are delivered is not guaranteed.

Delivery guarantee - A message is guaranteed delivery if the recipient actor exists. The message will be discarded if the actor does not exist.

No shared memory - The actors are not allowed to communicate using other mechanisms than using messages. This means no shared memory between the actors.

The *actor model* is in the foundation of languages like *Scala* and *Erlang*, which are languages optimized for highly concurrent programs.

Advantages The actors in the system have no shared memory. Concurrency challenges such as race-conditions and deadlocks are non-existing.

Disadvantages As discussed in Mackay (1997) a fundamental challenge with the actor model is the possible state changes caused because of actors spawning other actors. The behavior of a system becomes harder to analyze statically as much of the behavior in the system will be determined during runtime. This also means that it can be more difficult at compile time to determine the memory usage of the application statically, which for an system relying on the object model and method invocation often can be determined quite accurately at compile time. (Mackay, 1997).

4.2.4 Channel system

A way of building a message system is using channels. Instead of actors in the system communicating directly to each other, they are connected to a channel. The purpose of the channel is to move messages that are received in one end and deliver them to the other end. A bidirectional channel allows messages to be sent from both ends and a unidirectional only allows messages from one end. While it is technically possible to establish a bidirectional channel it is often not desirable, since it increases the complexity and in many cases communication between two actors should only be one-way. The components do not connect themselves to other components in the system. Instead the channel system relies on a higher level of abstraction that acts as a plumber and connects the components in the system with channels.

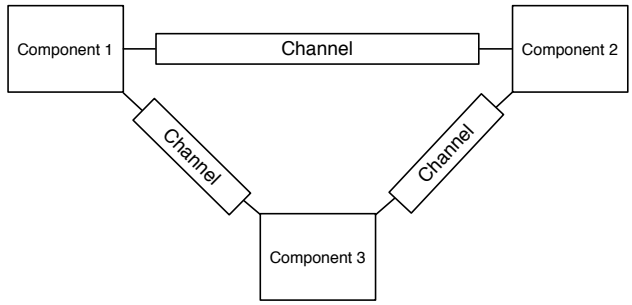

Figure 4.3: Channel system

An important aspect of the channel system is how the channels are set up between the components and what types of the channels are used. As explained in Hohpe and Woolf (2004) two basic types of channels exist as explained below.

Point-to-point channel

The point-to-point channel is used to send messages where there will be exactly one receiver. This means, that even though several receivers are connected to the channel, only one of them will receive the message. Further it means that messages are buffered in the channel if no components are ready to receive it yet. As illustrated in Figure 4.4 a component sends the messages `M1`, `M2` and `M3`. Because the channel is point-to-point each of the three receivers only get one of the messages.

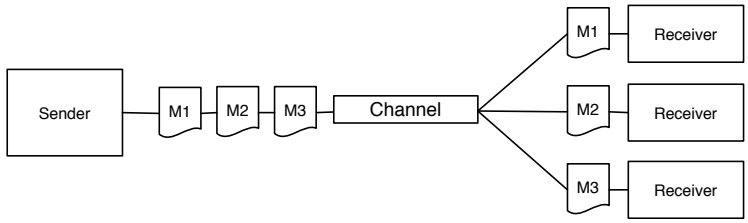

Figure 4.4: Point to point channel

A point-to-point channel is very similar to remote procedure call as seen in some languages, where a call should be executed exactly one time. Further it is useful for load balancing between several components and the channel will only deliver a message to a component that is ready to receive it.

Publish-subscribe channel

A publish-subscribe channel broadcasts an incoming message to all receivers. Further if the channel has no receiver the message will be discarded without being delivered to anybody. This is much like the *Observer pattern* as discussed in Section 3.3.3. The concept of a publish-subscribe channel is illustrated in Figure 4.5. All messages sent out on the channel are delivered to each receiver.

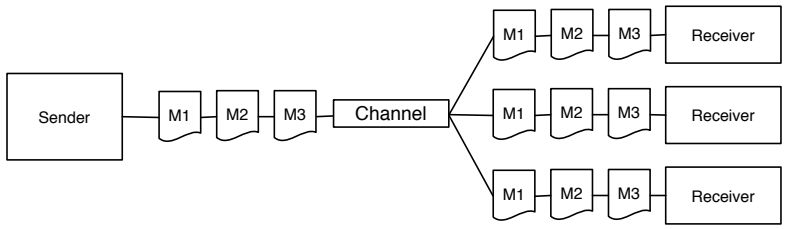

Figure 4.5: Publish subscribe channel

A publish-subscribe channel has many use cases. For instance is it useful for inspecting the communication on a channel as it only requires to add a component as a receiver on the channel, and all messages will be delivered to that component as well.

Advantages The channel system provides a very loosely coupled and flexible model. Components are running in their own context with no shared memory, which

lowers the complexity for modeling concurrency. As components do not spawn other components the effect of a single message is limited. Further the model provides the possibility for inspecting and debugging the message system.

Disadvantages It is not defined by the model to make the system very dynamic with components that are added during the execution of the system, this should be handled by other mechanisms.

4.2.5 Ports

Instead of connecting channels directly to the component, ports can be used. A port represents an opening between a component and the surrounding environment. Both the *actor model* and the *Channel System* can use the concept of ports. Figure 4.6 shows how ports can be used with the *Channel System*, where a port is put on the edge of a component and a channel is connected to the port instead of directly to the component. For the *actor model* an actor can act as the port of another actor.

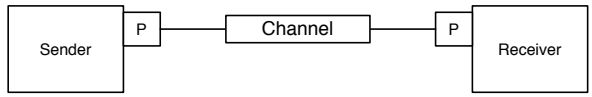

Figure 4.6: Ports

A port can contain the interface for the communication and thus act in a way for showing the surroundings which messages can be sent to a component. UML2 distinguishes between two kinds of ports. A service port, which is visible to the outside of a component and a non-service port which is only visible to the component itself. The non-service port allows the component to send messages to itself.

Advantages Using ports has the benefit that it is possible to define an interface visible to the outside. This makes it easier to correctly connect components that know how to communicate with each other.

Disadvantages Adding the need to specify ports on each component in order to use the *Channel System* introduces a bit of overhead compared to simply connecting the channels directly to the components.

4.2.6 Router

Communication between components can be modeled to go through a single actor in the system, called a router. The router is in charge of delivering messages to the right

receivers. The concept of a router in the *Channel System* is illustrated in Figure 4.7. It could also be used with the *actor model* where a special actor is the router in the system.

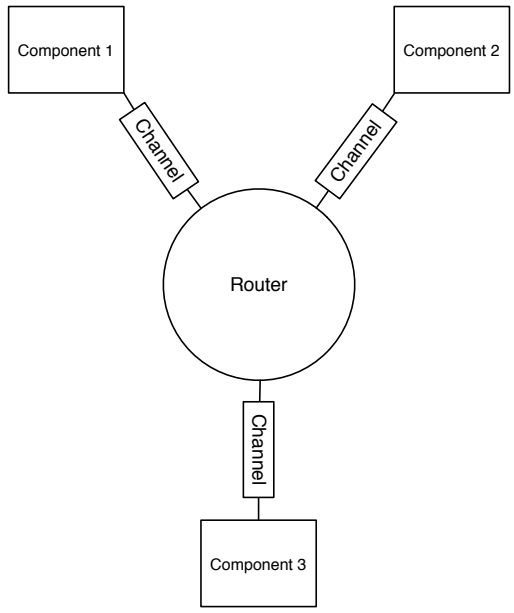

Figure 4.7: Message router

It is essential that the router is always ready to receive messages and thus it should queue incoming messages before routing them one by one.

Advantages The advantage of routing all messages through a single point is that inspections of the messaging system become easier. For instance, if the router can be set to log all messages passing through, which could be useful for debugging the system. The serial queue in the router brings a total ordering to the flow of messages in the system.

Disadvantages On the down side routing all messages through one point introduces a bottleneck for the communication system. It is essential for the system to work in practice with a high throughput, so that the messages queue in the router does not explode with messages that are waiting to be routed.

4.3 Statechart

As briefly presented in Section 3.3.9 a way of capturing the state is by using statecharts as presented by David Harel in 1986 (Harel, 1986). This section goes into the theory behind statecharts and its notation.

4.3.1 Transitions

In order for the statechart to change state a transition must occur. The concept is similar to the transitions as explained for the *finite state machine* in Section 3.3.6. A transition may occur as a reaction to an event. As illustrated in Figure 4.8, where a transition is performed from `State 1` to `State 2` when event `event1` occurs.

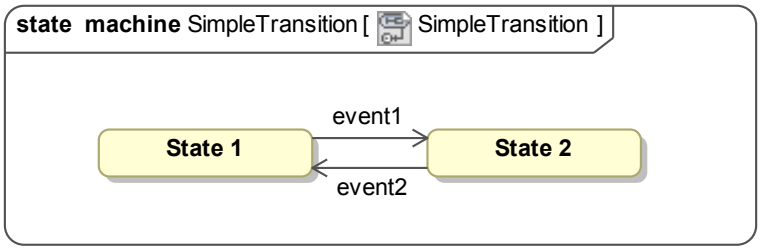

Figure 4.8: Simple transition

Self-transition

The possibility for a state to perform a transition to itself exists in the form of self-transitions as illustrated in Figure 4.9 where `event1` triggers a self-transition of `State 1`.

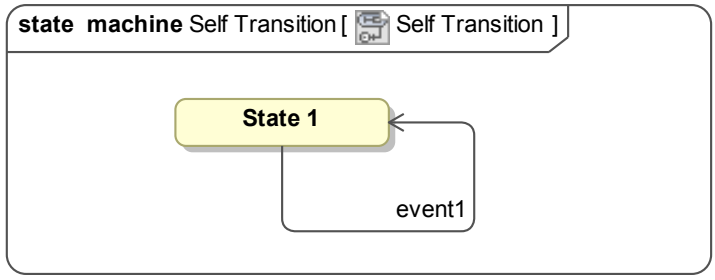

Figure 4.9: Self-transition

With self-transitions both the exit and later the enter action is called on the state.

Internal transition

The next type of transition is an internal transition. An internal transition is similar to a self-transition in the sense that it does not trigger a change of the active state configuration, however only the transition action is called when an internal transition occurs. The graphical notation of an internal transition is shown in Figure 4.10.

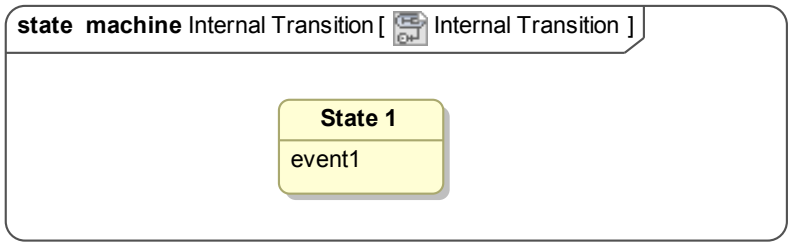

Figure 4.10: Internal transition

Initial transition

The state in which the statechart starts is determined by the initial transition, which is identified by a transition going from a solid circle to a state. This transition will be performed when the statechart is being started. In Figure 4.11 the initial transition takes the statechart into `State 1`.

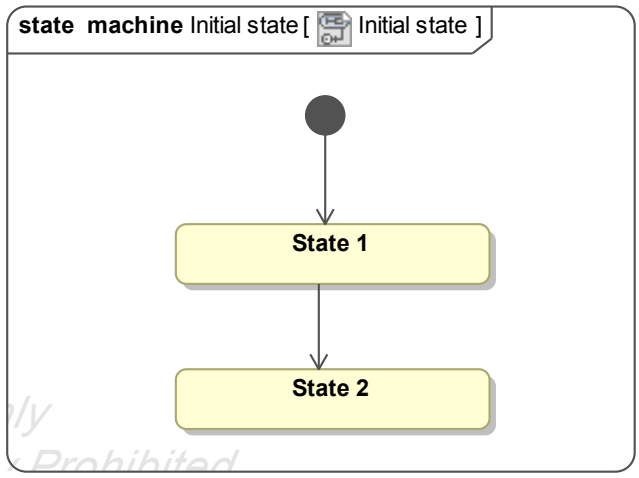

Figure 4.11: Initial transition

4.3.2 OR state

A statechart consists of states. An `OR state` is a type of state, which is similar to the states in the *finite state machine* as presented in Section 3.3.6, however the `OR state` introduces the concept of a hierarchy of states. An `OR state` that contains substates is known as a composite state. An `OR state` that does not contain substates is known as a leaf state. This is illustrated in Figure 4.12, where `State A` is a composite state with two substates `State A1` and `State A2`. When `State A` is active it can either be in `State A1` or `State A2`, but never both at the same time.

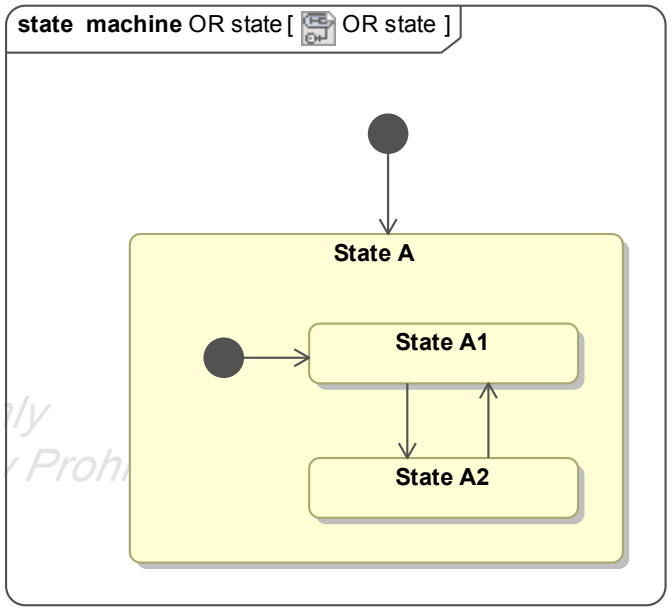

Figure 4.12: OR state

Notice how the concept of initial transition is used inside a composite state to specify the default state when entering the composite state.

4.3.3 AND state

Another kind of state is the `AND state`. An `AND state` contains one or more regions. Each region may consist of an `AND state` or an `OR state`. When an `AND state` is active all of the regions are active. Because of this property an `AND state` is also known as a orthogonal state. Figure 4.13 shows an `AND state` called `State A` with two

regions separated by a dotted line. Each region contains an `OR state` called `State A1` and `State A2`, that both contains substates. In the initial state configuration the states `State A`, `State A1`, `State A2`, `State A1 1` and `State A2 1` are all active. If `event1` occurs it will trigger two transitions. A transition from `State A1 1` to `State A1 2` and a transition from `State A2 1` to `State A2 2`. Performing a transition out of a region as done with `event4` results in leaving all of the regions.

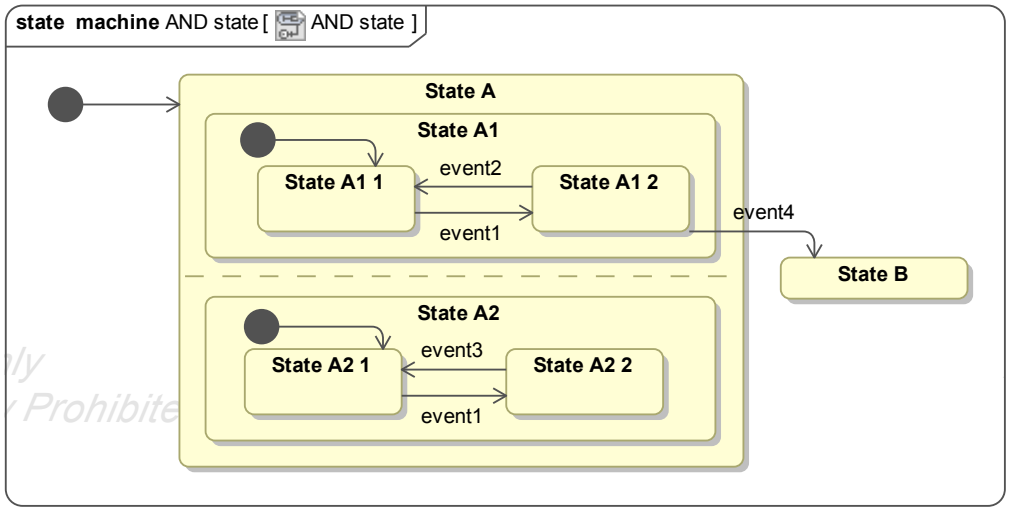

Figure 4.13: AND state

4.3.4 History connector

The initial transition as explained in Section 4.3.1 defines which substate of an `OR state` should be entered. However, sometimes the state has been entered previously and the last active substate should be entered instead. This can be done using the history connector. For instance consider Figure 4.14 with the active state being `State B`. Receiving `event3` makes a transition to the history connector of `State A`, which will automatically go to the last active substate instead of the initial transition. If no history exists, the initial transition will be performed instead.

4.3.5 Fork connector

The `AND state` as explained in Section 4.3.3 allows several regions to be active at the same time. A transition to a state in a region results in all of the regions becoming active. The default behavior is that the initial transition is used for each of the

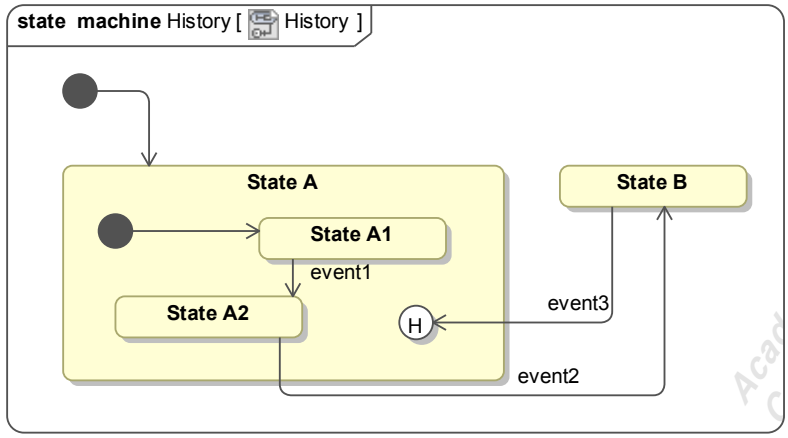

Figure 4.14: History connector

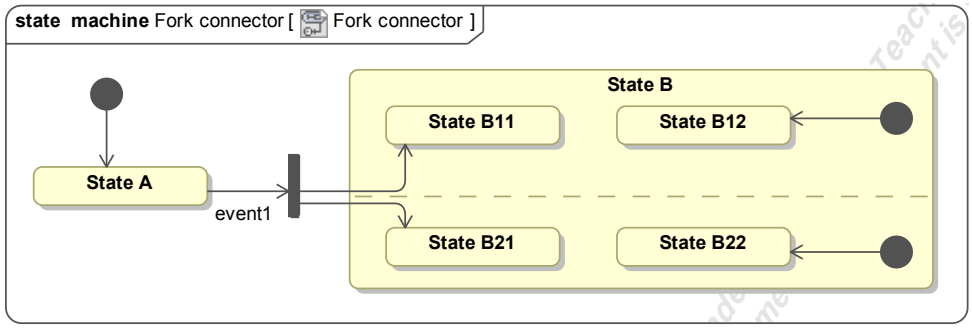

Figure 4.15: Fork connector

regions, except the one that caused the transition, however this is not always the desired behavior. The concept of a fork connector as defined by Harel and Kuger (2004, p. 15) makes it possible to define a transition from a single source state to multiple target states as shown in Figure 4.15, where a transition is made from `State A` to both `State B11` and `State B21` when `event1` occurs.

It is only supported when states are located in different `regions`. Using it to make a transition to two states in the same region is not supported.

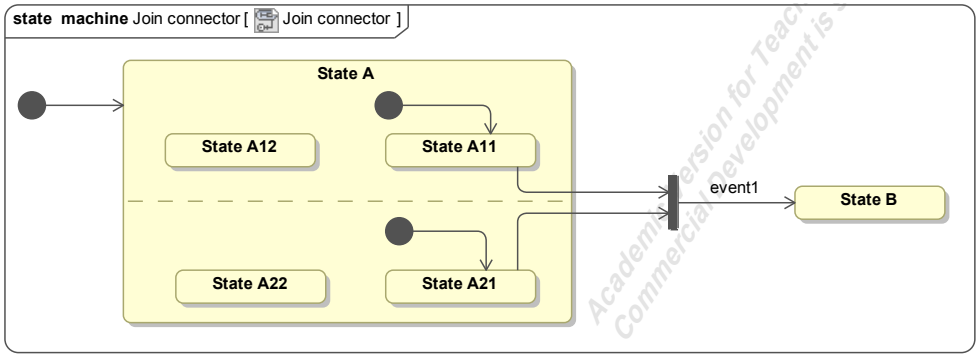

Figure 4.16: Join connector

4.3.6 Join connector

In a certain situation when making a transition from a state inside a region to a state outside of the region, the transition should only be made if a certain state is active in another region inside the same `AND state`. This can be modeled using a join connector as illustrated in Figure 4.16. If both `State A11` and `State A21` are active when `event1` occurs, then a transition to `State B` is performed.

As well as with the *Fork connector*, the `Join connector` should be used with caution in order not to model an invalid statechart.

4.3.7 Transition guards

So far a transition has been defined as a reaction to an event triggering a state change. Using guarded transitions it is possible to only perform the transition if a condition evaluates to be true.

> *"Guard conditions affect the behavior of a state machine by enabling actions or transitions only when they evaluate to TRUE and disabling them when they evaluate to FALSE."* (Samek, 2008, p. 64)

Moreover, it makes it possible to define transitions that automatically trigger a state transition when their conditions becomes true, without having an event defined on them. Figure 4.17 shows two transitions, each with only with a guard. Once the condition of the guard becomes true, the transition is performed. In the example `State2` is entered once the extended state variable `i` becomes larger than 2, and `State1` is entered when it comes less or equal to 2.

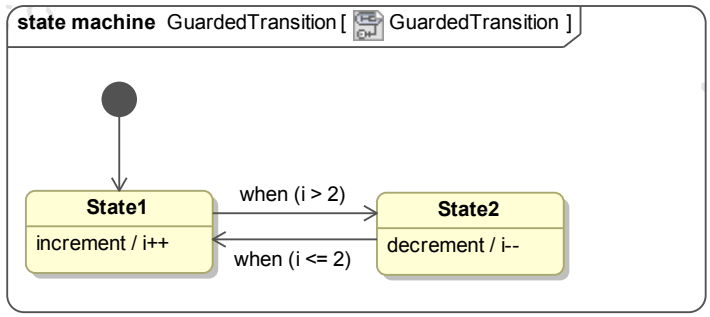

Figure 4.17: Guarded transition

4.3.8 Timeouts

Behavior that is determined by time can be modeled using timeouts(Horrocks, 1999, p. 73). By a timeout a transition can automatically be triggered once the timer runs out. This can be useful when modeling a user interface, where changes to the UI are being performed according to time. Figure 4.18 illustrates how a timeout can be put on a transition. In the example a transition from `State1` to `State2` is automatically performed after 10 seconds.

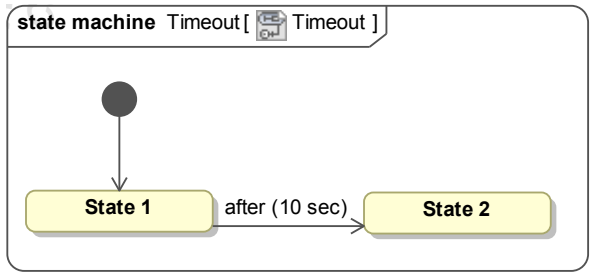

Figure 4.18: Timeout

Deferred event

Modeling a statechart that is ready to handle any incoming event at any time is a difficult task. For this reason `UML` has defined the concept of event deferral. As explained by Samek (2008, p. 220) *"The solution is to defer the new request and handle it at a more convenient time"*. The mechanism makes it possible for a state to have a list of events that it defers. Whenever a transition occurs in the statechart, all deferred events are reposted to the statechart engine, in order to determine if the new state configuration is able to handle the event. The concept is illustrated in

Figure 4.19, where state `Working` defers the event `work`. When a transition is made to state `Waiting`, the event is recalled and a transition to state `Working` is performed.

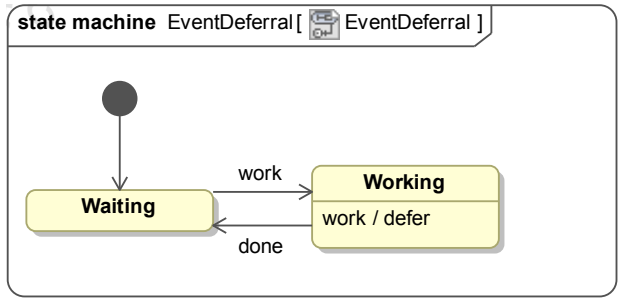

Figure 4.19: Event deferral

Submachines

In order to support reusability within a statechart `UML` extends the original statechart notation with the concept of submachines. As explained by Arlow and Neustadt (2002, p. 343) *"Submachine states are a way of simplifying statecharts by allowing you to refer to a submachine that has been fully defined on a different diagram"*. In other words, submachines make it possible to embed a statechart as a part of another statechart. This makes it possible to define reusable statecharts that can be embedded as submachines inside other statecharts. The concept is illustrated in Figure 4.20, where the statechart on the right is used as a submachine inside the statechart on the left. The execution of the statechart is identical whenever submachines are used or not.

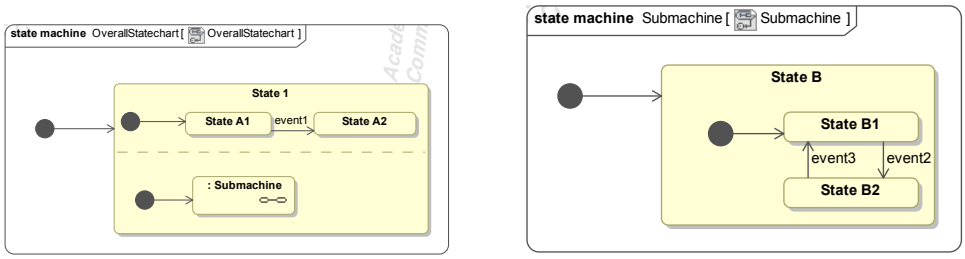

Figure 4.20: Example of a submachine (right) embedded inside a statechart (left).

