Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Statechart based support for implementing the behaviour of user interfaces

Op statecharts gebaseerde ondersteuning voor het implementeren van het gedrag van gebruikersinterfaces

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Toegepaste Informatica

Door: Yannik De Valck
Promotor: Prof.Dr. T. D'Hondt
Begeleiders: S. Goderis & Dr. D. Deridder
Augustus 2006

Abstract

Like applications, user interfaces need to show the ability to adapt, as some changes to the application logic cause the need for changes in the user interface (UI) as well. A problem with applying such changes to the user interface is that the UI logic often is entangled with and scattered throughout the underlying application logic, which makes adapting the UI and the application logic cumbersome for the developer. However, different concerns which also suffer from entanglement and scattering are also found within the UI itself. These concerns are not only entangled with and scattered throughout the application logic, they are also entangled with and scattered throughout each other, which makes matters even worse for the developer. Therefor, these concerns must not alone be separated from the application logic, but also from each other as much as possible.

In this dissertation we aim at disentangling such a concern from within the UI from the rest of the application and UI, and present a solution (which is based on statecharts) that provides support for doing this. This concern is the UI Behaviour. The separation of the UI Behaviour from the rest of the application and UI can help the developer to cope with the complexity of adapting and implementing an application, its UI and its UI Behaviour, as the developer is not forced to deal with the problems coupled to the entanglement and scattering of the UI Behaviour. To further aid the developer, the constructed solution is supported by a system, which can be used by the developer to automate most of this separation.

Abstract

Gebruikersinterfaces zouden, net zoals programmas, makkelijk aanpasbaar moeten zijn, aangezien sommige aanpassingen in deze programmas ook veranderingen teweeg brengen in de gebruikersinterface. Het maken van deze veranderingen in de gebruikersinterface wordt echter vaak bemoeilijkt door de verwevenheid van de logica van de gebruikersinterface met de logica van het programma, evenals door de verspreiding van de logica van de gebruikersinterface over het programma. Ook binnen de gebruikersinterface zelf vinden we onderdelen die verweven en verspreid zijn. Deze onderdelen zijn echter al niet meer enkel verweven met en verspreid over het programma, maar ook met en over elkaar, hetgeen aanpassingen nog meer bemoeilijkt. Daardoor willen we deze onderdelen van de gebruikersinterface niet enkel van het programma scheiden, maar ook van elkaar.

In deze verhandeling proberen we zo'n onderdeel van de gebruikersinterface te scheiden van de rest van het programma en de gebruikersinterface door een oplossing aan te bieden (gebaseerd op statecharts) die ondersteuning biedt om deze scheiding door te voeren. Het onderdeel van de gebruikersinterface dat wij zullen beschouwen, is het gedrag van de gebruikersinterface. Het scheiden van het gedrag van de gebruikersinterface van de rest van het programma en de gebruikersinterface zal de programmeur helpen om hieraan aanpassingen te doen, doordat de programmeur niet langer moet omgaan met de problemen die verwevenheid en de verspreiding van het gedrag van de gebruikersinterface meebrengen. Om de programmeur verder bij te staan, is een systeem gebouwd om onze oplossing toe te passen, hetgeen de programmeur toelaat om het grootste deel van de scheiding tussen het gedrag van de gebruikersinterface en de rest te automatiseren.

Acknowledgments

I would have never finished this dissertation without the help of a lot of people. Therefore I would like to express my gratitude towards all of them. Prof. Theo D'Hondt for promoting this thesis. Sofie Goderis and Dirk Deridder for being excellent advisors: not only did they come up with the subject of this dissertation but they also guided me throughout the whole process. They helped me at every step of the way from giving suggestions and valuable advice to proof-reading and correcting the mistakes in this document. Their door was always open, and never did they complain when I walked in for the 34th time to ask them a question. Without their help this dissertation would have never seen the daylight. The researchers at the Programming Technology Lab and my fellow thesisstudents for providing a fun and motivating environment to work in, sitting through my presentations and finding the courage to provide feedback afterwards. The Vrije Universiteit Brussel and Departement Toegepaste Informatica for providing an excellent education.

On a more personal note, I would also like to thank Phillip, Charlotte, Jo, Stijn, Andoni, Frederik and Pierre for bringing some joy in the thesisroom when it was needed the most, and for the few discussions that did not involve pirates, monkeys, ninjas or zombies. A big thanks goes out to all the people at the scouts, who gave me courage when I felt low and provided me with much needed distraction, lots of fresh air and an extra motivation to finish this thesis. Furthermore, I thank all my friends for supporting me throughout my thesis and for understanding that my thesis came before my social life. A big "thank you" also to Joke, Timothy and Joris, for entertaining me and letting me win while playing cards. A warm thanks goes out to my family, who also supported me, and understood when I said I could not explain what this thesis was about in a way that they would understand.

A very special thanks goes out to Margriet and Nathalie, two very special ladies whose support, listening ears, encouragements and so much more were crucial for me to survive this thesis. Last but not least, I thank my parents for supporting me all these years, believing in me and giving me the possibility to obtain a higher education. If I forgot to thank anyone in particular, blame it on the mind, not on the heart.

I would like to dedicate this thesis to both of my grandfathers; I hope I make you proud.

Contents

Chapter 1

Introduction

As developers improve and change their applications, the application's source code needs to show the ability to adapt. This does not only affect the application logic, but also the user interface [23], as some changes to the application logic cause the need for changes in the user interface (UI) as well. Consider for example a video-rental-application. Here a trusted client gets the new extra ability to reserve a video he wishes to rent up front, and then pick it up later. This means there will have to be new components added to the UI to allow the trusted client to reserve the video, to correspond with the extra application logic that has been added to (at least) separate trusted clients from occasional clients and appropriately grant access to the reserving system.
A problem with applying such changes to the user interface is that the *UI logic is entangled with the underlying application logic*. In addition to the entanglement, the *UI logic is scattered throughout the application logic*, which makes adapting the UI and the application logic cumbersome for the developer. In the case of the video-rental-application, the application logic (the decision of which client is "trusted" or not) influences the UI (showing the extra components for reserving or not). This means that the adaptations made to the UI will be entangled with the application logic.
Like in the video-rental-application, when the need arises to make such a change to the UI of an application, the scattering of the UI logic leaves the developer scanning through the application logic in search of that part of the UI that needs to be adapted.
The entanglement of the UI logic with the application logic also hampers the developer when adapting the UI, as the developer is forced to make a mental separation between UI logic and application logic before making any adaptations. Adapting the code without a clear view on which part of the code belongs to which part of the logic raises the chance of making errors by breaking other code. Therefor, *UI logic and application logic should be separated as much as possible*.

However, not only UI logic and application logic should be separated as much as possible. Different concerns, which also suffer from entanglement and scattering are also found within the UI itself. Therefor, similar to distinguishing different concerns in the

application logic (e.g. the logging concern), we can distinguish *three different concerns in the UI*, which we call UI concerns [23]. Firstly, *visualization* is what the UI looks like and what widgets are provided. Secondly, *application interactions* describe how the UI hooks into the application. The third concern is *UI Behaviour*, which can be described for now as the way in which widgets influence each other [1]. More precisely, the UI Behaviour is described as the behaviour of a widget, with an impact on the UI. This means that when an event takes place on a widget (for example when it is clicked), a certain action in the UI can be triggered. What this action is, is specified in the behaviour of the widget. For example when we click a certain button in the UI, the background color of the UI is changed from gray to white. Hereby, an event takes place on the button, namely the clicking, and the action of changing the background color was triggered. The changing of the background color is seen as the behaviour for that button.
It is on this last concern that we will focus throughout this dissertation, as it is our goal to separate this UI concern from other concerns.

The three UI concerns (as described above) are not only entangled with and scattered throughout the application logic, they are also entangled with and scattered throughout each other, which makes matters even worse for the developer to deal with entanglement and scattering when the need arises to adapt (a part of) such a UI concern. Therefor, these UI concerns must *not alone be separated from the application logic, but also from each other as much as possible*.

In this dissertation, we present a solution that provides support for separating the UI Behaviour concern, as well from the application logic as from the other UI concerns. We will focus on disentangling one particular kind of UI Behaviour, being state-based UI Behaviour (see section 3.3). This separation of the state-based UI Behaviour can help the developer to cope with the complexity of adapting and implementing an application, its UI and its UI Behaviour, as the developer is not forced to deal with the problems coupled to the entanglement and scattering of the UI Behaviour, as described above. To aid the developer further, we construct a solution that is supported by a system, which was used by the developer to automate some parts of this separation.
We find that, in order to achieve this separation of the UI Behaviour, the global problem can be split up into three issues that can be solved separately, as the solutions to these issues can then be combined into a solution to the global problem. These three issues and their respective solutions presented in this dissertation are the following.

First, we should be able to represent the UI Behaviour in a way that it is separated from other concerns in our application as much as possible. This can be achieved through the use of statecharts, as the UI Behaviour for a particular UI can be expressed in a statechart, in which as little as possible of other concerns or parts of the appli-

[1] This concern is discussed further in section 3.

cation are expressed [31]. This would mean that the UI Behaviour represented in the statechart is separated from the rest as much as possible.

Secondly, we should be able to create an application-level equivalent for this representation of the UI Behaviour, as we want an executable version of the UI Behaviour (which is described in our statechart) to use in our application. Therefor, the statechart holding the UI Behaviour should be transformed to a number of application-level statements.

Thirdly, as we want to use the UI Behaviour in the application, we should also couple this UI Behaviour to our application, so that the appropriate UI Behaviour can be addressed at the appropriate time. This can be achieved through the use of aspects which hold information about the UI Behaviour. These aspects can then be woven on the application on the appropriate places, which would allow the application to call on the UI Behaviour whenever it is needed.

As the three separated issues were provided and a solution to each of them was proposed, these solutions can be combined into a conceptual solution to the global problem. In this dissertation, we will also provide a practical equivalent of this conceptual solution, which means the developer is guided through the manual steps of the solution, and furthermore a system is implemented to aid the developer in using the solution by automating some parts of the solution. In these guidelines for the manual steps, it is described in what particular way the developer should draw a statechart containing the UI Behaviour, and how the developer can obtain an application-level equivalent of that statechart. Once the application-level equivalent is provided, the system supporting the solution takes control and creates the necessary executable code and couplings to the application in order for the UI Behaviour to be used directly. Once the system has finished, the UI Behaviour should be integrated in the application, and the application and its UI should work normally, as if the UI Behaviour had never been separated.
The solution, the supporting system and the benefits of using this solution are illustrated and validated in this dissertation through the use of a case study in chapter 6. This case study consists of implementing the application and the UI for an advanced calculator application, hereby using the solution that was proposed in this dissertation. By using this experiment, we show for example how the UI Behaviour is separated from other concerns during the implementation of the calculator application, and validate if the proposed solution truly works for the calculator application.

Overview of the dissertation

This dissertation begins with a chapter (chapter 2) describing the problem that is tackled. The global problem is sketched, before it is narrowed down to a more specific problem on which we will focus. Chapter 3 describes the user interface relations and continues with a thorough explanation of the UI concern on which we focus, being the

UI Behaviour. Chapter 4 contains the definitions of important terms and the explanation of technologies and artifacts that are used throughout the dissertation to construct our solution. Chapter 5 provides a solution for the problem stated in chapter 2. It presents the conceptual solution to the problem, followed by a description of the solution on a practical level. To conclude chapter 5, an overview of benefits and shortcomings of the proposed solution is given. Chapter 6 reports on how this solution is applied to a calculator application and validates our solution. Chapter 7 surveys existing techniques and approaches used to specify and disentangle parts of the UI by using statecharts. The final chapter holds a short summary of the dissertation, followed by ideas for future work and a conclusion.

Chapter 2

Applying separation of concerns on UIs

This chapter contains the explanation and specification of the problems we attack in this dissertation. The problems are explained more thoroughly and boundaries are set to what type of problems are tackled in this dissertation. We first define the core problem, which is followed by an existing solution to this core problem. This leads us to a refinement of the core problem, on which we will focus in this dissertation.

2.1 Entanglement and scattering: the core problem

As programs and applications are changed and updated regularly in today's software systems, they should be constructed with adaptability in mind. This does not only apply to the application's source code, but also to the user interface (UI). A problem with adapting the user interface is that the user interface logic is *entangled* with the application logic and *scattered* throughout the application logic. These problems can be seen as the core problem, i.e. the main problem from which the specific problems that are tackled in this dissertation are derived. Because of this, adapting the user interface logic has become a difficult task for the programmer. Due to scattering, the programmer is forced to scan through the application code in search of pieces of UI logic. When the pieces that need adaptation are found, typically, similar changes to the UI are made several times in different places in the code. For example, when adding one button to every window in the entire UI, a similar piece of code to add the functionality of this button is added to all window-related code.
Furthermore, making these changes to the *user interface logic is complicated due to the entanglement* with the application logic, which forces the programmer to be extra careful. As it is not always directly clear what parts of the code belong to the user interface logic or to the application logic, the programmer must first spend time on figuring out what pieces of code belong to the kind of logic that is being adapted. This is typically repeated during future adaptations of the logic, as the developer is not most likely to remember to what kind of logic the different pieces of code belong.
As stated in [31], applications that suffer from entanglement and scattering of the UI

logic are *difficult to enhance*, which is explained as follows. When UIs are written, it is typically obvious for the one writing them what relations are present. However, many events can affect or make use of the same information, and if a part of the application needs to be changed, it is necessary to understand the dependencies that exist between the different parts of the application. Therefor, it is difficult to understand the impact of modifying one part of the code without making the change and then testing the application to ensure no side-effects have been introduced. This makes enhancing an application a trial and error process, which is not good. Furthermore it is *difficult to get the applications to work correctly*, as errors in the application are hard to spot because we have to search the entire application to find them [31]. Errors are also made quicker while implementing the application, as the implementation is difficult due to the presence of entanglement and scattering. Furthermore it is *nearly impossible to get an abstract view* of the application just by looking at the code [31]. Therefor, the bigger picture is lost and all that remains is a line-per-line view, which leaves us trying to reconstruct that bigger picture by going through the code step by step.

2.2 Separation of concerns

A ”general approach” to cope with the core problem (as just described in section 2.1) already exists, and it is called ”separation of concerns”. *Separation of concerns* (SoC) is described as the process of breaking a program into distinct features that overlap in functionality as little as possible [40] [14]. A concern is any piece of interest or focus in a program. However, a separation of concerns that the developer has in mind is not always feasible, and just letting the developer break up the program into separate parts is not sufficient when this is not also applied to the application’s implementation.
When designing an application, the developer tries to design the application in a clean and logical manner. As there is no benefit to mingling the different parts of the application, the developer tries to keep those parts separated, so that reasoning about every part by itself is possible. Therefor, in the mind of the developer, the different parts of the application are molded separately, and not directly as a whole. For example, the developer can design the UI first, and then later on specify the application logic corresponding to the built UI, or the other way around. This means that the developer applies the principles of separation of concerns from the start, as the different concerns of the application are separated in his mind. However later on, during the actual implementation of the application, it is often not possible to *maintain this separation*. As the developer applied the principles of separation of concerns all along, he would also like to keep this separation when the application is being implemented. This would make the code that is written during the implementation free of entanglement and scattering, and therefor easier to understand, maintain and adapt (as is shown in section 2.1).
An example of how the programmer distinguishes concerns and how this separation is lost further on, is presented in the following pieces of Smalltalk-code, depicted by code snippets 2.1, 2.2 and 2.3. This example consists of a small part of the code of an

existing and working calculator-application. We can see how the programmer specifies the application logic (business logic) in a method ”checkAge”, which is represented by the green and blue pieces of Smalltalk-code in code snippet 2.1. In this method, it is checked if the user is old enough to be able to use roman numbers in the calculator. The programmer also specifies the UI logic in a method ”switchRoman”, which is represented by the red and magenta pieces of Smalltalk-code in code snippet 2.2. In this method, the buttons of the calculator are adapted to displaying roman or arabic numbers. Up to this point, the separation is still intact and it is still clear to which concern a certain piece of code belongs. However, when the application logic and the UI logic are combined into a working application, entanglement and scattering are introduced into the application, which is visible by the mixed colors and line-numbers in the method ”switchRomanWithAgeCheck” in code snippet 2.3.

```
checkAge
1  (self userAge < 11)
2   ifTrue:[Transcript show:
3              'You are not old enough for roman numbers'.
4   ifFalse:[Transcript show:
5              'You are now using roman numbers'.
6            self calculateInRoman].
```

Code Snippet 2.1: Code for the application logic (business logic)

```
switchRoman
7  (self roman)
8   ifFalse:[self numbers do:
9              [:each | |button|
10             button := self builder componentAt: each.
11             self makeArabic: button.]]
12  ifTrue:[self numbers do:
13             [:each | |button|
14             button := self builder componentAt: each.
15             self makeRoman: button.]]]
```

Code Snippet 2.2: Code for the UI logic

```
switchRomanWithAgeCheck
7 (self roman)
8  ifFalse:[self numbers do:
9            [:each | |button|
10           button := self builder componentAt: each.
11           self makeArabic: button.]]
12 ifTrue:[
1    (self userAge < 11)
2     IfTrue:[ Transcript show:
3                   'You are not old enough for roman numbers'.
8            self numbers do:
9                  [:each | |button|
10                 button := self builder componentAt: each.
11                 self makeArabic: button.]]
4     ifFalse:[Transcript show:
5                   'You are now using roman numbers'.
6              self calculateInRoman.
12             self numbers do:
13                 [:each | |button|
14                 button := self builder componentAt: each.
15                 self makeRoman: button.]]]
```

Code Snippet 2.3: Combination of application logic and UI logic

We would like to point out to the reader that this is a naive implementation of a small part of a calculator application, but it is only used to illustrate the entanglement and scattering. However, when we change this implementation to a less naive version, the entanglement will still be present. A less naive implementation is for example to put all the colored pieces of code into different methods (all the code of one color in one method), which makes the code more readable. However, the entanglement will still hold, as the different methods still need to be combined. In that case, code snippet 2.3 will then hold entangled methods representing the pieces of code, instead of the real pieces of code as is now the case.

As we have seen in code snippet 2.3, by combining UI logic and application logic we have introduced entanglement in the method ”switchRomanWithAgeCheck” and lost our separation of concerns. This has some consequences, as mentioned in section 2.1.
Say that we now want to adapt the ”switchRomanWithAgeCheck”-method in such a way that also the background of the calculator changes whenever it switches from the roman to the arabic state or the other way around to make it even clearer in what state we are in. This means that we want to add the functionality of changing the background (which is part of the UI logic) to the pieces of code that belong to the UI logic concern that are responsible for switching between the roman and arabic state. However, as those pieces of code are blended in with the application logic, it is hard for the developer to instantly see which parts of the code need adaptation. This is shown in code snippet 2.4, in which the background-changing functionality is added in brown (with line-number ”XX”). We can see that the background-changing functionality is added just after the pieces of code that belong to the UI logic concern, as the developer was forced to distinguish pieces of code that belong to the UI logic concern from those that belong to the application logic concern. It can be noticed that in this example,

distinguishing the UI logic from the application logic remains fairly simple. However, this is just a small example and one should picture this in a real-life application with a significant number of lines of code and several developers adapting the code. In this situation, one can imagine the difficulties that arise when an adaptation similar to the background-changing example needs to be made.
Furthermore, we also see that the entanglement causes similar background-changing functionality to be added to different places in the code. In code snippet 2.4, even the exact same functionality is added in two different places in the code. Again, for this example, this may not seem like a big deal, but when put in the context of a real-life application, we can say that we do not want having to make similar (or even equal) adaptations throughout the code, while having to search for the right pieces of code to adapt.

```
switchRomanWithAgeCheck
7 (self roman)
8  ifFalse:[self numbers do:
9            [:each | |button|
10           button := self builder componentAt: each.
11           self makeArabic: button.]
XX          makeBackgroundArabic.]
12 ifTrue:[
1    (self userAge < 11)
2     IfTrue:[Transcript show:
3                'You are not old enough for roman numbers'.
8             self numbers do:
9                  [:each | |button|
10                 button := self builder componentAt: each.
11                 self makeArabic: button.]
XX            makeBackgroundArabic.]
4     ifFalse:[Transcript show:
5                'You are now using roman numbers'.
6             self calculateInRoman.
12            self numbers do:
13                 [:each | |button|
14                 button := self builder componentAt: each.
15                 self makeRoman: button.]
XX            makeBackgroundRoman.]]
```

Code Snippet 2.4: Adding logging functionality

Through these examples and their discussed difficulties, it is shown that we want to avoid entanglement and scattering as much as possible. Therefor, we want to apply the principle of separation of concerns as much as possible, not only in applications, but also in UIs.

2.3 Separation of concerns for UIs

As mentioned in section 2.2, problems such as entanglement and scattering can be tackled by using the principles of separation of concerns. This is used in several techniques such as aspect-oriented software engineering [19] [5]. However, the principle of separation of concerns can not only be applied to applications. In this dissertation

we *apply this principle onto UIs*. In order to separate the UI from the application, we distinguish the concerns UI visualization and UI Behaviour [23]. Visualization is what the UI looks like and what widgets are provided. UI Behaviour will be defined and explained further in chapter 3, but for now, one might think of it as referring to how widgets relate to and influence each other. However, as we define these two concerns to separate the UI from the application, we immediately see the need for a *separation within the UI itself*. Therefor, also a separation is made between UI visualization and UI Behaviour.
Up to this moment, we still lack a clean way to achieve this separation and couple the UI logic to the application logic, as current solutions (such as MVC [32] and UI builders [35]) to achieve a separation of concerns for UIs only provide a partial solution, as stated in [23]. In this dissertation we will focus on *separating the UI Behaviour from other concerns*. As UI Behaviour is very general, we will focus on one kind of UI Behaviour, which is the state-based UI Behaviour. This is explained more thoroughly in chapter 3.

In this chapter, we have defined the problem that we want to tackle in this dissertation. This problem definition can now be used throughout this dissertation to find a valid solution for our problem, and to validate our solution in later stage. However, we have not defined UI Behaviour fully yet, which is needed to specify clearly what we will actually separate from other concerns. Therefor, the following chapter works towards a definition of UI Behaviour, which can then be used in the remainder of this dissertation.

Chapter 3

Specification of user interface relations: defining UI Behaviour

In this chapter, a specification of which relations are found *within, from, and to* User Interfaces is presented. This specification is based on the work of Moens in [33], which we extended and adapted with our own research, to eventually achieve this taxonomy. This chapter provides an overview of how certain events that take place in the UI or the application have an impact on certain aspects [1] of the application or the UI. We start this chapter with shortly explaining the possible events that can have such an impact and the aspects that can be had an impact on. When these events and aspects are explained, we continue with specifying and explaining the impacts that these events can have on these aspects, accompanied by an example for each of the impacts. We conclude this chapter by selecting a number of these impacts and using them to define UI Behaviour, on which we will focus in this dissertation. This is important, as we need to define and set boundaries for the UI concern (see chapter 2) we wish to disentangle in the remainder of this dissertation. Therefor, the UI Behaviour is defined in this chapter in order to pinpoint just which concern within the UI we are going to handle in this dissertation.

3.1 Defining the building blocks of impacts: cause and effect

In order to be able to specify the different impacts in section 3.2, we first need to specify what situations in an application or UI can have an impact, and what parts can be had an impact on. Therefor, in this section we give an overview and explanation of the situations and parts that we found necessary to specify the impacts.
First the situations that can occur in a UI or an application and can have an impact

[1] The word aspects in this context is not related to aspects as they are used in aspect-orientation. It is just used to describe certain elements of an application or a UI on which events can have an impact.

on parts of the UI or application are described. We call these situations *events*. When such an event occurs in the application, we call it a *system event* or a change of a *system value*. When it occurs the UI, we call it an *input-event on a widget*, in which a widget is a component of the user interface, and in this case a component with which the user can interact, such as a button or an input-field.
Secondly, the parts of the UI or application that can be affected by an event are described. We call these parts *aspects* [1]. When there is an impact on the application, a *system value or system event* can be affected. When the impact takes place on the UI, then the *widget appearance*, *inter-widget layout* and/or the *UI-related widget behaviour* can be affected.

It can be noted that a system value/event can be an event as well as an aspect. In the first case the system event or the changing of a system value causes the impact, whereas in the second case the system value or event is affected by the impact.

3.1.1 Causing an impact: key events

The first kind of event is the **input-event on a widget**, which is an event that occurs in the UI. This means that an event takes place on a certain widget, i.e. the widget is selected or is given input to. This is the case when, for instance, a widget is selected by using the mouse (e.g. a button that is clicked) or selected by using the keyboard (e.g. use a keyboard shortcut to select a widget and press enter). Giving input to a widget is for example when the keyboard is used to enter a string or a number in an inputfield. It should be noted that, however the input-event on a widget is mostly triggered by a user, it is also possible that it is triggered by the application, for example when the application fills in a string in an inputfield. In this case, we can already see an example of an impact, being the impact of a system event (filling in the inputfield) on the UI (the inputfield has received input). As the inputfield is filled in, this is considered as an input-event on the widget and this may in its turn have an impact on another aspect.