4.3.9 Transition execution

A traditional finite state machine with a flat structure has a simple execution, however having a hierarchy of states and orthogonality introduces a much more complex execution.

State configuration

An important concept of a statechart is the state configuration which represents the currently active states. In a flat structure as presented with the *finite state machine* only a single state can be active at a time, however with composite and orthogonal states, the statechart consists of a configuration of states that are active.

Responding to events

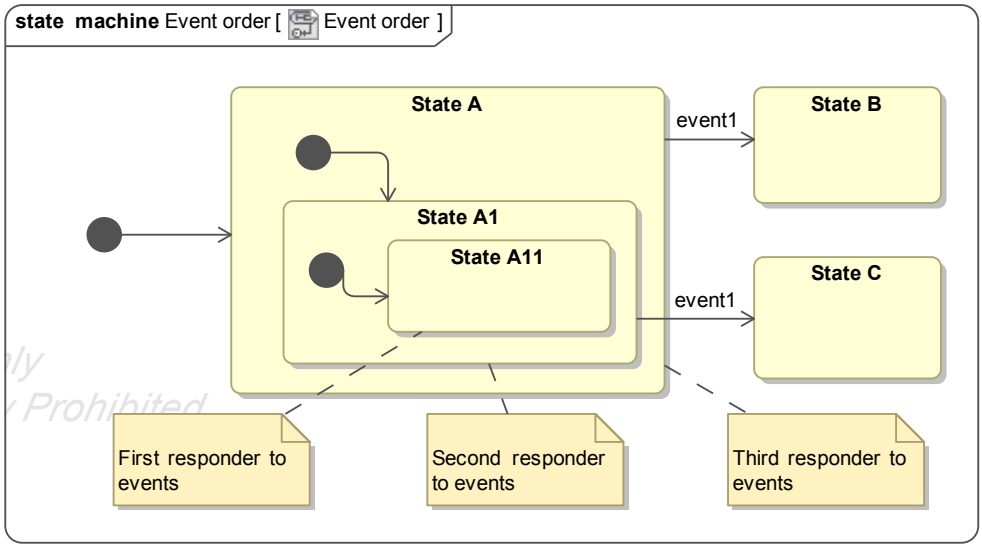

Figure 4.21: The responding order for events.

As explained in Section 4.3.1 states have transitions that respond to events. Several states in the statechart may respond to the same event. Having a hierarchy of `OR states` introduces the possibility that several states, currently a part of the active state configuration, respond to the same event. This is shown in Figure 4.21. For this reason an order, for how active states respond to an event, must be defined. The rule is that the lowest composite state that is a part of the current state configuration is the first responder. If the state does not respond to the event the superstate will

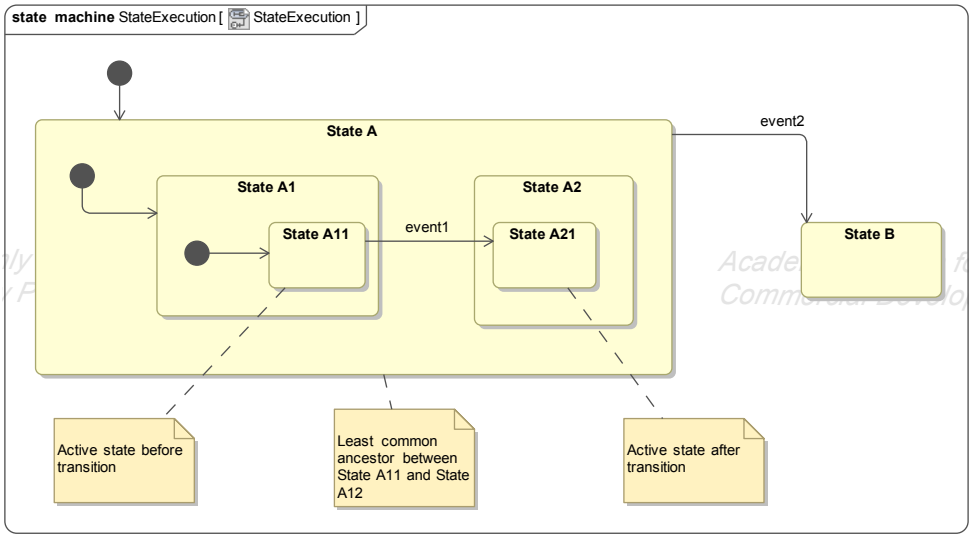

Figure 4.22: Finding lowest common ancestor before transition.

have a chance to respond. This continues until the root state is reached. If no state responds to an event it is discarded. For Figure 4.21, where the system is in the initial state configuration it is `State A11` which is the first responder for all events. Then `State A1` and finally `State A`. Consider the case where `event1` is being triggered. Since `State A11` does not respond to `event1` it will be forward to `State A1` and a transition to `State C` will be performed.

Actions

When performing a transition certain actions are performed. All states that are being exited get the chance to fire an action, further the transition can fire an action and finally all the states that are being entered can fire an action. The order of the actions is the following:

1. Exit actions
2. Transition actions
3. Enter actions

Lowest common ancestor

In order to determine which states that should be exited and which that should be entered it is necessary to find the lowest common ancestor(LCA) of the source and

target state of the transition. The LCA is defined as the lowest composite state that is superstate of both the source and target of the transition. For instance consider Figure 4.22 with a transition from `State A11` to `State A21`. The LCA is `State A` since it is the lowest composite state that is a superstate of both `State A11` and `State A21`.

During a transition all the states from the source state up to and not including the LCA need to be exited and all the states from and not including the LCA to the target state need to be entered. For Figure 4.22 this would result in `State A11` and `State A1` being exited and then `State A2` and `State A21` being entered.

A special case exists where a composite state causes a transition to a substate as illustrated in Figure 4.23, where `State A` contains a transition to `State A11` on `event1`. The rule is that all the states up to but not including the composite state that handles the event must be exited, before entering down again to a leaf state again. If `State A11` is active, both `State A11` and `State A1` are exited. Then the action of the transition is called before `State A1` and `State A11` are entered again.

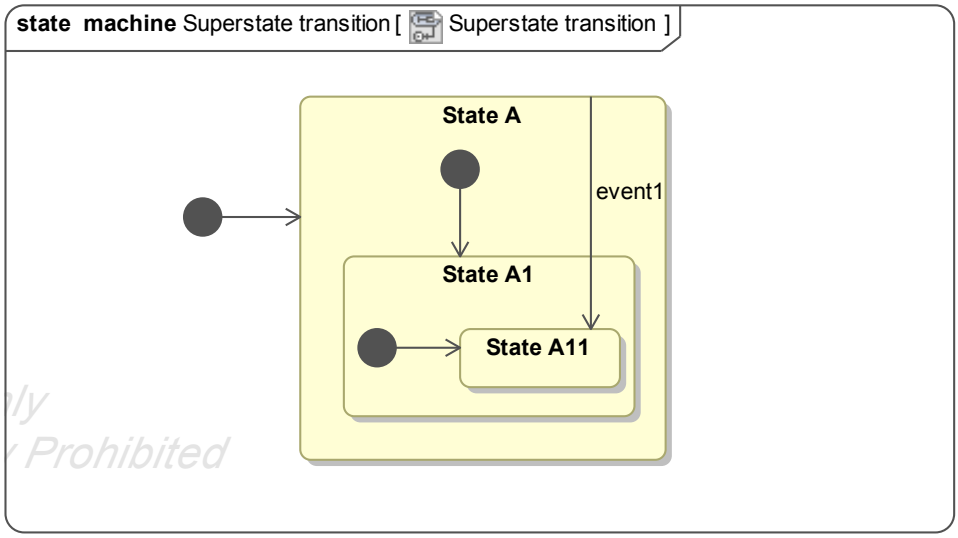

Figure 4.23: Transition from composite state to substate.

4.4 Summary

The theory behind the methods has been covered by looking at concurrent components, message passing and the statechart notation. Based on this knowledge the

next chapter is going to design a solution for modeling applications.

CHAPTER 5

Design

Based on top of the theory presented in the previous chapter, this chapter covers the design decisions taken for the framework to be implemented. This includes designing the concept of a component and in particular choosing the right execution context for a component. Further the messaging system is designed to make communication between components possible. The design of the statechart engine is covered, which provides the ability to keep track of the state in a component. Lastly based on the designed system, debugging and inspection functionalities offered are discussed.

5.1 System of components

As analyzed in Section 3.1 dividing an application into separate components allows for separation of behavior in an application. This is illustrated in Figure 5.1. The introduction of components gives a higher level of abstraction compared to seeing an application as a collection of objects and because of that it allows for more control over the flow in the system. This includes both the flow inside of a component as discussed further in Section 5.3 and the flow between components as discussed in Section 5.2. The requirements for the component system are first established before defining the context of execution for a component.

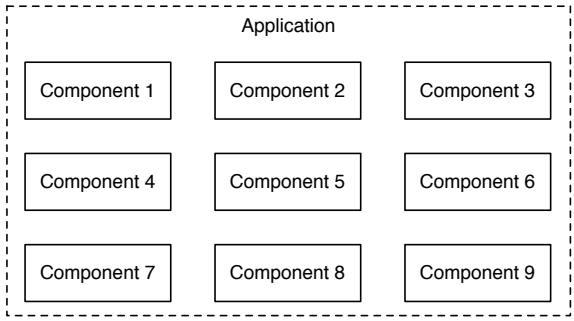

Figure 5.1: Application constructed of components

5.1.1 Requirements for component system

In order to evaluate approaches and make design decisions for the component system, the following requirements have been established.

- **Low memory footprint** - As the component is going to be a substantial part of an application, the components should not put a heavy memory footprint on the system.
- **Independency between components** - The execution of each component in the system should be independent of other components. The load and activity of one component should not influence other components.

5.1.2 Component execution context

An important property of components is that they run independently of each other. Having this property eases the challenges of modeling concurrency, since a single component can safely perform tasks, without having to worry about other parts of an application. For instance this means that if the execution of one component is blocked, other components will not suffer from this. Two overall approaches have been considered in order to establish this property and they are discussed below.

Unique thread per component

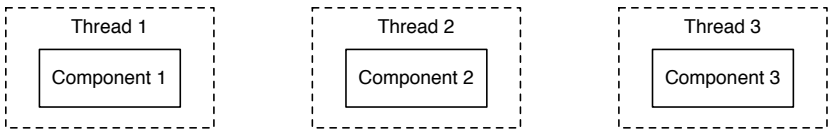

Figure 5.2: Using a thread for each component in the system.

One solution to allow each component to have its own context is to instantiate a thread for each component in the system. By allowing each component to manage its own unique thread, the components run independently of each other, since blocking one thread does not affect other threads in the system. This idea is illustrated in Figure 5.2.

Using this approach the number of threads in the system becomes equal to the number of components. Since one of the requirements is to maintain a low memory footprint, the memory usage of having many threads is investigated further in order to validate the solution. Apple (2014b) approximates that it requires around 512 KB of memory on the stack to store a running thread. Depending on the number of components running in an application this could lead to high memory usage. For instance using

this approximation to run 100 components would take up 50 mb. It is important to notice that this memory is only for keeping the threads alive. However, as stated by Apple (2014b) "the actual pages associated with that memory are not created until they are needed". This means that when a thread is allocated, the stack size is low and increases, as more memory is needed on the stack by that thread. In other words, the actual memory consumption depends on the application it is used in. As the amount of memory is limited when running on a mobile device as discussed in Section 1.3.2, a complex application with many components may end up using too much memory and thus getting terminated by the operating system. The documentation for iOS does not state the amount of memory an application may use before being terminated. For this reason a test has been performed as presented in Example 4.

Example 4 *In order to determine how the operating system reacts on iOS when applications consume much memory a test has been performed on several generation of devices. The test has been performed in a setup using many threads in order to simulate the situation of having a thread for each component. In order to have a realistic stack size for each thread, the example allocates around 512 KB on the stack for each thread created. In the example project instances of the class* `Thread` *is created. This is a subclass of* `NSThread`*, which when instantiated creates a thread from the perspective of the operating system. The* `ThreadMemoryTest` *project is included in the resource as explained further in Appendix A.*

Profiling the project as explained in Example 4 with 100 instances of `Thread` results in a memory usage of around 54 MB, which corresponds to 558 KB for every component. As also explained further in Section 7.2.1 the operating system sends a memory warning when an application is using too much memory in order to ask it to release memory. When the operating system needs memory for other applications it will automatically terminate the applications currently consuming the most memory. Because of that it is desirable to use as little memory as possible. In Table 5.3 different iOS devices have been tested with a variant number of instances of `Thread` objects. For each test, the memory consumption has been recorded together with a status, as shown in Table 5.3. The status indicates whenever the memory usage is so high that the application crashed received a warning or ran normally. The status `Warning` means that the application received one or more memory warnings from the operating system, The status `Crashed` means that the application was terminated during execution because of high memory usage and finally the status `Normal` means that the application kept memory usage within the limits and execution were normal.

# Threads	Model	Memory usage	Memory status
100	iPhone 5S	55 MB of 1024 MB	Normal
100	iPhone 5	54 MB of 1024 MB	Normal
100	iPhone 4	54 MB of 512 MB	Normal
100	iPod Touch 4 gen.	58 MB of 256 MB	Normal
250	iPhone 5S	132 MB of 1024 MB	Normal
250	iPhone 5	132 MB of 1024 MB	Normal
250	iPhone 4	132 MB of 512 MB	Normal
250	iPod Touch 4 gen.	140 MB of 256 MB	Warning
500	iPhone 5S	262 MB of 1024 MB	Normal
500	iPhone 5	263 MB of 1024 MB	Normal
500	iPhone 4	262 MB of 512 MB	Warning
500	iPod Touch 4 gen.	Out of memory	Crashed
750	iPhone 5S	394 MB of 1024 MB	Normal
750	iPhone 5	393 MB of 1024 MB	Normal
750	iPhone 4	Out of memory	Crashed
750	iPod Touch 4 gen.	Out of memory	Crashed
1000	iPhone 5S	521 MB of 1024 MB	Warning
1000	iPhone 5	523 MB of 1024 MB	Warning
1000	iPhone 4	Out of memory	Crashed
1000	iPod Touch 4 gen.	Out of memory	Crashed

Table 5.3: Threads memory usage on different devices.

As shown in Table 5.3 it seems that a memory warning is issued if around half of the memory available on the device is used by an application. An older device such as iPod Touch 4. gen. with only 256 MB of ram receives memory warnings with 250 threads running. It crashes if this amount is increased to 500. Running on some of the newer models such as iPhone 5(S) allows to more than 750 threads before running into memory issues.

It is important to note that the memory usage from Table 5.3 only includes the memory needed to have the threads running. Using a thread for each component would require way more memory as the data and state of the application would take up much memory too. As a low memory usage is crucial for keeping the application alive using many threads should be avoided and thus the solution with a thread for each component is not recommended.

Shared thread pool

Instead of allocating a dedicated thread for each component as discussed in the previous section another approach is to have a pool of threads. This pool is shared by

all components in the system. A way of modeling this is by having a serial queue, which a component has to go through in order to get access to a thread from the pool. The use of a serial queue where components wait for a queue to be available is illustrated in Figure 5.4. Once a component is done using the thread it is automatically released for other components to use. As also explained by Haller and Odersky (2007, p. 11), the size of the thread pool needs to be flexible. As more components need to use a thread, more threads are created. Further as threads are left unused in the pool, they are deallocated again. The number of threads running does not need to be equal to the number of components in the system. A component only acquires a thread when it has a task that needs to be executed, and as soon as it finishes, the thread is released again. If the number of available threads in the pool is zero and a new component needs to use a thread, a new thread will be instantiated, in order for a component not to wait for another component to release its thread.

With the shared thread pool the threads are only taking up memory when they are needed. Since a system can consist of many components that only execute code when certain events occur, this calls for a much more efficient memory usage. Further, since new threads are spawned when needed, the components are still independent of each other. For these reasons it has been decided that the components execution context should be based on a shared pool of threads instead of a unique thread per component.

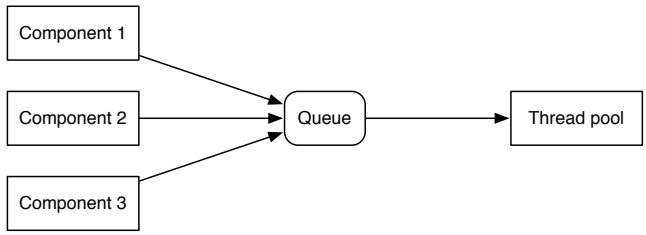

Figure 5.4: Using a pool of threads to share between components.

5.2 Communication system

So far a way of structuring an application into components has been defined. However, this introduces the requirement to allow communication between components. Such a communication system defines the way the components communicate with each other.

5.2.1 Requirements for communication system

The following requirements have been established for the communication system.

- **Loosely coupled** - By designing the system to be loosely coupled, components can easily be exchanged. For instance during testing where certain components can be exchanged with mocks[2] to make it easier to test the behavior of specific components.
- **Deterministic** - In order to understand and debug the system an important property is determinism. In the sense of the messaging system this means that the order in which the messages are delivered is deterministic. This creates an order in which the messages have been delivered. Having such an order helps understanding the behavior of the system.
- **Inspection** - Designing the message system so it can be monitored and inspected live enables many features such as seeing the messages that are being sent in the system at a given time.
- **Integration** - The message system should fit in with existing frameworks and system events.

5.2.2 Message

As discussed in Section 3.2.2, a message passing system is built around the idea of messages. A message is used to communicate between components. Each message should have the following properties.

- **Identifier** - Unique identifier that can be used to identify the message.
- **Timestamp** - Date for when the message was sent.
- **Body** - The content included in the message.
- **Message Type** - Statically defined type that can be used by the receiver to identify the kind of message that was received.

Defining a type for a message is important, since otherwise the receiver would not be able to determine how to react to the message.

5.2.3 Synchronous vs. Asynchronous

As presented in Section 4.2.1 and Section 4.2.2, an essential design decision for the messaging system is if the communication should be synchronous or asynchronous. Both of which have advantages and disadvantages.

[2] A mock is an implementation of a class that can be setup to have a specific behavior, which is useful during situations where the class is expected to act in a specific manner.

As mentioned in Section 1.1 one of the challenges when developing iOS applications is to keep the user interface interactive at all time. Traditional method-invocation in Objective-C uses a synchronous method passing system. While this makes sense for most cases, certain operations need to be handled asynchronously in order not to block the main thread as discussed in Section 2.4. As presented in Section 5.1.2, a component is running in its own context. Sending a synchronous message to another component would, as presented in Section 4.2.1, require the sending component to wait until the receiving component is ready to receive the message. However, this would violate the requirement of having independent components, since the execution of the sending component depends on the execution of the receiving component. This dependency can be removed by simply designing the whole messaging system between components to be asynchronous. Further with all communication being asynchronous, handling events from the system becomes much easier, since they are also asynchronous. In fact, in order to integrate the components in the event-driven environment, events can be captured and send as asynchronous messages to the components for them to react upon.

In order to meet the requirements it has been decided that communication between components in the system should be asynchronous only, and the designed communication system should therefore be built around this decision.

5.2.4 Actor model vs. Channel System

As presented in Section 4.2.3, the *actor model* is one way of doing an asynchronous messaging model and as presented in Section 4.2.4 the *channel system* is another way. The two solutions will be evaluated according to the requirements established in Section 5.2.1.

The *actor model* divides an application into independent parts, which share no memory. This maps directly to the components used in the design. While the actors are running independently of each other, the actors still have a coupling to each other. This comes from the fact that in order to send a message a sending actor needs to know the address of the receiving actor. Hence the actors have a close relation to each other, as they know exactly whom they are sending messages to. Because of this the *actor model* does not allow actors to be exchanged easily. The *channel system* takes a different approach. By the introduction of a channel concept, the components do not know each other. Instead they share a channel, which they can send messages on and retrieve messages from. This allows for a much more loosely coupled system. Since each component only knows a channel it can send messages on and not other components that are connected to the same channel. This means that a component can easier be exchanged without affecting other components.

The *actor model* defines how actors can spawn other actors in the system. This means that the actors themselves take part in creating the components in the sys-

tem. With the *channel system* in order to archieve a more decoupled system, the job of creating and connecting the components can be done outside of the components. This allows the individual components to operate unaware of structural changes in the system.

Based on the fact that the *channel system* provides a very loosely coupled and flexible model it has been decided to use that as the foundation for the messaging system to be implemented instead of the more tightly coupled *actor model*.

5.2.5 Routing vs. direct communication

Having a system of components that can communicate with each other using channels introduces the problem of how these channels should be connected between the components and where the responsibility for delivering messages should be put. One way of doing it is to let the channels be in charge of delivering the messages to the receiving components. This gives a direct communication between the components. Another approach it to use a router that handles this as explained in Section 4.2.6. Using a router the messages are sent to a router, which directs them to the correct receivers.

An advantage of having all traffic going through a single point is that a complete ordering of the delivery of the messages can be recorded, by having all messages go through a queue in the router and timestamp them as they are being processed. This means that the order in which the messages have been delivered can be logged and used for debugging. It gives a very deterministic messaging system. Direct communication between components only allows making partial order, since some messages may be delivered concurrently. This can be done using algorithms such as `Lamport timestamps`(Krzyzanowski, 2006). Applying such algorithms into the messaging system increases the complexity, but on the other hand having a central router where all traffic passes introduces a bottleneck that may influence the performance of the system. If the router is unable to process the messages, as they are added to the queue, larger latency is being put on each message.

It was at first decided to design the system around a direct communication model, because of the simplicity of the model, however as discussed further in the implementation in Section 6.2.2 it was decided to extend with a central router, changing the design to a routing approach, where a router took over the responsibility of delivering messages. This design change allows better debugging and inspection capabilities.

5.2.6 Communication protocol

As explained in Section 5.2.4 each component contains channels as a way of sending and receiving messages. However, since a component does not know to whom it is sending messages, it is also unaware about which kinds of messages it can send. Sim-

ilarly since a component does not know who is sending messages to it, it is unaware of which kinds of messages it might receive. For this reason the interfaces going into a component must define the messages a component can receive and similarly the interfaces going out of a component should define the messages a component can send. This can be done by having a protocol, which defines the messages a component understands. One way of doing this is to have a protocol on each channel as illustrated in Figure 5.5, where a message of each of the types `M1`, `M2` and `M3` is sent on the channel by the sender. The protocol defined on the channel defines these three kinds of messages, and hence they are all valid to be sent on the channel and since the channel is a *publish-subscribe* channel as explained in Section 4.2.4 they are all delivered to the receiver. However consider the case where only `Receiver 1` understands the message of type `M1`, and only `Receiver 2` understands the message of type `M2` and finally only `Receiver 3` understands the message of type `M3`. This cannot be modeled using a single channel and instead it is required to have a unique channel between the sender and each of the three receivers. However, having to create a unique channel between every component in the system that needs to communicate makes the system less flexible.

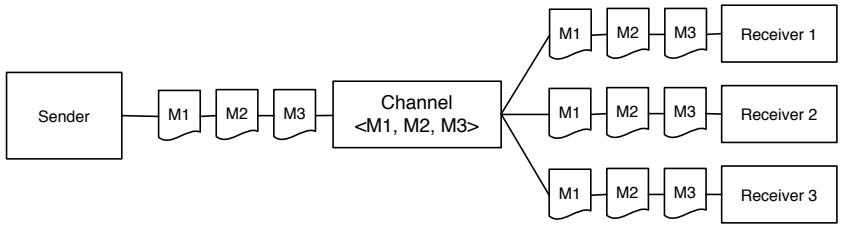

Figure 5.5: Protocol defined on channel.

Alternatively, in order to deal with this problem, the concept of ports can be used as introduced in Section 4.2.5. By placing a port in front of a component and defining the protocol on the port instead of the shared channel, it is possible to share a channel between several components, while each component only receives messages it understands. This is illustrated in Figure 5.6, where each port defines a protocol. The output port of the sending component includes all three message-types in its protocol, while the input port of each component only specifies the kind of message that the component understands. This guarantees that a component only receives messages defined on one of its input ports and only sends messages defined on one of its output ports.

Three basic kinds of ports are defined having different responsibilities. An *input*

port is used for receiving messages, an *output port* is used for sending messages and finally an *event port* is used for internal messages inside a component. Additional two special kinds exist called *input proxy port* and *output proxy port*. A input proxy port is used to receive messages from a channel on behalf of another port. As messages are received they are forwarded to another port. Similarly a proxy output port can be used to send messages to a channel on behalf of another port. Messages received by another port are forwarded and sent out of the proxy port.

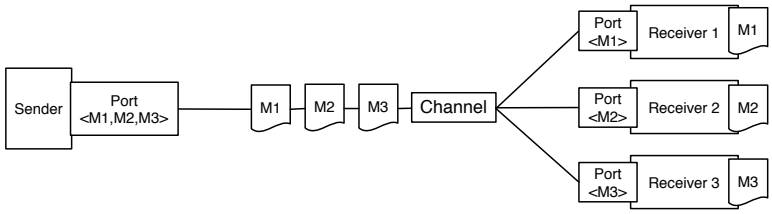

Figure 5.6: Protocol defined on ports.

5.2.7 Network bridging messages

The loosely coupled messaging system allows for designing very flexible applications. While the main purpose of the messaging system is to be used for sending messages between components running in the application, it is also often useful to communicate with other devices on the local network or over the internet. Since a component is unaware of how the channels in the application have been connected, and since all communication is happening asynchronously, it is possible to create a network bridge where each message is packed, then sent over the network and afterwards unpacked, before being delivered to a component running on another device than the sending component. This process is illustrated in Figure 5.7, where a component running in application on `Device A` sends a message on one of its output ports. The port is connected to a channel where a `Bridge Component` is a receiver. The `Bridge Component` is a component that can pack and unpack messages sent over the network. When the `Bridge Component` on `Device A` receives a message on its input port it will pack the message in a suitable format and send it over the network. As messages are received by `Device B`, they are unpacked and sent on the output channel. The receiving component gets a message, it is unaware that it actually originated on another device. It works as if the two components were connected using a channel on the same device.