The second kind of event is the **system event** or the **changing of a system value**, which is an event that occurs in the application. This means that a value that is stored in the application changes or an application-event is triggered. An example of a changing value is the changing of a value of a system variable, for instance updating the string in the variable ”name”. A triggered application-event is for example the execution of some code in the application.

3.1.2 Effect of an impact: key aspects

We first describe the aspects that can be affected when an event has an impact on the UI. These aspects are *widget appearance, inter-widget layout and UI-related widget behaviour*.
When we use the term **widget appearance**, we refer to the look of a widget, such as its

color, shape, length, width, etc. An impact on the widget appearance can for example cause a widget to change its color.
Inter-widget layout is the way widgets are positioned with regard to one another. For example a widget is horizontally or vertically aligned with another widget, or a widget is 2 cm to the left of another widget. An impact on this aspect may cause widgets for example to reposition themselves with regard to other widgets.
UI-related widget behaviour is described as the behaviour of a widget, with an impact on the UI. When an event takes place on a widget (for example when it is clicked), an action is triggered. What this action is, is specified in the behaviour of the widget. For example when we click a certain button in the UI, the background color of the UI is changed from gray to white. Hereby, an event takes place on the button, namely the clicking, and the action of changing the background color was triggered. The changing of the background color is seen as the behaviour for that button.

Secondly, we describe the aspect that can be affected when an event has an impact on the application. This is the **system value/event**. This means that a value that is stored in the application or an application-event is affected. For example when we click a certain button, the application calculates the product of 2 numbers we entered or when we enter a string, the application stores the name of the user.

As the possible events and aspects are discussed, we can now have a look at what impact such an event has on a particular aspect and what events have impact on what aspects. This is explained in the next section.

3.2 Impact of events on aspects

In this section, we present what events have an impact on what aspects of the UI or application. There are three situations in which these impacts can take place, which are illustrated in figure 3.1.
First, an event in a UI can have an impact on an aspect of the UI, so the impact remains *within the UI* (Impact number 1 in figure 3.1). Here an input-event on a widget can have an impact on a UI-related aspect, such as widget appearance, inter-widget layout or widget behaviour.
Secondly, an event in a UI can also have an impact on an aspect of its application, so there can be an impact going *from the UI to the application* (impact number 2 in figure 3.1). An input-event on a widget can have an impact on a system value or system event in this case.
And thirdly, an event in the application can have an impact on an aspect of the UI, so the impact is going *from the application to the UI* (impact number 3 in figure 3.1). This means that a system value or a system event can have an impact on a UI-related aspect.
In the remainder of this section, these situations are explained more thoroughly and illustrated with specific examples.

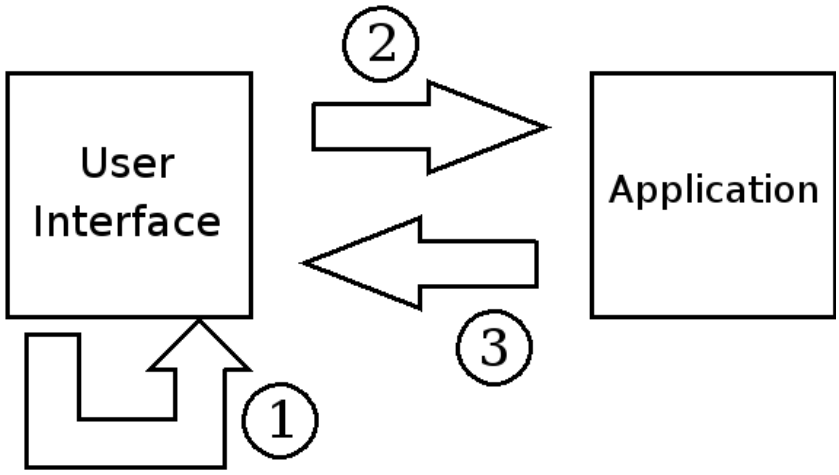

Figure 3.1: Global overview of the types of impacts

As one common application, on which the examples for all the scenarios are based, we selected a calculator application. The application for this calculator holds all functionality that a calculator needs, and has some extra features that are used in the examples. There is also a UI for the application, which represents a calculator-like interface by providing buttons to enter numbers and operators and an inputfield where numbers also can be entered straight from the keyboard. This UI is depicted in figure 3.2.

3.2.1 Impact within UIs

When an event in the UI has an impact on an aspect in the UI, the impact remains within the UI. In this case, we can distinguish three different situations: an *input-event on a widget* can have an impact on *widget appearance, on inter-widget layout or on UI-related widget behaviour*.

When there is an **impact of an input-event on *widget appearance***, this means that an input-event (selecting or giving input to a widget) takes place on a widget and the appearance of one or more widgets changes. For example when we enter a positive number in the inputfield of the calculator, the inputfield turns green, and when we enter a negative number, the inputfield turns red. This is shown in figure 3.3.

An **impact of an input-event on *inter-widget layout*** means that an input-event takes place on a widget and the position of one or more widgets with regard to other widgets changes. An example is when selecting a radiobutton to choose between a basic or advanced mode of the calculator, widgets are added to the operator-group to reach the advanced mode for the calculator. This means that the position of other wid-

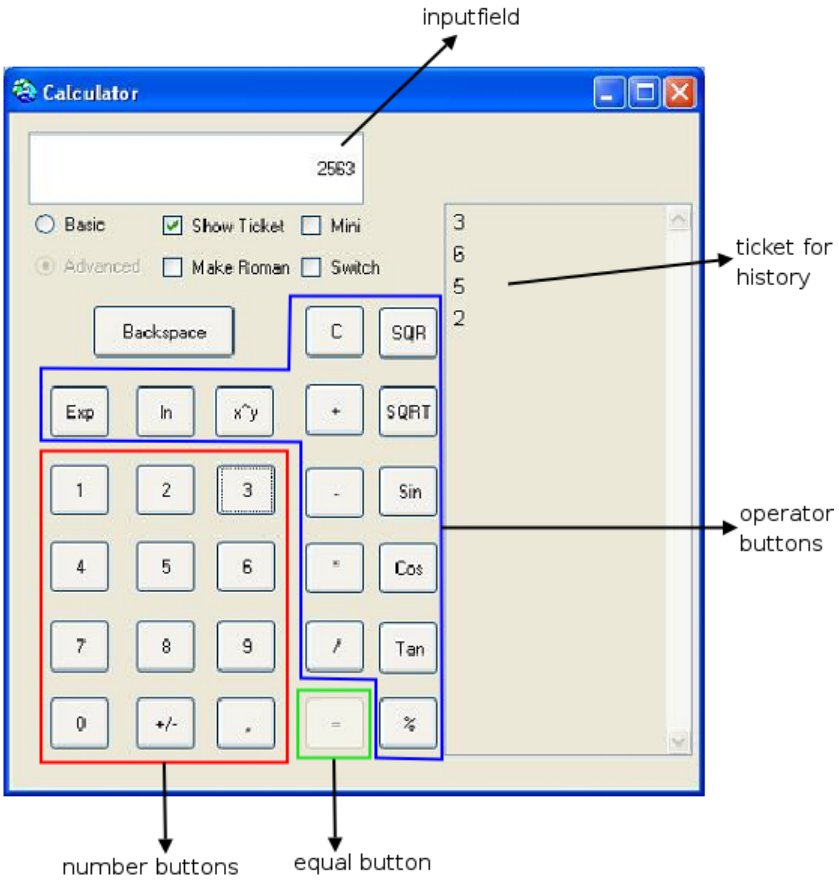

Figure 3.2: The UI for the calculator application

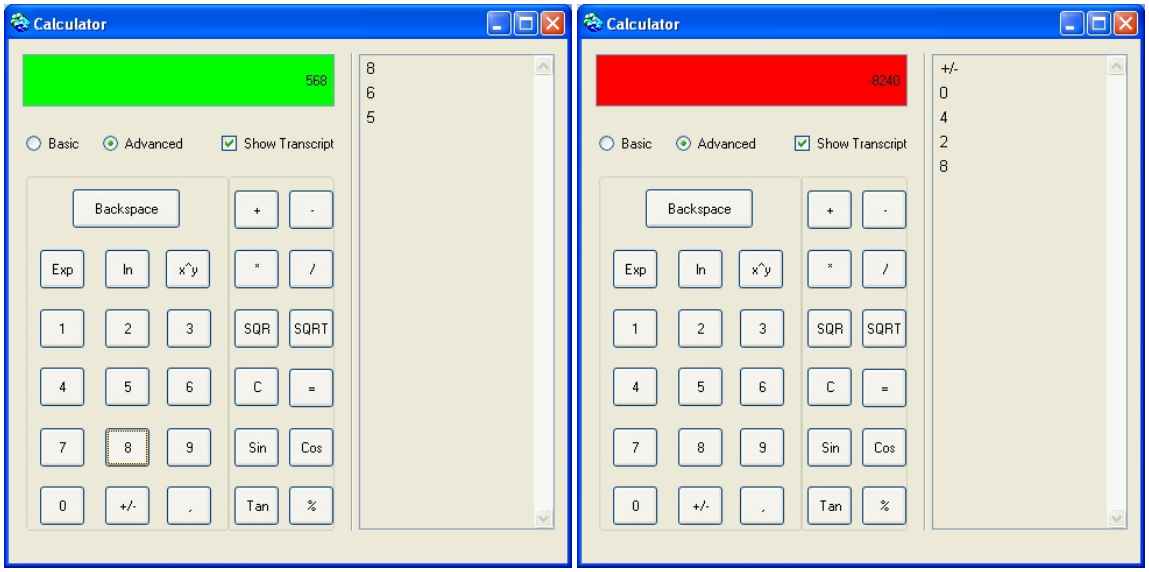

Figure 3.3: The calculator with a positive number (left) and a negative number (right)

gets must be adapted, in order for the new widgets to fit in the calculator. This can be seen in figure 3.4.

Finally, there can be an **impact of an input-event on *widget behaviour***. Here, an input-event takes place on a widget and the UI-related behaviour of one or more widgets changes. This is the case when we select a radiobutton that switches the calculator from accepting input as normal numbers to complex numbers, because then the behaviour of the operator buttons must be adapted to handle the imaginary part of the complex numbers. The input-event has an impact on the behaviour of the widgets in the UI. This is illustrated in figure 3.5.

3.2.2 Impact of the UI on the application

When an event in the UI has an impact on an aspect in the application, there is an impact going from to UI to the application. In this case, we only distinguish one situation: an *input-event on a widget* can have an impact on *a system value/event*.
The **impact of an input-event on a system value/event** means that an input-event takes place on a widget and an application-event (a system-action) gets triggered or a system value changes. As an example, we consider a situation where we select the equal button in the calculator, and therefor the system is asked to calculate the result of the operation we just entered. An alternative example is that we enter something in the inputfield, and then a validation action takes place in the system, which checks if the input is valid or not (e.g. in the calculator a string is invalid, a number is valid).

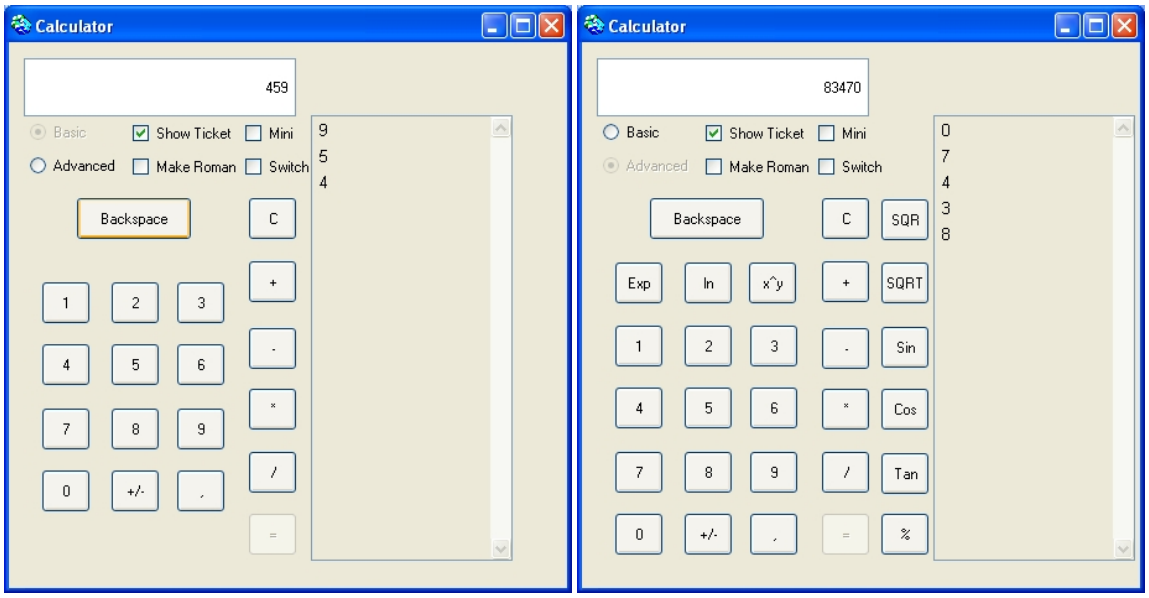

Figure 3.4: The calculator in the basic state (left) and the advanced state (right)

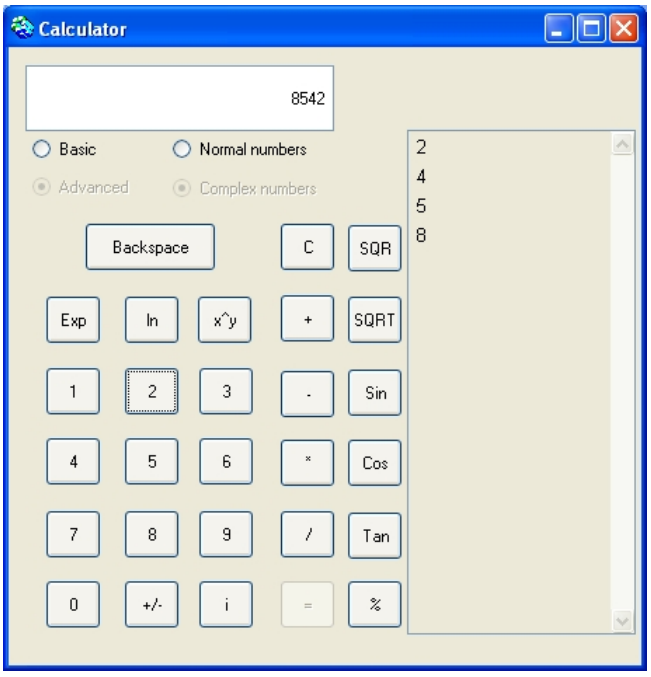

Figure 3.5: The calculator, adapted to handle complex numbers

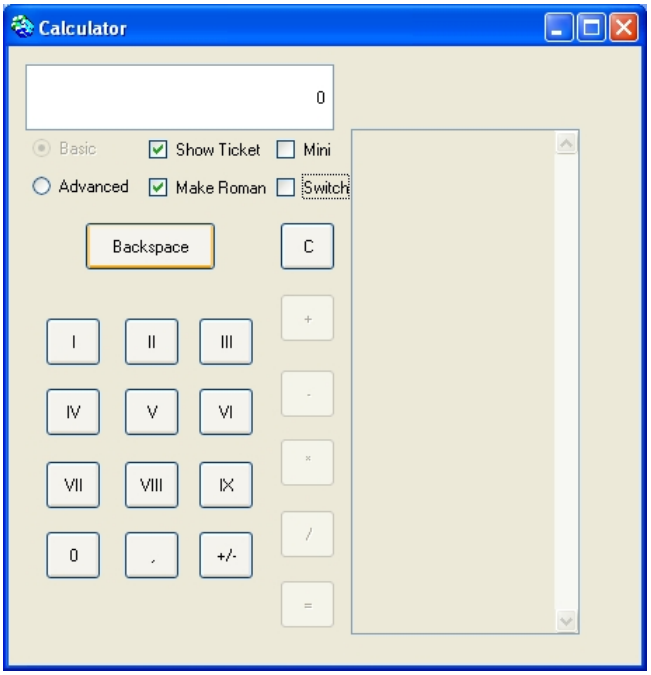

Figure 3.6: The calculator with Roman numbers

3.2.3 Impact of the application on the UI

When an event in the application has an impact on an aspect in the UI, there is an impact going from to application to the UI. In this case, we can again distinguish three different situations: an *system value/event* can have an impact on *widget appearance, on inter-widget layout or on UI-related widget behaviour*.

When there is an **impact of a system value/event on *widget appearance***, a system value changes or a system event takes place and the appearance of one or more widgets changes. For example, when we change the system value that keeps track of where we live. Depending on the area where you live, the appearance of the widgets changes to another language, or in the case of numbers, the number buttons switch in appearance from arabic numbers to roman numbers. This can be seen in figure 3.6.

An **impact of a system value/event on *inter-widget layout*** means that a system value changes or a system event takes place and the position of one or more widgets with regard to other widgets changes. For example, when it is checked that we have clearance to make use of this option, new advanced buttons are added to the calculator to save our calculations. This means that the position of the other widgets should change, to fit these new buttons in the calculator. As another example, the system detects that we use a certain operator a lot, so the button representing that operator is moved to the top of the operator buttons to be quickly accessible. This is illustrated in

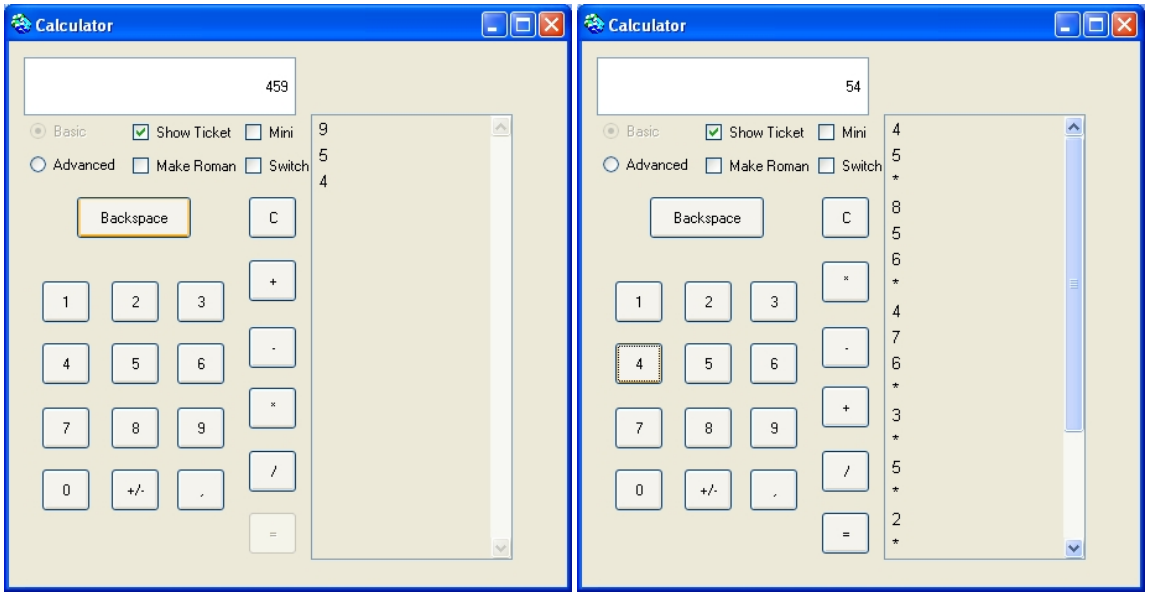

Figure 3.7: The calculator in the begin state (left) and with the most used operator button on top (right)

figure 3.7.

Finally, there can be an **impact of a system value/event on *widget behaviour***. This means that a system value changes or a system event takes place and the UI-related behaviour of one or more widgets changes. This is the case when we change the system value that holds in what area we live. As we have changed the area in which we live, the UI-related behaviour of the widgets must be adapted accordingly. For example when we live in the United States of America and we change the area in which we live to somewhere in Europe, we want the decimal point to be changed into a decimal comma, as this is used in Europe. Therefor the behaviour for the ”decimal”-widget is changed from behaviour that handles and displays the decimal point notation to behaviour that handles and displays the decimal comma notation. Also the behaviour of the operator-buttons should be changed, to be able to handle the decimal notation at hand. Such a switch in decimal notation is shown in figure 3.8.

In figure 3.9 a tree-shaped overview is presented of the impacts that were presented and discussed in this section. As we now have a general overview of the impacts and know what they stand for, we can use this knowledge to specify a number of impacts on which we want to focus throughout this dissertation. This is explained in the following section.

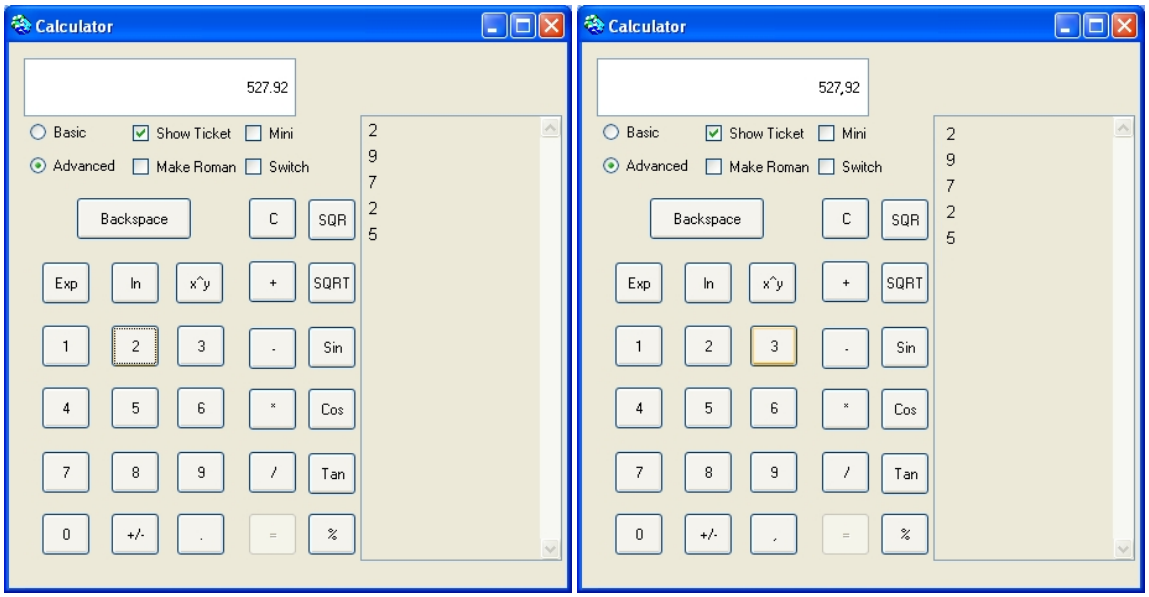

Figure 3.8: The calculator with the decimal point notation (left) and the decimal comma notation (right)

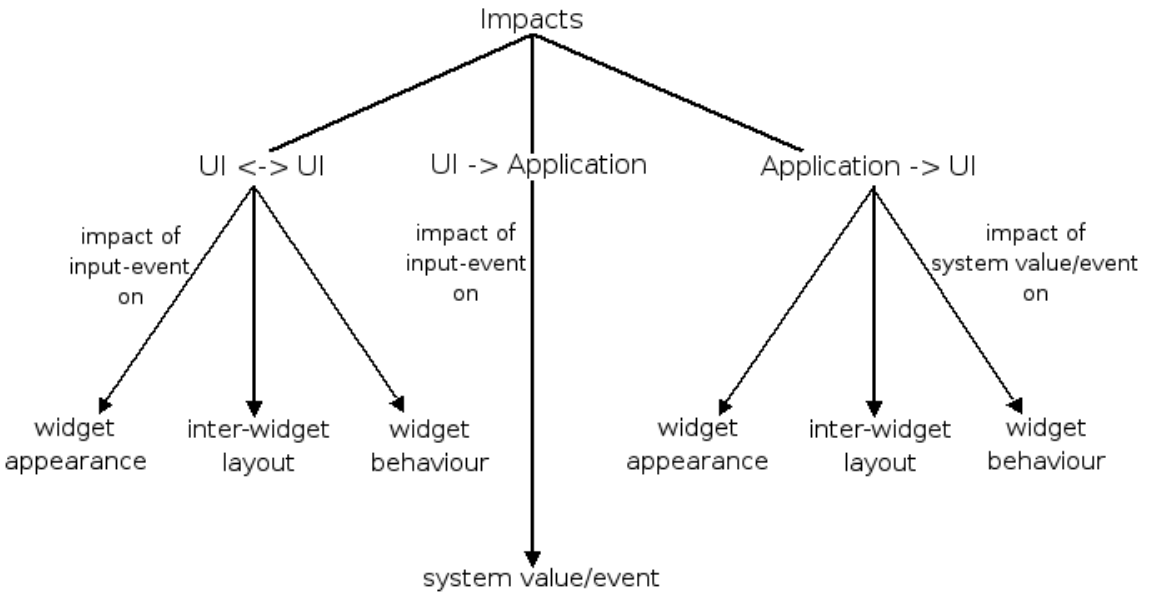

Figure 3.9: Tree-shaped overview of the different impacts

3.3 UI Behaviour

As we have seen the possible impacts of events on aspects, it is now possible to make a selection out of these impacts and focus on it. The impacts we select and focus on throughout this dissertation are *the impact of an input event on widget behaviour, widget appearance and inter-widget layout*, as described in section 3.2.1. These impacts are all within the UI, and can be found in the utter left branch of the tree in figure 3.9. We call these impacts *UI Behaviour* [2].
This brings us to the following definition of UI Behaviour: the behaviour that is executed whenever an event takes place in the UI, with this behaviour in its turn having an impact on the UI. An example is the clicking of a particular button in a UI which causes the background for this UI to change (which is an example of an input-event having an impact on widget appearance).

There are a number of kinds of UI Behaviour at hand, but in this dissertation we focus on state-based UI Behaviour (section 3.3.1).
State-based UI Behaviour is the kind of UI Behaviour that enables the execution of UI Behaviour corresponding to the state the UI is in. For example, when the UI is in a state where its background is black, the color of all widgets is changed to white to enhance visibility. State-based UI Behaviour is explained and defined more thoroughly further in this section.

Furthermore, UI Behaviour is comparable to what we describe as application behaviour, but the difference is that we define application behaviour as the behaviour that is executed whenever an event takes place in the UI, but with this behaviour having an impact on the application, and not on the UI as is the case for UI Behaviour. An example of application behaviour is the clicking of a button in a UI which causes the application to calculate a series of mathematical equations.
However, it is not always possible to categorize some specific behaviour as being UI Behaviour or application behaviour. Some specific behaviours find themselves in something that we can call ”the grey zone” between UI Behaviour and application behaviour, for which it is not clear to which kind of behaviour the specific behaviours belong. As this is the case, the categorization of these specific behaviours is left open for the person who is using them to decide depending on the situation in which they are used to which kind of behaviour they should be assigned in that particular case.
As an example of such behaviour that can not be clearly categorized, we consider the kind of behaviour that is executed whenever an event takes place in the UI and that has an impact on the way the UI and the application interact, for example when the user can mark in the UI how this interaction should be done. This can be considered as well as having an impact on the UI (when the situation has the most impact on the

[2] We write this with capitals, as it is a term we introduced which has more meaning than ”the behaviour of a UI” for which it can be seen without capitals.

UI, for example if the user can choose between switching between a graphical UI and a command-line interface) as having an impact on the application (when the situation has the most impact on the application, for instance when the user can choose a language in which commands can be given in a command-line interface).

With the definiton of UI Behaviour in mind, in the following section we focus on one kind of this UI Behaviour, being the state-based UI Behaviour.

3.3.1 State-based UI Behaviour

We described state-based UI Behaviour above as *the kind of UI Behaviour that enables the execution of UI Behaviour corresponding to the state the UI is in*. This means that it consists at least out of an ordered list of states which the UI can go through, accompanied by the list of UI events (events that happen in the UI) that can bring the UI from one state to another.
Furthermore it consists of pieces of UI Behaviour, which are individually coupled to a particular state of the UI. When all these parts are brought together, it has become possible to look at the UI Behaviour in a state-based way (hence the name). In simple terms this means that whenever a UI event takes place, the UI is possibly brought into a new state and the UI Behaviour corresponding to that state is executed.
For example, when we have a calculator application that has just been started, the UI for the calculator will be in its initial state, as no input has yet been given. However, when we push a number-button, an event on the UI takes place, and the UI for the calculator should change, as the operator-buttons should now be enabled (since a number is entered, we can now also enter an operator). This means that the UI for the calculator will no longer be in the initial state, but that it has moved to a state in which a number was entered to the calculator, which created the need for the changing of the state and the execution of the appropriate UI Behaviour.
We consider state-based UI Behaviour as being a kind of UI Behaviour, as it is behaviour that is executed whenever an event takes place in the UI, with this behaviour in its turn having an impact on the UI. Since only pieces of UI Behaviour are described and coupled to the states in our context, we do not take possible impacts on the application further into account, but from hereon only focus on the impact on the UI.
As it is explained what state-based UI Behaviour is, we mention that the use of the terms state-based UI Behaviour and UI Behaviour in the remainder of this dissertation always refers to this specific kind of UI Behaviour as just described.

Now that the problem was stated in chapter 2 and UI Behaviour has been defined in this chapter, we can start to find a solution for the mentioned problem. Therefor, in the following chapter, we describe the most important artifacts and definitions that will be needed to construct this solution, whereafter the solution is thoroughly explained in chapter 5.

Chapter 4

Foundations for a separation of concerns for UIs

In this chapter, we provide an overview of the most important terms and artifacts that are used further in this dissertation as a foundation to construct a solution for the problem specified in chapter 2.
First a number of terms are explained through definitions. These definitions are given to specify exactly what is meant when a certain term is used in this dissertation, and to provide the appropriate background-knowledge.
The chapter is concluded with an overview of technologies and artifacts that proved useful for solving the problem (as described in chapter 2). Hereby, we present statecharts and we explain the basics of statecharts shortly as we are planning to use statecharts to capture UI Behaviour in (see chapter 5). We also explain CoBro, which we will use in our solution for its concept-centric approach and graphical support (see chapter 5). Furthermore, we also present Carma, which we will use for an aspect-oriented technique (see chapter 5). Finally, we conclude this chapter by presenting Deuce, as the research in this dissertation fits into the research conducted in relation to Deuce.

4.1 Definitions

In this section we provide the reader with the definitions of some important terms that are used further in this dissertation. First some terms that are related to aspect oriented programming (AOP) are explained, followed by a definition of a statemachine as it is used in this dissertation.

4.1.1 Aspect oriented programming terms

We can say that an aspects implementation consists of two conceptually different parts [5]: the aspect *functionality* code and the aspect *applicability* code. The aspects func-

tionality code is not very different from 'regular' code and is executed when the aspect is invoked. *When* and/or *where* this invocation of the aspect takes place is determined by the aspect applicability code. This code contains statements that specify this where and/or when. In standard AOSD terminology [50], the aspect applicability code is referred to as a *pointcut* and the aspect functionality code is referred to as the *advice*.

Joinpoints and pointcuts

In all aspect-oriented programming languages, aspects can only be invoked at some well defined points in the programs execution. These points are called *joinpoints* and the possible kinds of joinpoints are described in what is called a *joinpoint model* [5]. The term joinpoint is described in [50] as follows: *"A joinpoint is a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed"*. A joinpoint model on its turn, is described in [50] as: *"A joinpoint model defines the kinds of joinpoints available and how they are accessed and used"*.
Possible kinds of joinpoints are, for example, assignment statements, method calls, variable references, etc. For each aspect-oriented programming language, a joinpoint model describes these possible kinds of joinpoints and it described how they can be determined in a *pointcut*. Since an aspects behavior can only be invoked at a joinpoint, a pointcut describes the set of joinpoints where the aspects advice needs to be invoked [5].
In [50], a pointcut is defined as *"a predicate that matches joinpoints. A pointcut is a relationship joinpoint -> boolean, where the domain of the relationship is all possible joinpoints"*. In simple terms, a pointcut determines and describes the particular places where we want the advice of our aspect to be executed. What an advice is, is explained in the following part.

Advices

As mentioned, we can describe the *advice* as the part that holds the functionality for an aspect, and this aspects advice code is not significantly different from other code. However, there may be an interesting note.
While many AOP languages have advice code that basically can contain the same kind of procedural source code as standard functions or methods in that language, there are AOP languages where the advice code is expressed in a different language or even in a different paradigm [5]. In the so-called 'general-purpose' AOP languages, it is often possible to use aspect-specific constructs in the implementation of an advice, which do make it a little different from regular method or function implementations.

Weaver

Whenever the advices and pointcuts are determined, AOP languages rely on a specific kind of compiler (or interpreter), called a *weaver*, which is used to insert the aspects

implementation at the desired places in some code.
Compilers for AOP languages are called weavers because they need to weave the aspect code into the modules that are crosscut by the aspect [5]. As a lot of the existing AOP languages were created as extensions to existing object-oriented languages, the weavers mostly transform the programs written in the AOP language into an object-oriented program where the aspect code is inserted (or woven) into the object-oriented implementation modules [5].

4.1.2 Statemachine

Here we give a short definition of a statemachine. In general [53], a statemachine is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. To summarize, a statemachine can be described as:

- An initial state or record
- A set of possible input events
- A set of new states that may result from the input
- A set of possible actions or output events that result from a new state

As we only need this simple definition of a statemachine in this dissertation, we do not explain this further. For a more thorough definition of a statemachine and an overview, one should turn to [22] and [10].

4.2 Building blocks for separating UI Behaviour: technologies and artifacts

In this section some existing artifacts and technologies are presented and explained. These artifacts or technologies are all used later on in this dissertation to solve certain issues and problems (as described in section 5.1), and they are therefor important within the performed research. Therefor, it is necessary for the reader to understand these technologies and artifacts, in order to understand the remainder of this dissertation. First an overview is given of what a *statechart* is and of what parts it consists. Then *CoBro*, which is a concept-centric tool to facilitate the use of domain knowledge, is explained. Thereafter *Carma*, a language for Aspect-Oriented Programming (AOP) in Smalltalk, is presented. To conclude this section, the *Deuce*-framework is explained and the content of this dissertation is positioned within that framework.

4.2.1 Statecharts

Statechart diagrams, also referred to as state diagrams or statecharts, are used to document the various modes (”states”) that a system or an object can go through, and the events that cause a state transition. For example, a calculator can be in the *Off* state, and when the power button is pressed, the calculator goes into the *On* state. Pressing the power button yet again causes another state transition from the *On* state to the *Off* state. Unlike the other behavioral diagrams in UML [20], which model the interaction between multiple objects, state diagrams typically model the transitions within a single object or system.

A statechart diagram is a graph that represents a statemachine (as described in section 4.1.2). States in the statemachine are rendered by appropriate state and pseudostate symbols in the statechart diagram. Transitions in the statemachine are rendered by directed arcs that interconnect those symbols. This can be seen in figure 4.3, in which a statechart diagram is drawn for a calculator application. This statechart is explained further in this dissertation, but for now it is only used to illustrate the parts of a statechart diagram, which are discussed in the remainder of this section.
It should be noted that not all elements that are possibly part of a statechart diagram are discussed here. Only the elements that are relevant for this dissertation are mentioned and explained. All the elements discussed are conform to the OMG UML Specification, as can be found in [39]. For a full overview, one can consult [28] and [29].

State

A state is depicted by a rectangle with rounded corners [39] (as can be seen in figure 4.1) and represent the states that an object can be in. A state can be divided into two parts, optionally separated from each other by a horizontal line. The first part is the *name compartment*. This compartment holds the name of the state as a string. The second part is the *internal transitions compartment*. This compartment holds a list of internal actions or activities that are performed while the object is in that particular state.
Such an internal action consists of two parts: an *action label* and an *action expression*. The action label identifies the circumstances under which the action, specified by the action expression, will be invoked. A number of action labels are reserved for special pre-set purposes and, therefore, they can not be used as event names. The action labels that are reserved [39] are the following:

- The entry-label (OnEntry) identifies an action (which is specified in the corresponding action expression) which is performed upon entry to the state (entry action).

- The exit-label (OnExit) identifies an action (which is specified in the corresponding action expression) that is performed upon exit from the state (exit action).

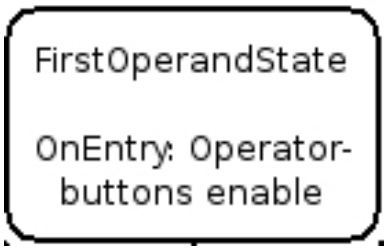

Figure 4.1: An example of a state

- The do-label and include-label are also reserved, but are not further discussed here.

Furthermore, there are special kinds of states [39], such as *composite states*, an *initial state* and an *end state*.

An *initial state* is marked by a solid filled circle. It is the state we are in before any transition has been done in the statechart. For example for objects in an object-oriented environment, this could be the state when instantiated. Only one initial state is allowed in a statechart diagram. The initial state in figure 4.3 can be found in the top left corner. An *end state* is marked by a solid filled circle with a surrounding circle. It is the final point of the diagram. For example for objects in an object-oriented environment, this could mean the destruction of the object whose state we are modeling. Only one end state is allowed in a statechart diagram. The end state in figure 4.3 can be found in the bottom left corner.

A *composite state* can be decomposed into substates. Any substate of a composite state can be a normal state or, in its turn, can also be a composite state. In addition to the name and internal transition compartments, a state may have an additional compartment that contains a region holding a nested diagram, which is used to depict composite states. An expansion of a state into substates is shown by showing a nested statechart diagram within the graphic region.
In some cases, it is convenient to hide the decomposition of a composite state. For example, the nested diagram within a composite state can be very large and it is possible that it simply does not fit in the graphical space available. In that case, the composite state may be represented by a simple state graphic. The contents of the composite state are then shown in a separate diagram. It should be noted that the hiding of the decomposition is just a matter of graphical convenience and that it has no semantic significance.

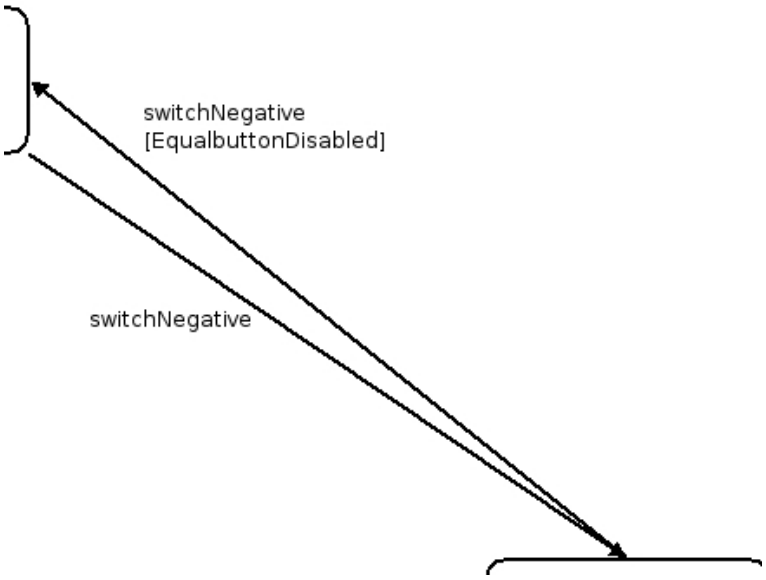

Figure 4.2: An example of a transition

Transitions

A *transition* is a relationship between two states, indicating that an instance that is in the first state can possibly enter the second state when a certain event occurs [39]. When this is the case and we go from the first tot the second state, the transition is said to fire. The trigger for a transition is the event that is labeling the transition.
An *event* can be described as a ”noteworthy occurrence” [39]. For use in statechart diagrams, it is refined to an occurrence that may trigger a state transition. Events are represented as text alongside the transition arrow (labeling the transition), which represents the event that has to take place in order for the corresponding transition to be triggered.

A transition is shown as a solid line originating from the source state and terminated by an arrow on the target state [39]. It can be labeled by a transition string, which possibly consists of an *event* (which was already discussed) and a *guard-condition*, as is shown in figure 4.2.

The guard-condition is a boolean expression, which should be written between square brackets and is placed behind the event of a transition. It describes a condition that is checked when the corresponding transition is triggered. This condition needs to evaluate to ”true” at the moment when the corresponding event triggers the transition in order for that transition to be fired.

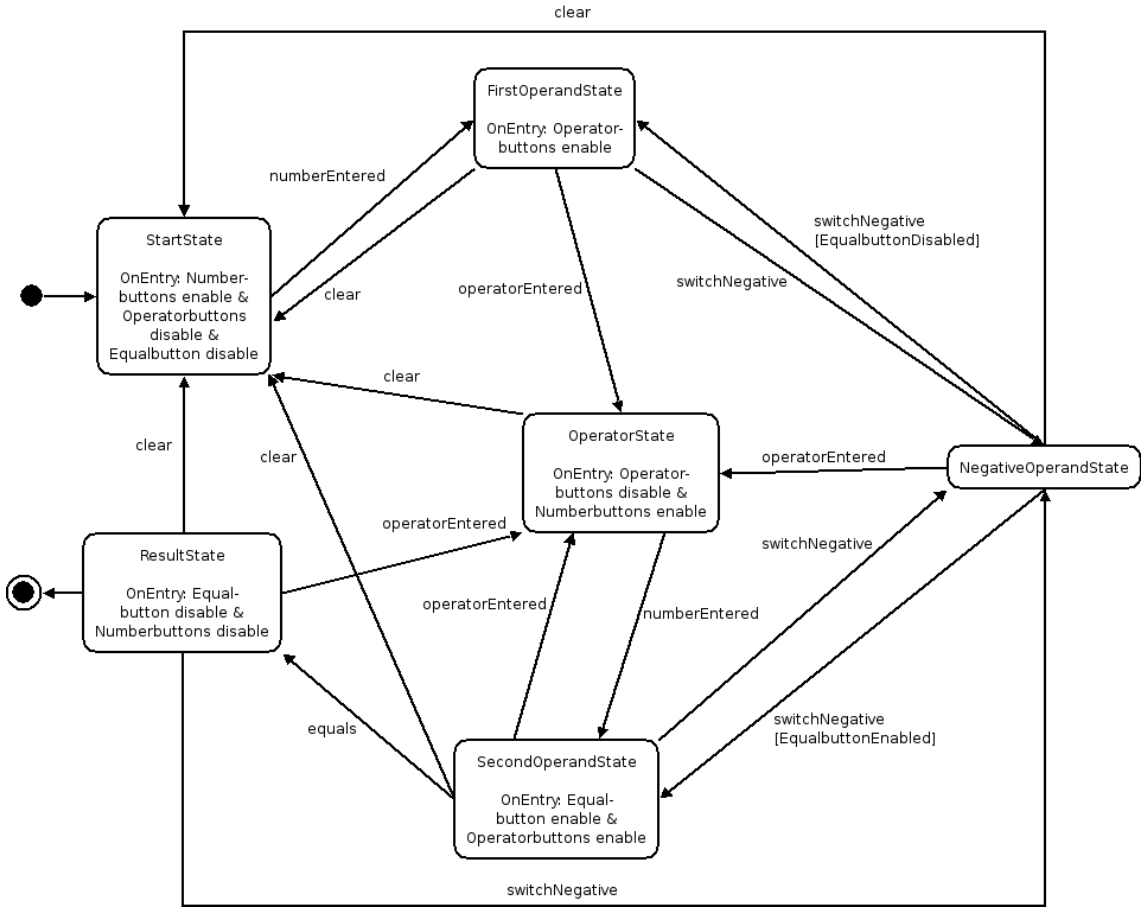

Figure 4.3: An example of a Statechart

For a more thorough and complete explanation of statecharts, one can consult [28] and [29].

4.2.2 CoBro

CoBro is a tool developed at the Programming Technology Lab at the VUB [12] [44]. CoBro, or the Code-to-Concept Browser, is an extension to the standard Smalltalk programming environment. It is a tool that facilitates the active use of domain knowledge by allowing a programmer to capture domain knowledge in an explicit way, namely by representing it in a concept network. To minimize the overhead for the programmer, CoBro is implemented in symbiosis with Smalltalk. This way, it is transparent for the programmer whether he is working with concepts in CoBro or with objects in Smalltalk [44].
Explained in simple terms, CoBro is a tool to allow the programmer to deal with domain knowledge in an explicit way. It represents this knowledge in a *concept network*,

which is a collection of concepts, connected to each other by relationships. Because of this representation, the programmer is able to cope with the complexity of domain knowledge and can add or extract knowledge in a simple manner [12].

Overview of the most important parts of CoBro

When using CoBro, built concept networks should be stored in a way, as we do not want to define the same concept network over and over again. A database for storing all concepts, hence the name *conceptbase*, is therefor created when CoBro is installed. In this conceptbase all *concepts* and *relationships* between them (which are also concepts) are stored and they can be retrieved at any time. These combinations of concepts and the relationships between them form *concept networks*.
Everything in the conceptbase is either a *concept* or a *terminal*.
A concept is something of which we want to keep additional information (in contrast to a terminal, which is explained further). Therefor, a concept consists of a number of entries (slots) in which this information is stored. These entries will contain a relationshipconcept and a collection of destinations, and are used to represent the ”properties” of a concept. For example the concept ”Calculator” may have an entry which consists of the relationship ”hasButton” and the concept ”PlusButton” (a destination for the relationship).
A terminal is something that we do not want to define further. It is a place where we just want to enter a specific value, without further relations. An example of a terminal is a stringvalue of a comment slot (a slot with the relationship ”comment” and its destination being the stringvalue, which we do not want to define further). If we want to define a more advanced entity to which we do not wish to assign just one value, such as a string, then we represent it as a concept.
To illustrate the possible uses of terminals and concepts, consider for example the calculator of a rocket scientist. Novice users with a normal calculator would just represent the functions of the calculator (such as adding, dividing, etc.) as strings (hence terminals) which would represent the name or symbol of the calculator-function. A rocket scientist however could want to define a certain function of the calculator as being used in a certain type of calculation and could want to keep track of what the function exactly does, considering that it is most likely that he will have a calculator with a huge number of functions. Maybe also special properties of the functions can be kept, e.g. that it is invalid to divide by zero. In that case one would represent the functions of the calculator as concepts that have slots that make it possible to represent this information.
Now that we discussed concepts, terminals and relationships, we can have a look at some examples.

A concept example

The following is an example of how concepts are created and how they look in general, based on the mentioned calculator. Firstly, this is the code used when we want to create a new concept in Smalltalk. It is written in CoBro-CML, a language specific for CoBro in which concepts can be expressed, accessed, deleted, etc.

```
c := Concepts new: #{Dividing}.
c hasPreferredLabel: 'The dividing function'.
c superconcept: Concepts.Concept.
c comment: 'This is used in the launch-angle calculation'.
c comment: 'It is invalid to divide by zero'.
c save.
```

In this example, we can see that a new concept (in this case a concept for the calculator functionality of dividing) is created in the beginning with the new-method and saved at the end with the save-method. This is necessary for it to be stored in the conceptbase. Between the creation and the storing of the concept, the concepts entries are specified. This concept has 5 relationships, being the hasPreferredLabel-relationship, the superconcept-relationship, two times a comment-relationship, and a hidden fullyQualifiedReference-relationship. This last relationship is omitted, since it is covered in the new-method in Smalltalk, where the first parameter for this method is matched with the fullyQualifiedReference-relationship.
Every concept should at least consist out of the top three slots of the example, being *a fullyQualifiedReference, a preferredLabel and a superconcept*. A fullyQualifiedReference is used to store the concepts in an unambiguous way in the conceptbase. No two concepts may have the same fullyQualifiedReference. For the concept in the example, the fullyQualifiedReference is #{Dividing}. A preferredLabel is just a string that the user may assign to identify and recognize the concept. Hereby, it is allowed that different concepts have the same preferredLabel. A superconcept is necessary to allow hierarchies to exist. The root of the hierarchy is always the ”Concept”-concept, and therefor everything inherits directly or indirectly from this concept. There are also two extra slots, which are the comment-slots. In these slots the user can specify a comment, which is just a string.
In this example we can also distinguish in which situations terminals and concepts are used. It is shown that for instance the comment-slots and the preferredLabel-slot contain nothing more than just a string, and these relationships will therefor point to a terminal. On the other hand, the superconcept-slot contains a certain concept, and therefor the superconcept-relationship will point to a concept.

Another example is the creation of a new relationship:

```
r := Concepts new: #{isUsedInCalculation}.
r hasPreferredLabel: 'is used in calculation'.
```

```
r superconcept: Concepts.Relationship.
r multiplicity: '#(1 n)'.
r comment: 'A function can be used in a certain calculation'.
r allowedDestinations: '#(#{Concept})'.
r save.
```

In this example we see that a new relationship is created, being the relationship that allows calculator functions to be coupled to certain calculations. Again, a fullyQualifiedReference, a preferredLabel and a superconcept are needed. In this case, the superconcept for our created relationship is not the ”Concept”-concept, but the ”Relationship”-concept (which is an existing concept in our conceptbase), indicating that this newly created concept is a relationship. When defining a relationship, we also need to specify the *multiplicity and allowedDestinations* of the relationships. The multiplicity indicates if the relationship is one-to-one or one-to-many [1]. The allowedDestinations-slot restricts the destinations of the relationship to certain concepts (e.g. the only allowed destination is the Dividing-concept) or more generally to certain kinds of entities (e.g. all terminals or all concepts are allowed destinations). In this example, we see that the relationship can only point to concepts, and not to terminals, as only a concept is an allowed destination. This means we can use the relationship in the example in combination with for instance the ”Dividing”-concept we just created, but not with a string that contains for instance the word 'Dividing'.