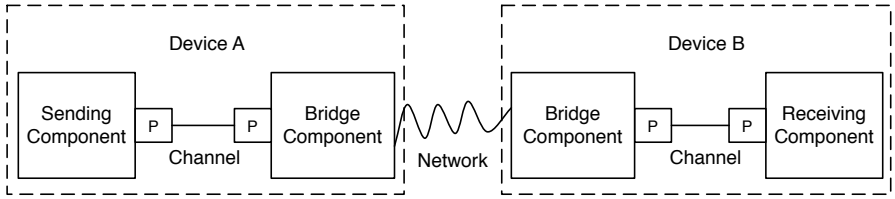

Figure 5.7: Network bridge

In order to send messages over the network a simple protocol for the endpoints to use has been defined that allows sending objects over a socket connection. An object can be seen as a chunk of data that needs to be transmitted. The protocol makes the separation between a header and a body as illustrated in Figure 5.8. This is similar to other communication protocols like `Hypertext Transfer Protocol`(HTTP). The header part is of a fixed size. It contains information about the type of data being transmitted and the length of the body. The length of the body depends on the objects that need to be transmitted.

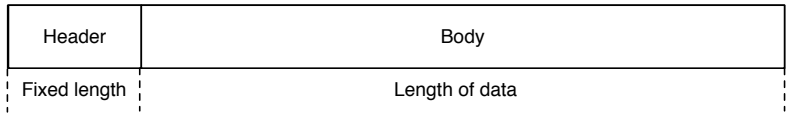

Figure 5.8: Header and body for transmitting messages over network.

By considering a stream of objects being sent as illustrated in Figure 5.9, the receiver can retrieve the body part of each message and decode it into an object again, since the header contains the number of bytes needed to be read to retrieve the whole body and the size of the header is fixed and known by the receiver.

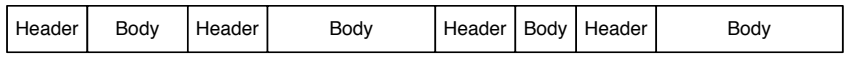

Figure 5.9: Stream of messages send.

In Section 5.4 it is discussed further, how the capabilities of the message network bridge can be used in designing tools essential for building asynchronous software.

5.3 Capturing state in components

In order to capture the state in each component, statecharts are used as analyzed in Section 3.3.9 and with the notation as presented in Section 4.3.

5.3.1 Requirements for statecharts

In order to take the definition of statecharts and apply them in a practical setup for developing iOS applications, many design decisions need to be considered. The requirements that the model of a statechart must satisfy are listed below.

- **Hierarchy** - The model must support modeling a hierarchy using `OR states`.
- **Orthogonality** - The model must include the power of orthogonality offered by `AND states`.
- **Extendible** - The model should be easily extendible with functionality from the statechart notation such as deferred events, merge and forking connecters as described in Section 4.3.
- **Reuse of logic** - The design should allow logic from a statechart to be reused in another statechart.

5.3.2 Modeling statecharts

An important part of the design of statecharts is how they are modeled. The model is two folded. On one hand it covers the application specific statechart that is being designed by the developer of an application. On the other hand it is the statechart engine that is able to execute these statecharts. Both the model of the engine and the statecharts should take the requirements of the statechart model into account.

First a solution for modeling statecharts will be considered before designing a model for a statechart engine that can execute statecharts according to the rules specified in Section 4.3.9. Three models for constructing a statechart are considered in the following sections.

Event-action table

A way of modeling a simple statechart is using one big table. The simple *finite-state machines* discussed in Section 3.3.6 can be modeled by a state transition table (Wagner, 2006, p. 4) where for each state a mapping is made to the next state for each event the state responds to. However, because of the hierarchy of a statechart a more

complex table is needed. As presented by Horrocks (1999, p. 82) an event-action table can be used. This table contains information about the transitions in the statechart and the actions performed, while taking the hierarchy into account. Consider the simple example of a statechart allowing to first fetch and then parse data from the internet as shown in Figure 5.10 with four states. This simple statechart does not satisfy the requirements established in Section 5.3.1, however it is used for simplicity in order to make an early validation for modeling the statechart.

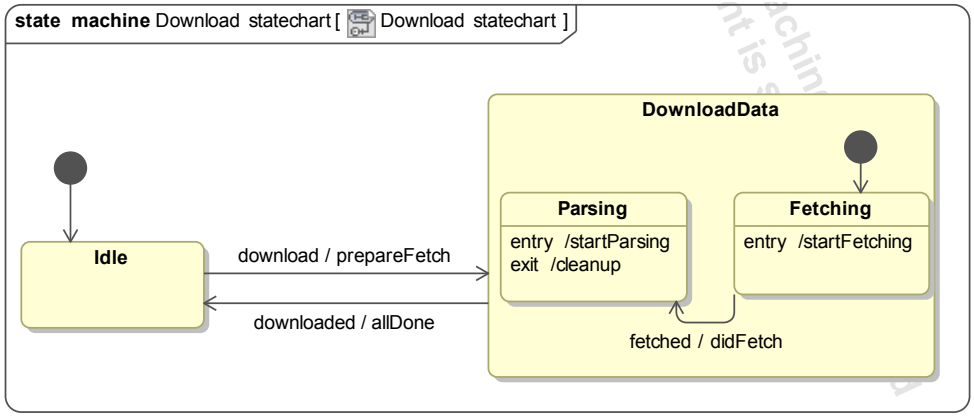

Figure 5.10: Simple statechart that downloads and parses data.

The three transitions could be modeled using an event-action table as shown in Table 5.11. The table shows for each event which current state they can be triggered from, which action that the transition is firing and lastly the next states that the statechart should make a transition to. For instance, when being in state `DownloadData` and the substate `Fetching`, the event `fetched` would take the system to state `DownloadData` and the substate `Parsing` and trigger the action `didFetch`.

Current state					Event	Action	Next state				
Root	Idle	Download-Data	Fetching	Parsing			Root	Idle	Download-Data	Fetching	Parsing
Idle					download	prepareFetch	DownloadData	Fetching			
DownloadData					fetched	didFetch	DownloadData	Parsing			
DownloadData			Fetching, Parsing		park selected	allDone	Idle				

Table 5.11: Event-action table of the statechart from Figure 5.10

For a simple statechart as the one from Figure 5.10 the event-action table gives a simple overview of the system. The number of rows in the table is equal to the

number of transitions in the system, the columns are equal to two times the number of states plus two columns for the event and action respectively. This means that as the complexity of an application grows the overview in the table quickly disappears. Further as noted in Horrocks (1999, p. 82) the table does not easily allow several people to work on the same statechart at once, without being faced with conflicts when merging changes in the table together. For that reason the use of a table is too simple for modeling something complex and a better model is needed.

Single function with cases

Instead of using a table it is possible to provide a single function that is able to handle an event, fire the actions and return the next active state. This code is shown in Listing 5.1 for the statechart from Figure 5.10. In this simple approach a state and an event are represented by an `int`, but any other simple data type could have been used as well. The function expects three reversed events to be provided for being able to handle initial, entry and exit transitions. Since all the logic is put into a single function, the hierarchy must be included in the function. This is done by having a switch on the event type that is provided as an argument to the function. Further, the current state is passed as an argument to the function and for each case of event, the current state is checked to see if it responds to the event. If so, the next state is returned. If the current state did not respond to the event, zero is returned. Further if the transition defines an action, it is fired just before the next state is returned.

Listing 5.1: Structuring the statechart using a switch cases.

```
1  //states
2  int idle = 1;
3  int downloadData = 2;
4  int fetching = 3;
5  int parsing = 4;
6
7  //events
8  int initial = 1;
9  int entry = 2;
10 int exit = 3;
11 int download = 4;
12 int downloaded = 5;
13 int fetched = 6;
14
15 - (int)handleEvent:(Event)event forState:(int)state {
16         switch(event) {
17                 case initial: // initial transition
```

```
18                  if (state == root) {
19                          return idle;
20                  } else if(state == downloadData) {
21                          return fetching;
22                  }
23                  break;
24                  case download:
25                  if (state == idle) {
26                          [self prepareFetch];
27                          return downloadData;
28                  }
29                  break;
30                  case fetched:
31                  if (state == fetching) {
32                          [self didFetch];
33                          return parsing;
34                  }
35                  break;
36                  case downloaded:
37                  if (state == fetching || state == parsing) {
38                          [self allDone];
39                          return idle;
40                  }
41                  case entry: {
42                          if (state == fetching) {
43                                  [self startFetching];
44                          } else if (state == parsing) {
45                                  [self startParsing];
46                          }
47
48                          //event was handled
49                          return 0;
50                  }
51                  case exit: {
52                          if (state == parsing) {
53                                  [self cleanup];
54                          }
55
56                          //event was handled
57                          return 0;
58                  }
59          }
60
```

```
61          // No state responded to the event
62          return 0;
63  }
```

While the solution suits as a correct way of modeling the statechart from Figure 5.10, it contains some downsides. Even with a simple example the single method is becoming hard to read and understand. The hierarchy of the statechart is defined by the order of the cases in the switch statement, since substates should be able to handle an event before its superstate, as described in Section 4.3.9. This way the model quickly becomes difficult to maintain and understand, as the complexity of the statechart grows. Further trying to model the requirement of orthogonal regions would dramatically increase the complexity and would not be feasible for complex applications with larger statecharts. Thus a better model is needed.

Functions for each state

The problem with having one function for handling events in the statechart can be avoided by making a separation into several functions, as shown in Listing 5.2 for the statechart in Figure 5.10. This approach is similar to the one suggested by Samek (2008, p. 39-48). Instead of abstracting a state into a simple `int`, as in the previous model, each state is instead modeled as its own function. Each function takes an event as the parameter and checks if it responds to the event. If it does, the action corresponding to the event is fired and the next state the engine should make a transition to is returned using the `Transition(state)` macro. The macro is used instead of simply returning the state as a way of using the return type to return different types of values. For instance, if the state responds to the event with an internal transition it simply performs the action and returns the `Handled()` macro to tell the engine that the event was handled in this state. If the state is not responding to the provided event it returns a reference to its superstate using the `Superstate(superstate)` macro. This allows the statechart engine to propagate the event up in the statechart hierarchy using a bottom-up approach, until the root is reached. The root state of the statechart that defines the initial transition and handles all events in order to stop the propagation.

Listing 5.2: Structuring the statechart using methods.

```
1  //events
2  int initial = 1;
3  int entry = 2;
4  int exit = 3;
5  int download = 4;
```

```
int downloaded = 5;
int fetched = 6;

- (StateResult)root:(Event)event {
        if (event == initial) {
                return Transition(idle);
        }

        return Handled();
}

- (StateResult)idle:(Event)event {
        if (event == donwload) {
                [self prepareFetch];
                return Transition(downloadData);
        }

        return Superstate(root);
}

- (StateResult)downloadData:(Event)event {
        if (event == initial) {
                return Transition(fetching);
        } else if (event == downloaded) {
                [self allDone];
                return Transition(idle);
        }

        return Superstate(root);
}

- (StateResult)fetching:(Event)event {
        if (event == entry) {
                [self startFetching];
        } else if (event == fetched) {
                [self didFetch];
                return Transition(parsing);
        }

        return Superstate(downloadData);
}

- (StateResult)parsing:(Event)event {
```

```
49          if (event == entry) {
50                  [self startParsing];
51                  return Handled();
52          } else if (event == exit) {
53                  [self cleanup];
54                  return Handled();
55          }
56
57          return Superstate(downloadData);
58  }
```

This solution is a significant improvement because of the separation into several functions, one for each state. It gives a much more clean and easy to read structure. Further the hierarchy is now explicitly given by each state by having it specify its superstate. This puts the logic of the hierarchy into the execution engine and makes it responsible for calling the correct functions instead of given the responsibility to the developer of the statechart. This means that this approach is suitable for modeling hierarchical statecharts. Because of the bottom-up approach, trying to model orthogonal regions introduces complexities and as explained in Samek (2008, p. 231) *"Orthogonal regions are a relatively expensive mechanism [since] [...] each orthogonal region requires a separate state variable (RAM) and some extra effort in dispatching events (CPU cycles)"*. Instead the solutions suggest to use orthogonal components, where orthogonality is achieved by having several statecharts executing at the same time. However orthogonal components introduce complexity for the developer of the statechart and do not provide the same power as orthogonal regions. This comes from the fact that modeling dependencies between the orthogonal parts are not defined for orthogonal components. The requirement of allowing reusability of the statechart and include it as a part of another statechart is only partly supported. The problem lies in the fact that both the actions and the hierarchy are defined together in each function. In order to change a transition one would have to replicate the action logic as well. This would result in much redundancy in the design of a complex application. For this reason a model is needed that separates the hierarchy and action logic, and that makes it possible to use a top-down approach in order to model orthogonality.

Object for each state

In order to higher the abstraction level and flexibility it is possible to take advantage of the fact that the solution is to be applied in an *object-oriented* environment. By modeling a state as an object the model becomes much more powerful and extendible. By further separating the action logic from the structure of the statechart, reusability becomes feasible. Further, object inheritance makes it easier to reuse logic across statecharts. Listing 5.3 shows how the statechart from Figure 5.10 can be modeled using a unique class for each state. These states are all subclasses of a generic state

class called `State`. All states implement the events it is responding to as methods. When the action of an event is being triggered, the method is simply called by the statechart engine. A reversed method is used for defining the entry and exit actions. The hierarchy and relation between the states is collected into a single configuration method called `setupStatechartWithRoot`, which gives the root state as a parameter in order to add substates to it. An instance of each state is instantiated and the hierarchy is specified by setting the substates of a state. Lastly the transitions between states are created.

Listing 5.3: Structuring the statechart using objects.

```
@class Idle: State {
    - (void)download {
        [self prepareFetch];
    }
}

@class DownloadData: State {
    - (void)downloaded {
        [self allDone];
    }
}

@class Fetching: State {
    - (void)entry {
        [self startFetching];
    }

    - (void)fetched {
        [self didFetch];
    }
}

@class Parsing: State {
    - (void)entry {
        [self startParsing];
    }

    - (void)exit {
        [self cleanup];
    }
}
```

```
32
33 - (void)setupStatechartWithRoot:(State)root {
34     Idle *idle = [Idle new];
35     DownloadData *downloadData = [DownloadData new];
36     Fetching *fetching = [Fetching new];
37     Parsing *parsing = [Parsing new];
38
39     root.subStates = @[idle, downloadData];
40     downloadData.subStates = @[fetching, parsing];
41
42     [idle onEvent:download transitionTo:downloadData];
43     [fetching onEvent:fetched transitionTo:parsing];
44     [downloadData onEvent:downloaded transitionTo:idle];
45 }
```

Object-oriented programming has many benefits as it makes the model much more powerful and understandable because of the separation between actions and hierarchy. It provides the possibility of easily creating `AND states`. This is is simply done by providing a new generic class, that differs from the `State` class by introducing the concept of regions instead of substates. This means that the model is suitable for implementing orthogonal regions. Further the model can be extended with more features from the statechart notation as explained in Section 5.3.3. For this reason the object-oriented model having an object for each state has been chosen as the most suitable and it will be used as the foundation for the implementation in the next chapter.

Execution model

The focus of the design so far has been on modeling the statecharts. Another aspect is how to model the execution engine of a statechart. The execution of a basic statechart supporting `OR states` and `AND states` basically comes down to the following execution requirements.

- **Finding transition for an event** - When an event occur the engine first needs to figure out if the current state configuration responds to the event. This is done by finding a transition for the event. If no transition is found the event can be discarded, unless the event is being deferred as discussed further in Section 5.3.3.
- **Finding the lowest common ancestor** - Once a transition has been found it is necessary to identify the LCA of the source state and target state of the transition, as discussed in Section 4.3.9 Lowest common ancestor.

- **Exit to lowest common ancestor** - Once the LCA has been identified the statechart engine should exit from the source state up until the LCA.
- **Perform transition action** - As the statechart has been exited to the LCA the transition action is fired before entering to the target state.
- **Enter from lowest common ancestor** - Finally the target state should be entered from the LCA

The sections below go through each of the above requirements in order to establish a design for how the statechart engine should fulfill them. The statechart from Figure 5.12 will be used as an example during the design of each requirement, since it both contains `OR states` and an `AND state` with two orthogonal regions.

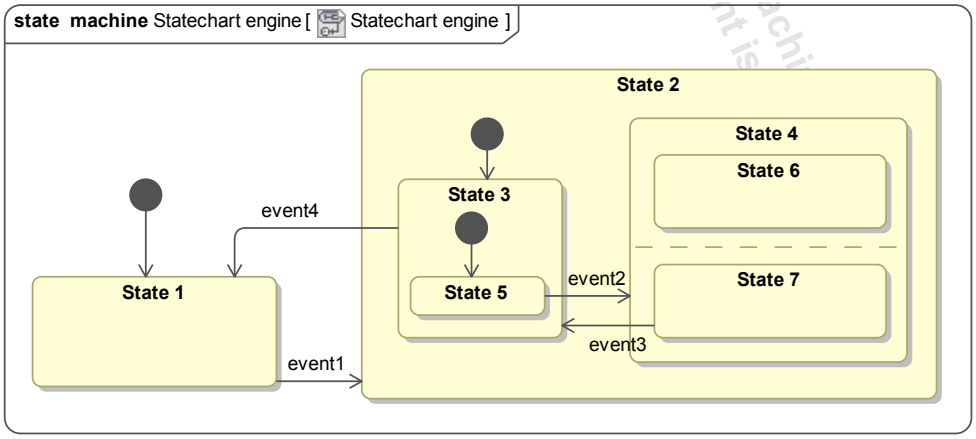

Figure 5.12: Example of statechart used to design the statechart engine.

As explained above, a statechart is modeled using objects. All states contain a reference to their superstate. For `OR state` they have an array of substates and for `AND states`, they have an array of regions. This information makes it possible to use an alternative graphical notation to model a statechart, using a graph. The statechart from Figure 5.12 is modeled as a graph in Figure 5.13. The graph notation is used in the following sections to explain how the engine executes. The root of the tree is the root state of the statechart. The root is connected to the substates using lines. A Dashed line is used to connect an `AND state` with its regions.

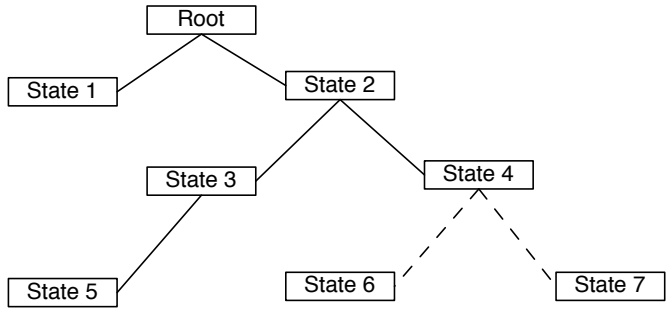

Figure 5.13: Tree of statechart.

Finding transition for an event

In order to perform a transition for an event, the statechart engine needs to search the tree using a top-down approach. The interesting part of the tree is all of the paths from the root to all active leaf states. This is the active state configuration. In order to search down the paths two cases should be considered. First the case where only `OR states` are involved is considered. This means that the system is in a single active state leaf state. Consider Figure 5.14 where the active leaf state is `State 5`. The traversal of the tree starts at the root. Since each `OR state` knows the currently active substate, the traversal continues down the tree. However, in order to avoid traversing all the way up again, once the leaf has been reached, each state check whether it responds to the event and propagates that information to its active substate. In Figure 5.14 since neither the root state or `State 2` respond to `event4` it will simply pass the value nil to the active superstate. However since `State 3` responds to `event4`, it will pass the transition to `State 5`. If a substate responds to the event it will overwrite the transition information. In the example `State 3` was the lowest state that responded to `event4` and thus transition from `State 3` to `State 1` is returned to the engine.

The other case is where an `AND state` is included in the active path. This means that more states are active. For instance as with `State 4` in Figure 5.15, where the search needs to be continued down in all regions. However, in order to collect all transitions into a set, each region must return the transition to the `AND state`, so it can be collected and provided to the statechart engine. This is illustrated in Figure 5.15 where `event3` is being forwarded to both `State 6` and `State 7`. Since `State 7` responds to `event3` it sends this information back to `State 4` that can provide it to the statechart engine.

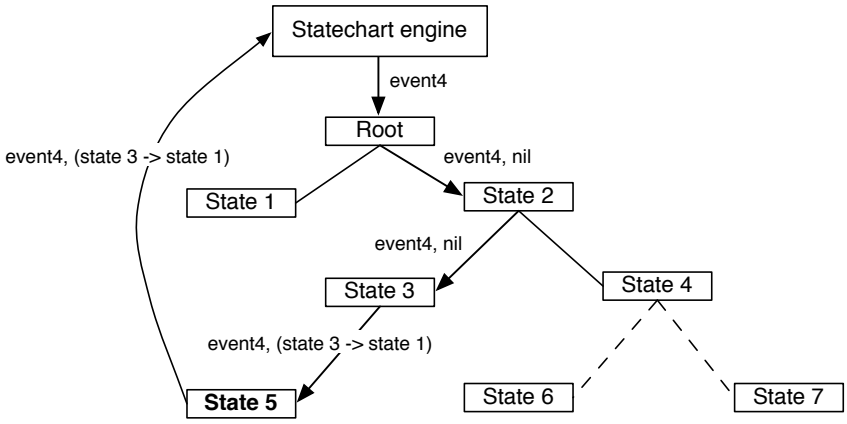

Figure 5.14: Finding transition with active leaf state being `State 5`.

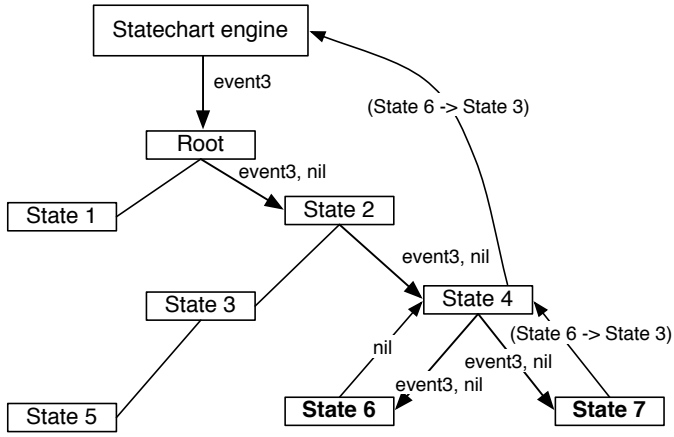

Figure 5.15: Finding transition with active leaf states being `State 6` and `State 7`.

The running time of finding the transitions for an event is $n + n \times k$ where n is the number of states in the statechart and k is the constant time it takes to determine if a state responds to an event. A statechart only consisting of `AND states` would

require a traversal of every state twice and for each state it should be determined whether it responds to an event or not. In big O notation this is $\mathcal{O}(n)$.

Finding the lowest common ancestor

The next step is to find the LCA between the source and target of a transition. This can be done using the following algorithm. First the path to the root state is found for both the source and target state as illustrated in Figure 5.16. The LCA is the state where the two paths intersect. This is `State 2` in the example.

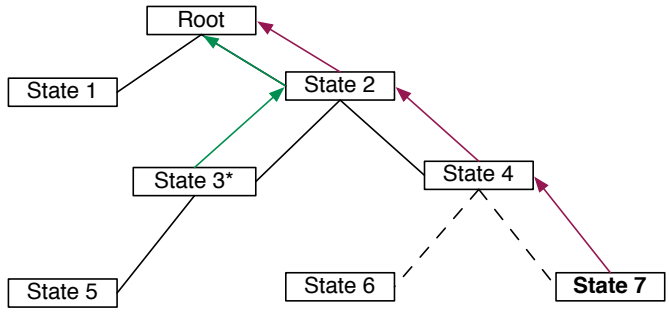

Figure 5.16: Finding lowest common ancestor.

The time complexity for finding the LCA is $2 \times h$, where h is the height of the tree. This is because a statechart where a transition is performed from two leaf states would require a traversal of the height, twice. In big O notation this is $\mathcal{O}(h)$.

Exit to lowest common ancestor

Now that the LCA has been found the statechart engine should exit up to the LCA. It can do so by simply exiting each state from the source state until the LCA is reached. This is indicated by the blue arrows in Figure 5.17 where an exit to `State 2` is performed. Notice how both `State 6` and `State 7` are exited since a transition outside of the regions is performed.

The running time is h since an exit from a leaf state to the root would require to exit a sequence of states equal to the height of the tree.

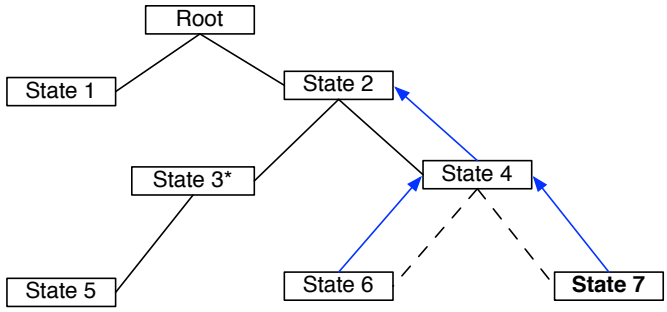

Figure 5.17: Exit to lowest common ancestor.

Perform transition action

The next thing is simply to perform the action. Since the found transition has a reference to the action it is simply a matter of calling a method.

Enter from lowest common ancestor

Finally the engine should enter from the LCA to the target state. The way this is done is similar to exiting, except the path from the LCA to the target is traversed and each state is entered. It is illustrated with the blue arrow in Figure 5.18.

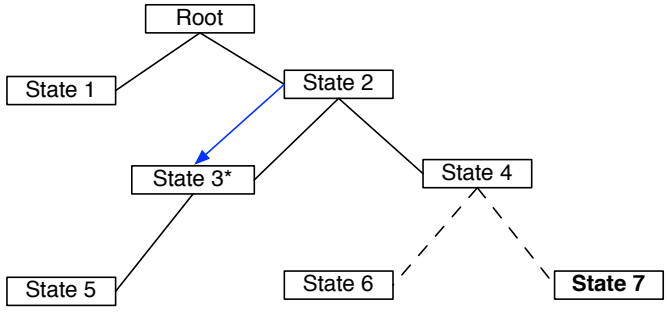

Figure 5.18: Enter from lowest common ancestor.

Doing so has a running time of h since entering a leaf state from the root state would require entering a sequence of states equal to the height of the tree. This makes a total running time of the statechart execution engine equal to $2 \times n + k + 2 \times + h + h = 2n + 4h + k = \mathcal{O}(n + h)$. This means that the running time depends on the number of states in the statechart plus the depth of hierarchy.

5.3.3 Supporting additional statechart features

Both the models for defining a statechart and the engine have been designed in order to support the fundamental notation of statecharts to model hierarchy and orthogonality. However, an important property of the design is to make it extendible for further functionality. Some of these are discussed below.

History

The history mechanisms as explained in Section 4.3.4 can be achieved by having a variable in each state that keeps track of the last active substate. This variable is updated every time the state is being exited. A transition to the history connector of a state results in the last active substate being entered, as it is stored in the variable. If no history has been recorded the initial transition is performed.

Fork & join

A fork transition as described in Section 4.3.5 can be supported by the engine by allowing a transition to have several targets. The engine as explained above would then simply have an array of targets, which all would be entered instead of just a single target. Similarly the join transition could be supported by having a transition with several sources and a single target.

Guards and timeouts

Guards can be put on a transition as explained in Section 4.3.7. This can be included in the model by simply providing a boolean expression that is validated when determining if a state responds to an event. This means that only transitions that have a guard that is true are propagated to the statechart engine. As explained in Section 4.3.7, a guarded transition can be taken simply by the condition of the guard becoming true, if no event is specified on the transition. This can be achieved by having an extra check after each transition, where all such guarded transitions are evaluated, to see if their condition is true. If any of them have become true a special event is posted to the statechart, making it perform the transition immediately. Since this might introduce another transition, it continues until no guards are true. Notice that this potentially could cause an infinite cycle of transitions. A way of solving this is to provide a maximum number of iterations that the statechart engine should check for guarded transitions, triggered only by a condition. An important property of a guard is that it contains no side effects. This means that checking the condition

of a guard changes no outside variables or calls any methods. The statechart engine should be allowed to check the condition of a transition at any time without triggering any behavior as it otherwise could result in unexpected behavior.

Further, the support for timeouts as explained in Section 4.3.8 can be achieved by starting a timer once a state has been entered. This timer is again stopped once the state is exited. However, if the timer runs out before the state is exited an event is posted to the statechart engine with the transition to be performed.

Deferred events

Deferred events as explained in Section 4.3.8 can be achieved. When the statechart engine processes an event, besides checking if one of the active states triggers a transition, it also checks if any active state defers the event. If this is the case the event is added to an array of deferred events. With each change of state, the array of deferred events is posted back into the engine. This would result in a check for each event to see if the new state configuration responds to the event.

Submachine

The model can be extended with submachines as explained in Section 4.3.8 by simply allowing a statechart to be instantiated with its root state. This makes it possible for a statechart to use another statechart as a submachine by instantiating it with a state placed in the hierarchy, where the submachine should be placed. This state suits as the root state of the submachine.

5.4 Debugging & inspection

The abstractions designed allow for a rich set of functionality for debugging and inspecting the system on a higher level compared to normal debugging tools such as a debugger. Tools for debugging and inspecting are discussed further below.

5.4.1 Overview of components

As an application is composed by components, it is possible to get an overview of the application by seeing which components are live in the system and how they are connected using the messaging system. It can be achieved simply by keeping track of all the components in the system. This is useful for debugging as it is possible to provide a graphical notation for showing the whole system, which makes it possible to spot if channels between ports are missing or components are not set up correctly.

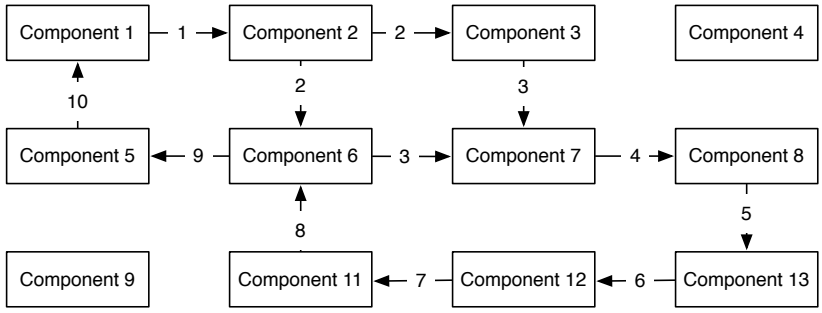

Figure 5.19: Message flow in a system of components

5.4.2 Inspecting messages

The only way for components to influence each other is through the messaging system, since they share no state. For this reason a powerful way of tracking the behavior in the system, is to look at the messages sent between the components. Because of the deterministic property of the messaging system, it is possible to simulate the communication between the components in order to debug the system. The simplest way of tracking messages sent in an application is simply to log every time a message is sent between two components. The information reveals which components are communicating with each other. It is a helpful way of understanding the system. However, even more information can be retrieved by looking at the messages flowing between the components. As explained by Roestenburg (2012) by understanding how messages flow in the system it is possible to obtain a better understanding of how the system works. By tracing how messages originate and cause a chain of messages to be send in the system gives an overview of what effect an event in the system has.

Figure 5.19 shows a flow of messages for a system consisting of 13 components. The flow originates in `Component 1`, which sends the first message to `Component 2`. It further creates two messages, which are sent to `Component 3` and `Component 6`. The flow ends with a message back to `Component 1`. Each message is provided with a number indicating the order in which they are sent. In order to design such a message tracing system the following must be known.

- **Sender** - From which component is a specific message sent.
- **Receiver** - To which components is a specific message sent.
- **Trigger message** - The trigger the message could either be another message, which would then extend an existing flow, or it could be because of an event in

the sending component, which would mean that a new flow has started.

In order to keep track of the trigger messages, an incoming message to a component is stored temporarily. If the message results in another message being sent, the stored message is recorded as the trigger message.

5.5 Summary

During the chapter, the framework to be implemented has been designed. The design consists of a shared thread pool for providing a context for each component that uses a low amount of resources. Further the messaging system has been designed, defining how components communicate with each other using channels that are connected to ports. Each port has a protocol defining the messages it understands. The channels are sending messages through a router, which is in charge of delivering the messages to the right receiving components. How to model a statechart with objects as states has been defined, which allows for modeling hierarchy and orthogonality as well as other features from the statechart notation. Further, the execution engine for the statechart has been designed. Lastly debugging and inspection possibilities have been explored that can be applied on top of the designed abstractions. The next chapter uses the described designs in order to make an implementation of the framework that can be used when developing iOS applications.

CHAPTER 6

Implementation

The design from the previous chapter ended with the blueprints for how a framework which introduces the concept of a component, a message system and statecharts can be designed. This chapter uses the design in order to make an implementation of a framework, that allows building applications for iOS using these abstractions. Since the implementation is done in Objective-C the framework also runs on Mac OSX and can be used for developing desktop applications.

Notice that classes implemented as a part of the framework, are all prefixed with *SHP*. For instance, the class representing a statechart is called `SHPStateChart`. Classes prefixed with *NS* are part of the systems framework provided by the iOS/OSX platform. An example of this is for instance `NSInvocation`.

The full implementation of the framework is available in the resources as explained in Appendix B.

6.1 Implementing components

The concept of a component has been implemented in the class `SHPComponent`. The class is supposed to be subclassed when using it to model an application. Each subclass should capture a behavior and provide an interface for other components to take advantage of. Defining this interface is discussed further in Section 6.2.3. The class `SHPComponent` provides the basic functionality for having a component with its own context. The application specific behavior of the component should be implemented as a method inside of the subclass. The power of statecharts can be used within a component in order to keep track of the state of the component. This is done using the subclass `SHPStateChart`, which can be seen in Section 6.3

As discussed in Section 5.1.2, it was decided to use a shared pool of threads for the components in the system in order to keep the memory usage low. The iOS/OSX platform framework provides an implementation of this, by using *Dispatch Queues* as presented in Apple (2012a). More specifically the object `dispatch_queue_t` represents a dispatch queue, which uses an underlying shared thread pool. When a task is dispatched to an instance of a `dispatch_queue_t`, it results in the task being executed by a thread from the underlying thread pool. The order in which tasks are being handled by the queue is configurable, but by making it serial the tasks are exe-

cuted one by one in first-in-first-out order. As explored in Section 5.1.2, new threads should be allocated and deallocated according to the needs of the components. However, an undocumented property of *Dispatch Queues* is the number of threads, that at maximum can be alive in the thread pool. It is stated in Apple (2012a, p. 1) *"The system can scale the number of threads dynamically based on the available resources and current system conditions"* still, it is unclear how many threads the system at maximum allows to be spawned. In order for *Dispatch Queues* to be suitable for providing a unique context for a component, it is crucial that the pool increases to a high number of threads, as components need them. Otherwise it would result in a component having to wait in the queue for another component to release the thread, and the independency between components would be lost. For this reason a test project has been created to record data that can be used to determine how a `dispatch_queue_t` behaves.

Example 5 *In order to determine the maximum size of the thread pool used by Dispatch Queues the example project* `DispatchQueueComponentTest` *has been created. It can be accessed through the resources as explained in Appendix C. In the project a variable number of* `dispatch_queue_t` *instances are created. Each of them is given the task to sleep for two seconds on the thread that they have been given from the thread pool. The maximum time among all* `dispatch_queue_t` *instances to do this is then recorded. Based on the results it is possible to determine how many threads the thread pool increased to, by running the test with a various number of* `dispatch_queue_t` *instances and for each run look at the maximum executing time. If all* `dispatch_queue_t` *instances had been given a unique thread, the maximum runtime would be around 2 seconds. If any of them would have to wait for a thread to become available, the maximum would be above 2 and no new threads can be created.*

The result of Example 5 tested on an iPhone 5S is shown in Figure 6.1. Other iOS devices had been tested as well and provided the same result. It shows that with more than 512 `dispatch_queue_t` instances, the maximum execution is increased by 2 seconds. Using this information, it can be concluded that the thread pool has an upper limit of 512 threads.

Given the fact that a single application needs more than 512 occupied threads before a `dispatch_queue_t` instance would have to wait for a thread to become available, it has been decided to use *Dispatch Queues* as the implementation of the shared thread pool design. Notice as explained in Section 5.1.2, the components living in the system only acquire a thread when they have a task that should be executed. This means that more than 512 components can be alive in an application without the thread pool reaching its limit.

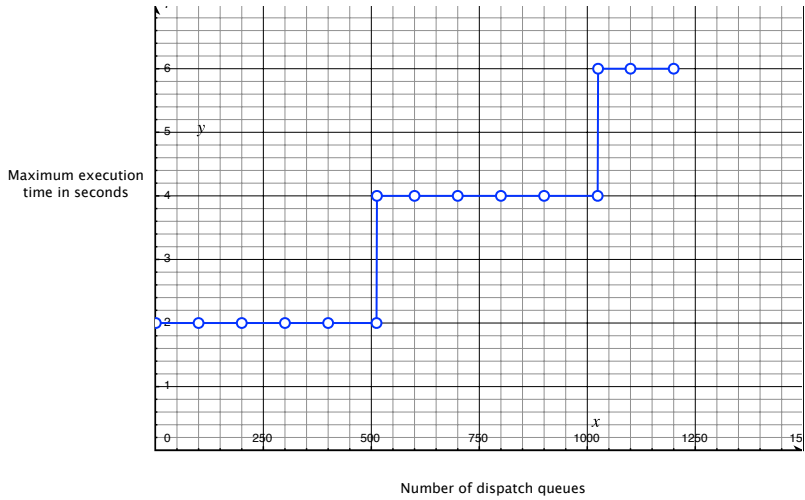

Figure 6.1: Graph showing maximum execution time with increasing number of `dispatch_queue_t` instances as explained in Example 5.

Each `SHPComponent` instance holds a serial `dispatch_queue_t` property called `messageQueue`. Every execution of a component must be dispatched on this `dispatch_queue_t` property in order to avoid concurrency problems as analyzed in Section 2.4. Further it is built directly into the messaging system as explained in Section 6.2.3.

6.1.1 Defining a component

In order to create a component a object subclassing `SHPComponent` is created as shown in Listing 6.1.

Listing 6.1: Defining a component

```
1  @interface MyComponent : SHPComponent
2
3  @end
4
5  @implementation MyComponent
6
7  - (void)doSomething {
8      dispatch_async(self.messageQueue, ^{
9          //running in own context.
10     });
```

```
11 }
12
13 @end
```

It is important that all tasks, that use shared variables within the component, are dispatched on the `messageQueue` as shown in line 8-10 in Listing 6.1.

6.2 Implementing the messaging system

The communication between components is an essential part of the solution and thus it is important that it is convenient to use the framework as required in Section 1.4. Because of this a lot of work and many iterations have been required, in order to end up with an implementation of the messaging system that satisfied this requirement. The following sections will walk through the major iterations and explain why further improvements were needed before ending up with a solution.

6.2.1 Message

An essential part of the messaging system is the messages floating between the components. Below, first the job of receiving a message is being discussed and then the job of sending a message.

Receiving a message

A key element of receiving a message is being able to pull information out of it, so it can be understood. This means, that the sender needs to provide information about what kind of message is being sent. The first iteration of the messaging system contained several classes for representing a message. The idea was to have a class for each kind of message. For instance the class `SHPCommandMessage` was used to send commands between components. A command represented a type of message where the sender wanted the receiver to do something and then reply with a result. Another kind of message was the `SHPEventMessage`, it was used to send notifications between components. When sending an event message, the sender did not expect a reply from the receiver. Since having a few generic classes for representing a message were not enough for the receiver to identify and understand a message, each message also had a field called `type`, which contained additional information about the message. If a message contained any content, it was sent in a content field that could contain any kind of object. If more objects needed to be sent, they had to be wrapped up in an array.

The combination of the class and the type allowed the receiver to identify the meaning of an incoming message. However, it was found that as the number of messages,

a component is able to understand, increases the job of identifying incoming messages becomes complex and messy. Further, getting the content out of the message required the receiver to cast it into the correct type. An example of how messages were received in the first iteration is shown in Figure 6.2.

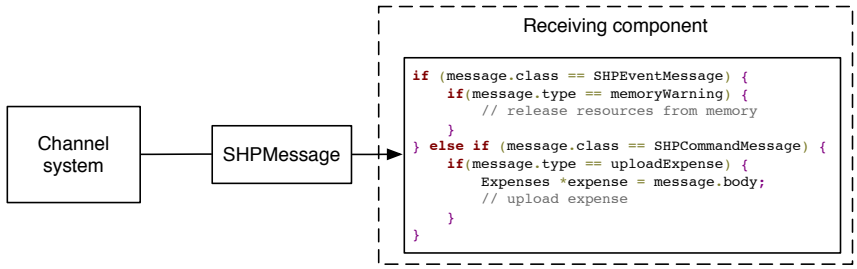

Figure 6.2: First iteration of receiving a message.

The solution required a lot of matching code in the receiving component. This quickly became messy and therefore could easily introduce bugs. Further, the code was not coherent and not satisfying the purpose of being convenient to work with for the developer.

The goal of the second major iteration, which became the final solution, was to make it much easier for the receiving component to identity the type of message and to retrieve the content from the message, without having to cast in into the right type of object. The idea was to implement a solution where messages were delivered in the same fashion as when a method is called on an object. The receiver would then simply implement a method for each message type, and the content of the message could be passed in as arguments to the method.

The underlying messaging system still uses a generic class for sending a message from component to component, however the overall message representation has been refactored down to a single class called `SHPEventMessage`. Instead of containing a generic `body`, it contains an `invocation` property, which has the type class `NSInvocation`. A `NSInvocation` instance is used by the system for handling method calls. It includes the name of the method and all the arguments and their values. Instead of delivering the `SHPEventMessage` to the component, the message system invokes the `NSInvocation` on the component, resulting in a simple method call. With this solution all message types matching code disappears from the receiving component. Further there is no need to cast the object provided by the message. The second

iteration is illustrated in Figure 6.3, where methods are simply implemented in the receiving component and automatically called when a message matching one of them is received.

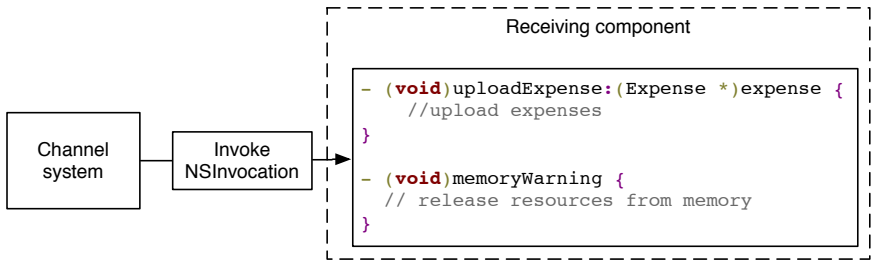

Figure 6.3: Second iteration of receiving a message.

Using the second iteration a much cleaner way of receiving messages can be defined.

Sending a message

As the implementation for receiving a message was improved, so was the implementation of sending. The two iterations are discussed further below.

In the first iteration, the sender needed to create an instance of one of the generic message classes and set the correct type of message. The message would then be posted into the messaging system by giving it to an outgoing port, as examined further in Section 6.2.3. An example of sending a message with the indention of uploading an expense is shown in Figure 6.4.

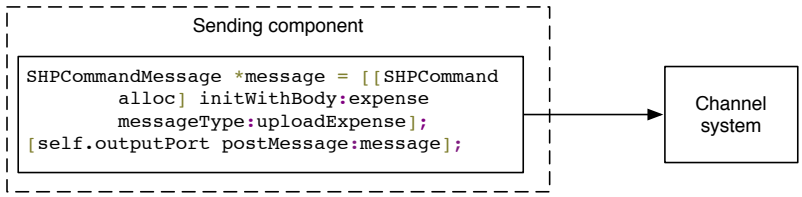

Figure 6.4: First iteration of sending a message.

While the solution provided a solid way of sending a message, the extra overhead of constantly having to instantiate objects in order to send a message seemed unnecessary. So the goal for the second iteration was to make it easier to send a message. By allowing it to be done in a similar way as when invoking a method, the task for sending a message became much simpler. The second iteration removed the need for a component to ever create an instance of a message. Instead this logic was built into the port system in a generic manner. A component should simply invoke a method on the output port, in order to send a message on it. The port then converts it into an `SHPEventMessage` by constructing an `NSInvocation`. The `SHPEventMessage` is then sent into the channel system. This way sending messages became much cleaner as illustrated in Figure 6.5.

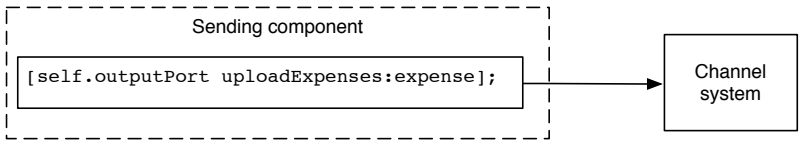

Figure 6.5: Second iteration of sending a message.

6.2.2 Channel

In order to define which components communicate with each other, channels are used between the components. Each channel is either of the type point-to-point or publish-subscribe as explained in Section 4.2.4. The concept of a channel has been implemented in the class `SHPChannel`.

The first iteration of the channel system was based on channels that directly handled the communication between the components. When a message was given to a channel it made sure that components in the other end would receive the messages when ready. Each channel had its own `dispatch_queue_t`. When a message was sent over the channel, it would result in the message being added to the queue. The channel processed the messages in the queue one by one in a first-in-first-out order. When the channel scheduled execution time, it would dequeue a message from the queue and send it to the receiver of the channel. While working as expected it was decided to introduce the concept of a central router as discussed in Section 5.2.5. This was done in order to send all messages in the system through a central point, for easier monitoring of the traffic between components.

The responsibility of delivering the messages was moved from the channel to the router. The router is implemented in the class `SHPMessageRouter`, which uses the

singleton pattern as presented in Section 3.3.2, for providing a single globally accessible instance. The router contains a serial `dispatch_queue_t` for processing messages. The messages are dequeued from the queue one by one and delivered to the right components.

6.2.3 Ports

As presented in Section 5.2.6 ports are used to define the protocol of a component. This can either be for input, output or internally. The implemented port system consists of many classes which all have the system class `NSProxy` as the root class as illustrated in Figure 6.6. The `NSProxy` class allows invoking a method on an instance of the class on behalf of another object. The use of `NSProxy`, is what makes sending and receiving messages similar to the invocation method in its usage. The subclass `SHPInvocationProxy` extends the `NSProxy` by introducing a `Protocol` field. In Objective-C a `Protocol` is a list of methods. With the `SHPInvocationProxy` class only methods specified in the protocol are being forwarded. The `SHPPort` subclass defines the base class for all port types by having a port containing a reference to the `SHPChannel` instance it is connected to. Further, a port has a reference to the component it is used in.

The basic `SHPPort` subclasses are explained in the following sections.

Output port

The `SHPOutputPort` class is used for sending messages out on a channel. This is simply done by invoking a method on an instance of a `SHPOutputPort`. The method must be defined in the protocol of the output port in order for the invoked method to result in a message being sent.

Input port

The input port is implemented using `SHPInputPort`. Messages received by an input port, must be included in the specified protocol in order for them to be processed. A `SHPInputPort` instance delivers incoming messages to its component by using invoking the method `inputPortdidReceiveMessage:`. The `SHPComponent` class implements this method and the default behavior is invoking the `NSInvocation` contained in the message on the component itself. A `SHPStateChart` class however, processes the received message as an event instead, as discussed in Section 6.3.5.

Event port

As treated in Section 5.2.6, an event port exists which a component can use to post messages to itself. This is implemented in `SHPEventPort`, which is a subclass of

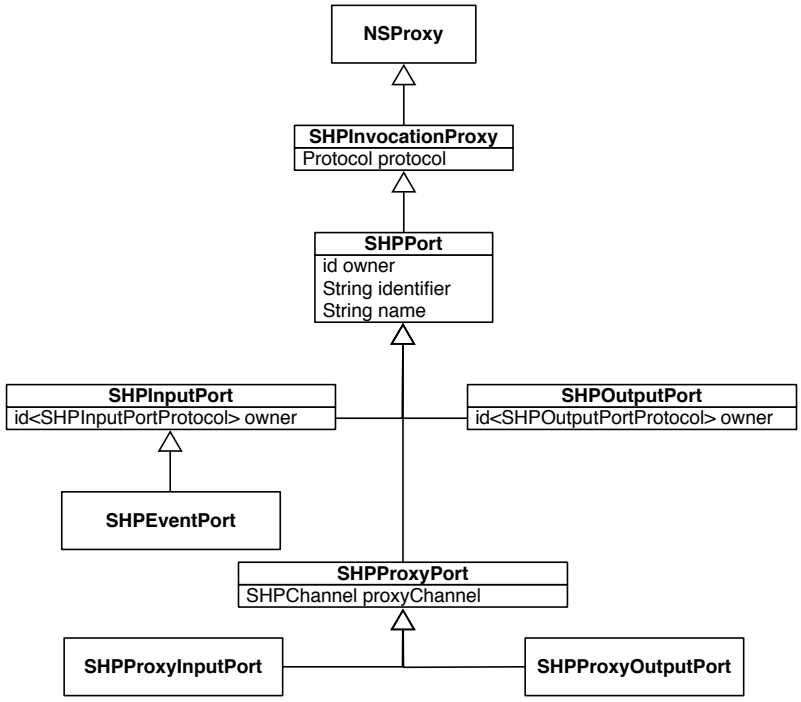

Figure 6.6: Class hierarchy of port system.

`SHPInputPort`. The event port behaves as a mixture of an output and input port. Methods invoked on it result in a call of `inputPortdidReceiveMessage:` on the component. This means that internal messages are handled the same way as external ones.

6.2.4 Proxy ports

Two special kinds of ports exist that can be used for forwarding messages to a port as explained in Section 5.2.6. This is implemented in the classes `SHPProxyInputPort` and `SHPProxyOutputPort`. They both inherit from the class `SHPProxyPort`. A `SHP-ProxyPort` class includes an extra `SHPChannel` called `proxyChannel` used to connect to the port being proxied.

6.2.5 Setting up channels between components

In order to setup channels between components and take advantages of the above-explained classes all that needs to be done is defining ports on each component and

connecting them. Listing 6.2 shows how a system consisting of three components, each with an input and an output port, can be connected using the implemented framework.

Listing 6.2: Connecting three components.

```
- (void)createAndConnectComponents {
        Component1 *component1 = [Component1 new];
        Component2 *component2 = [Component2 new];
        Component3 *component3 = [Component3 new];

        component1.outputPort.channel =
                component2.inputPort.channel =
                [[SHPMessageRouter sharedInstance]
                        publishSubscribeChannel];
        component2.outputPort.channel =
                component3.inputPort.channel =
                [[SHPMessageRouter sharedInstance]
                        publishSubscribeChannel];
        component3.outputPort.channel =
                component1.inputPort.channel =
                [[SHPMessageRouter sharedInstance]
                        publishSubscribeChannel];
}
```

6.3 Statechart implementation

As a part of the framework the concept of a statechart has been implemented. It has been applied in a way to integrate with the concept of a component in order to easily keeping track of the current state configuration. It has been implemented into the class `SHPStateChart`, which is a subclass of `SHPComponent`. This means that a statechart takes advantage of the independent context offered by the components implementation and further makes it possible to use the implemented messaging system to communicate with other statecharts or components. In order to show how to model a statechart using the implemented framework, the statechart from Figure 6.7 will be used as an example.

6.3.1 Defining the states

An essential part of the statechart implementation is the states. As explained in Section 5.3.2 two bases exist for providing the possibility of modeling, `OR state`s and `AND state`s. They are implemented in the `SHPOrState` class and `SHPAndState` class respectively. They are both subclasses of the abstract class `SHPState` that contains

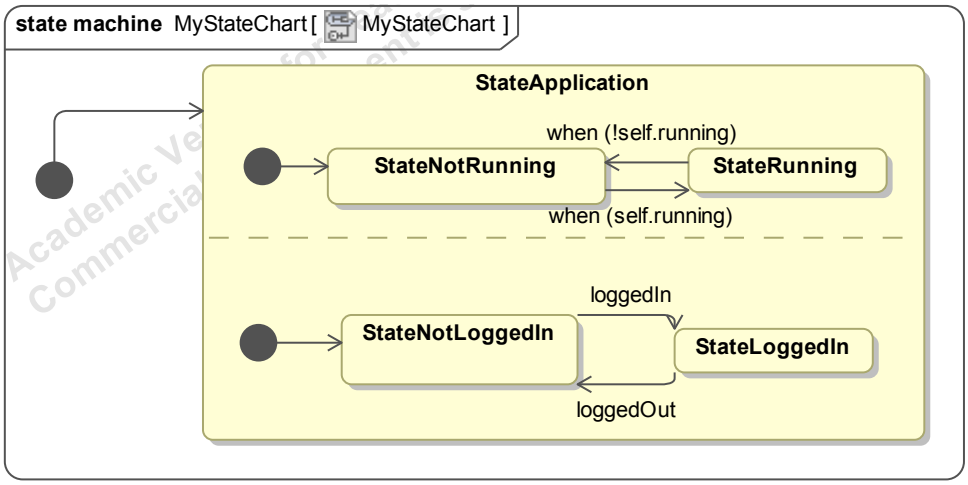

Figure 6.7: Simple statechart.

the logic shared between both types. When defining the states of a statechart these two base classes should be subclassed for each state, in the same fashion as presented in Section 5.3.2. For instance Listing 6.3 shows how to define the four `OR state`s and the single `AND state` from Figure 6.7.