Graphical support

CoBro also offers graphical support for most of its functionality, which helps to understand the available domain knowledge and deal with the concept networks. Some of the supporting tools that are available in CoBro are:

- The basic viewer allows us to view concepts present in the conceptbase and their slots.
- The basic editor allows us to edit earlier created concepts (and their slots) or lets us create new concepts.
- The concept locator enables us to search the conceptbase for certain concept, e.g. concepts with a certain preferred label.
- The CoBro-Nav tool allows us to ”navigate” through a concept network, as it shows concepts and the relationships between them.

For a more thorough and complete description of these and more CoBro tools, one should consult [13].

[1] One-to-many holds the possibility to have a relation from one entity to an unspecified number of entities as well as to have a relation from one entity to a specified number of entities (e.g. one-to-three)

4.2.3 Carma

CARMA (formerly Andrew), developed at the Programming Technology Lab [25] [27] [7], is a language for Aspect-Oriented Programming (AOP) [2] [5] for Smalltalk, based mostly on the AspectJ language (an Aspect-Oriented extension for Java), but which makes use of logic meta programming for the specification of crosscuts. As any other Aspect-Oriented Language, it can handle all the general AOP-constructs, but alongside of this, CARMA has several more advanced features as well, which are benefits of using a crosscut language based on logic meta programming. From logic programming it gets the use of unification as a more advanced wildcard mechanism than what is supported in other crosscut languages, the use of logic rules for writing reusable crosscut specifications, and the use of defining multiple rules for the same predicate for writing variants of a crosscut specification. The logic programming language used by CARMA is Soul [26], a Prolog-like language which is implemented in symbiosis with Smalltalk. Furthermore, from logic meta programming it gets features for writing crosscut specifications based on structural properties of the program being crosscut. As such, CARMA is used to write more robust pattern-based crosscuts [44].

An aspect example

This is how an aspect can be defined in Carma:

```
Andrew.TestAspects defineAspect: #CalculatorAspect
    superAspect: #{Andrew.Aspect}
    ofEach: #CalculatorClass
    instanceVariableNames: ''
    aspectVariableNames: ''
    category: 'Calculator-Category'.
```

This is quite similar to how classes are defined in Smalltalk. Like when defining classes, we first have to give the aspect we are creating a name, which is CalculatorAspect. Then we have to define a superaspect, from which the new aspect will inherit. When the new aspect does not inherit from any earlier created aspect in particular, it inherits from the root aspect, which is the Aspect-aspect. Then we can specify a Smalltalk class with which we want to associate the aspect, and then specify the variablenames of the instancevariables and aspectvariables (which are equivalent to classvariables) present in the aspect. To conclude the definition, we specify a category to which the aspect will belong. Now that the aspect has been defined, we need to define our pointcuts, advices and joinpoints. An example of how this is done is given.

```
before ?jp matching {reception(?jp, #calculatorMethod)} do
    Transcript show: 'calculatorMethod is being called'.
```

This is an example with the effect that when a method with the methodname calculatorMethod is called, a string saying it was called will appear on the Smalltalk

transcript. It can be divided into 3 parts. A first part which specifies when the advice [2] is executed, in this case ”before” the joinpoint [2]. The joinpoint is represented by ”?jp” in the example. It is also possible to execute the advice after the joinpoint, for which we should write ”after ?jp” in the example.
In a second part we specify the pointcut [2]. This is the part after the keyword ”matching”. In this example the pointcut means ”at the reception of a message with message-name calculatorMethod”. However, other specifications of the pointcut then the one using the keyword ”reception” are also possible. Other possibilities are for example pointcuts using the ”send” keyword (meaning ”at the moment a particular message is sent”) and pointcuts using the ”assignment” keyword (meaning ”at the moment a new value is assigned to a particular variable”).
In a third part (the part after the keyword ”do”), we define the advice, which is writing something to the transcript in this example. As this is only a simple example, the advice in real situations is often a much longer and more complicated block of Smalltalk-code.
When all the parts are completed, the aspect is woven and the joinpoints are defined by the weaver [2]. This means that for the example, the weaver will check the system for places where a message calculatorMethod is possibly received, and define those places as joinpoints for the newly created aspect. Later, when we run the system and a method with the methodname calculatorMethod is called, the desired text is printed on the transcript.

4.2.4 Deuce

Deuce, or Declarative User interface Concern Extrication, is a framework for creating high-level declarative user interfaces and is currently being developed at the Programming Technology Lab at the VUB. The early stages of Deuce can be found in [23]. The Deuce framework aims for the disentanglement of UI visualization, UI Behaviour and application interactions from the application logic by enabling UI creation from declarative UI specifications. These concerns are described within the Deuce framework as follows. Visualization is what the UI looks like and what widgets are provided. Application interactions describe how the UI hooks into the application and UI Behaviour refers to how widgets relate to and influence each other. These terms are explained more thoroughly further in this section.
By avoiding manual adaptations to the UI and replacing them by UI creation from declarative UI specifications, Deuce avoids having entanglement and scattering introduced to the UI when it is being specified. For these declarative specifications, the logic language SOUL (as mentioned in section 4.2.3 and in [26]) is used, therefor enabling UI creation from certain specified SOUL rules.

[2] These terms are explained in section 4.1.1

Declarative User Interfaces

The goal that is set in Deuce for separating UI logic from application logic, is to express the UI-concern as well as the application-concern declaratively in such a way that the UI is specified on a higher level. This would mean that the programmer does no longer have to deal with the low level entanglement [23].
To reach this goal, it is first needed to specify the UI logic, for which an expressive medium can be useful [23].
Secondly, it has to be possible to generate certain parts of UI actions and interactions, as the programmer should be freed of the burden of maintaining those certain parts [23].
As Deuce is not intended to be an ad hoc system, it wants to provide a general solution that can help the programmer in creating UI logic. To provide such a general solution, *declarative programming* is used as a way to express UI concerns.

Now the goal for Deuce was discussed, the several concerns that need to be taken into consideration when separating UI logic from application logic are explained more thoroughly: *UI visualization, UI behaviour with respect to interactions on the UI level and UI behaviour with respect to interactions between the UI and the application.*

UI visualization

The UI visualization describes for instance the specification that a label with a certain kind of properties should be positioned above an input field (and not for example next to the input field). The specification for the visualization concern in Deuce thus describes the graphical elements of a UI through a number of facts and rules. By the use of declarative rules, more general visualizations can be described. However, even though the positioning of UI elements is a part of the UI visualization, it still can be influenced by the application, for example when some application value had an impact on how the UI looks (see also chapter 3). This can mean that the UI visualization needs to be changed while it is in use and elements of the UI may need to be repositioned. This would mean that a mechanism for automatically laying out the elements is required [23].
To achieve this, a layout relation can typically be transformed into a linear constraint equation [23], for which a *declarative reasoning mechanism* can be used to perform this transformation. A *linear equation constraint solver* then resolves these layout relations and automatic layout can be achieved through the constraint system [23].

UI behaviour: Interactions at the UI level

An example of an interaction at the UI level is the disabling of a certain button while a specific input field is empty. These kind of interactions between UI elements are inevitable when creating UIs [23].
It is said that a UI has a certain state at a certain moment, and certain actions or events

can possibly change that state. Often programmers express this behaviour by means of *statecharts* (see section 4.2.1), which are then implemented manually into the application code, which again results in entangled code. In Deuce, expressing statecharts will be approached as a declarative process that can be done by the use of facts and rules, which in their turn can be translated into actual UI actions. By reusing these facts and rules, it also becomes possible to reuse certain states or behaviours of the UI.
When wanting to make runtime adaptations to the UI (such as adapting the behaviour), dynamic interactions are required [23]. From a declarative point of view this means that whenever a new fact is known, other new facts should be inferred. Therefor Deuce will make use of a *forward chainer*, and thus data-driven reasoning.

UI behaviour: Interactions between UI and application

Clicking a certain button in a UI may not have an impact on the UI, but have an impact on the application, as a UI event (i.e. clicking the button) can trigger certain application events (such as the execution of a method). This means there is a link between UI and application.
Like the other concerns in Deuce, how UI actions relate to application actions is again described declaratively. As linking the UI to an application involves code generation as well as code adaptation at the application level, Deuce considers *aspect oriented programming* to tackle this problem [23].

Declarative UI specification framework

Within Deuce, the three levels of specifications are combined, which leads to a declarative user interface specification framework that helps the programmer when developing the UI logic. This framework can be extended with different strategies, specifications or reasoning mechanism if wanted to [23]. However, the declarative UI specification framework already has the necessary declarative reasoning mechanisms built in.
As a starting point, Deuce uses the *declarative meta programming language* SOUL (as mentioned in section 4.2.3 and in [26]) which has been implemented on top of Smalltalk. SOUL is used to specify UIs by means of facts and rules, while the application logic is implemented in Smalltalk. SOULs reasoning mechanism uses the UI facts and rules to generate an actual Smalltalk UI.

Contribution to Deuce

As the Deuce framework is still under development, more insights are continuously acquired as more research is done and more contributions are made. This dissertation is such a contribution to the research involving Deuce, as it disentangles UI Behaviour from the rest of the application, based on statecharts. As the Deuce framework aims to disentangle as well the UI visualization, as the UI Behaviour and as the application interactions from the application logic, it is clear that the scope for Deuce is much

broader than the scope for this dissertation. Nevertheless, the solution presented in this dissertation and the conducted research are closely related to solving the issue of disentangling only UI Behaviour in Deuce. Therefor we can say that this dissertation fits into the Deuce framework, as it handles just one problem of the problems stated for the Deuce framework, being the disentanglement of UI Behaviour from other concerns.

Now that we discussed all artifacts that are needed as a foundation for our solution and the most important terms are explained, in the following chapter a solution is constructed by using these artifacts. This solution is aimed at solving the problem as stated in chapter 2, and also uses the definition of UI Behaviour, as presented in chapter 3. After the construction of the solution in the following chapter, the solution is validated in chapter 6.

Chapter 5

Disentangling state-based UI Behaviour

In this chapter we provide a solution to the entanglement of state-based UI Behaviour (as defined in chapter 3), a problem that was explained in chapter 2.
We begin this chapter with an overview of the separate conceptual components that need to be solved in order to solve this entanglement, accompanied by the conceptual solutions to the problems in these components. This helps in getting an abstract overview of how the solution is constructed and how it works.
Then these conceptual solutions are translated into a practical solution, which is explained step by step. Hereby the system that is implemented to support this practical solution is thoroughly explained. We will also find the (in chapter 4.2) presented artifacts to be used in this practical solution.
We conclude this chapter with an overview of the most important benefits and shortcomings of the presented solution. These will provide a good overview of what the strong sides of the solution are, as well as discussing the points where there is still work to be done in improving the solution.
This solution (and its benefits) is then validated in chapter 6.

5.1 Disentangling state-based UI Behaviour: a conceptual view

This section explains the conceptual solution to the problem, as we described it in chapter 2. The solution is constructed by reasoning about what approaches and concepts are necessary to resolve the problem, which is split up into separate components. It is found that the following *four components of the problem* should be taken into account in order to solve the stated problem of applying separation of concerns on UIs, while focussing on the UI Behaviour.
The *first* component is the need for a way to *represent the UI Behaviour* on its own, as we want to separate it from other concerns in our application.
As a *second* component, a way to *create an application-level equivalent* for this rep-

resentation of the UI Behaviour should be provided, so we are able to use the UI Behaviour in our application.
The *third* component is that we need a way to *couple this UI Behaviour to our application*, so that the appropriate UI Behaviour can be addressed at the appropriate time.
The *fourth* issue is that we generally need to guarantee that the *amount of overhead* for the developer is reduced in comparison to the solution that was used before by the developer. Otherwise the developer will not be compelled to use our solution.
These four components will be explained and resolved in this section. How this conceptual solution is then transformed into a real system, which makes it possible to apply the solution to real applications, is described in section 5.2.

5.1.1 Separating UI Behaviour from other concerns

As mentioned in chapter 2, we want to apply the principle of separation of concerns to UIs, and then more specifically to UI Behaviour. How this separation is already feasible inside the developers mind is illustrated in figure 5.1, where we see the application that is split up inside the developers mind into UI Behaviour and the rest of the application (which is possibly also split up into separate parts, but those are not considered here), visible at point number one. It is exactly this separation that we also want to keep outside the developers mind. A first step to achieving this, is to find a way to *represent this UI Behaviour separated from other concerns*. This means that only the UI Behaviour should be grasped within this representation, with as little as possible of (pieces of) other concerns represented in this same representation.
When such a separation of the UI Behaviour from the rest of the application is achieved, we should ensure that the method for achieving such a separation is *not overly complex or time consuming* for the developer. This is explained further in section 5.1.4.
However, as this method should be kept from being overly complex and time-consuming, we should also ensure that this method is *able to represent the UI Behaviour* in all its forms and with its necessary details. When the chosen method does not have the appropriate means to express UI Behaviour in all its forms, the developer will encounter situations in which he is not able to express the needed UI Behaviour. This will refrain the developer from using the presented solution.

We have now mentioned a number of conditions that should be met in order to achieving a first step towards a separation between UI Behaviour and the rest of the application. To meet all these conditions, we decided to use *statecharts* (as described in section 4.2.1). Statecharts are accepted as a conventional way of specifying behaviour [20], and are also suitable for specifying UI Behaviour [31].
Furthermore, statecharts are already known and used by a large number of developers and integrated in a large number of real-world systems, as stated in [36] and [1]. Therefor we are interfering as little as possible with the developers way of working, as it is most likely that statecharts are already in use and it is therefor possible for the developer to keep using them as before. By using statecharts, we also avoid the need to

introduce a new designmethod, which would bring along a certain learning overhead for the developer. Finally, statecharts are also suitable for representing UI Behaviour because they are expressive enough, as stated in [31].
As we hereby selected statecharts as a suitable way of specifying the UI Behaviour, accordingly, the first step in the practical solution (which is described in section 5.2) will be to specify the wanted UI Behaviour in a statechart. This can also be seen in figure 5.1 at point number two, where we see how in the first step towards a separation the UI Behaviour is specified in a statechart.

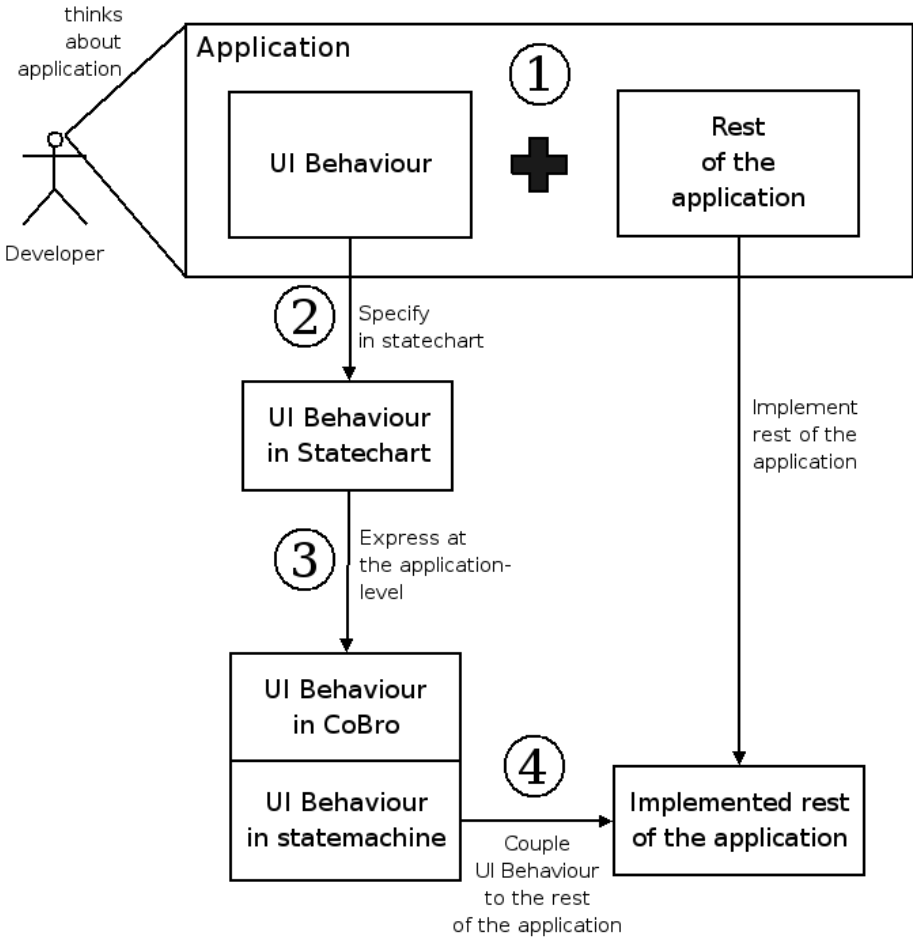

Figure 5.1: A graphical representation of the conceptual process

5.1.2 Expressing and separating UI Behaviour at the application level

As the UI Behaviour has been separated conceptually by the developer from the rest of the application (as explained in section 5.1.1), we want this *separation to remain at*

the code level as the UI Behaviour is implemented. This means that we want a separation of the UI Behaviour code from the other application code while implementing the application, which facilitates the implementation (as mentioned in chapter 2).

However, in order for a separation at the code level to be possible, there first needs to be an *application-level equivalent of the UI Behaviour*. By this we mean that the UI Behaviour that was captured in the statechart (as described in section 5.1.1) should be expressed in such a way that it can be used by the rest of the application. As the UI Behaviour captured in a statechart can not directly be used by the rest of the application (since it was expressed by a statechart drawn on paper or drawn in an UML-tool), we need to provide an equivalent that can be used, otherwise the rest of the application can never use the UI Behaviour.

Note: this manual specification of the application-level equivalent by the developer needs improvement, as in the future we would want the developer to draw a statechart (for example in CoBro or a UML-tool) and then automatically transform this statechart to an application-level equivalent. Nevertheless, this is not yet the case, and therefor the developer still needs to do this manually. This is discussed further in section 5.3.2.

Furthermore, if the developer is forced to make the separation between UI Behaviour and the rest of the application manually, the advantage of using the presented solution is lost, as the developer is still responsible for maintaining the separation while implementing the application, which we wish to avoid. Therefor, we should ensure that it is integrated in our solution that the *developer only needs to specify the UI Behaviour* and *provide an application-level equivalent of this UI Behaviour for the application to work with* [1], and that the separation is handled by the solution, and not by the developer. This means that the separation of UI Behaviour and the rest of the application at the code level would become automatically (by our solution) instead of manually (by the developer). Note that this manual approach is exactly what is proposed in [31], in which a separation is achieved through a manual transformation of a statechart containing UI Behaviour into code.

For allowing the developer to provide an application-level equivalent of the UI Behaviour, we should provide a way of expressing this UI Behaviour that reduces the overhead for the developer (as mentioned in section 5.1.4). We should keep in mind that we also want the developer to specify the application-level equivalent in such a way that it *stays adaptable* later on. If not, with every change of the UI Behaviour (specified in the statechart), the developer is forced to provide a completely new application-level equivalent, which is hardly better than having to search in the application to adapt the UI Behaviour.
Therefor, we let the developer express the application-level equivalent of the UI Be-

[1] See note.

haviour in a *concept-centric medium with graphical support*, in which the developer can create such an equivalent (mapping the statechart to an application-equivalent - in this case a concept network - is shown in section 5.2.6), and in which adapting the application-level equivalent later on is facilitated through the presence of a graphical representation, which can help the developer as this graphical representation of the application-level equivalent resembles the initial statechart [2].

To meet the conditions as described, we use the tool *CoBro* (as described in section 4.2.2) and a *statemachine* (as described in section 4.1.2).
CoBro is used as a concept-centric medium with graphical support [12] in which the developer can express the application-level equivalent of the UI Behaviour by the use of CoBro-specific language statements, called CoBro CML (as mentioned in section 4.2.2), which we will simplify even further specifically for UIs by adding syntactic sugar. Thanks to the graphical support in CoBro, the application-level equivalent of the UI Behaviour can be graphically viewed, which can help the developer to adapt the equivalent later on.
As CoBro is used by the developer to specify the UI Behaviour, we also need an application-level equivalent that can be used by the application, in order to execute the UI Behaviour. Hereby a statemachine is used to provide an executable version of the UI Behaviour (which was captured in a statechart), which can be called upon from in the application whenever UI Behaviour is needed.

This step is illustrated in figure 5.1, where we see at point number three that the UI Behaviour in the statechart is transformed into an application-level equivalent in CoBro.

5.1.3 Coupling UI Behaviour to the application

Once the UI Behaviour has an application-level equivalent (as described in section 5.1.2) that can be used by the rest of the application, the rest of the application still needs to access this UI Behaviour whenever it is needed in the application. Therefor we need a way to *couple the UI Behaviour to the application*, so interaction becomes possible. This coupling should also be automatic, and not done by the developer, as otherwise the developer still needs to manually make adaptations in the code of the application, which we wish to avoid. To achieve this, we can use a part of the technique that is used in Aspect-Oriented Programming (AOP), which is especially designed to weave separated concerns on an application. In this technique, pieces of code, representing a certain concern (named aspects), are taken and they are woven in particular places - which were defined earlier by the developer - in the rest of the application.
In our solution, we use the ability of the aspect-technique to couple the UI Behaviour to our application, by making aspects that represent the UI Behaviour and weaving these

[2] This is illustrated in section 5.2.7.

aspects on particular places in the application. These particular places are defined by the pointcuts (as explained in section 4.1.1), in which is specified in which parts of the application the UI Behaviour is used. The advice of the aspects then calls the right UI Behaviour (accordingly to which aspect was triggered) in the application-level equivalent of the UI Behaviour.

It should be noted that however we are applying a separation of concerns on the UI behaviour, the system supporting this separation is not programmed in a pure aspect-oriented way, as we only use a small part of AOP and also use this part in a different way than it is supposed to be used in AOP (this is explained in section 5.2.9). This also means that no matter what aspect-oriented language or tool that we use for our solution, it will always bring along a large number of unused features, since we only need and use the ability of AOP of weaving in pieces of code in existing code. This may cause the use of the aspect-technique to seem like ”overkill”. Nevertheless, these extra features that are available, can be of use in a later stage. This is discussed further in this dissertation.
Also, in the future we would want to evolve towards a solution that uses a pure aspect-oriented approach, which would be an improvement of the current system (see section 5.3.2). Therefor, already a small part of the aspect-oriented approach is integrated into our solution (even though we do not use AOP yet), which can be a foundation for introducing the pure aspect-oriented approach into our solution.

Furthermore, when we use the aspect-technique, we should ensure that the part that we are using of that technique is able to express the UI Behaviour (like was the case for the statecharts in section 5.1.1). Else the developer may encounter situations in which he has specified a part of the UI Behaviour, but this part can not be expressed with the used technique, leaving it impossible to couple that part to the application. This would mean that parts of the UI Behaviour would be lost, which is not acceptable.

To meet these conditions, we use *Carma* (as described in section 4.2.3), which is a language for Aspect-Oriented Programming in Smalltalk and which therefor can use the aspect-technique to couple the UI Behaviour to the rest of the application. This is shown in figure 5.1 at point number four, where the application-level equivalent of the UI Behaviour is coupled to the implementation of the rest of the application.
Carma also is able to represent the UI Behaviour as aspects. This expressivity comes from the use of Smalltalk (on which Carma is based) and is enhanced because of the integration of the possibility to use logic rules when wanted. As explained in section 4.2.3, Carma can use logic rules in the specification of the aspects, which make it possible to express constraints in the UI Behaviour. These constraints are not necessary to express UI Behaviour, but in some cases the constraints can be useful to express the wanted UI Behaviour (e.g. in complex cases). This is discussed further in section 5.2. The logic rules are also stored separately in a logic repository, which has the extra advantage that they can be adapted separately.

As mentioned before, we are only using a small part of the aspect-technique, and therefor there are a large number of features of Carma that we do not use. However, the possibility to use logic rules in Carma provides us with the opportunity to use this as an extra feature, for example to enhance expressivity (section 5.2.9).

As the components in sections 5.1.1, 5.1.2 and 5.1.3 only provided conceptual solutions for the given problems, we also provided a practical implementation of these solutions through the creation of a system that supports the separation of the UI Behaviour, which is explained thoroughly in section 5.2.