Listing 6.3: Defining four `SHPOrState` subclasses and one `SHPAndState` subclass.

```
1  @interface StateApplication : SHPAndState @end
2  @implementation StateApplication @end
3
4  @interface StateLoggedIn : SHPOrState @end
5  @implementation StateLoggedIn @end
6
7  @interface StateNotLoggedIn : SHPOrState @end
8  @implementation StateNotLoggedIn @end
9
10 @interface StateRunning : SHPOrState @end
11 @implementation StateRunning @end
12
13 @interface StateNotRunning : SHPOrState @end
14 @implementation StateNotRunning @end
```

6.3.2 Defining the hierarchy

In order to take the states and put them together into a statechart a subclass of `SHPStateChart` must be created. To configure the hierarchy and instantiate the states, the method `configureWithConfiguration:` must be implemented. The method is automatically called by the superclass. Listing 6.4 shows an implementation the statechart from Figure 6.7. The class `MyStateChart` is a `SHPStateChart` subclass. It implements the `configureWithConfiguration:` method. Line 29-35 instantiate instances of the states in the statechart. Notice how two extra `SHPOrState` instances are created in line 34 and 35. These are used as the overall region states. Subclassing one of the base classes is the recommended way of defining a state, however states that do not need to perform logic when responding to events, can be instances of one of the base classes directly. From line 38-44 the hierarchy of the statechart is being setup. The provided `SHPStateChartConfiguration` instance has a reference to the automatically created root state of the statechart. The `StateApplication` state is added as a substate to the root state. In line 40 the two regions are added. In line 41 to 44 the substates of the regions are added. Notice how the initial state is specified when adding a substates to a `SHPOrState` subclass.

Listing 6.4: Example of a `configureWithConfiguration:` implementation.

```
1  @interface MyStateChart : SHPStateChart
2
3  @property(nonatomic, strong) SHPInputPor
4          t<MyStateChartInputProtocol> *inputPort;
5  @property(nonatomic, strong) SHPOutputPort
6          <MyStateChartOutputProtocol> *outputPort;
7
8  @end
9
10 @implementation MyStateChart
11
12 - (id)init {
13     self = [super init];
14     if (self) {
15         _inputPort = [self
16                 inputPortWithProtocol:@protocol(My-
17                 StateChartInputProtocol) name:@"Input"];
18         _outputPort =
19                 [self outputPortWithProtocol:@protocol(My-
20                 StateChartOutputProtocol) name:@"Data input"];
21     }
```

```
22
23      return self;
24  }
25
26  - (void)configureWithConfiguration:(SHPStateChartConfiguration *)
27                  configuration {
28      // Init states
29      StateApplication *stateApplication = [StateApplication new];
30      StateLoggedIn *stateLoggedIn = [StateLoggedIn new];
31      StateNotLoggedIn *stateNotLoggedIn = [StateNotLoggedIn new];
32      StateRunning *stateRunning = [StateRunning new];
33      StateNotRunning *stateNotRunning = [StateNotRunning new];
34      SHPOrState *region1 = [SHPOrState new];
35      SHPOrState *region2 = [SHPOrState new];
36
37      // Setup hierarchy
38      [configuration.rootState setSubStates:@[stateApplication]
39                  initialState:stateApplication];
40      [stateApplication setRegions:@[region1, region2]];
41      [region1 setSubStates:@[stateNotLoggedIn, stateLoggedIn]
42                  initialState:stateNotLoggedIn];
43      [region2 setSubStates:@[stateNotRunning, stateRunning]
44                  initialState:stateNotRunning];
45
46      // Setup transitions
47      [stateNotLoggedIn onEvent:@selector(loggedIn)
48                  transitionTo:stateLoggedIn];
49      [stateLoggedIn onEvent:@selector(loggedOut)
50                  transitionTo:stateNotLoggedIn];
51      [stateNotRunning transitionTo:stateRunning
52                  guard:[[SHPStateChartGuard alloc]
53                  initWithCondition:^BOOL{
54          return self.running;
55      } guardDescription:@"If running"] reverse:YES];
56
57      // Setup event protocol
58      configuration.eventHandlerProtocol =
59                  @protocol(MyStateChartEventProtocol);
60  }
61
62  @end
```

6.3.3 Defining the transitions

The transitions between states are modeled with the base class `SHPStateChartTransition`. This class represents a transition triggered by an event. It supports making fork, merge, internal or history transitions, which are explained in Section 5.3.3. Further the subclass `SHPStateChartGuardTransition` makes it possible to create guarded transitions. Each guarded transition uses a `SHPStateChartGuard`, which contains a condition that evaluates to either true or false. Further `SHPStateChartTimeoutTransition` is used to create transitions triggered after a time period. And lastly `SHPStateChartDeferredTransition` is used for creating deferred transitions. However, the need, to instantiate an object of any of these classes has been abstracted into `SHPState`. This means that a transition can be added between a source and a target state, simply by asking the source state to add it. For instance line 47 from Listing 6.4 shows how to add a transition from `StateNotLoggedIn` to `StateLoggedIn` when the event `loggedIn` occurs. Methods for adding the other supported transitions exist on all states. For example line 51 shows how to add a transition from `StateNotRunning` to `StateRunning` that is automatically being triggered when the provided condition becomes true. The specified `reverse` flag, for the transition, automatically creates the reverse transition, resulting in a transition from `StateRunning` to `StateNotRunning`, when the condition is false. Methods for defining fork, merge, internal, history and timeout transitions are defined for a state. The header file of `SHPState` should be examined in order see how they are called.

6.3.4 Defining the events

The last thing of the implementation of the statechart is to define the events. The `SHPStateChart` class defines a `SHPEventPort` for handling events. However, the protocol for it needs to be defined and provided to the statechart. The protocol is defined by creating an Objective-C `Protocol` that contains the events. As explained in Section 6.2.3, the events in the system are implemented the same way as messages. Listing 6.5 defines the two events used in the statechart from Figure 6.7

Listing 6.5: Example of `configureWithConfiguration:` implementation

```
1 @protocol MyStateChartEventProtocol
2 - (void)loggedIn;
3 - (void)loggedOut;
4 @end
```

Notice how the event names match the events from line 47 and 49 in Listing 6.4. Further, line 58 tells the statechart to use the protocol `MyStateChartEventProtocol`, when instantiating the underlying event port.

Starting the statechart

In order to use the statechart, an instance should simply be created and the method `initialTransition` should be called in order to perform the initial transition. This is shown in Listing 6.6.

Listing 6.6: Creating instance of `MyStateChart` class and calling `initialTransition`

```
1 MyStateChart *myStateChart = [MyStateChart new];
2 [myStateChart initialTransition];
```

6.3.5 Execution engine

The way of implementing a statechart has now been presented. When using the framework for implementing statecharts the underlying engine is abstracted away. The implementation of the statechart engine is based on the design described in Section 5.3.2. The main logic of the engine is implemented in the class `SHPStateChart`. The method `performEvent:` contains the logic for handling an event. Figure 6.8 shows a flow chart for how an incoming event is handled. The various tasks such as getting transitions, exiting to LCA and entering from LCA are defined in the `SHPState` subclasses, `SHPOrState` and `SHPAndState`, according to the model explained in Section 5.3.2.

The transitions for the provided event are first found. If a transition is found, the type is examined and the rules from Section 4.3.1 are followed. For instance a self transition results in the currently active state being exited, the transition action being performed and the state being entered again. If a transition will be performed the `checkForGuardTransition` flag is set to true. This makes sure that transitions that are not being triggered by an event, but only by a condition becoming true is checked to see if they have become true. The event has been completely handled once the transition triggered by the event has been taken and there are no more guarded transitions, which are a part of the active state configuration, that are true.

6.4 Overall framework structure

The overall structure of the core part of the framework is illustrated in Figure 6.9. The figure shows the class hierarchy with the filled arrows and the associations between the classes with the not filled arrows. The figure is slightly simplified compared to the implementation. It contains the implementation of the components, messaging system and statecharts in one big picture.

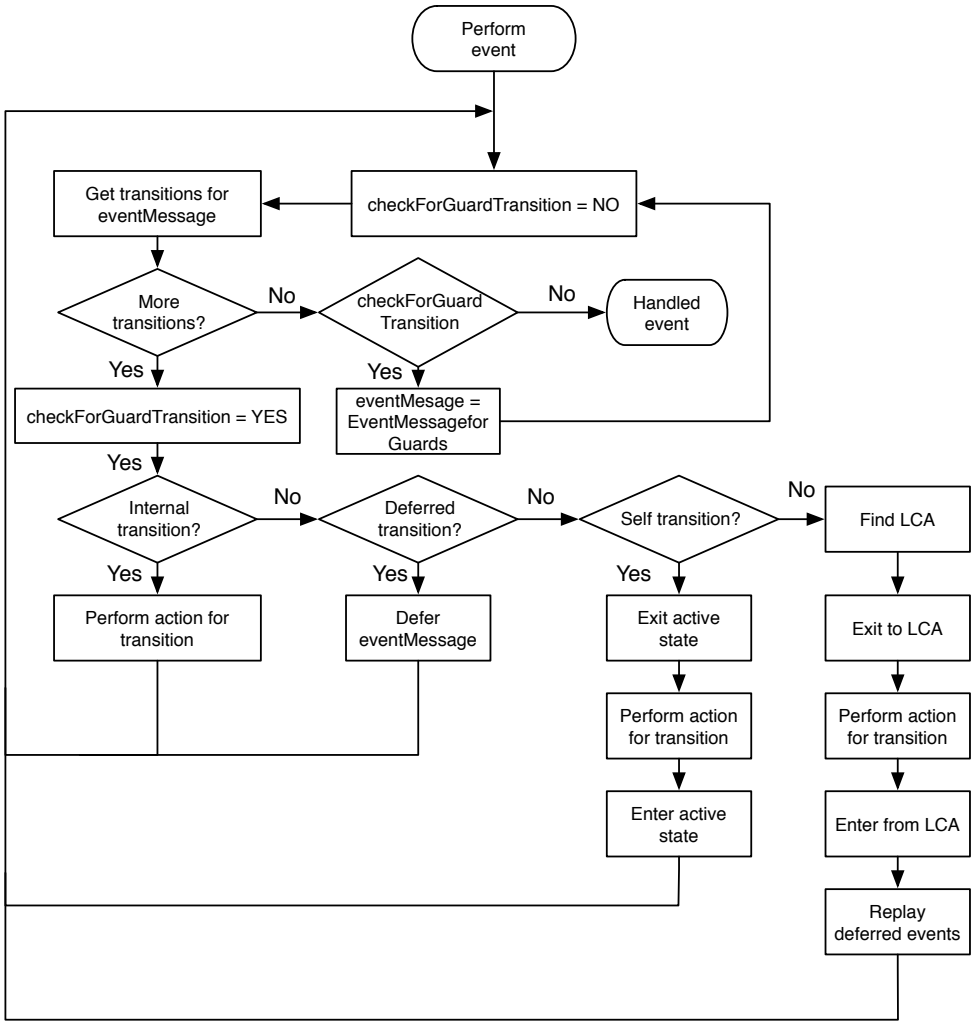

Figure 6.8: Flow chart of the `performEvent:` method.

For instance it shows how an `SHPOrState` has an array of `substates`, where a `SHPAndState` has an array of `regions`.

6.5 Runtime system

Based on the implemented core framework as described above, powerful abstractions have been implemented taking advantage of the framework. This has been put together into a runtime system that can be used when developing applications. The runtime consists of components and features that make it more convenient and powerful, to work with the framework.

6.5.1 Component register

An essential part of the runtime is the component register, which is implemented in the class `SHPComponentRegister`. The main purpose of the component register, as presented in Section 5.4.1, is to keep track of the components that are currently in the system. It is implemented as a `SHPComponent` subclass. It has an input port that follows the protocol shown in Listing 6.7. The protocol allows a component to register itself. This functionality has been built into `SHPComponent` so all subclasses automatically register themselves to the component register. Further it includes an output port where messages are sent, whenever a component is registered or removed.

The simple component register opens up the possibility for many interesting features when it comes to inspection and debugging of a running application. Seeing the components currently active in an application provides a way of getting an overview of the application.

Listing 6.7: Protocol for input port of `SHPComponentRegister`.

```
1 @protocol SHPComponentRegisterCommandProtocol
2 - (void)componentRegisterAddComponent:(SHPComponent *)component;
3 - (void)componentRegisterRemoveComponent:(SHPComponent *)component;
4 - (void)componentRegisterGetComponents;
5 @end
```

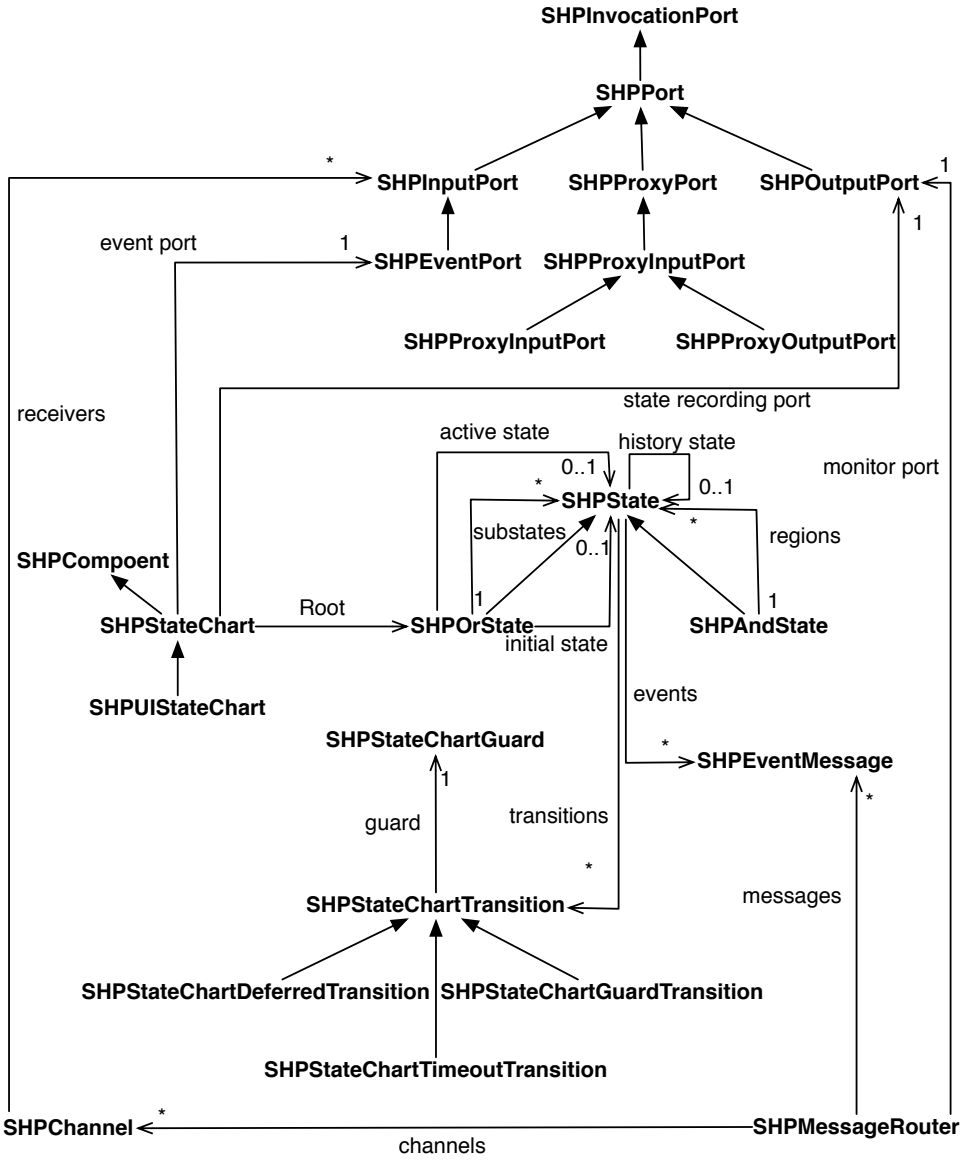

Figure 6.9: Overall structure of the framework implementation.

6.5.2 Message bridge

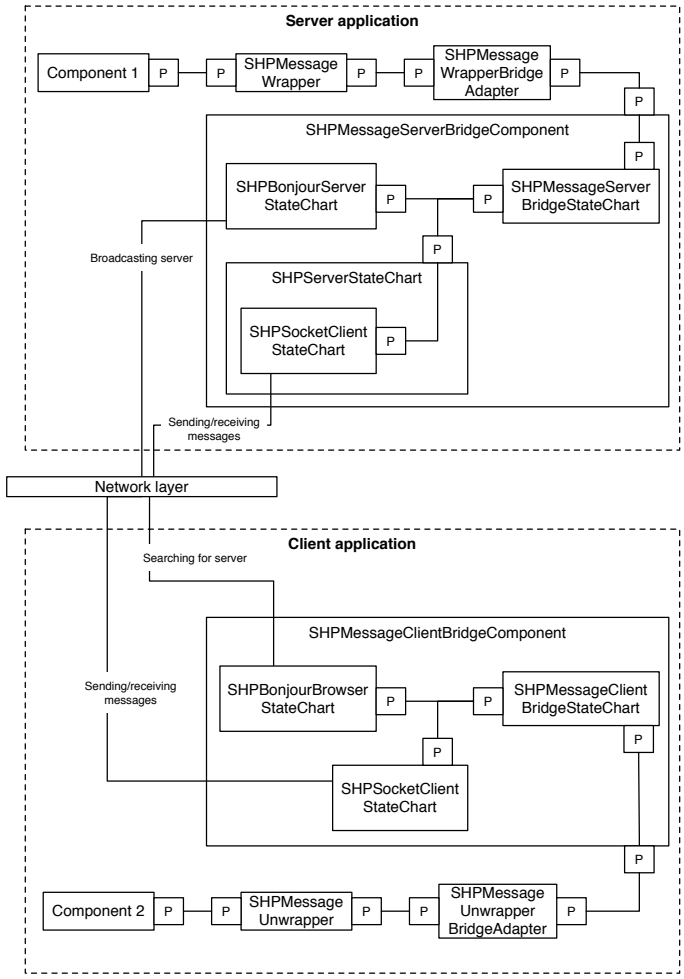

Figure 6.10: Messaging bridge.

An interesting feature allowed by the loosely coupled channel system is bridging between applications and even devices, as discussed in Section 5.2.7. This has been implemented as a part of the runtime system. It creates the possibility of transparently creating a channel between two components, which are running on different devices, on the same network. It has been implemented in a way that allows bridging between iOS devices, but also bridging to Mac OSX. For instance this makes it possible to easily build useful debugging tools running on OSX, that live inspect an

application running on iOS.

The implementation of the messaging bridge itself has been done using the core framework and the third party frameworks *CocoaAsyncSocket*(Hanson, 2014) for creating socket connections and *AutoCoding*(Lockwood, 2014) for encoding and decoding objects. The message bridge has been implemented using separated components that communicate using the messaging system. Many of the components are implemented as statecharts, because of their complexity. Overall the message bridge consists of two parts, a server part and a client part. In order to use it, one application must use the server part. Many applications can act as clients connecting to the same server. Once a connection has been established between a server and a client, messages can be sent both ways. The server application creates a socket server, which is broadcasted over Bonjour[3]. Client applications automatically connect to the server once discovered. The overall implementation of the system is illustrated in Figure 6.10. It shows how a component called `Component 1` located in an application acting as the server can be connected to a component called `Component 2` located in another application acting as a client. The only requirement is that the devices, in which the two applications are running, are connected to the same network.

It is important to emphasize that `Component 1` and `Component 2` are unaware and completely independent of the fact that they are running within different applications.

Configuring the server

Listing 6.8 shows how to configure an application as the server of the message bridge. In the example a component called `component1` is being put on the sending end of the channel, which is bridged over the network.

Listing 6.8: Configuring a message bridge server inside an application.

```
1  // Message bridge server
2  SHPMessageWrapperBridgeAdapter *serverAdapter =
3          [SHPMessageWrapperBridgeAdapter new];
4  SHPMessageServerBridgeComponent *messageServerBridgeComponent =
5          [[SHPMessageServerBridgeComponent alloc] init];
6  SHPMessageWrapperComponent *channelWrapper =
7          [[SHPMessageWrapperComponent alloc]
8                  initWithChannelName:@"myChannelIdentifier"];
9
10 // Channels
```

[3]A Bonjour service is a way of broadcasting over the network that a service is available

```
component1.outputPort.channel =
        channelWrapper.inputPort.channel =
        [[SHPMessageRouter sharedInstance] publishSubscribeChannel];
channelWrapper.outputPort.channel =
        serverAdapter.inputPort.channel =
        [[SHPMessageRouter sharedInstance] publishSubscribeChannel];
serverAdapter.outputPort.channel =
        messageServerBridgeComponent.messagesInputPort.channel =
        [[SHPMessageRouter sharedInstance] publishSubscribeChannel];
```

The `SHPMessageWrapperComponent` is in charge of wrapping messages being sent into `SHPWrappedMessage` instances. The `SHPWrappedMessage` simply contains the message together with a channel identifier. The channel identifier is used to make it possible to use the same socket connection to setup multiple channels between two applications. In Listing 6.8 it has the value `myChannelIdentifier`.

Configuring a client

How to configure the client part of a messaging bridge inside an application is shown in Listing 6.9. In this case it is connected to a component called `component2`, which is put on the receiving end of the channel, bridged over the network.

Listing 6.9: Configuring a message bridge client inside an application.

```
// Message bridge client
SHPMessageUnwrapperComponent *unwrapper =
        [[SHPMessageUnwrapperComponent alloc]
                initWithProtocol:@protocol(SHPStateChartInfoProtocol)
                        channelName:@"myChannelIdentifier"];
SHPMessageBridgeUnwrapperAdapter *adapter =
        [SHPMessageBridgeUnwrapperAdapter new];
SHPMessageClientBridgeComponent *messageClientBridgeComponent =
        [[SHPMessageClientBridgeComponent alloc] init];

// Channels
messageClientBridgeComponent.messagesOutputPort.channel =
        adapter.inputPort.channel =
        [[SHPMessageRouter sharedInstance] publishSubscribeChannel];
adapter.outputPort.channel =
        unwrapper.inputPort.channel =
        [[SHPMessageRouter sharedInstance] publishSubscribeChannel];
unwrapper.outputPort.channel =
```

```
19          component2.inputPort.channel =
20          [[SHPMessageRouter sharedInstance] publishSubscribeChannel];
```

Notice how the same channel name is used as in the server application from Listing 6.8. This means that messages sent by `component1` will be received by `component2`.

Limitations

The message bridge implementation has a few limitations that are important to notice. First of all only bridging between iOS and OSX is possible. This comes from the fact that the data sent over the network is serialized `SHPEventMessage` instances. These can only be deserialized on iOS and OSX. Further the content of the messages sent must be objects. Simple types are not supported by the serialization mechanism. Objective-C contains object wrappers for all simple types, so these should be used instead of simple types in order to take advantage of the messaging bridge.

6.5.3 Message tracing

A way of inspecting the messages being sent in an application, as explained in Section 5.4.2, has been implemented. This has been done in the `SHPMessageTracingComponent` class. The `SHPMessageRouter` singleton has a port called `monitorPort`, where all messages processed are sent to for other components to monitor. An instance of `SHPMessageTracingComponent` is supposed to be connected to this port. As messages are being sent between components, the message-tracing component will keep track of the traces.

6.5.4 Statechart event handling recording

To keep track of statechart changes and allow other components to monitor these, a recording mechanism has been implemented and added to the statechart event handling. Messages are sent to a port called `stateChartInfoPort` on the statechart in order to provide information about how an incoming event was handled. Two kinds of messages are sent on the port. First of all, an instance of `SHPEventHandlingResult` is sent when an event has been handled. It provides information about which event occurred, the transitions triggered and which states were exited and entered. Further, an instance of `SHPStateChartRepresentation` is sent to the port. It contains a description of the statechart hierarchy, including the name of each state and whether the state is part of the active state configuration or not.

This information can be used for various tools to inspect and debug a statechart. Knowing how the statechart executes helps getting an understanding of the statechart and makes it easier to discover bugs.

6.6 Prototype application

The implementation of the message bridge system explained in Section 6.5.2, uses many aspects of the implemented framework in order to offer a bridge for sending messages. However, it does not deal with the challenges of modeling GUI applications. For this reason a prototype application has been developed that uses the implemented framework. The prototype is implemented according to the specification from Section 1.4.1. The overall elements of the prototype are presented in the following sections, using figures showing how it has been implemented. The full source code for the prototype application is available in the included resources as explained in Appendix D.

6.6.1 Overall structure

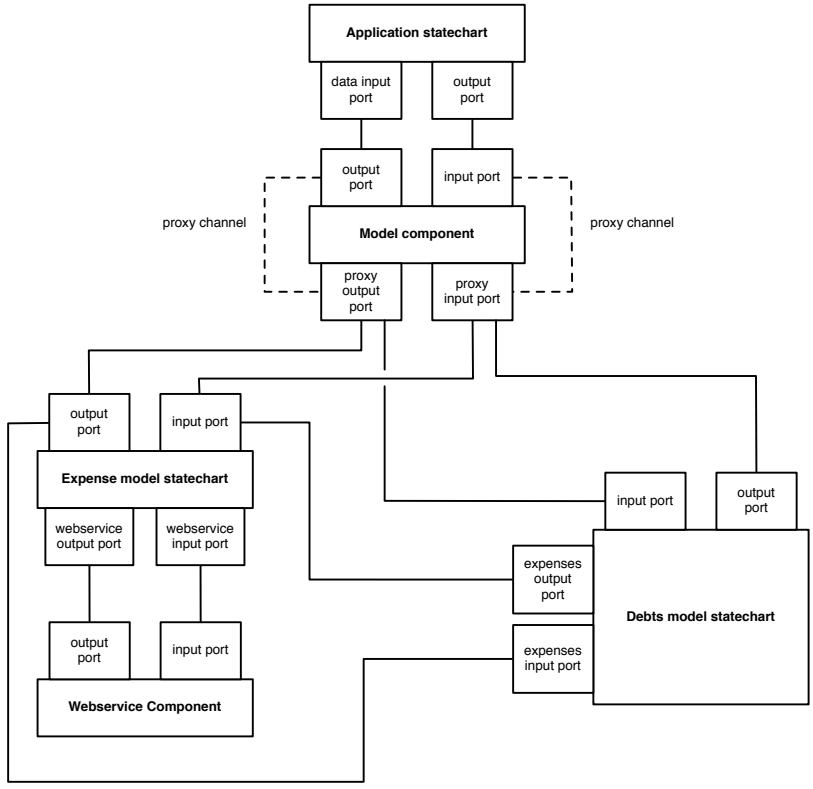

Figure 6.11: Overall structure of the implemented prototype application.