5.1.4 Reducing complexity

The previous components described the individual problems that should be solved in order to solve the greater problem of applying separation of concerns on UIs, while focussing on the UI Behaviour. As we have seen, it is possible to find a solution to all of these problem-parts, therefor resulting in a solution for the greater problem as stated in chapter 2. However, next to providing a solution for the problem, we should also ensure that the *solution we offer does not become too complex and time consuming* for the developer to actually use it. By this we mean that the presented solution should not be more complex and time consuming for the developer than the ad hoc solution that the developer used before to separate UI Behaviour from the rest of the application.
With this in mind, the solution is constructed in such a way that components are automated where possible, in order to reduce complexity. As most parts of the presented solution are automatic, the developer only needs to *specify the UI Behaviour in a statechart* and then *provide an application-level equivalent* of this UI Behaviour, which we define in this dissertation as *less complex than manually maintaining the separation* between UI Behaviour and the rest of the application while implementing the entire application. Since the developer only needs to do the above mentioned two steps (and the rest is automated), the developer does not need to come in contact with Carma, and in limited contact with CoBro, which allows the developer to know for example nothing about Carma, but still use our application.
By using our solution, the task of the developer is switched from making manual adaptations to the code while maintaining the separation into drawing a statechart and transforming this statechart into an application-level equivalent. Therefor, in this dissertation, we consider our solution as less complex and less time consuming than the ad hoc solution.

5.2 Disentangling state-based UI Behaviour: a practical view

This section explains the practical solution to the problem as it is described in chapter 2, based on the conceptual solution that was presented in the previous section. We begin this section by explaining a basic calculator application, which we will use throughout this section as an example. Then we discuss a preliminary step that is not part of the solution, but is necessary in the development process. Then an overview is given of the different parts of the solution, separating the parts in the solution which are supported by our system from those that are not. This overview is illustrated by a graphical representation of the solution. In the remainder of the section, the different steps in the solution are separately explained thoroughly and illustrated with the calculator application. At the end of this section, when all steps have been executed, a new UI and application will have been created, during the creation of which the UI Behaviour will have been separated from other concerns.
As a further validation and example for this solution a more advanced calculator-application is presented in chapter 6.

5.2.1 Case study: a basic calculator

Throughout this section we will use one application consistently as an example to illustrate the different parts of the solution as they are explained separately. This application is the basic calculator application. A screenshot of this application is provided in figure 5.2.

This calculator application is a simple application, which contains the most basic behaviour for a calculator. At the top of the screenshot we see an inputfield, in which numbers can be entered through the keyboard or by using the number-buttons provided. As a number is entered, the user can press an operator button (such as +, -, etc.) to perform the wanted operation and then again enter a number. At the end, the equal button can be pressed, and the result of the calculation is displayed in the inputfield. The calculator can be reset by pressing the clear button (”C”). Numbers can be made negative or positive by using the ”+/-”-button.

In figure 5.2 we find, besides a screenshot of the calculator application, also a series of numbers, indicating the impacts that certain events in the UI can have on certain aspects of the UI or application (as described in chapter 3). These impacts are described in the following list, where the number in the list matches the number of the impact indicated in figure 5.2. It should be noted that we only take the impacts within the UI into consideration, as this is the part on which we focus throughout this dissertation (as mentioned in chapter 3).

1. An input-event on a widget (the inputfield) has an impact on widget appearance:

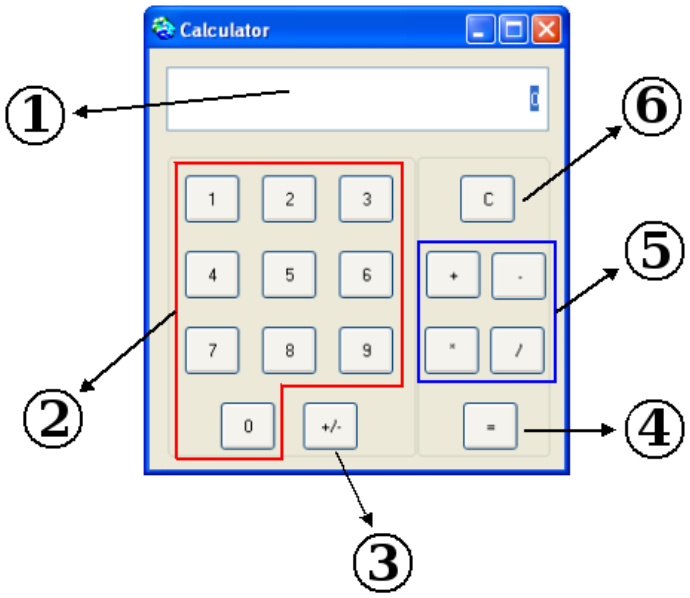

Figure 5.2: The UI for the calculator application, with the corresponding impacts

the operator-buttons are enabled (when entering a first operand of a calculation) or both the operator-buttons and the equal-button are enabled (when entering a second operand).

2. An input-event on a widget (any number-button) has an impact on widget appearance: the operator-buttons are enabled (when entering a first operand of a calculation) or both the operator-buttons and the equal-button are enabled (when entering a second operand).

3. An input-event on the ”+/-”-button only has an impact on a system value (in which the current calculation value is stored). It has no impact within the UI and therefor is not further discussed.

4. An input-event on the equal-button has an impact on widget appearance: the number-buttons are disabled, the operator-buttons are enabled and the equal-button disables itself.

5. An input-event on any operator-button has an impact on widget appearance: the number-buttons are enabled (if they were disabled in the first place) and the operator-buttons themselves are disabled.

6. An input-event on the clear-button has an impact on widget appearance: it brings the calculator back into its original starting mode, which means that the operator-buttons and the equal-button are disabled and the other buttons are enabled.

These impacts within the UI describe the UI behaviour that we wish to separate from the rest of the application. This is explained further in section 5.2.4.

Now that we explained the basics of the calculator application and have shown the different impacts (within the UI) in it, we can use this application throughout this section to explain the different parts of the solution. This solution is illustrated with an abstract overview in section 5.2.3.

5.2.2 Developing the UI

The first step in the process of developing a UI is depicted in figure 5.3. This step is not as such a part of our solution, but it is a necessary preliminary step that should be made by the developer to develop the UI [41] [45]. In this step, the developer needs to design the UI for the desired application, hereby specifying the visualization of the UI (depicted as layout), as well as the behaviour for the UI, which in its turn can be split into 2 parts. We can speak of application interactions (how the UI hooks into the application) and UI Behaviour, as described in section 3. These parts can be seen as separate concerns, and thus the developer can make a separation of concerns while developing the UI. It is this separation of concerns that we want to hold on to while implementing the entire application. However, as mentioned in chapter 2, in this dissertation we only focus on the UI Behaviour, which is marked in figure 5.3 with a circle. Therefor the separation of the other concerns is not further explained and investigated.
In order to reach such a separation of concerns for the UI Behaviour, the developer can use our solution, which is supported by an implemented system and which is presented in the remainder of this section. The developer should then follow our solution while developing an application and its UI, which will result in the creation of that application and its UI, but with the UI Behaviour separated from other concerns during the development.
To provide a global picture of our solution, we first give an abstract overview, which is illustrated in an accompanying figure. Then the different parts of the solution and the supporting system are thoroughly discussed.

5.2.3 An abstract overview

As we look at figure 5.4, we can directly distinguish two big parts. The part within the red box consists of preliminary steps that should be made by the developer. These steps are done manually [3]. The part within the blue box depicts the system that is implemented in order to support our solution. The steps in this box are done automatically by the system, given the input from the developer from in the ”manual” steps. Keeping this in mind, we now provide a short abstract overview of our solution, based

[3] As mentioned before, we would like to improve the solution in such a way that the representing of the statechart in language statements also becomes automatic, but this is not the case at the moment.

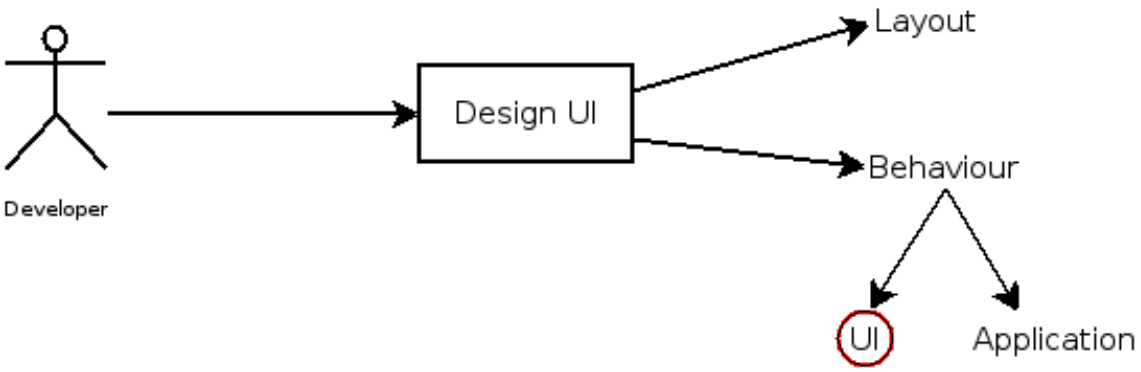

Figure 5.3: Designing a new UI, already separating concerns

on figure 5.4. For a detailed explanation of each part of the solution, we refer to the corresponding section in the remainder of this dissertation.

As mentioned, the part within the red box consists of preliminary, manual steps done by the developer. Firstly a statechart should be drawn to represent the UI Behaviour (by which we still mean the state-based UI Behaviour, as said in section 3.3) for the application the developer wants to create and its UI. This was explained conceptually in section 5.1.1 and is explained with a more hands on approach in section 5.2.4.
Furthermore, the wanted application and the UI should be implemented by the developer while leaving out the UI Behaviour. During this implementation, the developer should not take the UI Behaviour into account, except on the places in the application where the UI Behaviour needs to be used. On these places the developer should follow a certain naming convention (which is explained further in this dissertation) for methods, which then can be used to couple the UI Behaviour to the application later on (section 5.2.5).
To end the steps that should be made by the developer, the developer should transform the earlier drawn statechart into a application-level equivalent, by representing it in language statements. This was described conceptually in section 5.1.2 and is explained further in section 5.2.6.

Once the manual steps are done, the developer is no longer involved in the steps and the system supporting the solution takes over. This means that the system performs the rest of the solution automatically. This is depicted in the blue box.
First, the system takes the language statements that are given by the developer as input, and constructs a concept network based on this input (as mentioned conceptually in section 5.1.2 and further explained in section 5.2.7).
This concept network is then used by the statemachine builder and the aspect generator, which respectively build a statemachine (section 5.2.8) and generate aspects (section 5.2.9), along with specifying the joinpoints. The statemachine is used to be able to

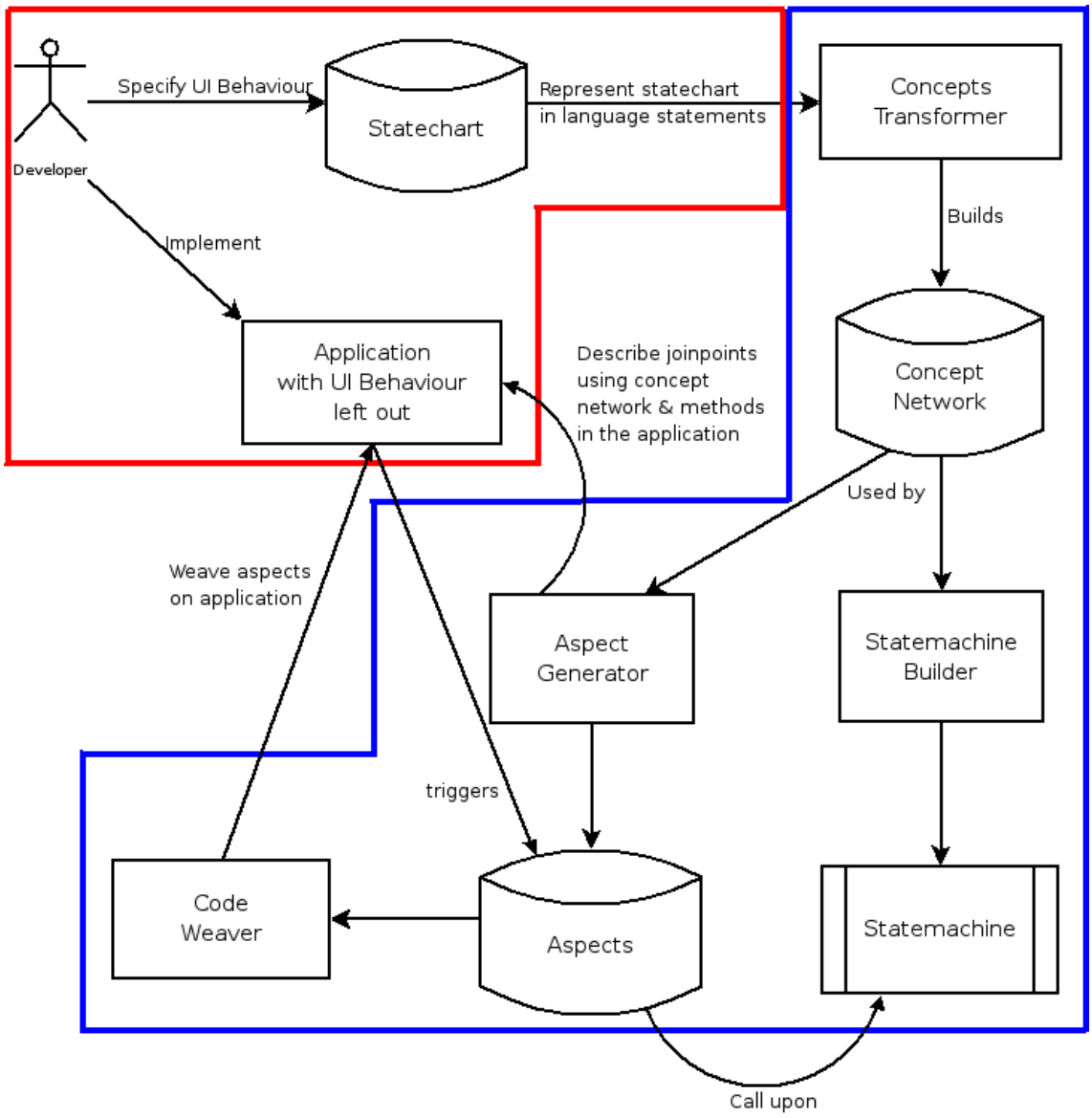

Figure 5.4: The process of creating an application and its UI with our solution

execute the UI Behaviour when it is needed. The aspects are used to call upon the UI Behaviour in this statemachine whenever the aspects are triggered in the application, meaning that UI Behaviour is needed at that place in the application.
These aspects are then woven on the application, providing a coupling between the application and the UI Behaviour (section 5.2.10, and explained conceptually in section 5.1.3), as the aspects can now be triggered by the application, allowing UI Behaviour to be executed by the statemachine. The joinpoints (where the aspects should be woven) are described by the aspect generator, which makes sure that the aspects are woven in the places in the application where UI Behaviour is needed, based on the concept network and methods in the application itself.

When all these steps are done, the wanted application and its UI will be created and will work normally, but during the development, the UI Behaviour will have been separated from other concerns.
The remainder of this section is used to explain and illustrate each part in the given overview separately.

5.2.4 Drawing a statechart

The first step in our solution is to *draw a statechart in which the developer specifies the UI Behaviour* for the particular application and UI that need to be created. This statechart should represent the state-based UI Behaviour (as described in chapter 3) and should be drawn by the developer in the standard way of drawing statecharts, as explained in [31]. As we suppose that the developer used common practice design-methods [31] [20], we expect this statechart (or more statecharts) containing the UI Behaviour to be drawn already during the design-phase. Such a statechart for the basic calculator application can be found in figure 5.5.
Furthermore, the developer should make sure that all states and transitions necessary for representing the state-based UI Behaviour are specified in the statechart. For example when the state-based UI Behaviour can come to an end in a certain state of the UI, the developer should represent this in the statechart by marking the appropriate state as the end-state. This can be seen in figure 5.5, as we have marked the ResultState as a possible end-state. Another example is a button that needs to be enabled when entering a certain state of the UI. This can be achieved by entering the enabling of that button in the ”On Entry”-field of the state in the statechart (as described in section 4.2.1) for which, when entered, the button should be disabled. This is the case for the equal button in the calculator, which is enabled whenever we reach the SecondOperandState, as seen in figure 5.5.

To specify the statechart and/or check if the wanted UI Behaviour is expressed in it, the developer can use the impacts presented in chapter 3. When the developer has speci-

fied all different impacts that are possible within the UI [4] of the application that needs to be implemented, these *impacts can be used to specify and check the statechart*.
When the developer is specifying the states of the statechart representing the UI Behaviour of a certain UI, the impacts within this UI, more specifically the aspects [5] of these impacts (see chapter 3), can be used to specify the entry- and exit-actions of these states. This can be seen in the calculator statechart (in figure 5.5), in which the entry-actions of the states are exactly the same as the aspects of the impacts within the calculator-UI, presented in section 5.2.1.
Furthermore, when specifying the transitions of the statechart representing the UI Behaviour of a certain UI, the events [6] of the impacts within this UI can be used to specify these transitions. It is shown in figure 5.5 that the events of the transitions in the statechart are the same as the events of the impacts as described in section 5.2.1.

This can be illustrated by using our basic calculator example, for which the possible impacts within the UI are described in section 5.2.1 and the statechart is shown in figure 5.5. Therefor, we will now briefly explain the statechart containing the UI Behaviour for the calculator application and discuss the connection of this statechart to the impacts within the calculator-UI (as described in section 5.2.1).

In the statechart depicted in figure 5.5, we see that we have six different states. One of these states is the StartState, where the application has just been started and no input has been received yet. At this point, the user should only be able to enter a number, and not for instance an operator, as there is nothing to apply the operator on. Therefor, only the number-buttons are enabled. This is also exactly what is described in the aspect of impact number six in section 5.2.1, where the impact of pushing the clear-button (which resets the calculator in his initial state) is exactly the enabling of the number-buttons and disabling of the operator-buttons and the equal-button.
Whenever a number is entered, we enter the next state, which is the FirstOperandState. The entering of a number can be compared to the input-event in impact one and two, where a number is entered by pushing a button or by entering it in the inputfield. Here we see that this input-event is translated to a transition in the statechart. In the First-Operandstate, a number has been entered and the user is allowed to enter an operator, so the operator-buttons are enabled. This can be compared to the aspects of impacts one and two.
At this point, there are two possible states that we can reach.
When an operator is entered (the input-event of impact five), we reach the Opera-

[4] We only focus on the impacts within the UI, as they are useful in specifying the statechart containing the UI Behaviour. The other impacts may also be useful for other designing purposes, but these are not investigated further in this dissertation.

[5] We are not referring to the term aspect as used in aspect-oriented terminology, but to the term aspect that was introduced in chapter 3.

[6] How events are described is found in chapter 3.

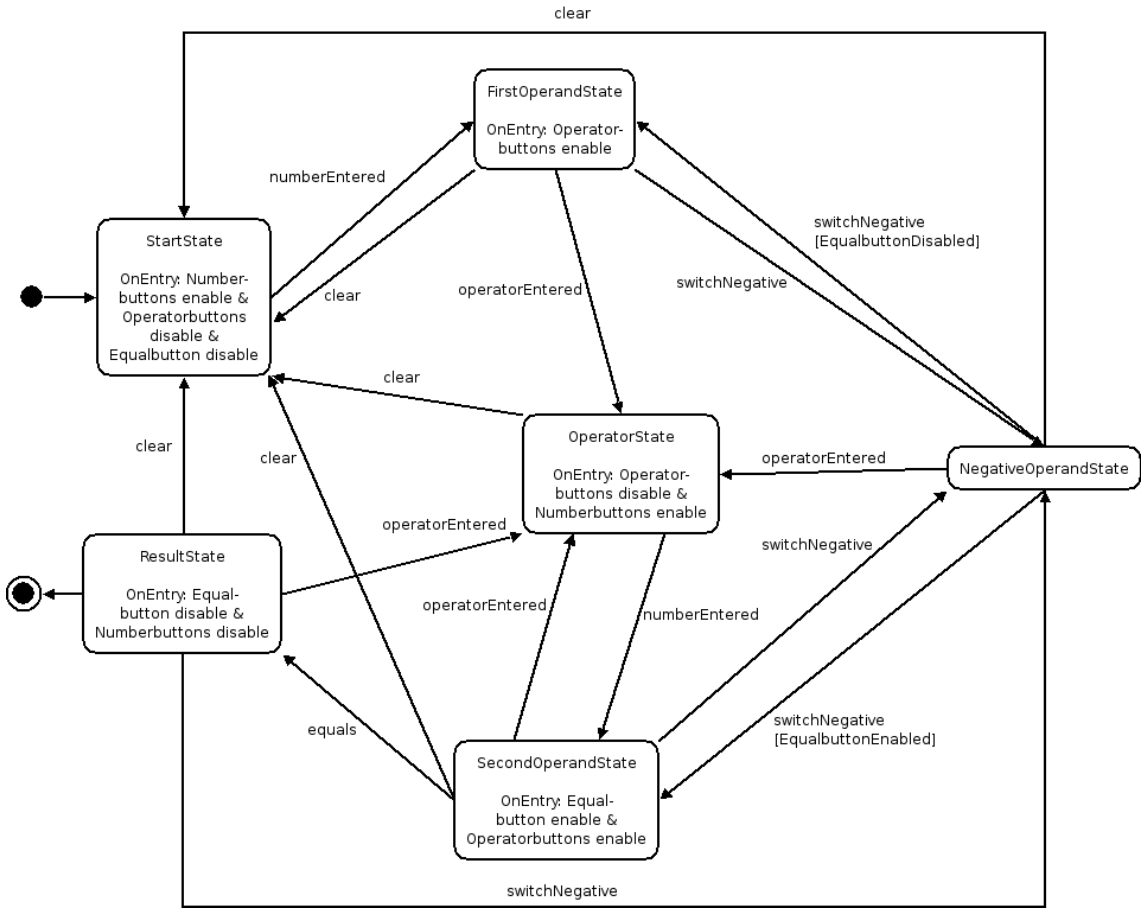

Figure 5.5: A statechart representing the state-based UI Behaviour for the calculator application

torState, in which the operator-buttons are disabled again (the aspect of impact five), as the user again only is allowed to enter a number at this point.
However, when the ”+/-”-button is pressed in stead of entering an operator, we reach the NegativeOperandState, which means that the entered number has now become negative. This can be compared to impact three, where we do not have an aspect that is affected within the UI (remember that only the application was affected), but we still have the possibility of an input-event. This is depicted by a state without an entry-action (since there is no affected aspect), but with the input-event of pushing the button translated into a transition. Whenever the ”+/-”-button is pressed when we are in the NegativeOperandState, we go back to the state that we came from, which is ensured by the guards on the transitions, which check from which state we entered the NegativeOperandState by checking the enabling of the equal-button.
We also have a SecondOperandState, which is reached when a number is entered after we pushed an operator-button (event of impact five). In this case, we have entered the first operand, the operator and the second operand of our calculation, so the user is allowed whether to continue the calculation by pressing an operator-button or press the ”equals”-button now, which is enabled (aspect of impact five).
When this last button is pressed (event of impact four), the ResultState is reached, and the result of the calculation is displayed in the inputfield. At this point, the user is only allowed to enter an operator to proceed with the calculation, and therefor everything is disabled except the operator-buttons (aspect of impact four). In any state, when the user presses the ”C”-button, the calculator is reset and we go back to the StartState (event of impact six).

When the developer decides that the wanted UI Behaviour is captured in the statechart (and possibly has checked that the wanted UI Behaviour is specified in it by comparing the statechart with the possible impacts within the UI), the developer can move on to the next step in the solution. In this step, the developer uses the created statechart to identify the specified UI Behaviour, so the application, apart from the UI Behaviour, can be implemented and methods in the implementation can be foreseen to couple the UI Behaviour to the application in a further stage.

5.2.5 Implementing the application

The second step in the solution consists of implementing the entire application, apart from the UI Behaviour. This should be done manually by the developer. The application should be implemented in Smalltalk [7], and this should happen in the same way as the developer would implement any other application, with one big difference. Instead of implementing the UI Behaviour - that was captured in the statechart - by writing

[7] Since the system that supports our solution is implemented in Smalltalk and uses Smalltalk-related tools, it can only work in a Smalltalk environment. Nevertheless, the conceptual idea behind the solution can be ported to other environments.

code for it, the developer now just *leaves out the UI Behaviour from the implementation*. However the developer is asked to follow a naming convention when creating methods in which, later on, the UI Behaviour should be added. Why this is the case and which naming convention is meant, is explained later in this section.

First we should explain an important part of the implementation of an application and its UI in Smalltalk, which will be used throughout this section.
As the application is implemented in Smalltalk, it will have a class that is a subclass of the so-called *Application Model-class*: this is the class on which the UI is installed. As Smalltalk uses the MVC-pattern, the visualization of the UI is installed on a certain class, which must be a subclass of the standard Smalltalk Application Model-class, and which then becomes the model for that UI. However, as this class is the model for the UI, it also receives all method calls that come from the UI. For example when a button is clicked in the UI and a method is called in response to that buttonclick, this method call shall pass through the model class. For our calculator, this means that we can for example have a class called CalculatorModel, which is a subclass of the Application Model-class and which receives for instance the message numberOnePressed whenever the button for the number one is selected.
Now that we know that the application will have a such a ”model class”, we can proceed by explaining why the developer should follow a naming convention and which naming convention this is.

As the application, apart from the UI Behaviour, is being implemented, there should still be a way for the developer to identify places in the application where the UI Behaviour should be inserted later on. Otherwise there are no *access points for the application to call the UI behaviour* when necessary, for instance when the UI changes its state. For example, when the equal-button is pushed in the calculator, the developer would like to specify that whenever that button is pushed, a corresponding method should be executed. This method would be the place where the UI Behaviour should be inserted later on by our system, in order for the appropriate UI Behaviour to be executed whenever the equal-button is pushed. However, when the developer does not know in which method the UI Behaviour will be inserted by the system, this is not possible.
Therefor, the developer must build the application in a way that all the code which holds the UI Behaviour is not written down explicitly in the code, but that there still can be *references by the application to the UI Behaviour, by providing certain ”hooks”*. These hooks are the points that the application uses to reference the UI Behaviour and to call the UI Behaviour, as these hooks will be the places to which the appropriate UI Behaviour is coupled later on. Practically, this means that the hooks are just methods, in which the appropriate UI behaviour can be inserted later on by our system.
However, we still need a place for these hooks in the application.
As the developer is creating the Smalltalk classes for the implementation of the application, the model class (as mentioned in the beginning of this section) for the UI of the

application is also created. It is exactly in this class that *the hooks should be placed*, as the UI Behaviour is always triggered by events in the UI, which all go through the model class.

Furthermore, we distinguish two kinds of hooks: *hooks that are manually made* by the developer and *hooks that are generated* by the system.
The hooks that are manually made by the developer are nothing more than methods (as mentioned before in this section) in the model class of the application. These methods are created by the developer *when an event in the UI* (e.g. pushing a button) *does not trigger UI Behaviour alone*, but also triggers other parts of the application [8].
For example, when in the calculator application a number-button is pressed, this does not only mean that the corresponding UI behaviour (in this case enabling the operator-buttons) should be done. The system variable that holds the current value of the number that is being entered in the calculator should also be updated. Therefor the developer will create a method in the model class, which is executed whenever a number-button is pressed, and which holds the functionality to update the mentioned system variable. As the developer is using our solution, the UI Behaviour is left out of the method and only the updating of the system variable is done.
However, we want that whenever the UI Behaviour for pushing a number-button is inserted, it is inserted exactly in that method that was created to be executed whenever a number-button was pushed. This means that we should find a way to let the system know which UI Behaviour corresponds with which method in the model class.
To achieve this, we can use a *naming convention* (as was mentioned in the beginning of this section). By this naming convention, we mean that the methodnames for the methods representing the hooks that are manually created may not be chosen freely by the developer, as specific names make it possible to match the UI Behaviour in the statechart with the existing hooks in the application. Therefor, the methodname for every hook must be the same as the name of the event of the transition in the statechart with which it matches.
As an example for our calculator, it is shown in figure 5.5 that there is a transition between the StartState and the FirstOperandState in our statechart which has the event ”numberEntered” and which ends in the FirstOperandState. The FirstOperandState has an Entry-action, being the enabling of the operator buttons. When the developer now wants to ensure that the UI behaviour for the FirstOperandState (which is the enabling of the operator buttons) is inserted in the appropriate method (in which the functionality of updating the system variable was already expressed), this method should have the same name as the event of the transition leading to the FirstOperandState. In this case, the transition leading to the FirstOperandState has an event which is called numberEntered. Therefor, the hook (and therefor the method in the model class) should also be named numberEntered.

[8] This can also be described as: when an event in the UI does not only has an impact within the UI, but also on the application. See chapter 3.

There are also hooks that are generated by the system. These hooks are also methods, but more specifically they are empty methods. These methods are created by the system *when there are transitions in the statechart that have no matching methods (hooks)*. This means the developer has not created a hook that corresponds to a certain transition in the statechart (using the naming convention), as that hook would only contain UI Behaviour, which the developer is leaving out of the implementation.
For example, when there would be a button in our calculator that switches the background-colour from blue to gray and the other way around, pushing this button would only trigger its corresponding UI Behaviour, and no other functionality would be needed.
As the developer is following our solution and leaving the UI Behaviour out while implementing the rest of the application, the developer should not create a method that is executed when the background-button is pushed, as this is purely UI Behaviour and therefor the body of the created method would be empty. Nevertheless, there should be a method, for the UI Behaviour to be inserted in by the system later on, and for the developer to use in its UI to reference to the UI Behaviour [9]. Therefor, the system generates an empty method in which the UI Behaviour can be inserted later on, and which can be used by the developer to reference the UI Behaviour from in the UI.
These methods are created by the system at the moment the aspects are defined (see section 5.2.9), and they follow the same naming convention as the methods that are created by the developer. In our example this would mean that the method in which the UI Behaviour for the switching of the background-colour should be inserted, should have the same name as the event of the transition in the statechart that leads to the execution of this UI Behaviour.

5.2.6 Creating a concept network: representing a statechart at the application level

As the statechart is drawn, the application (with its UI) is implemented and coupling points are foreseen in the application, this is *the third and last step that has to be done manually by the programmer*. This third step is to represent the statechart holding the UI Behaviour in a way so it can be given as input to the system.
Hereby an important note should be made. However this step is presented in this dissertation as a step that should be made manually by the developer, we would actually also *want this step to be automated*. In the future it should become possible to draw the statechart directly in CoBro [10], so the need for a manual transformation of the statechart is avoided. Then it should be possible as well to draw the statechart in CoBro

[9] However the developer is leaving out UI Behaviour from the implementation, the developer should still indicate in the UI what UI Behaviour should be executed whenever a certain event takes place in the UI. This is done only with methodnames, and not with actual code.

[10] An example of how an existing statechart looks in CoBro can be found in figure 5.9.

and give this as input to the system, as to just express the statechart at the application level, which can then be used as an alternative way of specifying the statechart.

However, this improvement to our system does not yet exist, and therefor *support for the manual approach* of this step is provided. We distinguish two kinds of support that is offered to support the manual construction of an application-level equivalent of a statechart: the *graphical support* and the *textual support*.

The graphical support consists of two tools: the concept maker (figure 5.6) and the statechart builder (figure 5.7).
The concept maker is a tool that can be used by the developer to create an application-level equivalent (which are CoBro concepts) for the different states and transitions in a statechart. This means that the developer should enter the names of the states or transitions in the statechart into the concept maker, which then automatically creates the appropriate application-level equivalent (i.e. states are transformed into state-concepts and transitions into transition-concepts, which are explained in section 5.2.7).
After the creation of the application-level equivalents of the different states and transitions, the developer can use the statechart builder to construct a statechart using these equivalents. The developer can choose which states should be connected to each other with which transition by selecting them in the tool and adding them to the list which holds the different connections (visible in the bottom right corner of figure 5.7). When the build-button is pushed, the selected transitions are used to connect the selected states, hereby creating an application-level equivalent of a statechart (i.e. a concept network, as described in section 5.2.7).

However, representing a statechart with this graphical support can get time-consuming and long-winded fast as bigger statecharts may need to be represented, which means every state and transition should first be made manually with the concept maker, to connect the states and transitions later on with the statechart builder. Taken that it is a big statechart, this could take a while.

Therefor, textual support is offered to provide a quicker and compacter way to represent statecharts than is offered by the graphical support. As textual support, a series of language statements for expressing statecharts is presented to the developer. These language statements are based on the way concepts are created in CoBro and can just be executed in Smalltalk (e.g. in the Smalltalk transcript). Hereby, the language statements are syntactic sugar for the CoBro CML language (as mentioned in section 3), and are specifically created and used to represent statecharts in a natural way. This way, the developer is not confronted with specifying the statecharts in CoBro CML, but is still provided with a way to naturally represent the statechart at the application level. As the textual support is less time-consuming than the graphical support, we use the textual support throughout the remainder of this section.
Applying the textual support to our calculator, we need to translate our statechart, depicted in figure 5.5, to a series of language statements, representing this statechart. As

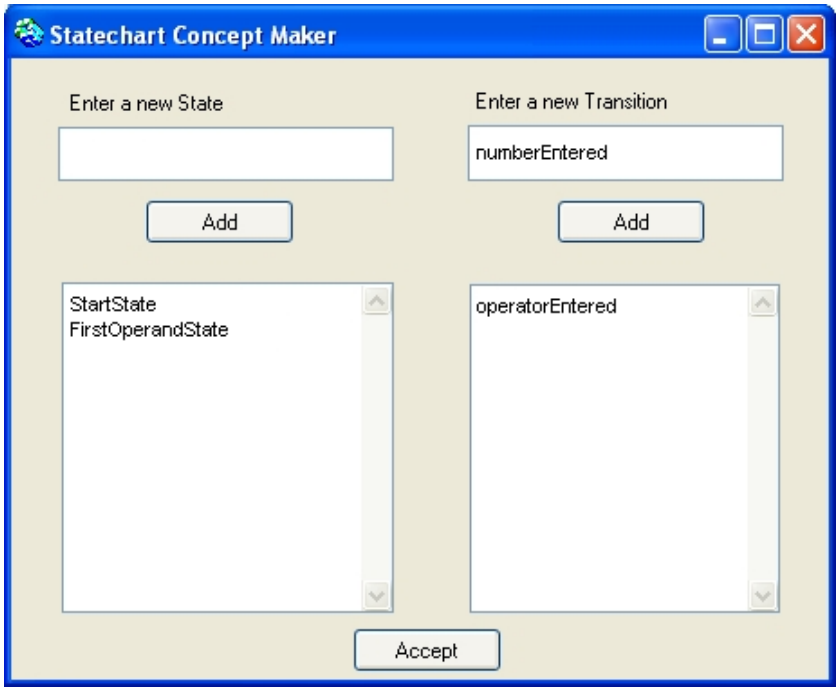

Figure 5.6: A part of the statechart representing UI Behaviour for the calculator

this is just an example, we will not represent the whole statechart, but only a part of it, with which we illustrate the most important aspects of the language statements. This part of the statechart is shown in figure 5.8, which is followed by the corresponding language statements for the states and transitions present. The code for all the states and transitions in the entire statechart of the basic calculator application is included in the appendix.

The states for figure 5.8 are represented in language statements as follows.

```
a := State new: #StartState.
b := State new: #FirstOperandState.
c := State new: #NegativeOperandState.

a beginState.

a hasOnEntry: 'Numberbuttons enable'.
a hasOnEntry: 'Operatorbuttons disable'.
a hasOnEntry: 'Equalbutton disable'.

b hasOnEntry: 'Operatorbuttons enable'.
```

The first language statements represent the available states in the selected part of the statechart for the calculator application. Furthermore, we indicate that the StartState

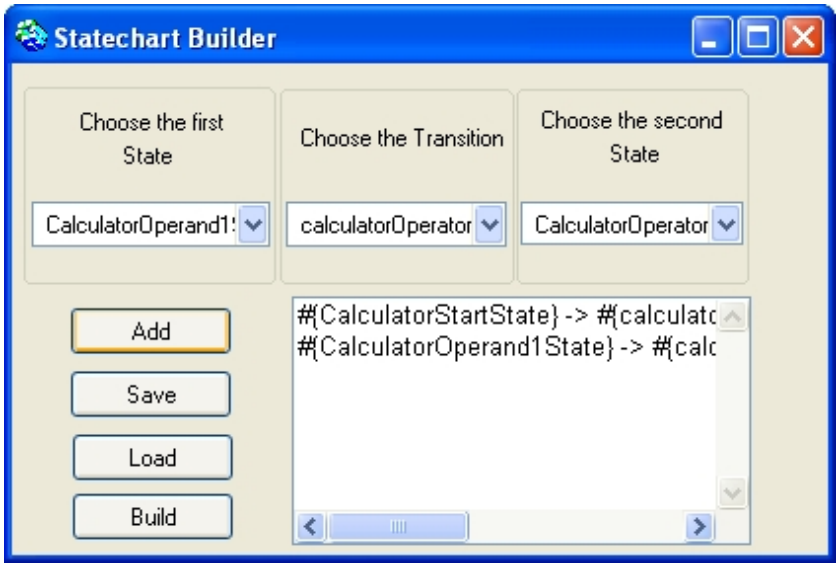

Figure 5.7: A part of the statechart representing UI Behaviour for the calculator

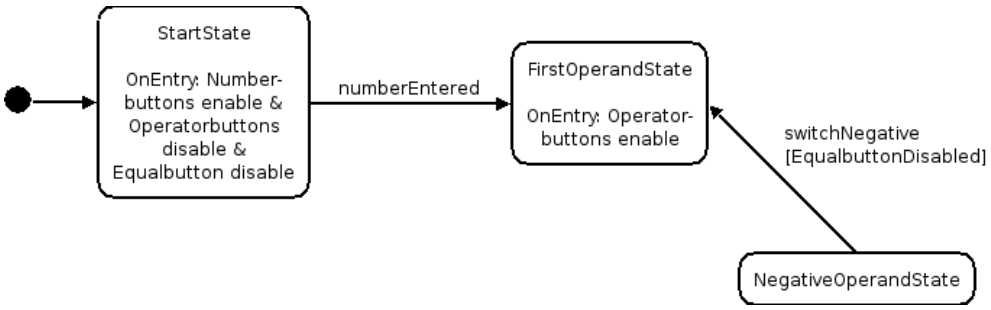

Figure 5.8: A part of the statechart representing UI Behaviour for the calculator

is marked as the beginstate in the statechart. We also specify the entry-actions that are shown in the statechart for the appropriate states. We now continue with representing the available transitions for figure 5.8 in language statements.

```
t1 := Transition new: #numberEntered.
t1 hasEvent: 'numberEntered'.
a -> t1 -> b.

t2 := Transition new: #switchNegDis.
t2 hasEvent: 'switchNegative'.
t2 hasGuard: 'EqualbuttonDisabled'.
c -> t2 -> b.
```

Here it is shown how transitions are represented and how those transitions can be

used in combination with states. In this example, we see that we create the transition t1, with the event ”numberEntered”, and we use it to connect the StartState and the FirstOperandState (as specified earlier) by putting the transition in between them. For transition t2, we notice that the guard-condition on the transition is expressed.

Now that the statechart is represented at the application level, the system supporting the solution takes control. However it is hidden from the developer, every language statement triggers one or more statements in the CoBro CML Language. As these statements are executed, a concept network is created, which is explained in the following section.

5.2.7 Creating a concept network: from representation to concept network

As the statechart is represented at the application level by the developer, the system, supporting our solution, can take this as input. The system will then take control and execute this input to form a concept network (as described in section 4.2.2), by the use of CoBro. Each language statement in the input is executed, hereby triggering the corresponding statements in CoBro CML, as the language statements are just syntactic sugar for these CoBro CML statements.
For every state and transition specified in the language statements, with their corresponding properties, *a concept is created*, and these concepts are linked together, forming a *concept network* with states as concepts and transitions as relationships. As the state-concepts are linked to other state-concepts through the use of transition-concepts, we can recognize the form of the original statechart in the concept network. This is illustrated in figure 5.9, in which a graphical representation of a part of the concept network for the calculator example is shown in CoBro-Nav (as mentioned in section 4.2.2). In this concept network, we can recognize three states of the calculator statechart (as shown in figure 5.5) and two transitions between these states.
This concept network thus represents the statechart, as it is constructed out of a representation of the statechart, and therefor the concept network also represents the state-based UI Behaviour that was represented in the statechart all along.

When applied to the calculator application, the language statements as specified in section 5.2.6 are now used as input to create a concept network representing the statechart for the basic calculator application (figure 5.5). This concept network is constructed as the language statements are translated into their corresponding CoBro CML statements and executed, and therefor CoBro concepts are created, which form a concept network.
How this transformation from language statements to CoBro CML statements is done is illustrated by the following concepts for the basic calculator. These concepts form

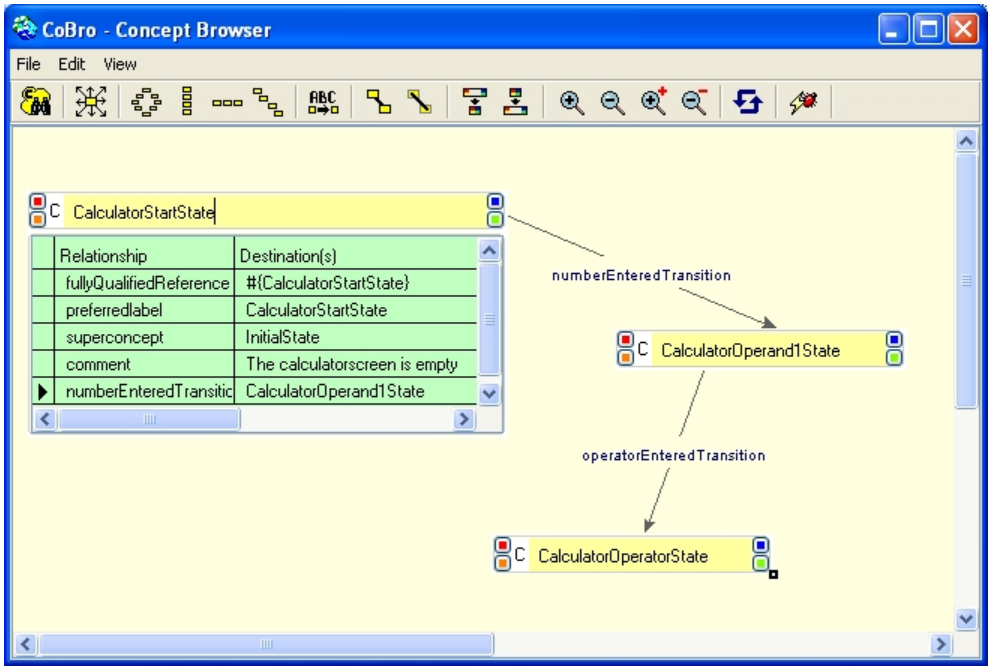

Figure 5.9: The concept network for a part of the calculator statechart, presented graphically in CoBro-Nav

the concept network representing the partial statechart depicted in figure 5.8. As these concepts represent the same states and transitions as specified in the language statements in section 5.2.6, they can be compared to one another. Again, not all concepts are given for the complete statechart in figure 5.5, but the full list of all concepts can be found in the appendix.

```
(a := Concepts new: #{StartState})
    hasPreferredLabel: 'StartState'.
    a superconcept: Concepts.BeginState.
    a hasOnEntry: 'Numberbuttons enable'.
    a hasOnEntry: 'Operatorbuttons disable'.
    a hasOnEntry: 'Equalbutton disable'.
    a save.

(b := Concepts new: #{FirstOperandState})
    hasPreferredLabel: 'FirstOperandState'.
    b superconcept: Concepts.State.
    b hasOnEntry: 'Operatorbuttons enable'.
    b save.

(c := Concepts new: #{NegativeOperandState})
```

```
    hasPreferredLabel: 'NegativeOperandState'.
    c superconcept: Concepts.State.
    c save.
```

Here we can see the concepts representing the three states in the partial statechart depicted in figure 5.8. We recognize these states from the language statements, which we specified in section 5.2.6.
In the code for these concepts we can see that a state-concept can have an extra slot (the basic and extra slots are mentioned in section 4.2.2) to indicate whether the state has an entry-action [11].
Furthermore, it should be noted that the StartState specification in CoBro CML, which still has the name StartState, and which still has the same entry-actions as we defined in the language statements, highly resembles the StartState specification in the language statements. Like in the language statements, it is also indicated that the StartState is the beginstate. The entry-actions for the states are also specified in a very similar way as in the language statements.

As the state-concepts are created, the transitions for the statechart for the calculator application that can link these state-consepts can now be created. They form the relationships within the concept network.

```
(t1 := Concepts new: #{numberEntered1})
    hasPreferredLabel: 'numberEntered'.
    t1 superconcept: Concepts.hasTransition.
    t1 hasDestination: Concepts.StartState.
    t1 hasSource: Concepts.FirstOperandState.
    t1 hasTransitionEvent: 'numberEntered'.
    t1 save.

(t2 := Concepts new: #{switchNegDis})
    hasPreferredLabel: 'switchNegDis'.
    t2 superconcept: Concepts.hasTransition.
    t2 hasDestination: Concepts.NegativeOperandState.
    t2 hasSource: Concepts.FirstOperandState.
    t2 hasTransitionEvent: 'switchNegative'.
    t2 hasTransitionGuard: 'EqualbuttonDisabled'.
    t2 save.
```

Here we see the creation of concepts representing the two different transitions present in the statechart depicted in figure 5.8. These transitions form the relationships within the concept network, as they connect the state concepts to one another.
It is shown that a transition-concept can have four extra slots, which are necessary to

[11] This can also be an exit-action, in that case the relation hasOnExit should be used.

define the beginstate (source) and the endstate (destination) of the transition, as well as specifying an event or a guard on the transition.
Furthermore, we see that in the concept network for the calculator application the Start-State is connected to the FirstOperandState with a transition t1 with an event "number-Entered". We notice that every transition has a source-state and a destination-state, as well as a transition event. In the case of the last transition, also a guard is represented in the concepts.
It should also be noted that, when there are transitions with the same name and event, these transitions are given a fullyQualifiedReference consisting of their name and a number, in order for them to have a unique fullyQualifiedReference and therefor to be uniquely selectable in the concept network.

At this point, a concept network has been created, representing our statechart and consequently the UI Behaviour that is represented within that statechart. We should mention again that this concept network was created *automatically* by the system (as explained in section 5.2.7), using the input (the language statements) the developer *manually* provided (as said in section 5.2.6).
This concept network can now be used further within the system to extract the needed UI Behaviour out of the statechart. This UI Behaviour will be needed and used by the supporting system for building a statemachine and generating aspects, which are necessary to complete our solution. The building of a statemachine is described in the next section, and the generating of the aspects is explained in section 5.2.9.

5.2.8 Building a statemachine

As the concept network is established (as described in section 5.2.7), the developer has created an application-level equivalent [12] of the UI Behaviour that is adaptable later on (as mentioned in section 5.1.2). Furthermore, this concept network can provide the developer with an abstract view of the UI Behaviour (by resembling the original statechart), hereby enhancing adaptability and CoBro provides the developer with a way to specify an application-level equivalent of the UI Behaviour without having to manually implement a statemachine [13].
However, such a statemachine would come in handy, as we want to be *able to execute the UI Behaviour* (that was first captured in the statechart and is now captured in the concept network) without having to access the concept network every time the UI Behaviour is needed.

Using a statemachine would make the creation of the aspects (described in section 5.2.9), which are used to couple the UI Behaviour to the application, less complicated, as executing the UI Behaviour in a statemachine is more automated than in the concept

[12] For the moment still through the use of simple language statements, but we mentioned before that this should be automated in the future.

[13] We assume that specifying a statechart in language statements (as described in section 5.2.6) is an improvement over manually implementing a statemachine for that statechart.

network. By this we mean that we only need to send one message, triggering a transition, to a statemachine in order for the corresponding UI Behaviour to be executed and the statemachine to be brought in the right following state. This opposed to a concept network, in which the UI Behaviour has to be extracted from the concepts and then executed, and also the next state needs to be looked for in the concept network. Therefor, to facilitate the creation of the aspects (we find sending one message to a statemachine to execute the correct UI Behaviour easier than manipulating and extracting a concept network to do this), we introduced a statemachine.
Therefor we want to *automatically generate a statemachine out of this concept network*, creating a kind of "operational statechart" in Smalltalk which can be used by our application to execute UI Behaviour.
It should be noted that this transformation from a concept network to a statemachine is not strictly necessary, as we could also use the concept network in CoBro to execute the appropriate pieces of code, but as mentioned, this would influence the creation of our aspects. However, it should also be mentioned that by introducing the statemachine, we also partially lost the possibility of making runtime adaptations, as is discussed in section 5.3.2.
Nevertheless, as the concept network represents the statechart holding the UI Behaviour for the application, generating a statemachine out of this concept network would enable us to execute this UI Behaviour, depending on what state we are in. This is exactly what we want, as we want to execute the appropriate UI Behaviour, depending on what state the UI is in. How the statemachine is warned when the UI changes his state, is explained in section 5.2.9.

The generation of a statemachine from the concept network can be achieved in three steps. First, a *statemachine should be created* in Smalltalk. Secondly, states, transitions and UI Behaviour should be added to this statemachine, by *extracting this data out of the concept network*. This will give the statemachine its functionality. Finally, in order for the application to be able to use the statemachine, it should be *initialized and become operational*. These steps are illustrated in figure 5.10 and are explained in the remainder of this section. It should be noted that all these steps for generating a statemachine are done automatically by our system.