The behavior of the application has been divided into components. The user interface has been modeled in a single statechart component. The different parts of the model layer of the application have been separated in its own components. The overall structure is shown in Figure 6.11.

In total the prototype consists of five components of which three are statecharts. Each component abstracts away details and encapsulates its own behavior as discussed further below.

6.6.2 User interface

The user interface is modeled in a single statechart called `ApplicationStatechart` as shown in Figure 6.12. However, because of the complexity of the user interface, the concept of submachines is used heavily in order to separate the implementation into smaller parts. For instance the `ApplicationStatechart` uses the submachines `ExpensesNavigationControllerStateChart`, `DebtsStateChart` and `TabBarStateChart`. These are approached further down.

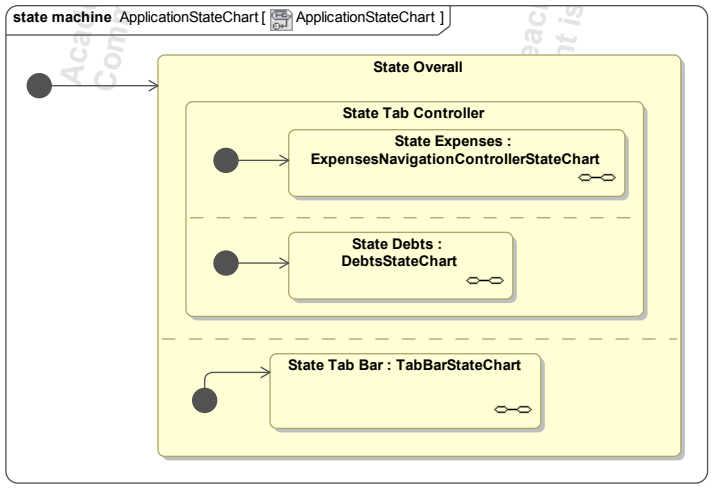

Figure 6.12: Application statechart with submachines.

Tab bar

The main navigation in the prototype application is built around a `UITabBarController`(Apple, 2013b). This is a standard way for iOS applications to provide basic navigation between the views. The tab bar allows the user to select between two tabs,

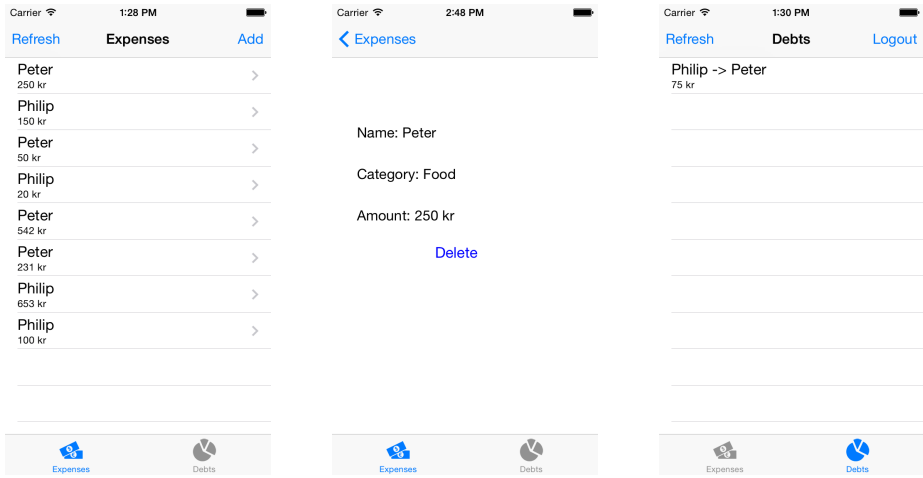

Figure 6.13: Screenshots of expenses tab selected(left), debts tab selected(right) and specific expense selected and details shown using navigation controller(middle).

one for expenses and one for the debts. This is illustrated in Figure 6.13. Only one tab can be selected at a time, however when a tab is not selected, the view representing the tab is still alive even though it is not visible. In order to keep a view active even though it is not visible the two submachines `ExpensesNavigationControllerStateChart` and `DebtsStateChart`, representing the two tabs, are added to two different regions as shown in Figure 6.12.

In order to keep track of which tab is selected another submachine is used, namely `TabBarStateChart` as shown in Figure 6.14, where the `OR state` called `State Tab Controller` changes between `State Tab Expenses` and `State Tab Debts` according to the selected tab events.

Navigation bar

The `UINavigationController`(Apple, 2013a) is another common control for providing navigation on iOS. It allows managing a stack of views, where new views can be pushed on top of other views. In the prototype it is used to go between the list of all expenses and details for an expense when the expenses tab is selected as shown on Figure 6.13(left and middle). This control has been modeled in Figure 6.15,

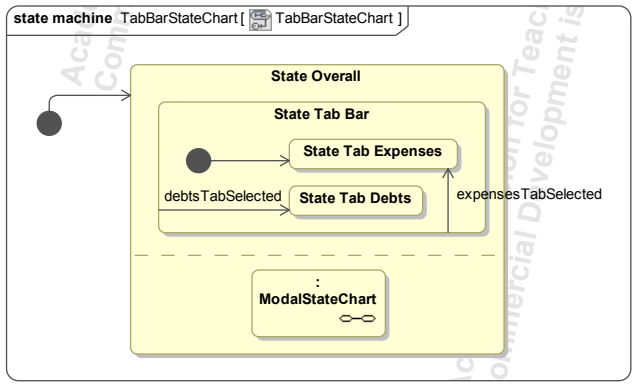

Figure 6.14: Tab bar statechart.

where the views are modeled using the submachines `ExpensesListStateChart` and `ExpenseDetailStateChart`, which are added to their own regions resulting in them always being active. Further the currently selected view is modeled in its own region using the `State Navigation Bar`. Notice how the state `State Root Detail Animation` is used when going from `State Root View Controller` and `State Detail View Controller`, since the change between the two views is done using an animation.

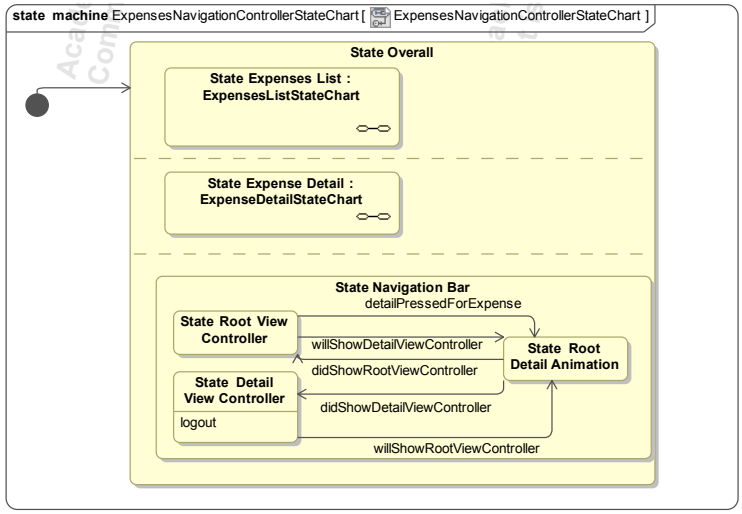

Figure 6.15: Expenses navigation controller stateChart.

Modal views

Modal views can be used as a way of presenting a view on top of all other views. This is often used in situations, which require the users attention before interacting with other views in the application(Apple, 2012b). In the prototype application this is used to present a view on top of the tab bar and the selected view. It is done by having a region in the `TabBarStateChart` submachine from Figure 6.14, which holds a submachine for this modal representation. It is used for two views. The login view showed when the user is not logged in, and the view for creating a new expense. The submachine is shown in Figure 6.16, where the same pattern as with the tab bar and navigation bar is used. One region is used to keep track of whether the login view is selected, the create expense view or none of them. Furthermore two regions are used one for each of the two submachines `AddExpenseStateChart` and `LoggedInState-Chart` in order to keep them active all the time.

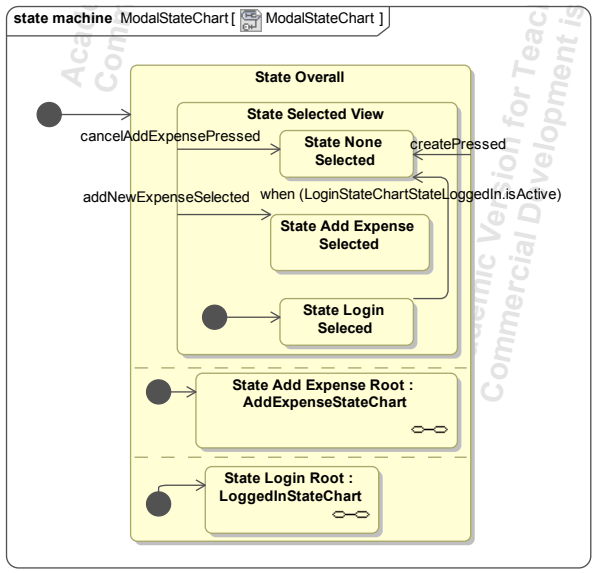

Figure 6.16: Modal statechart.

Logging in

The login flow as shown in Figure 6.17(right) allows the user to provide an email address and a password in order to login. It is an example of a view that is presented modally.

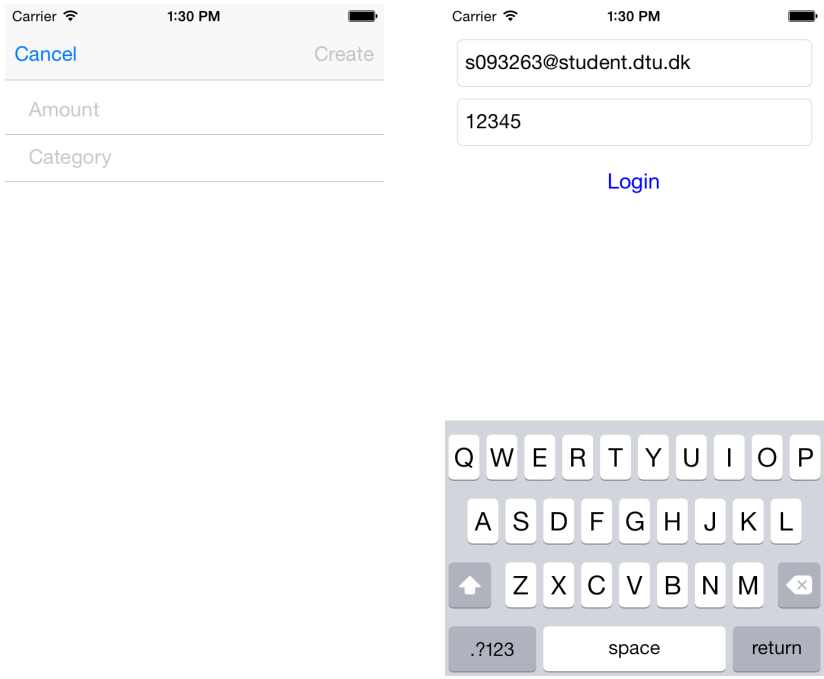

Figure 6.17: Screenshots of creating a new expense(left) and login(right)

The login button is only enabled when the application is in a certain state. This is the case when a valid email address has been provided, as well as a password with at least 5 characters. The statechart is used as a submachine for keeping track of the status, this is shown in Figure 6.18

In overall the `LoginStateChart` is either in the `Logged in` state or in the `Not logged in` state. The `Not logged in` state consists of 3 regions. The upper most region from Figure 6.18 represents the state of the provided email. It can either be valid or not valid. The same goes for the provided password. A transition from valid to not valid occurs automatically once the condition on the transition becomes true. The login button automatically becomes active, once both the `Email valid` state is active as well as the `Password valid` state. The event `login` makes a transition to the `Logged in` state if the values of the email and password correspond to the hardcoded values.

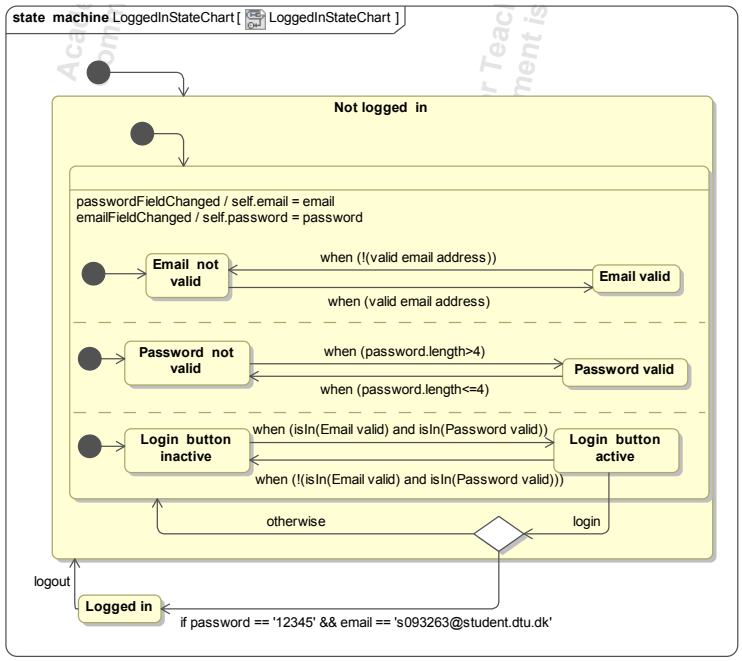

Figure 6.18: Login statechart.

List of expenses & debts

One of the main features of the prototype application is to see a list of expenses that have been added to the system as shown in Figure 6.13(left). The view consists of a simple list showing the expenses. As illustrated in Figure 6.19 the submachine consists of three states.

Another feature of the application is showing the debts in the system as illustrated in Figure 6.13(right). The submachine containing the state of the view is very similar to the one for showing the list of expenses. It is illustrated in Figure 6.20.

Notice how the `logout` event makes a transition from `State Showing Data` to `State Waiting`. This is done in order to remove all visible data when the user logs out.

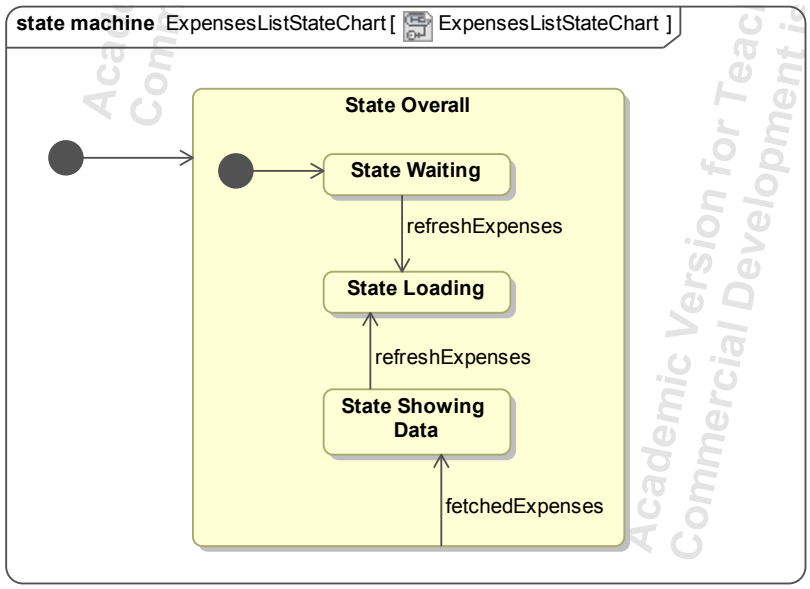

Figure 6.19: Expense list statechart.

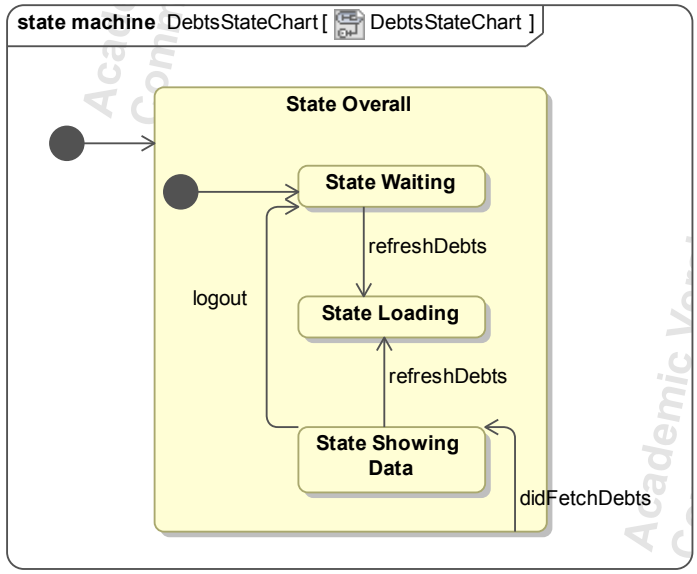

Figure 6.20: Debts list statechart.

Adding a new expense

The view for adding a new expense is shown in Figure 6.17(left). By specifying an amount and a category for the expense, it is possible for the user, who is logged in, to add a new expense. As with the login view, the input fields for adding a new expense have some requirements. As shown on the statechart in Figure 6.21 the provided amount must be larger than zero and the category must be at least three characters long.

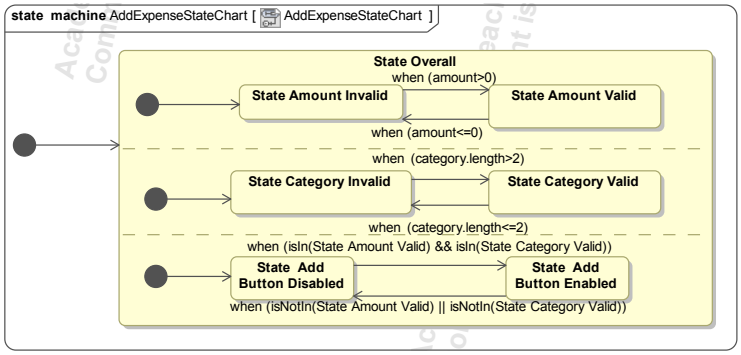

Figure 6.21: Add expense statechart.

6.6.3 Model

Another part of the application is the model. This is where the data logic of the application is placed. As shown in Figure 6.11 it consists of four components. The `Model Component` is used to interface with the `Application Statechart` and uses proxy ports to forward communication to the various model components. The `Expense model statechart` component is in charge of managing expenses. The protocols used for the input and the output ports are shown in Listing 6.10. The input protocol contains message types for fetching the expenses, deleting an expense and creating a new expense. The output protocol simply defines the method for notifying about new expenses.

Listing 6.10: Protocols used for the input and output port of the Expense model.

```
1 // input port protocol
2 @protocol ExpenseModelInputProtocol
3 - (void)fetchExpenses;
4 - (void)deleteExpense:(Expense *)expense;
```

```
- (void)createExpenseWithAmount:(NSNumber *)amount
        category:(NSString*)category;
@end

// output port protocol
@protocol ExpenseModelOutputProtocol
- (void)fetchedExpenses:(NSArray *)expenses;
@end
```

However as shown in Figure 6.11 the `Expense model statechart` has ports connected with channels to the `Webservice component`. This is because the `Webservice component` is in charge of communicating with the external web-service for deleting, creating and fetching expenses. In the prototype the `Webservice component` fakes the communication to the external web-service by maintaining data locally. This is done in order to ensure that the prototype is able to run without depending on an external web-service. The `Debts model statechart` provides the debts in the system. Since the debts can be calculated from the expenses, the `Debts model statechart` is connected to the `Expense model statechart` as illustrated in Figure 6.11.

6.7 Summary

The implementation of the framework has now been presented, by looking at how the concept of components has been introduced by the `SHPComponent` class. Further the messaging system has been developed by having a router implemented in the `SHPMessageRouter` class which routes message sent to `SHPChannel` instances that are connected to either `SHPInputPort` or `SHPOutputPort` instances. The statechart implementation has been covered by explaining how to define a statechart using the framework, by subclassing `SHPStateChart` and creating states by subclassing either `SHPOrState` or `SHPAndState`. Several implemented components and tools for inspecting the system have been presented, which together form a runtime that is usable for developing applications. Lastly, the implemented prototype has been explained, in order to show how the framework can be used in integration with the existing iOS SDK in order to model an iOS application, which takes advantage of the abstractions.

CHAPTER 7

Tests & Performance

The previous chapter covered the implementation of the framework. This chapter covers the testing of correctness and performance of the implementation. First the tests of the correctness of the statechart engine are presented. Next performance testing of the statechart implementation, with focus on memory usage and the throughput performance event, is processed. Finally the throughput of the messaging system is tested and compared to similar systems.

7.1 Statechart engine

The statechart engine implementation has been tested in order to make sure that it is executing correctly. All of these tests have been put together into a test project as explained in Appendix E. To test if a statechart is being executed correctly when an event occurs, the order of exit, action and enter of events is being recorded. A test is passed if the expected exit, action and enter calls are performed on the right states in the statechart and match the recorded ones. The expected result is found according to the rules of statecharts as explained in Section 4.3.9. For a detailed overview of the exact test cases, the `TestProject` in the Appendix E should be consulted.

- **Base orthogonal regions 1** - The handling of orthogonality has been tested using two statecharts. The first statechart is illustrated in Figure E.1. The source code is found in the `Scenario1StateChart` class in the `TestProject`. The purpose of the test is to verify that the `AND state` is entered correctly from an `OR state`. It is tested that when `event1` is fired, both state `State1b1` and `State1a1` are entered correctly. Further it is tested that a transition going outside of a region is being performed correctly. This happens when first `event1` is fired and then `event2`. Lastly it is tested that active regions are exited correctly when the `AND state` triggers a transition. This happens when `event1` and then `event3` are fired.

- **Base orthogonal regions 2** - The second statechart for testing orthogonality is illustrated in Figure E.2. The source code can be found in the `Scenario2StateChart` class in the `TestProject`. The purpose is to test that the transitions inside regions of an `AND state` occur correctly. This requires that all regions responding to an event result in the correct transition. This is the case when `event1` is fired, where two transitions should be triggered.

- **Merge transition** - The use of merge transitions has been tested in the statechart illustrated in Figure E.3. The source code is found in the `Scenario3StateChart` class in the `TestProject`. The purpose is to test that transitioning from two states in separate regions to another state resolves in a correct merge transition. This is tested by performing `event2`, which triggers a transition to `State2`. It requires both state `State1a1` and state `State1b1` to be exited correctly.

- **Fork transition** - Fork transitions have been tested using the statechart illustrated in Figure E.4. The source code can be found in the `Scenario4StateChart` class in the `TestProject`. The purpose is to test that transitioning to two states in separate regions from another state resolves in a correct fork transition. This is the case when `event2` is fired. It causes the statechart to enter `State1a2` and `State1b2`.

- **Internal transition** - Internal transitions have been tested using the statechart illustrated in Figure E.6.

- **Not responding event** - The purpose is to test that when an event is posted, where no states in the state configuration respond, nothing happens. The source code can be found in the `Scenario5StateChart` class in the `TestProject`. As illustrated in Figure E.5 this is the case when `event2` is posted to the statechart.

- **Self transitions** - Self transitions have been tested using the statechart illustrated in Figure E.6. The source code is found in the `Scenario6StateChart` class. When being in the state `State2a1` and `event2` occurs, a self transition to state `State2a1` should be performed. Further a special kind of self transition is being tested, where the state responding to the event is defined on a superstate of the leaf state, and performs a transition to the current leaf state. This is the case being in state `State2a1` and having `event1` occur.

- **Guarded transition** - The handling of guarded transitions has been tested. The statechart for this test is illustrated in Figure E.7. The source code is found in the `Scenario7StateChart` class in the `TestProject`. The purpose is to test that guarded transitions are handled correctly. It is tested that the initial transition results in `State Overall` being active. Further it is tested that when `StateA2`, `StateB2` and `StateC2` are active, a transition from `StateD1` to `StateD2` is automatically performed.

- **History transition** - The history mechanisms are tested. The statechart for this test is illustrated in Figure E.8. The source code is found in the `Scenario8StateChart` class in the `TestProject`. The purpose is to test that history transitions are handled correctly. It is tested that the last active substate of `State2` is being entered when a transition from `State1` is performed.

- **Submachine** - The use of submachines is tested and the statechart for this test is illustrated in Figure E.9. The source code can be found in the `Scenario9StateChart` class in the `TestProject`. The purpose is to test that submachines inside a statechart are executing correctly.

7.2 Statechart performance

The implementation of the statechart engine is complex and adds an overhead to the execution when used. A goal of the framework as explained in Section 1.4 has been to provide performance acceptable for practical usage. The most important factors are the memory usage and the event-processing throughput. For this reason these two factors have been tested. The example presented in Example 6 will be used to measure the memory usage and event processing performance of the statechart engine.

Example 6 *In order to test the statechart performance a small example of a statechart has been developed called* ***TikTok****. The project is available in the resources. It is illustrated in Figure 7.1. When the initial transition has been performed, the statechart will enter the state* ***Start*** *and automatically make a transition to the state* ***TikTok****. It will further enter the leaf state* ***Tik****. The entry action of state* ***Tik*** *will perform the event* ***performTik****, which will trigger a transition to state* ***Tok****. In the entry action of state* ***Tok*** *the event* ***performTok*** *is triggered, which causes a transition to state* ***Tik*** *again. In the action of* ***performTok*** *the extended state variable* ***numberOfToks*** *will be increased by one before. In the entry of state* ***Tik*** *a transition to state* ***Tok*** *is triggered again. This continues until the guarded transition on state* ***TikTok*** *becomes true and a transition to state* ***Done*** *is performed. The guard becomes true when the value of* ***numberOfToks*** *has become equal to a predefined* ***goal*** *variable.*

7.2.1 Memory usage

During the execution of the statechart it is important that the memory consumption is kept stable. Having spikes in memory usage in iOS causes memory warnings and the result of this may be that the system terminates the application (Apple, 2014a). A project has been created implementing Example 6 as explained further on Appendix F. It is used to measure the memory usage using `Instruments`[4] with the variable `goal` set to 100000. The result is shown in Table 7.2.