To create a statemachine (as illustrated at point number one in figure 5.10), a class was created in Smalltalk, which allows the supporting system to make instances of statemachines. This means that every time a new concept network is handed to the system, a new instance of the statemachine class is taken. This instance is always empty (i.e. no states and transitions have been entered yet, and the statemachine does not have any functionality) and can then be used to insert the data from the concept network (this will be discussed further in this section).
A (instance of a) statemachine is constructed in such a way that a number of possible states and a number of possible transitions can be added to it.

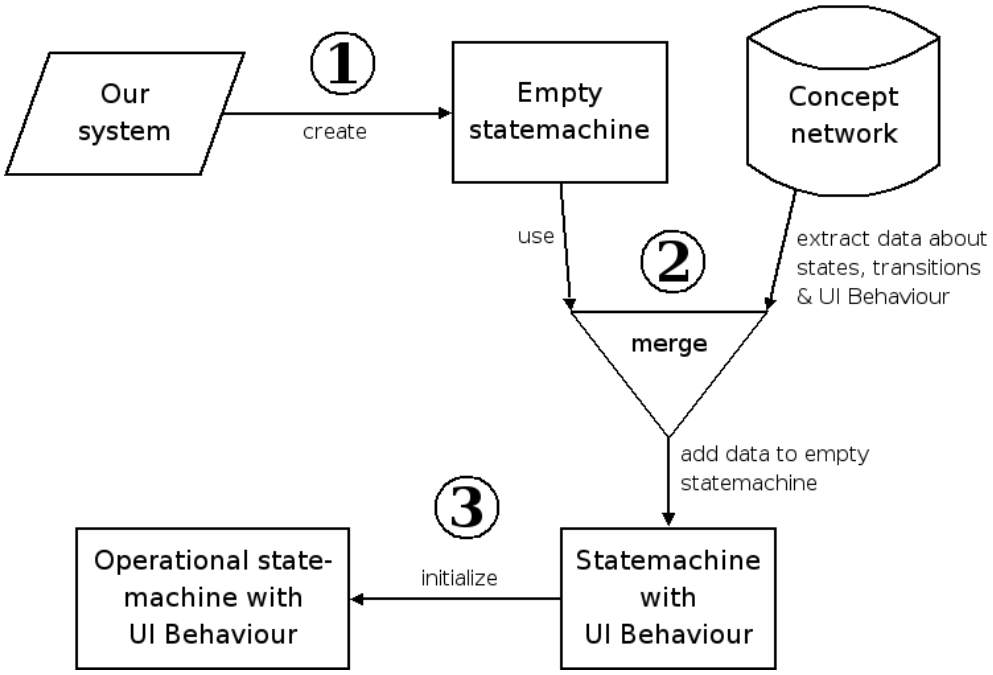

Figure 5.10: How the system generates a statemachine

The states of a statemachine are represented by strings. For every state in the statemachine, a string with the same name as the state is available in the statemachine.
The transitions of a statemachine are represented by Smalltalk-methods, and hereby, a transition can be performed by simply calling the corresponding method. For every transition, a method, which has the same name as the event of the transition it represents, is created within the statemachine.
A statemachine also holds its current state at all times in a variable and will only change this state when a valid transition is done from the current state to another state. In the case of the calculator application, for example a statemachine can be created with the name CalculatorStatemachine, which holds all the above described features.

Now that we have an empty statemachine in Smalltalk, we can extract the states, transitions and UI Behaviour from the concept network and insert them into the statemachine. This is illustrated in point number two in figure 5.10.
As we have seen in section 4.1.2, the prime elements that make something into a statemachine are an *initial state*, a set of *possible input events*, a set of *states* that result from the input and a set of *possible actions* that result from a state. It is exactly this set of prime elements that is added to the statemachine by using the data from the concept network.

As the concept network already holds states and transitions as concepts and relations,

the structure (of states and transitions) of the statemachine is already present in the concept network, so we can use this structure directly, without any transformations. This means that states that are connected by a certain transition in the concept network, will still be connected by that transition in the statemachine.
Therefor, when making a statemachine, we can first take all state-concepts out of the concept network and enter the names of these concepts as states in the statemachine (remember that states are represented by strings), as the states in the concept network and the states in the statemachine should correspond. Here we see that a first prime element was added to the statemachine, being *a set of states*.
Among the state-concepts in the concept network, there should also be a state-concept representing the initial state of the statechart. As we are adding states to the statemachine, this state should also be marked as the initial state of the statemachine. This brings us to a second prime element that has been added: an *initial state*.
Thereafter, the transition-concepts can also be taken from in the concept network and installed on the statemachine as methods. It should be noted that for every state and every transition, all properties represented in the concept network are also taken into account. By this we mean that the event for a transition is used to name the method which represents the transition, and that entry- and exit-actions for a state are placed as bodies in the methods that represent the transitions that enter or leave that state. This is clarified with an example in the following paragraph. We should first mention that this last part goes for all properties represented in the concept network, except for guard conditions on transitions. The guard conditions are a special case and they are handled separately in section 5.2.9. Furthermore we can also notice that the two final prime elements of a statemachine were added, therefor completing our statemachine. The *set of possible input events* is described by all the events of the transitions in the statechart (and therefor also in the concept network), which have just been added to the statemachine alongside with the transitions. Also, *the set of possible actions* is described in the statechart as entry- and exit-actions for the appropriate states, which have just been added as properties of the states that are included in the concept network. This is now clarified with an example.

As an example we use our basic calculator and we consider the concept network that we have constructed in section 5.2.7, and which is based on the partial statechart, as depicted in figure 5.8. In this concept network, we have three state-concepts, being the StartState, the FirstOperandState and the NegativeOperandState (set of states). We also have two transition-relationships, being the numberEntered-transition and the switchNegative-transition (set of input events).
Following the suggested solution, we take the names of the state-concepts and add these as states to the CalculatorStatemachine (the empty statemachine that was mentioned earlier in this section), while marking the StartState as the beginstate (initial state). Then we take the transition-concepts and add these as transitions to the statemachine, which creates methods in the statemachine with the names numberEntered and switchNegative.

Finally, we see that for instance the FirstOperandState has an entry-action, being ”Operatorbuttons enable” (set of actions). As this is an entry-action, we want it to be executed whenever we reach the FirstOperandState. Therefor, we will add this action to the body of the methods representing all transitions that have the FirstOperandState as destination, so the action is executed whenever a transition that ends in the FirstOperandState is called.

As we have created a statemachine and added states, transitions and UI Behaviour to it, we now still have to initialize it (as represented at point number three in figure 5.10). This is done by the system by calling the method ”initialize” on the statemachine, which sets the current state of the statemachine to the beginstate of the statechart that is represented by the statemachine.
For example when the CalculatorStatemachine is initialized, the current state will be set to the StartState, as this is the beginstate in the statechart (and in the concept network) for the calculator.

As a final example of how the statemachine works, a simple graphical representation is presented to further clarify a statemachine. In figure 5.11, we see the CalculatorStatemachine, which is now the statemachine that represents the whole statechart for the calculator application, as depicted in figure 5.5. We see that the statemachine contains all the states and transitions, and the current state is the StartState. Then, it is shown that the numberEntered-message is sent to the statemachine. As the statemachine notices that this is a valid transition, the transition shall be executed and the statemachine is updated. The current state has been updated to the FirstOperandState, and since this state has an entry-action, this entry-action is executed, as seen in the executing-box.

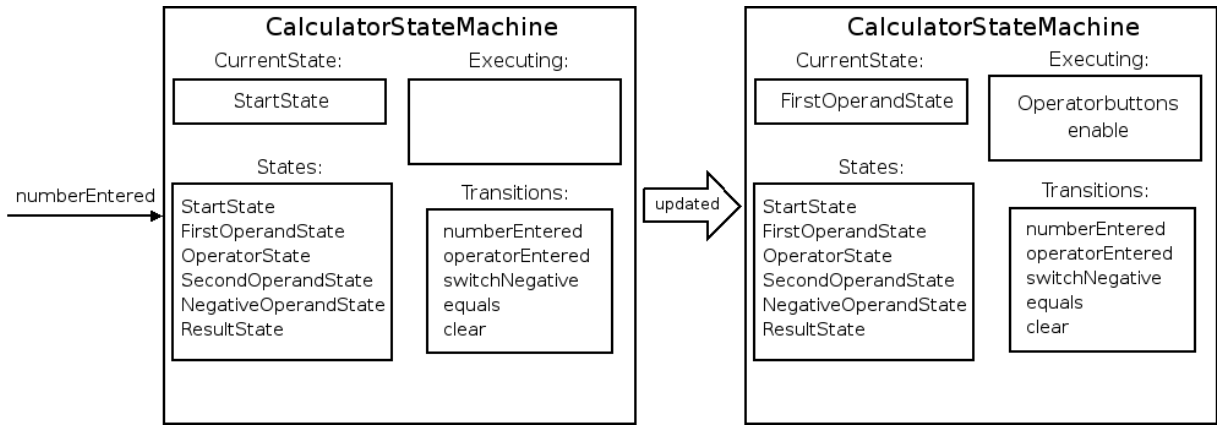

Figure 5.11: A representation of a statemachine for the basic calculator

After the system initializes the statemachine, we have an operational statemachine holding the UI Behaviour as it was specified in the original statechart. It should be noted that, as the statemachine contains the UI Behaviour, we are now executing UI Behaviour depending on what state the statemachine is in. Furthermore, as the statemachine is just a representation of the statechart which holds the UI Behaviour coupled to the states of the UI, we can say we are executing UI Behaviour depending on what state the UI is in.
However, the statemachine on its own does not know when the UI changes its state and which transition should be executed in response to this change. Therefor, aspects are introduced, which are explained in the following section.

5.2.9 Generating aspects

As a statemachine that can execute the UI Behaviour is provided (as described in the previous section), we should still provide a way to couple this statemachine to the application in order for the statemachine to be warned when the UI changes its state and the UI Behaviour is thus called upon by the application. For this purpose aspects can be generated automatically by the system that offer a coupling between the application and the statemachine.

First, an **important note** should be made for this section and the entire remainder of this dissertation. However aspects and Carma are used, we are *not using an aspect-oriented approach*. In fact we are only partially using the aspects and Carma, and this only for coupling the UI Behaviour to the application (this will become clear in the remainder of this section). This can be explained by our future wish to use an aspect-oriented approach, in which Carma can be used to the fullest. Furthermore, Carma should also be useful in the context of Deuce (section 4.2.4), as it can possibly be used to disentangle other concerns than UI Behaviour, and also has the benefit of using logic rules. Therefor, Carma was already introduced into our system, however only a small part of it is used. This is discussed further in this section and in section 5.3.2. With this note made, we can now have a look at how the UI Behaviour is coupled to the application.

This coupling is done automatically by the system by generating aspects, which is done in two steps. Before creating the aspects, the supporting system first should *gather some data* from the concept network and from the application to be able to specify the aspects correctly. Afterwards, when the necessary data is obtained, the system can *create the aspects* to couple the application to the statemachine, and consequently to the UI Behaviour.

To **gather the data** necessary for specifying the aspects, the system first takes a look at the *concept network*, that is provided in section 5.2.7, and it identifies all possible

transitions in this concept network. As all transitions are known, the names of the events of all these transitions are stored. Furthermore, the system checks which of the transitions have guard conditions, which is of importance when creating the aspects. This is explained at the end of this section.

As data was gathered from the concept network, the supporting system now gathers data from the *application*. The model class (as explained in section 5.2.5) of the application is inspected, and the system stores all the methodnames of the methods in this class. Among these methods in the model class, the hooks that were created manually by the developer (as explained in section 5.2.5) will be present. As explained, these hooks will be needed to couple the UI Behaviour to the application.
However, as also mentioned and explained in section 5.2.5, there also need to be hooks generated by the system, which can be done using the methodnames and eventnames. At this point, the list of methodnames of the methods in the model class is compared to the list of names of the events of the transitions in the concept network. Hereby, for each event that does not have a corresponding method (i.e. there is no method with the same name as the event), an empty method is created in the model class, with as methodname the name of the event. This means that all the necessary hooks are now available and can be used.
However, as the hooks that were generated by the system were added to the model class, the list of methodnames of the methods in this class needs to be updated. As this is done, we now have a list of all methodnames in the model class.
It should be noted that all methods in the model class are selected, and not only the hooks. This is done because all the methods in the model class possibly express the changing of the state of the UI. Whenever then UI Behaviour is added to the statemachine for these changes in the state of the UI, we already have provided a coupling between the application and the statemachine for the methods that express these changes, as they have become hooks.

This can be also be applied to the basic calculator. The system will then take a look at the concept network for the calculator (as presented in section 5.2.7), identifying the transitions, such as the numberEntered-transition, the clear-transition, etc. The system will then also notice that the switchNegative-transitions between the NegativeOperandState and the FirstOperandState on the one hand, and between the NegativeOperandState and the SecondOperandState on the other hand have guard conditions on them.
Furthermore, the model class in the calculator application is inspected and all the methodnames of the methods in it are listed. This list is then compared to the list of eventnames, which holds names such as numberEntered, clear, etc. Depending on which methods were already manually created by the developer in the model class, additional empty methods are then generated by system. In this case, all the events will already have a corresponding method, as these methods should have been already implemented by the developer [14].

[14] All the events also have an impact on the application, and should therefor already be implemented

Hereafter, the list of methodnames is updated, now holding all the names of the methods in the model class of the calculator application.

Using the data that was gathered from the concept network and the application, the system can **create aspects**. For each methodname in the model class, a different aspect is created, which is linked to its methodname. To create such an aspect, the system should specify two parts: a *pointcut* and an *advice* (as explained in section 4.1.1 and mentioned in section 4.2.3). These parts are first explained and then illustrated with an example.

The *pointcut* that is generated by the system for each aspect (which is linked to a certain methodname in the model class) has the same conceptual content for each aspect, being "execute the advice for every reception of a message". Hereby, the advice (which is discussed in the following paragraph) as well as the message are always filled in differently, depending to which methodname the aspect is linked. For example in the calculator application, the system will obtain for instance the methodnames "numberEntered" and "clear" from the model class. For each of these names, a corresponding aspect is created, with as pointcut "execute the advice of this aspect for every reception of the message numberEntered" in the first case, or "execute the advice of this aspect for every reception of the message clear" in the second case. This ensures that whenever a method (in this case: numberEntered or clear) is called in the model class of the application, the advice of the corresponding aspect is executed.
These pointcuts for the calculator can be expressed in Carma (as explained in section 4.2.3) as:

```
before ?jp matching {reception(?jp, #numberEntered)}
before ?jp matching {reception(?jp, #clear)}
```

As a different aspect is created for every methodname in the model class, the *advice* is also coupled to these methodnames. Also the advices for the different aspects have the same conceptual content for each aspect, being "send the same message as was specified in the pointcut to the statemachine". Also in this case the message that is mentioned is filled in differently per aspect, as it depends on the pointcut, and therefor indirectly on a methodname. As we again use the calculator example, and we use the same aspects as mentioned above, corresponding to the methodnames numberEntered and clear, we see that the advice will be "send the message numberEntered to the statemachine" in the first case, and "send the message clear to the statemachine" in the second case.
These advices for the calculator can be expressed in Carma (as Smalltalk-code) as:

```
TestCalculator statemachineInstance numberEntered.
TestCalculator statemachineInstance clear.
```

manually, as explained in section 5.2.5.

As the pointcuts and advices are specified, they can be combined into aspects (as many as there are methodnames in the model class of the application) that guarantee that whenever a method is called in the model class, the method with the same name is called in the statemachine. As the hooks (as well the manually created as the automatically generated ones) are all methods in the model class and all the names of the hooks correspond with the name of a transition-event, the calling of the hooks will trigger the corresponding transition in the statemachine. This ensures that whenever an event takes place in the UI that triggers a transition (and therefor changes the state of the UI), that transition is brought to effect in the statemachine.
So for example for the calculator application, this would mean that whenever an event takes place in the UI of the calculator that triggers the method "clear" (which is a hook), then the method clear is automatically also called on the statemachine, which will perform the corresponding transition if it is valid. The performing of the transition in the statemachine naturally brings along the execution of the appropriate UI Behaviour, as explained in the previous section.

An example of an aspect in Carma for the calculator application (in which we can recognize one of earlier specified pointcuts and advices) is the following:

```
Andrew.TestAspects defineAspect: #StatechartAspect
    superAspect: #{Andrew.Aspect}
    ofEach: #TestCalculator
    instanceVariableNames: ''
    aspectVariableNames: ''
    category: 'Calculator'.

before ?jp matching {reception(?jp, #numberEntered)} do
    TestCalculator statemachineInstance numberEntered.
```

We see that in this aspect we just send the message "numberEntered" through to the statemachine (which is contained in TestCalculator statemachineInstance) when it is received. This will make the CalculatorStatemachine do the transition "numberEntered" (if it is a valid transition at the time) and execute the corresponding UI Behaviour, for instance the enabling of the operator-buttons.

As another **important note** (following the one that we made in the beginning of this section), we should mention that, as the pointcuts of the aspects are specified in such a way that the advice will always be woven at the start of a method (when it is called), we never weave the UI Behaviour in between the existing application code. This means that we are only using the aspects to insert the UI Behaviour in a method, and not weave it in with the existing code, hereby not using an aspect-oriented approach. Therefor, we could also just use the Smalltalk "method-builder" for this task, with which we can insert parts of code (which would then be the advices of the current aspects) in particular methods (which would be the hooks, as they are used in the pointcuts).

However, nevertheless that Carma could be replaced by the method-builder for this task, we can still justify the use of Carma by the use of logic programming and our look at the future.
As we are already using Carma in our solution, even if we are hereby not using it to its fullest, we have already created a *foundation for using the full power of Carma in our solution in the future*. As the aspects can already be generated by the system and the hooks to couple the UI Behaviour to the application are provided, a foundation is given for evolving towards an aspect-oriented solution. The most important change that should be made in the future to achieve this, is the fine-tuning of the pointcuts that are used in our solution, so that they can be used to weave the UI Behaviour in between pieces of application code. Nevertheless, also further research should be done, investigating the consequences of such an aspect-oriented approach on the solution and exploring other changes to the system.
Furthermore, we use logic programming in our solution to specify guard conditions, which is not possible without the use of Carma. How this is done, is explained in the remainder of this section, and it is validated in section 6.5.
Now that this note is made, we can continue with explaining why transitions with a guard condition must be treated differently as simple transitions, as was mentioned in the beginning of this section, when the data was being gathered.

Transitions with a guard condition are treated separately, as the advice for the corresponding aspect is slightly adapted compared to the aspects for simple transitions.
When an aspect is created, it is always checked if the methodname for which the aspect is created corresponds to the name of a transition-event of a transition that has a guard on it. When the methodname corresponds to the event of such a transition, we know that we are dealing with a hook that represents a transition with a guard, and the advice should be adapted. The advice is adapted to this form: ”if the guard-condition is met, then send the same message as was specified in the pointcut to the statemachine”.
This guard-condition is extracted from the concept network. As we are putting the guard-conditions in the aspects, we can check the guard conditions on the transitions even before entering the statemachine. Furthermore, for this reason we are also able to use logic rules in our guard conditions through the use of Carma (see section 4.2.3).
Carma can be used for example to express guards on transitions for which logic reasoning can be used. This can be for instance when a parameter in a guard-condition can be different in different situations, which means it could be better off stored in a logic repository.

For example: when our calculator is used by a variety of people (going from small children to adults), we might want to add a feature that adapts the calculator according to the age of the user. This could mean that when, for instance, a child under the age of twelve is using the calculator, only the most basic operations are allowed and shown,

as this child is not taught things as squares and square roots yet. However, when the calculator is used by an adult, all the operations, including the most advanced, are shown. Adding this feature would bring along introducing parameters that hold the age-limits for allowing certain operations or not. However, as the educational system can be different depending on in which country we are, so can be the ages at which certain mathematical operations are taught. Therefor, as these parameters can change depending on in which country we are, it can be useful to let these parameters be handled by the logic repository, in which we have entered a logic rule depending on in which country we reside.

This is illustrated in the following (fictitious) aspect, in which it is checked if the user of the calculator is an adult, so all the operations can be shown.

```
before ?jp matching {reception(?jp, #adjustToAge)} do
((SOULEvaluator eval: 'if adult(TestCalculator user)') allResults)
    ifTrue:[TestCalculator statechartInstance adjustToAdult].
```

As an example of an aspect that contains a guard condition which can be found in our calculator statechart in figure 5.5, we can use the "switchNegative"-transition in the calculator. Hereby, the advice is adapted to the presence of the guard "EqualButtonDisabled".

```
before ?jp matching {reception(?jp, #switchNegative)} do
((SOULEvaluator eval: 'if EqualbuttonDisabled') allResults)
    ifTrue:[TestCalculator statechartInstance switchNegDis].
```

Here we see that the condition that is specified by the guard is first evaluated by the evaluator of SOUL, the logic language used by Carma, which evaluates logic rules. SOUL also evaluates Smalltalk, so if the guards are just Smalltalk code, this is not a problem. When the guard-condition was evaluated and the condition is met, which means that the equal-button is disabled, then the right transition (namely the switchNegDis-transition) is send to the statemachine. The guard condition is stored in the logic repository of SOUL and looks like this:

```
EqualbuttonEnabled
    if (TestCalculator builder componentAt: #equalbutton) isEnabled
```

Here we see that the guard conditions just checks if a certain widget, the equalbutton in this case is enabled or not.
In this case (for the equal-button) we actually did not need to specify the guard condition in SOUL (as this is just a simple situation), but we did this nevertheless to illustrate how this looks like. We could also specify the guard condition as:

```
before ?jp matching {reception(?jp, #switchNegative)} do
((SOULEvaluator eval: 'if (TestCalculator builder componentAt:
#equalbutton) isEnabled') allResults)
    ifTrue:[TestCalculator statechartInstance switchNegDis].
```

5.2.10 Weaving aspects

As a last step in our solution, the supporting system will weave the aspects created in the previous section onto the application, more specifically onto the model class. From the moment the aspects are woven, whenever the model class receives a message, this message is also automatically send to the statemachine [15].
The aspects are woven through the use of the Carma weaver (as explained in sections 4.1.1 and 4.2.3), which will identify all the places in the Smalltalk code for the application that match the pointcuts that are specified in the aspects. At these places, which are methods, the corresponding advices are added to the code already present in the method. Once the weaving is finished, the application can run like the UI Behaviour was never separated from the rest of the code. The right UI Behaviour shall be executed at the appropriate time, and the end user will never notice that the UI Behaviour was separated during the implementation of the application.

For the calculator application, this means that the specified aspects for the basic calculator are woven onto the model class of the calculator application. This means that the advice of the aspects is added to the code of the methods in the model class. For example in the method ”numberEntered”, there will now the piece of code ”TestCalculator statemachineInstance numberEntered.” at the beginning of the code in the method, as this piece of code was specified as the advice for the aspect corresponding with the numberEntered-method. This ensures, that whenever the numberEntered-method is called, the method with the same name is called in the CalculatorStatemachine.

As we have completed the description and explanation of our solution, we can now have a closer look at some benefits and shortcomings that this solution has. This is described in the following section.

5.3 Benefits and shortcomings

In this section the most important advantages and shortcomings of the solution, as it was presented earlier in this chapter, are described.

First, the benefits are presented, starting with the most important one, i.e. the disentanglement of the UI Behaviour, which leads to easier implementation of UI Behaviour and application (as mentioned in chapter 2). This can be seen as the biggest benefit this solution has to offer, as some of the other benefits are more or less inclined by the disentanglement.
Furthermore, the solution is constructed as such that it aims at reducing overhead for

[15] The notes that were made in the previous section should also be kept in mind during this section and the entire remainder of this dissertation.

the developer who is using it, for example by not obliging developers to learn complicated new design methods.
Also, by using the solution, the developer introduces adaptability and extensibility into the application. *As this chapter only holds a list of benefits, the mentioned benefits are all illustrated and validated in chapter 6.*

To conclude this chapter, the shortcomings of the solution are described. First, we discuss the shortcomings that are brought along by not using a pure aspect-oriented approach for our supporting system.
Furthermore, we show that we still need the developer to manually transform the statechart to an application-level equivalent, which should be avoided.
Thereafter, it is said that runtime changes to the supporting system are not supported, as for some changes it is necessary that the system is recompiled.