[4]Instruments is a profiling tool included as a part of the iOS SDK.

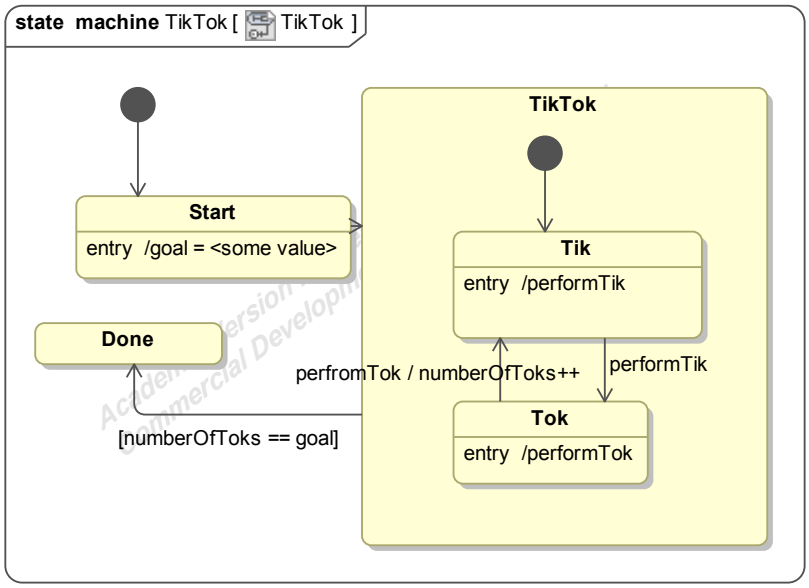

Figure 7.1: TikTok statechart suitable for testing statechart performance

Seconds	Live bytes	#Live objects	Overall bytes	#Overall objects
5	1.31 mb	14534	19.69 mb	542256
10	1.30 mb	14539	45.50 mb	1343524
15	1.30 mb	14472	73.08 mb	2198530

Table 7.2: Memory consumption over time.

As the statechart is being executed the number of overall created objects rises, which also results in an increase in the overall amount of bytes allocated. However the live bytes in the memory stays constant, as the statechart is constantly releasing old memory. This happens because of an auto release pool that has been set up around the event processing of the statechart. Doing so has a performance consequence as cleaning up memory is a costly operation, however it keeps the memory usage low and peaks are avoided.

7.2.2 Event processing

An interesting performance measure for the statechart is how fast it can process incoming events. The `TikTok` statechart explained in Example 6 is used to measure this using the implementation in the project from Appendix F. In the entry action of the state `TikTok` the current time is stored as an extended state and in state `Done` the

event processing performance is the measured. The test was performed on an iPhone 5S. The results are listed in Table 7.3

Goal	Seconds	Messages per second
100	0.09	1106
1000	0.23	4192
5000	0.87	5720
10000	1.60	6234
50000	7.69	7646
100000	13.07	7648
200000	25.52	7834

Table 7.3: Event processing for different goals.

Further the results are plotted in Figure 7.4. As the figure illustrates the performance stabilizes around a `goal` above 50000 messages, resulting in a bit more than 7500 messages per second.

The statechart clearly introduces an overhead in performance compared to trying to capture the state using simpler methods as discussed in Section 3.3.1. Running a profiling with `Instruments` on the `TikTok` example with the `goal` variable equal to 100.000, reveals in which methods the most time is spent. The results have been mapped to tasks and it is presented in Table 7.5.

Task	Time in percent
Enter target state	19.8%
Releasing memory	17.6%
Finding lowest common ancestor	15.7%
Check transition condition	12.7%
Perform state action	11.7%
Creating message identifiers	7.4%
Other	15.3%

Table 7.5: Percent of time spent doing different tasks.

The traversal of the states in order to find the LCA is the most time consuming task. As many objects are being created and released during the execution. Also

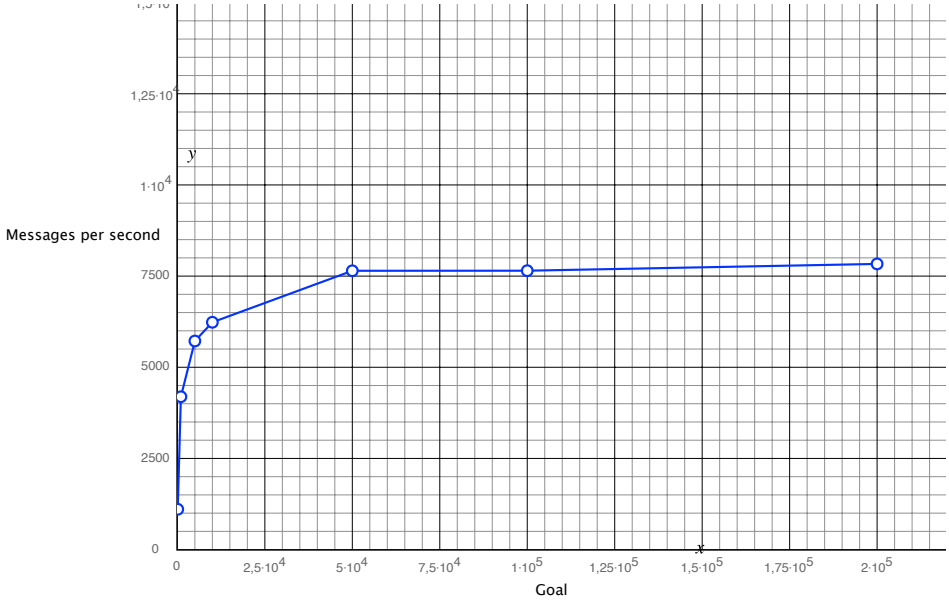

Figure 7.4: Plot of messages for event processing per seconds with different values of goal.

memory management by the system has a big impact on the performance. A surprising discovery is the noticeable time spent on creating the UUIDs which are the unique identifiers of the messages sent between the components.

7.3 Messaging system throughput

So far the focus has been on the performance of the statechart implementation. As an application may consist of many components communicating with each other, it is important that the messaging system delivers an acceptable performance. The throughput between the components may be a bottleneck in the system. As presented in Section 5.2.5, the system uses a single central router where all the traffic passes through. If the throughput is too low, messages will get queued up and eventually it could cause the application to run out of memory. It is important to notice that performance optimization has not been a main goal for the messaging system, but instead it is designed to give advantages for inspection and debugging, as discussed in Section 5.4. Example 7 is used to measure the throughput of the messaging system.

Example 7 *In order to measure the throughput performance of the messaging system, the example project* `PingPong` *has been created. The project is available in the resources. The project consists of two components called* `Component1` *and* `Component2`*. They are connected to each other as illustrated in Figure 7.6. When* `start` *is called on* `Component1`*, it will measure the current time before sending the message* `ping` *out on its output port. This port is connected to the input port of* `Component2`*, using a* `SHPChannel` *going through the* `SHPMessageRouter` *singleton. When receiving a* `ping` *message,* `Component2` *will the send a* `pong` *message on its output port. This port is connected to* `Component1`*s input port. Every time* `Component1` *receives a* `pong` *message it will decrease the value of a variable called* `goal`*. The communication will continue until the* `goal` *variable is zero and the execution time will be recorded.*

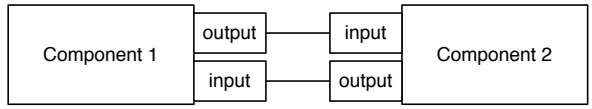

Figure 7.6: Showing how `Component1` and `Component2` are connected in Example 7.

The throughput test was performed on an iPhone 5S and the results are shown in Table 7.7 for different values of the `goal` variable.

Goal	Seconds	Messages per second
100	0.02	5366
1000	0.09	11467
5000	0.39	12850
10000	0.76	13055
50000	3.90	12822
100000	7.85	12733
200000	15.45	12941

Table 7.7: Throughput for channel system with different goals.

The messages sent per second stabilize around the initial goal variable having the value of 50000, which gives a throughput of about 13000 messages, as illustrated in Figure 7.8.

In order to compare the results, a similar test was performed with the normal method invocation in Objective-C. Instead of having two components sending messages to each

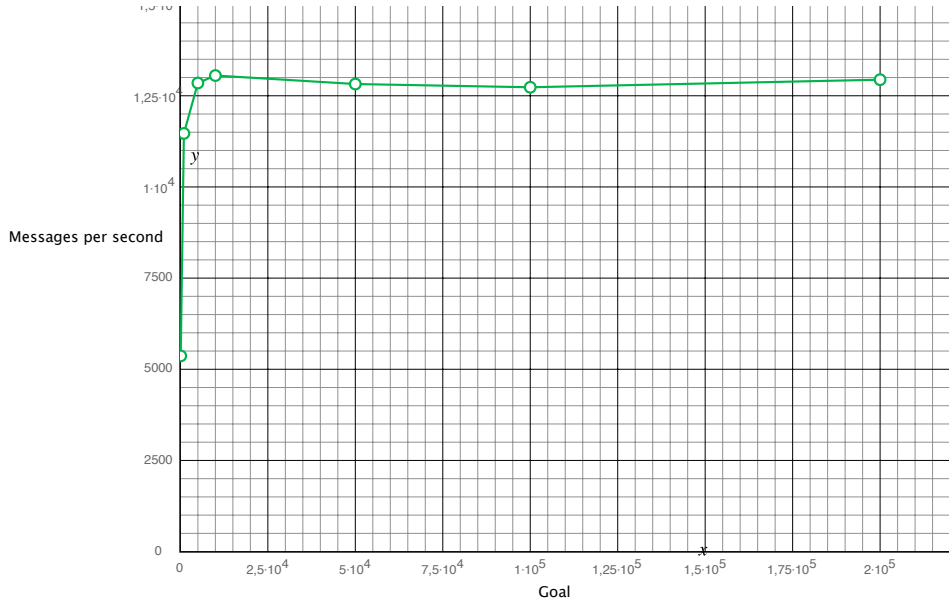

Figure 7.8: Plot of messages per seconds for message throughput with different values of goal.

other, two normal objects were created that called methods directly on each other. Method invocation is highly optimized for performance. The results are shown in Table 7.9

Goal	Seconds	Messages per second
100	0.004	22364
1000	0.02	55766
5000	0.05	104248
10000	0.09	112072
50000	0.45	109700
100000	0.83	120368
200000	1.59	121894

Table 7.9: Throughput for Objective-C method invocation with different goals.

As illustrated in Figure 7.10, method invocation is approximately 10 times faster than the implemented messaging system with more than 100.000 messages per second for 5000 messages. Which, as expected shows that the messaging system introduces

much overhead compared to plain method invocation.

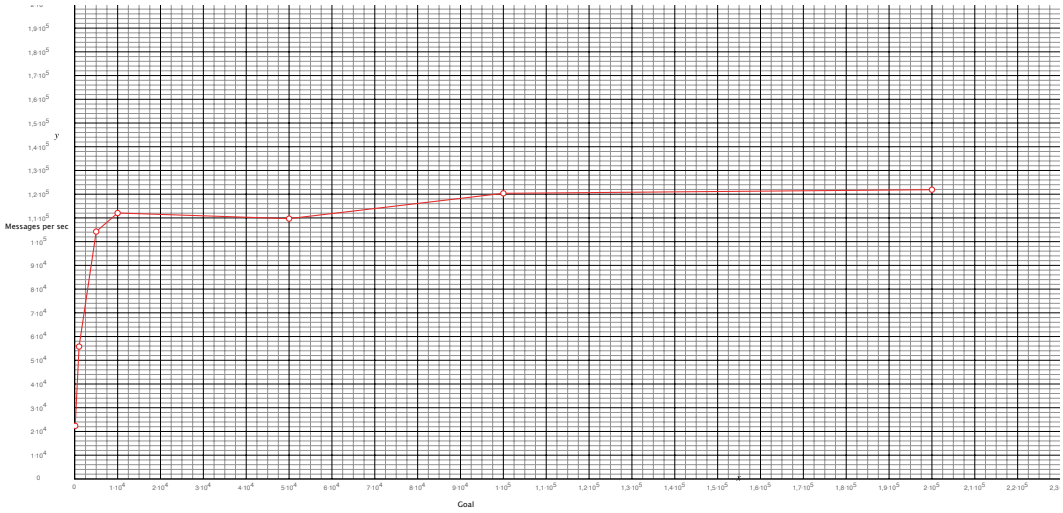

Figure 7.10: Plot of messages per seconds for objective-C method invocation with different values of goal.

In order to determine whenever the performance of the messaging system is useable in a real life application, a comparison of the performance with an existing and popular framework has been performed. The third party library `ReactiveCocoa`[5], which implements the use of the *Reactive Programming Model* as discussed in Section 3.3.5, uses signals as a way of reacting as events occur. This can be used to model the `PingPong` example described above. It is done by having two objects that each send out a signal when they receive a signal until a goal is reached. The results of such a program running on an iPhone 5S are shown in Table 7.11

[5]ReactiveCocoa is a popular Objective-C framework. https://github.com/ReactiveCocoa/ReactiveCocoa

Goal	Seconds	Messages per second
100	0.02	3752
1000	0.11	8682
5000	0.56	8912
10000	1.03	9646
50000	4.80	10410
100000	9.65	10360
200000	19.11	10464

Table 7.11: Throughput for Reactive Cocoa with different goals.

As illustrated in Figure 7.12 the messages per second is a bit lower than using the implemented messaging system with around 10.000 messages per second for 5000 messages.

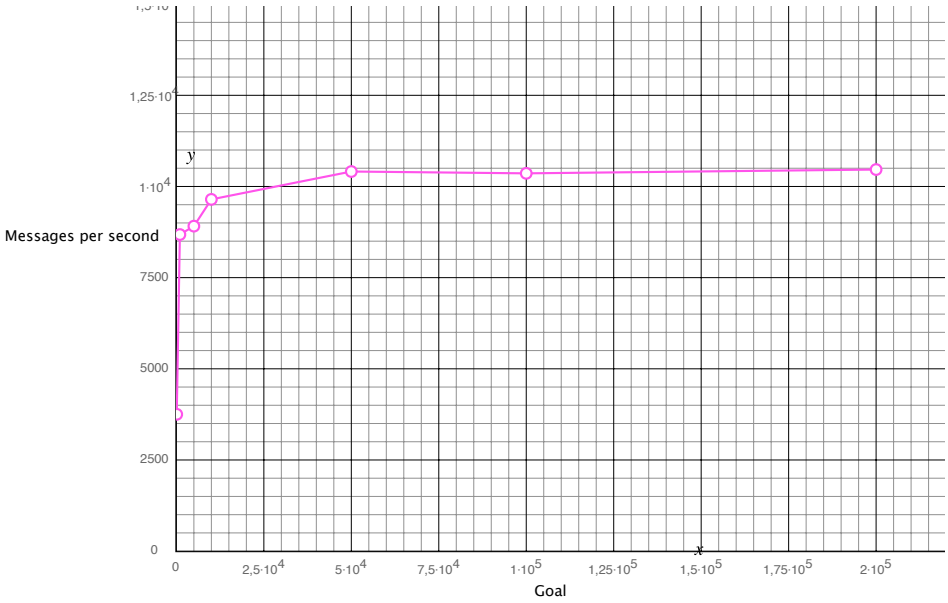

Figure 7.12: Plot of messages per seconds for ReactiveCocoa with different values of goals.

The implementation of all of the above throughput tests can be found in the `ComponentThroughput` project in the resources as explained on Appendix G

7.4 Summary

The chapter has covered the testing of the implementation. First the correctness of the statechart engine was tested during various test cases. Then the performance of the statechart implementation was tested by first considering the memory usage. This turned out to be kept constantly low over time as memory was cleaned during the processing of events. Next the performance of the statechart event processing was tested and it was found that for a simple statechart, the implementation is capable of processing almost 8000 events per second. Finally the throughput of the messaging system was tested and it was found that it could process almost 13000 messages a second.

CHAPTER 8

Discussion

The presentation of the implementation in Chapter 6 mostly focussed on what the framework offers. In this chapter the original goals are evaluated by discussing how to model large applications while keeping them convenient for the developer. Further it is discussed how the designed system can be used in order to provide new ways of handling errors. Lastly it will be talked about how the design and the implementation can be ported to other platforms that face similar challenges.

8.1 Modeling large applications

One of the initial goals for the developed framework was that it could be used to model large complex iOS applications. The prototype, which was explained in Section 6.6, showed how to use the framework to model an iOS application. Further it was integrated with existing frameworks and provided a very typical overall user interface navigation. While the prototype is constrained in its functionality, it shows that using the framework to model iOS applications is possible. Modeling larger applications is simply accomplished by creating more components for modeling the data logic of the application and expanding the statecharts representing the user interface with more submachines.

Moreover, the performance testing results from Chapter 7 show that the framework allows building an application consisting of many components, while keeping the memory footprint low and providing good performance. This is both the case for when many messages are sent in the system, as well as for when a statechart needs to process many events. This means that the implemented framework can be used for modeling large applications consisting of hundreds of components, and communicating with up to 13.000 messages a second between each other.

8.1.1 Dynamic behavior

As a large application is executed, depending on the input, the parts of the application being used might be dynamic. This can be modeled using the framework by having components being allocated and deallocated dynamically according to the needs. However within a single statechart, there is limited support for modeling this kind for dynamic behavior. This is because the state and the hierarchy of the statechart are determined at compile time and it does not change during execution. The

statechart notation does not define how to deal with this kind of dynamic behavior. In many cases it can be solved by having some extended states that keep track of the dynamic part, however this is not always a suitable solution. Allowing the states in a statechart to be defined dynamically complicates the model dramatically. However, a solution could be to allow the number of independent regions of a `AND state` to be dynamic during execution. By allowing this, dynamic behavior can be achieved inside of a statechart.

8.2 Implementation overhead

In order to make the framework convenient to work with, it is important that as much implementation overhead is removed as possible. Common tasks, such as creating a new statechart, defining the states and the hierarchy and setting up the channels between components, should be as convenient and easy as possible. Because of this, convenient methods have been defined to help doing common tasks. Below it is discussed how these are used to remove implementation overhead, when using the framework.

8.2.1 Transitions

An example of where efforts have been put in order to make it understandable and convenient is, when defining transitions between states. Instead of having to create an instance of `SHPStateChartTransition` and add it to the state, as shown in Listing 8.1, a much more convenient way of doing it is offered by simply calling a method as shown in Listing 8.2.

Listing 8.1: Complicated way of adding transition.

```
1    SHPStateChartTransition *transition =
2                [[SHPStateChartTransition alloc]
3                initWithEvent:@selector(someEvent)
4                fromStates:@[state1] toStates:@[state2]
5                timeout:nil internal:nil guard:nil
6                historyMode:HistoryModeNone];
7    [state1 addEventTransition:transition];
```

Listing 8.2: Convenient way of adding transition.

```
1 [state1 onEvent:@selector(someEvent) transitionTo:state2];
```

8.2.2 State names

As explained in Section 6.3.1 defining a state for a statechart is done by creating a class. This class must be a subclass of one of the two base state classes `SHPOrState` and `SHPAndState`. However, this introduces a practical problem when applied in languages that do not support namespaces. This is the case for Objective-C. Two classes cannot be defined with the same name in Objective-C. An Application consisting of many statecharts is likely to have two states with the same name. This means that the classes need to be prefixed. An obvious solution is to prefix them with the name of the statechart they are being used in as illustrated in Listing 8.3, where the `StateStart` state is prefixed with `MyStateChart`.

Listing 8.3: Defining state with prefix

```
1 @interface MyStateChartStateStart : OrState
2 @end
3 @implementation MyStateChartStateStart
4 @end
```

However having to prefix all state classes is complicated and inconvenient to work with. Luckily Objective-C supports a way of defining an alias for a class using `@compatibility_alias <alias> <name>`. The use of this is shown in Listing 8.4, where the state `StateStart` is defined by making a class called `MyStateChartStateStart` and making an alias from `MyStateChartStateStart` to `StateStart`. Further, in order to get the short name of the state, the method `name` is implemented, which returns the name of the state.

Listing 8.4: Defining a state using `@compatibility_alias`

```
1  @interface MyStateChartStateStart : OrState
2  @end
3  @compatibility_alias StateStart MyStateChartStateStart;
4  @implementation MyStateChartStateStart
5
6  - (NSString *)name {
7          return @"StateStopped";
8  }
9
10 @end
```

Defining states this way makes it possible to use the name `StateStart` within the `MyStateChart` class. However, this solution puts overhead on the implementation. For this reason a macro has been created for easily defining states. It is used by writing `STATE(prefix, statename)`. Listing 8.5 shows how to define the same `StateStart` state with the macro.

Listing 8.5: Macro used for defining state

```
1 STATE(MyStateChart, StateStart)STATE_END
```

A macro for defining an `AND state` exists as well using `AND_STATE(prefix, state-name)`.

8.2.3 Setting up channels

The introduction of components having ports and channels between them requires the need to setup channels. As shown in Listing 8.6, it introduces much overhead to first create a channel that goes through the router and then hooking that channel up between the two ports, defined on two components.

Listing 8.6: Defining a channel between two ports.

```
1 SHPMessageRouter *router = [SHPMessageRouter sharedInstance];
2 SHPChannel *channel = [[SHPChannel alloc]
3         initWithType:SHPChannelTypePublishSubscribe
4         messageRouter:router];
5 [router createMessageQueueForChannel:channel];
6 component1.outputPort.channel = channel;
7 component2.inputPort.channel = channel;
```

In order to make the channel system easier to setup, convenient methods have been created for connecting two port with a channel as shown in Listing 8.7, where a `publish-subsribe` channel is created between the output port of `component1` and the input port of `component2`.

Listing 8.7: Convenient way of defining a channel between two ports.

```
1 [[SHPMesageRouter sharedInstance] addPublish-
2         SubscribeChannelFromOutputPort:component1.outputPort
3         toInputPort:component2.inputPort];
```

A similar method exists for creating `point-to-point` channels.

8.3 Error handling

The implemented messaging system provides a way for components to asynchronously communicate with each other by sending messages. The state of each component can be captured using abstract states in a statechart, however out of the box, the proposed solution does not specify how errors are handled. As errors occur the application should handle these in a controlled manner.

A common solution to handle unexpected errors is to terminate the application. When using exceptions for handling errors, the behavior in many languages, including Objective-C, is to terminate the application when an uncatched exception occurs. This is usually the desired solution, since trying to recover from an unexpected error may cause data to get corrupted or calls undesired behavior. However with an application consisting of concurrent components, that share no memory, it is possible to only terminate the part of the application where the error occurred. Handling the error in the component where the error occurred has the same challenges as handling an error in an application where all parts share memory. Nonetheless, terminating only the component and letting another healthy component handle the error, simplifies the problem. This is known as *"remote detection and handling of errors."*(Armstrong, 2013, p. 197). The error is not handled in the component where it occurs. Instead the component is terminated and another component is notified in order to recover. If the notified component is not able to recover, it will simply be terminated as well, resulting in another component being notified. When a component is reached that is able to recover from the error, it can be done by creating fresh copies of the terminated components. The idea is illustrated in Figure 8.1, where the `Web Service Component` receives an unexpected error. It then notifies the `Expense Model` component and then terminates it. Since the `Expense Model` component is unable to recover from the situation it also terminates, and notifies the `Model Component`. The `Model Component` is able to recover from the situation and does so by creating a new instance of `Expense Model`, which causes a new instance of `Webservice Component` to be created.

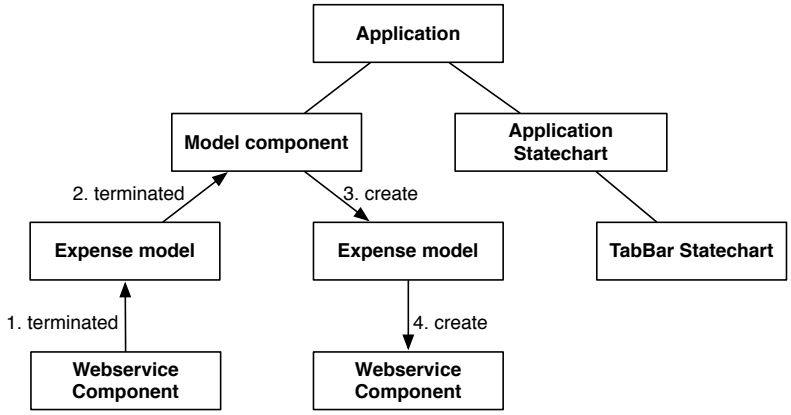

Figure 8.1: Error handling by terminating components and creating new ones.

By taking advantage of the fact that an application is structured into independent components, it is possible to provide an advanced error handling system, where each component can fail without the entire application to be terminated.

8.4 Porting to other platforms

The implemented framework introduces new concepts and ways of building applications for iOS. However, since many similar platforms exist that face the same challenges, it makes sense to look at the possibilities for porting it to other platforms. Obvious candidates are other mobile platforms such as Android or Windows Phone, but also other desktop platforms such as Linux and Windows.

Most of the implementation can be easily ported to another object-oriented language. As explained in Section 6.1 the implementation uses `Dispatch Queues` to use the concept of a shared thread pool to provide a context for each component. Porting the concept of components to another platform, would require implementing a shared thread pool system, if not already offered by existing frameworks on the platform.

The messaging system takes advantage of a special behavior for `NSProxy` subclasses, which as explained in Section 6.2.3 makes it possible to forward a method invocation. This means that any method can be called on a port, resulting in it being made into a message that is sent to the channel. The benefit of this approach is a very lightweight way of sending a message on a port, since it only requires a method call. Providing content with the message is done by simple adding arguments to the method. Achieving the same might not be possible in other languages. However, the implementation of the rest of the messaging system uses structures available in most other languages.

The implementation of the statechart engine is portable and a port to another object-oriented language could be done quite easily.

CHAPTER 9

Conclusion & future work

The thesis analyzed the challenges for architecting event-driven software for iOS. Based on the analysis, a design was proposed for developing applications. The design consisted of a method for dividing the application into components, where each component is running in its own context in order to remove dependencies between the components. Further the communication between components is done using a decoupled messaging system based on ports and channels. Lastly in order to keep track of the state inside a component, it can be modeled as a statechart. The implemented framework was used to implement a prototype application, in order to use it to model a complex iOS application. For testing the performance of the framework various tests were performed, which showed that the framework is suitable for architecting large applications. In overall the implemented solution suits as a foundation for architecting iOS applications by introducing new concepts. These concepts make it possible to create inspection and debugging tools in order to improve the development phase of an application and thus build applications with few bugs consisting of code that is maintainable.

9.1 Future work

The implemented methods used for architecting software provide a foundation, which opens up the door for many improvements and extensions. Below are listed some ideas for future work of the project.

- **Testing of statechart** - It is possible to test an application by looking at the behavior of it in different states. Statecharts introduce the concept of abstract states. Since the number of abstract states is fixed and for each state the transitions to other states is known, it is possible to test an application by forcing it into all possible states by sending events to the statechart. This can be used to create a way of automatically testing a statechart, with the goal of trying to force it into a particular state configuration, which should not be allowed. It can be done by creating a helper statechart that gets the same events as the statechart being tested. The helper statechart however, observes the other statechart and performs a transition to a fail state if the statechart

being tested ends up in an illegal state configuration. This is done in order to indicate that the test failed.

- **Debugging tools** - As discussed in Section 6.5 the introduced concepts make it possible to develop new tools for inspecting and easing the development of an application. This could include many different tools. For instance a tool to inspect the communication between components that allows pausing all communication or stepping through a trace of messages being sent, in order to debug the behavior.

- **Dynamic behavior** - As mentioned in Section 8.1.1 the statechart notation has limited support for dynamic behavior. As discussed further a solution could extend the notation with the possibility of dynamically defining regions to an `AND state`.

- **Contract on communication** - The messages sent between components in the system are in the implemented framework defined by a protocol put on the end points. However, this does not guarantee that the correct messages are sent at the correct times. For instance, if a component sends a message to another component and expects a message to be sent back as a reply, however the receiving component expects another kind of message first before being able to reply, the system ends up in a stuck state. An improvement could be to introduce the concept of a contract for the communication between components. Such a contract could be used as a definition for how components can communicate with each other.

- **Error handling** As discussed in Section 8.3 the introduction of components provides a way of handling errors by linking the components together and crashing a component once an error occurs, in order for another component to recover the system. By adding a mechanism for easily linking components a powerful way of handling errors can be archived.

- **Platform independent network bridge** - As discussed in Section 6.5.2 the message bridge has the limitations that objects being sent are encoded into a format that can only be decoded on either iOS or OSX. This means that it is not possible to bridge to other platforms. But instead using a more general format such as *JSON* a platform independent bridge can be archived.

- **Porting to more platforms** - As many platforms exist facing similar challenges, the framework can be ported as discussed in Section 8.4.

The findings in the thesis were found profitable for Shape A/S, hence the conceptual work, for several of the above mentioned additions have already been started in order to further improve the way software for iOS can be architected.

APPENDIX A

Thread memory consumption test

In order to run the project, Xcode 5 or newer must be installed. The memory consumption on iOS has been tested with the example project `DispatchQueueComponentTest`.

The project is included in `Projects/DispatchQueueComponentTest` in the resources. The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

A.1 Running the project

Simply open the `ThreadMemoryTest.xcodeproj` project file in Xcode or AppCode.

```
$ open ThreadMemoryTest.xcodeproj
```

In order to specify the number of threads created in the test open `AppDelegate.m`. Change the value of `numberOfThreads`

```
1 // Specify number of threads that should be created
2 // ************************************************
3 NSUInteger numberOfThreads = 250;
4 // ************************************************
```

The memory usage is printed in the console every 10th second.

```
1 Memory used 163086.3 kb (+163086 kb), free 49139.7 kb
```

APPENDIX B

Framework source

The framework can run on iOS7 or higher and OSX 10.9 or higher. The implemented sources are available in the `Source` folder in the resources.

The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

A *CocoaPod* `podspec` file has been included, which makes it possible to use the code by adding the following to the `podfile`.

```
pod 'SHPStateChart', :path => '<path to folder with SHPStateChart.podspec>'
```

The prototype application explained in Section 6.6 and available from Appendix D is a great resource for understanding how to use the framework.

APPENDIX C

Dispatch queue component test

In order to run the project, Xcode 5 or newer must be installed. The maximum size of the dispatch queues provided as a part of the iOS SDK has been tested with the example project `DispatchQueueComponentTest`.

The project is included in `Projects/DispatchQueueComponentTest` in the resources. The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

C.1 Running the project

Simply open the `DispatchQueueComponentTest.xcodeproj` project file in Xcode or AppCode.

```
$ open DispatchQueueComponentTest.xcodeproj
```

In order to specify the number of components created in the test open `AppDelegate.m`. Change the value of `numberOfComponents`

```
1 // Specify number of components that should be created
2 // ************************************************
3 NSUInteger numberOfComponents = 513;
4 // ************************************************
```

The result of the test is printed in the console.

```
1 max = 4.000778
2 min = 2.001259
```

APPENDIX D

Prototype application

In order to run the prototype application, Xcode 5 or newer must be installed. The source for the prototype application is available in the resources.

The project is included in the folder `Projects/ExpensesPrototype` in the resources. The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

D.1 Running the prototype

The project uses `CocoaPods`, which must be installed before running.

```
$ sudo gem install cocoapods
```

While in the root of the `ExpenesPrototype` project folder, install the depending pods.

```
$ pod install
```

Run the project by opening `ExpensesPrototype.xcworkspace` in Xcode or AppCode.

```
$ open ExpensesPrototype.xcworkspace
```

APPENDIX E

Test statechart engine

In order to run the project with the tests, Xcode 5 or newer must be installed. The test project is included in the project `TestProject`.

The project is included in `Projects/TestProject` in the resources. The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

E.1 Running the project

The project uses `CocoaPods`, which must be installed before running.

```
$ sudo gem install cocoapods
```

While in the root of the `TestProject` project folder, install the depending pods.

```
$ pod install
```

Simply open the `TestProject.xcworkspace` project file in Xcode or AppCode.

```
$ open TestProject.xcworkspace
```

The statecharts for the tests are listed below.

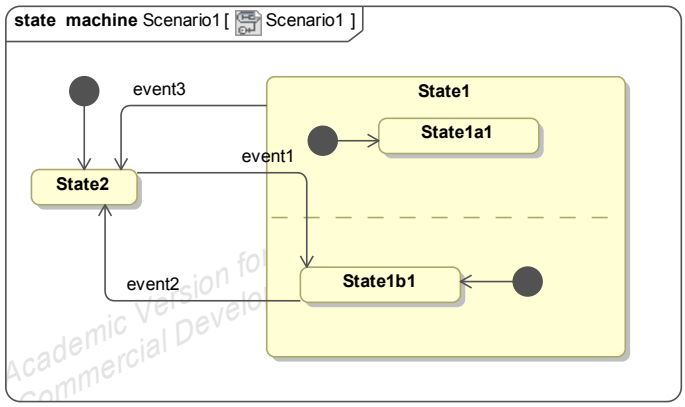

Figure E.1: Scenario1

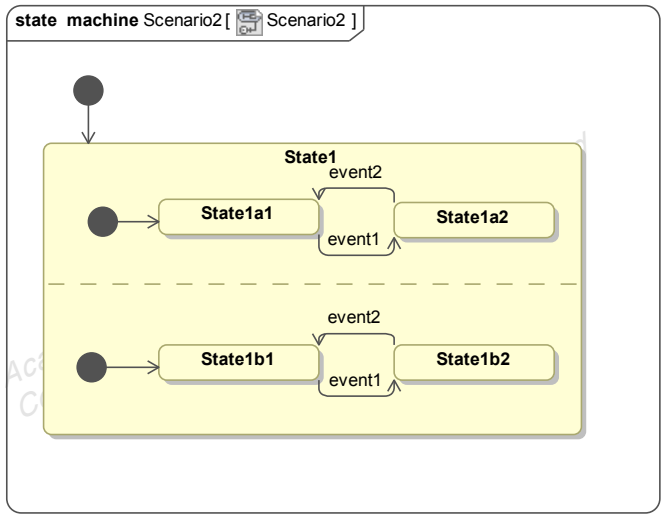

Figure E.2: Scenario2

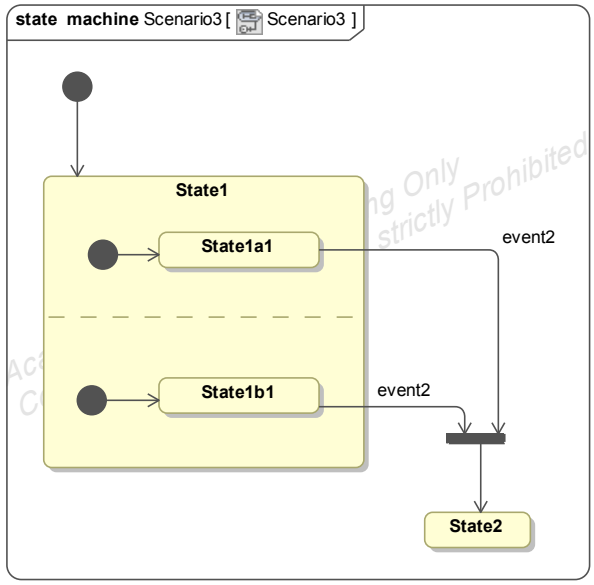

Figure E.3: Scenario3

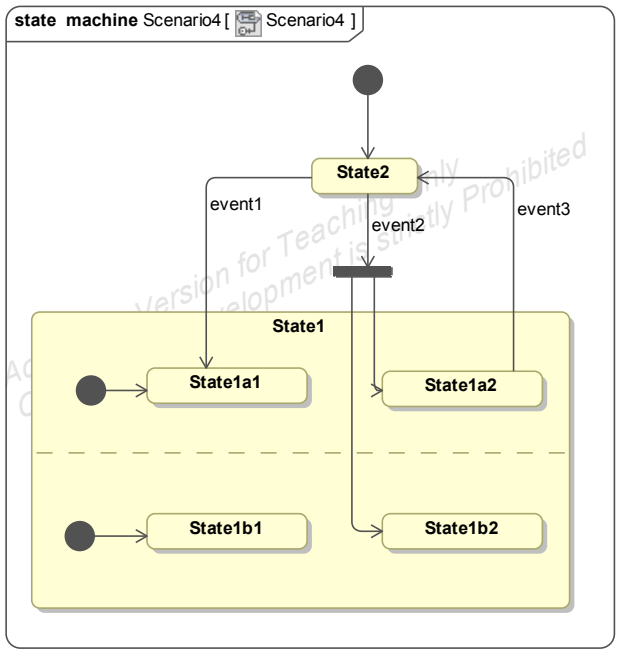

Figure E.4: Scenario4

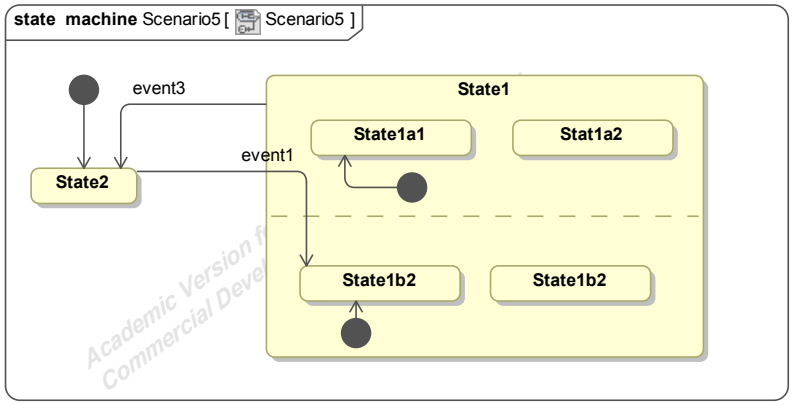

Figure E.5: Scenario5

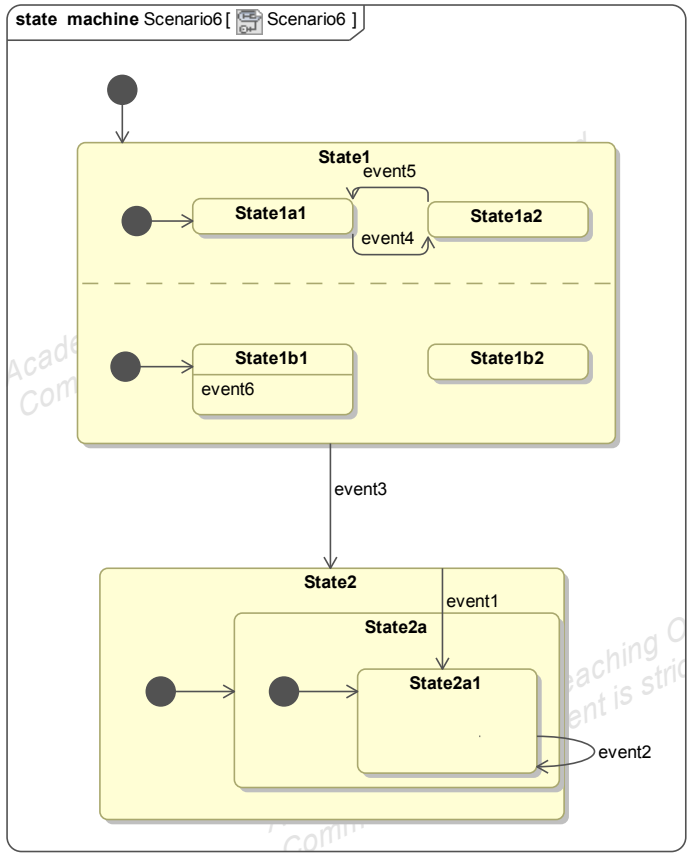

Figure E.6: Scenario6

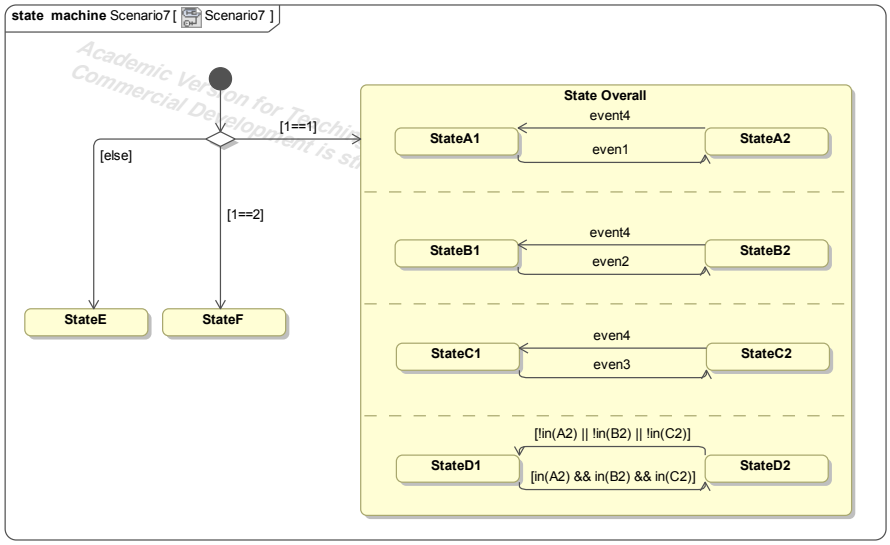

Figure E.7: Scenario7

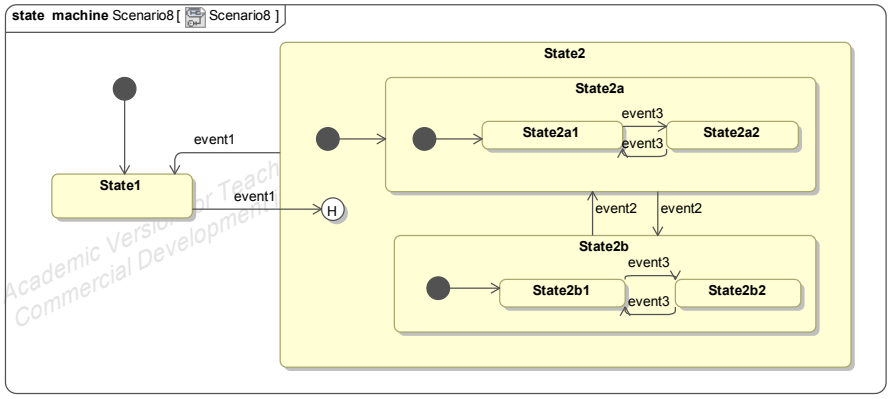

Figure E.8: Scenario8

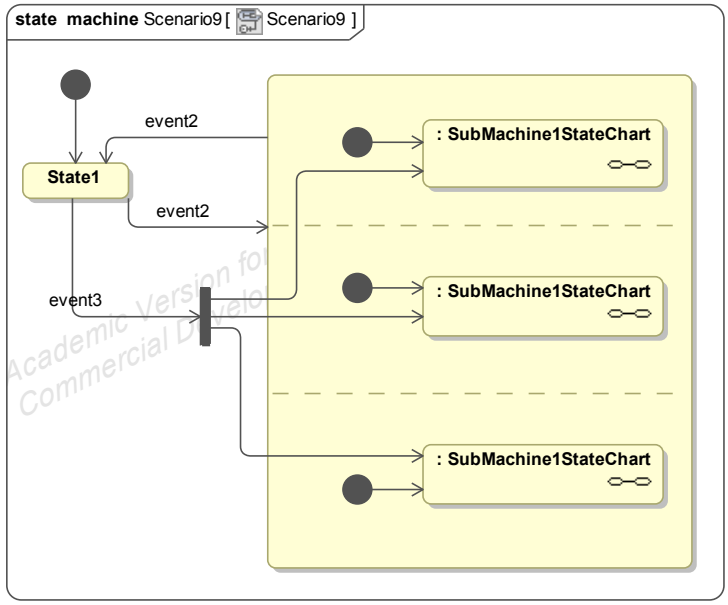

Figure E.9: Scenario9

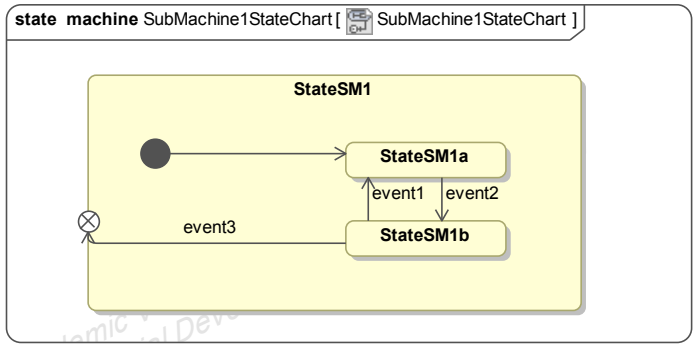

Figure E.10: SubMachine 1

APPENDIX F

TikTok project

In order to run the project, Xcode 5 or newer must be installed. The throughput tests for the different solutions is included in the project `TikTok`.

The project is included in `Projects/TikTok` in the resources. The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

F.1 Running the project

The project uses `CocoaPods`, which must be installed before running.

```
$ sudo gem install cocoapods
```

While in the root of the `TikTok` project folder, install the depending pods.

```
$ pod install
```

Simply open the `TikTok.xcworkspace` project file in Xcode or AppCode.

```
$ open TikTok.xcworkspace
```

In order to specify the goal variable used for the test open `AppDelegate.m`. Change the value of `goal`

```
1 // Specify the value of the goal variable
2 // ************************************************
3 NSUInteger goal = 50000;
4 // ************************************************
```

The result of the test is printed in the console.

```
1 7.69 sec / 50000 msg
2 7646.2 msg / sec
```

APPENDIX G

Component throughput test

In order to run the project, Xcode 5 or newer must be installed. The throughput tests for the different solutions are included in the project `ComponentThroughput`.

The project is included in `Projects/ComponentThroughput` in the resources. The resources can be downloaded from:
`http://www.student.dtu.dk/~s093263/thesis2014/resources.zip`.

G.1 Running the project

The project uses `CocoaPods`, which must be installed before running.

```
$ sudo gem install cocoapods
```

While in the root of the `ComponentThroughput` project folder, install the depending pods.

```
$ pod install
```

Simply open the `ComponentThroughput.xcworkspace` project file in Xcode or AppCode.

```
$ open ComponentThroughput.xcworkspace
```

In order to specify the goal variable used for the test open `AppDelegate.m`. Change the value of `goal`

```
1 // Specify predefined goal
2 // ************************************************
3 NSUInteger goal = 1000;
4 // ************************************************
```

Further in order to specify which test should run, change the value of the `test` variable.

```
// Specify what should be tested
// ************************************************
// TestComponent, TestMethodInvocation or TestReactiveCocoa
Test test = TestMethodInvocation;
// ************************************************
```

The result of the test is printed in the console.

```
0.063588 sec / 1000 msg
15726.3 msg / sec
```

Bibliography

Andrachek, J. M. R., Bender, G., Diehl, D., Gorbsky, M., Novak, R., Shapiro, B., and Zolyak, A. (2013). *The Impact of Mobile*. Segue Technologies, Inc.

Apple (2012a). Dispatch queues. `https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html`. [Online accessed 1-June-2014].

Apple (2012b). Presenting view controllers from other view controllers. `https://developer.apple.com/library/ios/featuredarticles/viewcontrollerpgforiphoneos/ModalViewControllers/ModalViewControllers.html`. [Online accessed 10-June-2014].

Apple (2013a). Uinavigationcontroller class reference. `https://developer.apple.com/library/ios/documentation/uikit/reference/UINavigationController_Class/Reference/Reference.html`. [Online accessed 25-Marts-2014].

Apple (2013b). Uitabbarcontroller class reference. `https://developer.apple.com/library/ios/documentation/uikit/reference/UITabBarController_Class/Reference/Reference.html`. [Online accessed 25-Marts-2014].

Apple (2014a). Performance tuning. `https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/PerformanceTuning/PerformanceTuning.html`. [Online accessed 1-June-2014].

Apple (2014b). Thread management. `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/CreatingThreads.html`. [Online accessed 1-June-2014].

Arlow, J. and Neustadt, I. (2002). *UML And the unified process - Practical object-oriented analysis design*. Addison-Wesley.

Armstrong, J. (2013). *Programming Erlang - Software for a Concurrent World*. The Pragmatic Booksehelff.

Bobbio, A. (2010). System modelling with petri nets.

Booch, G. (1993). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, second edition.

Buyya, R. (2009). *Object Oriented Programming with Java: Essentials and Applications*. Tata McGraw Hill Education Private Limited.

Collberg, C. (2005). Principles of programming languages.

Demetrescu, C., Finocchi, I., and Ribichini, A. (2011). Reactive imperative programming with dataflow constraints. `http://arxiv.org/pdf/1104.2293v1.pdf`. [Online accessed 13-June-2014].

DeRemer, F. and Kron, H. (1975). Programming-in-the-large versus programming-in-the-small.

DiLascia, P. (2004). What makes good code good? *MSDN Magazine*.

Eales, A. (2005). The observer pattern revisited. *Wellington Institute of Technology*.

Eshuis, R. (2009). Translating safe petri nets to statecharts in a structure-preserving way.

Gamme, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns - Elements of Reusable Object-Oriented Software*.

Gursoy, A. and Kale, L. (1994). Dagger: combining benefits of synchronous and asynchronous communication styles.

Haller, P. and Odersky, M. (2007). Actors that unify threads and events.

Hanson, R. (2014). Cocoaasyncsocket. `https://github.com/robbiehanson/CocoaAsyncSocket`. [Online accessed 25-June-2014].

Harel, D. (1986). Statecharts: A visual formalism for complex systems.

Harel, D. and Kuger, H. (2004). The rhapsody semantics of statecharts (or, on the executable core of the uml).

Harel, D. and Politi, M. (1998). *Modeling Reactive Systems with Statecharts*. McGraw-Hill.

Havelund, K. (2008). Runtime verification of c programs. *Jet Propulsion Laboratory*.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns - Designing, Building and Deploying Messaging Solutions*. Addison-Wesley.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2007). *Automata Theory, Languages, and Computation*. Pearson Education, Inc., 3rd edition edition.

Horrocks, I. (1999). *Constructing the User Interface with Statecharts*. Addison-Wesley.

IBM (2005). Designing a software application by using models. http://publib.boulder.ibm.com/infocenter/rsdvhelp/v6r0m1/index.jsp?topic=%2Fcom.ibm.xtools.modeler.doc%2Ftopics%2Ftwrkcs.html. [Online accessed 3-May-2014].

Jackson, D. (2005). Software abstractions - patterns of modelling analysis. http://sdg.csail.mit.edu/6.894/papers/dnj-book-volume2.pdf. [Online accessed 15-May-2014].

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley Sons, Inc.

Klein, L. (2013). *UX for Lean Startups*. O'Reilly Media, Inc.

Krzyzanowski, P. (2006). Clock synchronization.

Lockwood, N. (2014). Autocoding. https://github.com/nicklockwood/AutoCoding. [Online accessed 25-June-2014].

Mackay, P. (1997). Why has the actor model not succeeded? http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/. [Online accessed 1-June-2014].

Roestenburg, R. (2012). Discovering message flows in actor systems with the spider pattern. http://letitcrash.com/post/30585282971/discovering-message-flows-in-actor-systems-with-the.

Salvaneschi, G. and Mezini, M. (2014). Towards reactive programming for object-oriented applications.

Samek, M. (2008). *Practical UML Statecharts in C/C++*. Elsevier Inc., 2nd edition edition.

Scheifler, R. W. and Gettys, J. (1987). The x window system.

Schmidt, D. (2003). Programming principles in java: Architectures and interfaces. http://people.cis.ksu.edu/~schmidt/PPJv12pdf/. [Online accessed 12-June-2014].

Services, I. G. B. (2012). Creating a compelling mobile user experience.

Smith, M. H. and Havelund, K. (2008). Requirements capture with rcat. *Jet Propulsion Laboratory*.

Szyperski, C. (1999). *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley.

van Roy, P. and Haridi, S. (2003). Concepts, techniques, and models of computer programming.

Vasileios, T. (2011). Introduction to erlang concurrency processes. http://trigonakis.com/blog/2011/05/09/introduction-to-erlang-concurrency-processes/. [Online accessed 2-June-2014].

Wagner, F. (2006). What's all this state machine stuff? http://www.stateworks.com/active/download/TN15-Whats-All-This-State-Machine-Stuff.pdf. [Online accessed 7-June-2014].