5.3.1 Benefits

Disentanglement of the UI Behaviour

The first and most important benefit is the disentanglement of the UI Behaviour from the rest of the system. The problems associated with entanglement were already mentioned in chapter 2. Thanks to our solution however, it is now possible to separate the UI Behaviour from the other concerns [16], where earlier on UI Behaviour code, application logic and other UI concerns code were written as an entangled whole by the developer. This facilitates the implementation of the application and of the UI Behaviour for the developer (as was mentioned in chapter 2).
Furthermore, the disentanglement facilitates the making of changes to either the UI Behaviour or the application. For example when the need arises to change a part of the UI Behaviour, it is no longer necessary to scan through the application in search of the appropriate pieces of code to adapt [16]. Because of the disentanglement, the developer knows where the UI Behaviour is situated (as it is separated from the rest, in our case in a statechart) and changing a part of it does no longer require the developer to scan through the application. This disentanglement also helps the developer in making fewer errors, as the developer can no longer , by accident, wrongly adapt parts of the application logic when wanting to adapt the UI Behaviour, as the UI Behaviour is separated.
The disentanglement also provides the developer with a more precise global overview of the application, as when the separation of the UI Behaviour was not yet done, all concerns present in the application were entangled and now at least one concern (the UI Behaviour) can be understood separately, and one less concern is entangled in the application.
These points are illustrated and validated in chapter 6 through the use of an example.

[16] This is validated in chapter 6.

To illustrate the importance of disentangling UI Behaviour, more drawbacks of the entanglement of UI concerns can be found in [31].

Reducing overhead for the developer

A second benefit is that the presented solution is aimed towards reducing the overhead for the developer (that can be caused by using our solution) as much as possible.
As statecharts are already commonly used for designing a UI and are even considered as best practice when developing a UI [31] or an application [20], they are also used in a large number of real-world systems, as stated in [36] and [1]. As statecharts are so widely dispersed among developers, we may assume that the developer using our system is familiar with statecharts, and therefor not forced to learn a new formalism or design method to use our solution.
Furthermore, the developer is provided with a system (as described in section 5.2) supporting our solution, which can be used to perform a number of steps in our solution automatically, given the correct input.
Furthermore, the developer is not asked to make major changes to the development process that is used, as we assume that statecharts are already used in this process, since they are considered as best practice. The only change to the development process is that, when - before using our solution - there was the need to implement the entire application, including the UI Behaviour, now only the application without the UI Behaviour has to be written, as the UI Behaviour is specified separately.
The only thing the developer has to learn for using our solution, is the series of language statements (as mentioned in section 5.2.6) for expressing the statechart at the application level. However, as we have mentioned in section 5.2.6, also graphical support is provided for this task. Furthermore, it should be mentioned again that we would like to enhance our solution in such a way that the language statements are no longer necessary, and the developer just needs to specify the statechart and implement the application without the UI Behaviour.
However, as this is not yet the case, at the moment there is some overhead for the developer in the use of the language statements.

Adaptability

A third advantage is adaptability. This is highly connected to the disentanglement, since the disentanglement provides the way to adapt the UI Behaviour and the application more easily, as stated in the beginning of this section. As the UI Behaviour is separated from the application logic, making changes to one or both of them is facilitated [16]. This is because there is no longer a need to search the application for the appropriate pieces of UI Behaviour that need adaptation, and finding the pieces of application code that need adaptation is facilitated as the UI Behaviour code is no longer entangled and scattered throughout the application.
Furthermore, as we are using statecharts to express the UI Behaviour, it is possible for

the developer to change one thing (e.g. changing some UI Behaviour) in such a statechart and then let the supporting system step through the rest of the solution again with the altered statechart as input. This means that when the system finishes, the change will be noticeable when running the application.
One might mention that making a change to a complicated statechart is as hard as making this change in an entangled application. However, the statecharts are able to be extended and nested [16], which means that complex statecharts are typically build out of other, smaller statecharts, which can consist of previously defined statecharts, linked together through the nesting of states. Therefor, adapting the statechart is a matter of applying the wanted change to the appropriate ”sub-statechart”.
Furthermore, by using a medium with graphical support like CoBro, we provide the developer with a visual aid when expressing wanted changes (as mentioned in section 5.2.7). For example, when wanting to change some UI Behaviour in a statechart, one could make that change to the concept network representing that statechart and then let the supporting system go through the rest of the solution using this concept network. As this concept network is stored in CoBro, a graphical view of this concept network can be provided, making the concept network look like a drawing of the corresponding statechart. This helps the developer to visualize the wanted change and helps in finding the appropriate place in the concept network to apply the change.

Extensibility

As a last benefit, we can say that extensibility is supported in our solution [16]. Since the UI Behaviour is represented and stored as a statechart, it is possible to reuse parts of a certain statechart, or even statecharts as a whole, and extend the UI Behaviour in it. When the UI Behaviour in these parts or whole statecharts is ported to another statechart, additional parts can be added and changes can be made in order to create a new statechart. This allows the developer to ”build” some new, wanted UI Behaviour with the UI Behaviour that was already described earlier in a statechart, extending this UI Behaviour.
Statecharts can even be passed between several developers, as they are all able to read the statechart and use it in their application. The same goes for different applications of the same developer. This means that we only have to specify some UI Behaviour once in a statechart, and from then on, we can port the statechart to any application where that UI Behaviour is wanted, where we can change and extend this UI Behaviour to the needs of the application.

5.3.2 Shortcomings

Limited use of aspect-orientation

As discussed in section 5.2.9, we are not using an aspect-oriented approach in the system supporting our solution and we are currently only using a part of the power of

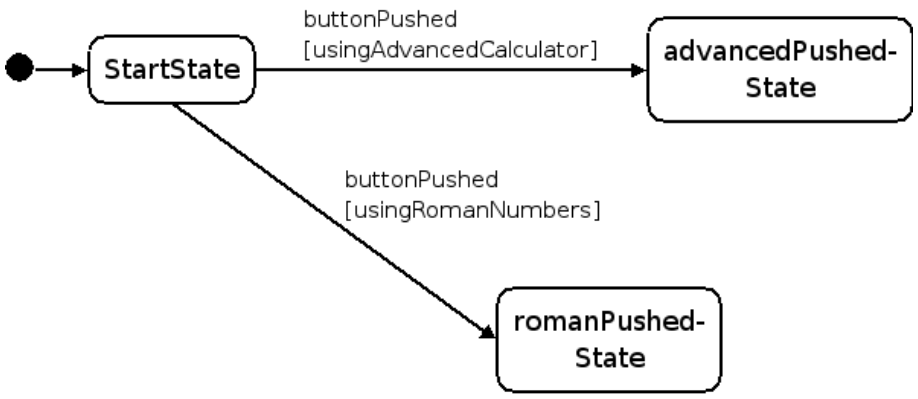

Figure 5.12: An example of problematic guards

Carma to couple the UI Behaviour to the application.
Since we are just adding the UI Behaviour at the beginning of methods (as explained in section 5.2.9) and not weaving the UI Behaviour in with the existing code, we are not using the aspects properly, as they were meant to be used in an aspect-oriented approach. This has some consequences, of which the two most important ones are discussed.

Firstly, this has consequences on the execution of the UI Behaviour in particular situations, which are now explained.
Consider a statechart that holds the UI behaviour for a calculator application, with in this statechart an initial state (e.g. the StartState), as shown in figure 5.12. Consider in this statechart also two transitions that have the same event (e.g. buttonPushed) but a different guard condition, which both start in this initial state and each end in a different state (e.g. the first one ends in the advancedPushed-state and the second one in the romanPushed-state). We could then say that the guard conditions on the first transition would then for example be that it is checked if we are using the advanced mode of the calculator, and on the second transition that it is checked if we are using the calculator in the roman numbers mode.

As we consider this situation, we can see that both guard conditions can be true simultaneously, as it is possible to be in the advanced mode and simultaneously be in the roman numbers mode.
When our solution is used to disentangle the UI Behaviour captured in the considered statechart, problems might arise. As we are just inserting calls to the statemachine - in which the checking of the guard condition is integrated - at the beginning of the methods instead of weaving them in the code, we can not control the order in which the transitions are executed. As both guard conditions for our transitions in our calculator-example could be true at the same time, either of these transitions could be executed,

depending on which call to the statemachine was inserted first in the method.
However, the order in which transitions are executed is important, as this has an impact on the statemachine (e.g. when we first execute the first transition, the statemachine will be in state advancedPushed, and the executing of the second transition will have no effect). This could be avoided by weaving in the calls to the statemachine at the appropriate places in the application, as then the order of execution is then determined by the execution of the application and no longer randomly determined by which transition-call was inserted first.

A second consequence is the doubling of the tests that have an influence on the UI as well as the application. This is explained with an example.
Consider the same statechart as described above (see figure 5.12). When we are implementing the application logic for the calculator as described above, we will most likely have a method in which it is tested if we are in the advanced mode of the calculator, and if so, some application logic should be executed. In the same method (as the transitions have the same event) there will also most likely be a test that checks if we are in the roman numbers mode, and if so, also some appropriate application logic is executed.
Hereby, if we were using aspects in a proper way, we would weave in the appropriate calls to the statemachine just after these tests (that were already there for the application logic), hereby avoiding these tests to be done again by our system. However, as we are not using the aspects to the fullest, we are just inserting code at the beginning of methods, ignoring the already available tests.
This means that for the example, we will have a method where, at the beginning of the method, the two tests if we are in the advanced mode and/or in the roman numbers mode are done, hereby calling the appropriate transition on the statemachine. In the same method however, just below these tests, the same tests, that were originally meant to be used to execute the appropriate application logic, will be repeated.
This way, we can see that we test certain situations twice, while they should only be checked once. This is especially important if these tests are time-consuming or complex, as we then most certainly want to minimize the number of times they should be executed.
Furthermore, when certain situations are tested twice when this is not necessary, we could find ourselves in inconsistent situations. Consider for example a situation in which the first test is true, and before the second test is done, things change so that the second test becomes false. This could leave us with an inconsistency between the state of the application and the state of the statemachine, which is not acceptable.

These consequences of not using aspects properly can be solved by using aspects as they are supposed to be used and by using the full power of Carma and AOP. This was already discussed in section 5.2.9.

Manual transformation of a statechart

As we mentioned in section 5.2.6, the statechart that was drawn by the developer currently still needs to be transformed manually to a series of language statements in order to be passed to the supporting system of our solution. This is seen as an overhead for the developer, as this manual transformation can be avoided. Therefor, our solution should be improved in the future to automate this manual transformation. This can be done in two ways.
The first way is to introduce a tool into our solution that can automatically transform statecharts that are drawn in a UML-tool to a series of language statements representing those statecharts, or directly to the appropriate concept networks. Hereby, the developer only needs to specify the statechart in a UML-tool, and the transformation of the statechart is done automatically by the tool, which would then become a new part of our supporting system.
The second way is to add graphical tools to Cobro, to enable the developer to draw and specify statecharts graphically directly in Cobro (e.g. through the use of CoBroNav, as in figure 5.9). This way, as the developer is creating a statechart, actually the concept network for this statechart is directly created, and the manual transformation from a statechart to language statements as well as the automatic transformation of the language statements into a concept network are not needed anymore.

No runtime changes supported

As CoBro supports runtime changes [12] to concept networks, this would be a benefit to hold on to in our supporting system. However, by introducing the statemachine and the aspects, we are using the data in the concept network to generate other structures (i.e. the statemachine and the aspects). This means that the data in the concept network is duplicated and split up into different parts, which are then inserted in the correct structure. Hereby, we lose the benefit of making runtime changes, as it is no longer enough to adapt the data in the concept network, since that data was duplicated and dispersed over the statemachine and the aspects.
When we want to adapt the data in the concept network (being the representation of the initially drawn statechart to represent the UI Behaviour), a recompilation of aspects (section 5.2.9) and the statemachine (section 5.2.8) is necessary. In this recompilation, the updated data in the concept network is dispersed over the appropriate structures, which then also can use the updated data.
An example of such a runtime change that will not work, is the changing of a guard condition on a transition. In this case, a new aspect should be created, with the advice of the aspect holding the new guard condition. This can not be done at runtime, because the new aspect should be woven on the application.
However, this shortcoming is related to the implementation of the solution, and not so much to the conceptual solution, so it can be seen as a implementation issue, and not so much as a shortcoming of the idea behind the solution.

This issue can be solved by moving all runtime-related behaviour to the concept network in CoBro, where it can be edited at runtime. This would mean that we would not generate a statemachine and that the guards of the aspects, along with all available UI Behaviour code are handled in CoBro, where these parts can be adapted at runtime.

As the solution for our problem is presented and explained in this chapter, we still need to validate the mentioned benefits of this solution, and, what is even more important, validate that our presented solution really solves the problem as stated in chapter 2. This is done in the following chapter.

Chapter 6

Disentangling state-based UI Behaviour: put to practice

In this chapter we validate the solution presented in section 5.2 and the benefits for our solution, as they are stated in section 5.3.1, illustrated by applying the solution for an advanced calculator application, which is our case study.
First the calculator application is discussed. This application is meant to represent a more advanced calculator than the one used in section 5.2, and therefor it will have some extra features, so the UI Behaviour comes out more explicitly. The design of this application was done in a conventional matter (which means that statecharts were used) and the application is written in Smalltalk. We use this application throughout this chapter to validate and illustrate the different steps in the solution.
Hereafter, we follow our solution for the construction of the advanced calculator application, starting with drawing a statechart, as seen in the previous chapter. Hereby we illustrate the extensibility of the statecharts, as it was mentioned in section 5.3.1.
After this, the most important section of this chapter comes up, as we validate the solution on the disentanglement of the UI Behaviour, which is the initial problem of this dissertation. Again, the calculator application is used to show that the disentanglement is a fact.
To conclude this chapter, we validate if the separation of UI Behaviour and application is maintained up to the point that both need to be coupled to one another, and we validate if the solution supports adaptability, as mentioned in section 5.3.1.

6.1 The advanced calculator

A screenshot of the advanced calculator application is depicted in figure 6.1.

In this screenshot we notice the advanced options of the calculator, compared to the basic calculator used in section 5.2. First of all, the user has the possibility to switch between the advanced calculator and the basic calculator, by using a radiobutton. Depending on which mode the calculator is in, certain widgets (such as the advanced

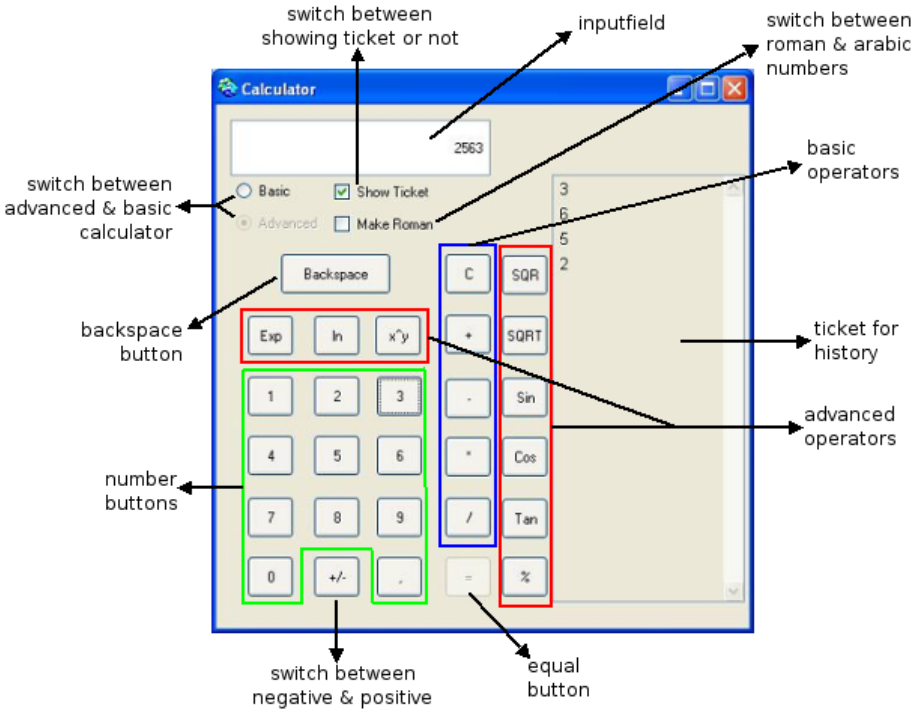

Figure 6.1: A screenshot of the advanced calculator application

operators, the ticket, etc.) will be visible or invisible. The basic calculator was already explained thoroughly in section 5.2.1, and the part that has stayed the same in the advanced calculator will therefor not be explained again.

However, when the calculator is in the advanced mode, some extra functionality is enabled, which may need explanation.
It is shown that a number of new operator-buttons are added to the calculator, so more advanced calculations are enabled. This includes for example the sinus and the square root. Furthermore, it has become possible to enter decimal numbers, by using the decimal comma, which can be found at the bottom of the calculator. If the comma is used when no number was entered, it will automatically generate a zero before the comma. Also a backspace-button is introduced, so the user can correct errors in numbers that were entered. Furthermore, the possibility is given to enable or disable a ”ticket”, which is a list of all things that were entered. This is done through the use of a checkbox on top of the calculator. One can look at it as a kind of history for the calculator. For example if the number two is entered in the inputfield, the ticket will display the number 2. The user also gets the choice between displaying the number-buttons in arabic numbers or in roman numbers. This again is done through the use of a checkbox.

As the advanced calculator and its features are explained, we now want to implement

an application and a UI for it. As we want the UI Behaviour to be separated from the other concerns during the implementation, as this facilitates the implementation (as stated in section 5.3.1), we follow the solution presented in section 5.2 and use the supporting system.

6.2 Drawing a statechart

As we wish to separate the UI Behaviour from other concerns (i.e. the rest of the application) and we are following the solution that is presented in this dissertation, the first step is to specify the UI Behaviour for our application in a statechart [1].
The statechart representing the UI Behaviour for the advanced calculator is depicted in figure 6.2. As we use this particular statechart throughout this chapter, we will explain the different parts in the statechart shortly.

In this statechart we see a number of new elements, as well as elements that are already known from the statechart for the basic calculator as depicted in section 5.2.4. Since the statechart for the basic calculator was explained before, we will now only focus on the newly added elements.
It can be noted that the part of the statechart that lies within the blue square (which represents a nested state) is not that different from the statechart in section 5.2.4, apart from some small changes. Only some advanced elements were added to this part, such as a backspace-transition, an entry-action that allows data to be written to the ”ticket” and an entry-action that makes the advanced operator-buttons visible. These advanced operator-buttons are referred to as ”AOperatorButtons” in the statechart. All the operator-buttons together (advanced and basic) are referred to as ”OperatorButtons”.
We also notice that the name of the states has been adapted and the letter A has been added in front of the name of every state, to point out that these are states of the ”Advanced” calculator. Furthermore, it is shown that an advancedOperatorEnteredtransition is introduced. This transition is performed whenever an operator is used that can be applied to one operand, as is the case for square, square root, sinus, etc. The transition ”operatorEntered” is performed whenever an operator is used that needs to be applied to two operands, such as multiplication, taking a percentage, etc.
Outside the blue square, the statechart for the basic calculator is used to represent the basic part of the calculator, as it is possible to switch between the advanced mode and the basic mode of the calculator, as mentioned in the introduction of this chapter. Not the whole statechart for the basic calculator is repeated, as this would clutter the statechart for the advanced calculator, and therefor it is just referred to as a composite state (see section 4.2.1). Switching to the basic mode of the calculator happens through the

[1] This step can already be done during the design phase.

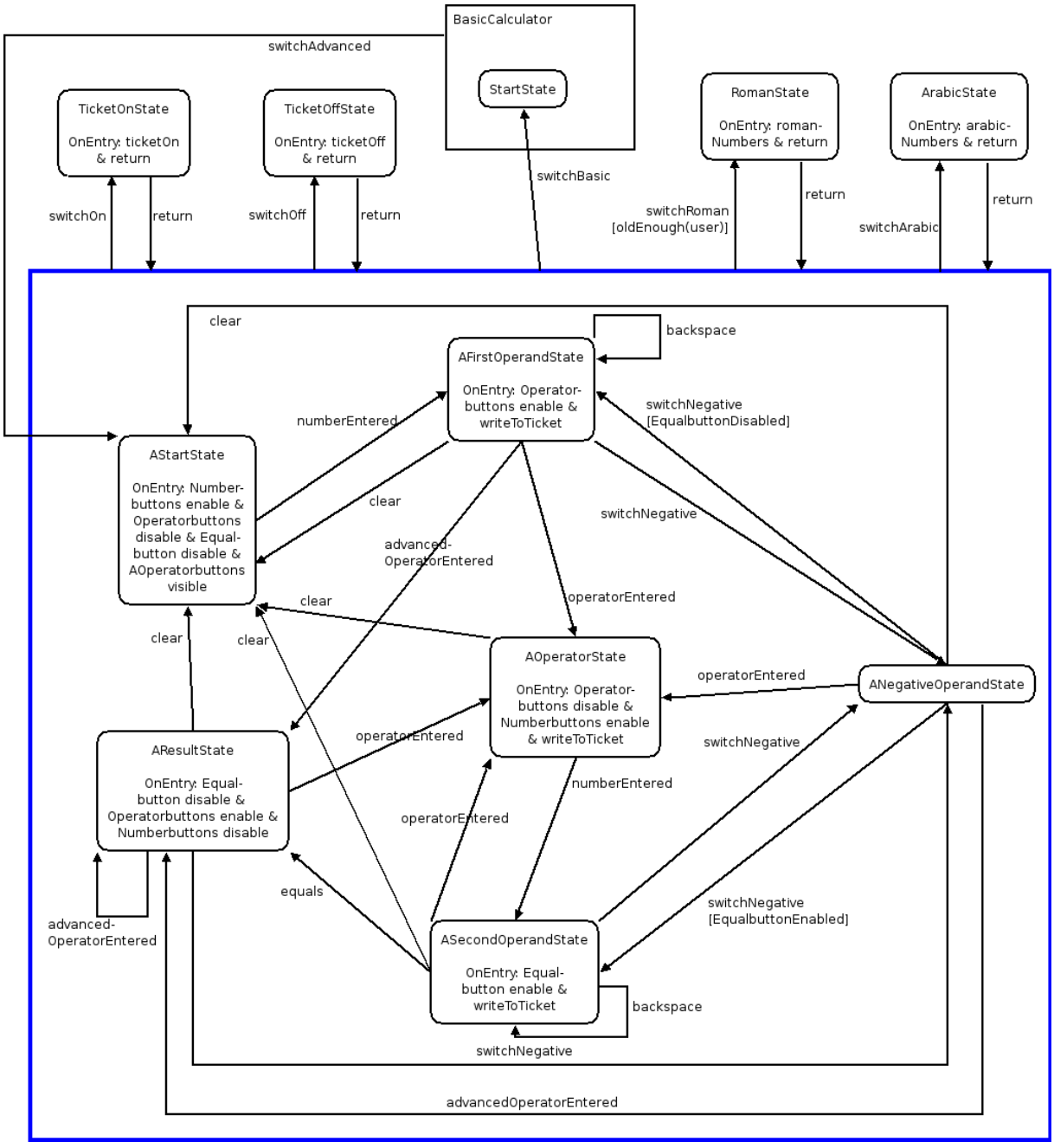

Figure 6.2: A statechart representing the UI Behaviour for the advanced calculator

switchBasic-transition, which brings us in the startstate of the basic calculator. Switching back brings us in the startstate of the advanced calculator.
More states were added outside the big square to represent the possibility of turning the ”ticket” on and off and switching between roman and arabic numbers.

As the UI Behaviour for the advanced calculator is represented in this statechart, we can mention that the *specification of this UI Behaviour has been facilitated*. As the UI Behaviour is handled separately in the statechart, we do not have to search for the appropriate places in the application code to place the UI Behaviour code and we do not have to integrate the UI Behaviour code with the application code. Since this is avoided, we consider the specifying of the UI Behaviour in a statechart an improvement over manually inserting pieces of UI Behaviour code in the application.
This is a result of the separation of the UI Behaviour from the rest of the application, which is explained further in section 6.4.
This is also the case for the UI Behaviour in the advanced calculator statechart. In figure 6.2, we see that we specify UI Behaviour, such as the enabling of the Equalbutton and the writing to the ticket when we reach the ASecondOperandState, in the advanced calculator statechart, and not in the application. In section 6.4, we will illustrate this further.

6.3 Extensibility

As we have seen the statechart for the advanced calculator application in figure 6.2 and this statechart was explained in section 6.2, it should be noted that this statechart resembles the statechart for the basic calculator, as it was presented in section 5.2.
More specifically, the part of the statechart that lies within the blue square highly resembles the statechart for the basic calculator in section 5.2.4, apart from some small changes. This is because the statechart for the basic calculator was used to construct the statechart for the advanced calculator, since the UI Behaviour for the basic calculator was partially the same as that for the advanced calculator. Hereby, we extended the UI Behaviour in the statechart for the basic calculator, adding the appropriate states and transitions and adapting the entry-actions in order for more advanced features to be added.
This is also the case for the UI Behaviour for the basic mode of the advanced calculator application, as we are just inserting the statechart for the basic calculator into the statechart for the advanced calculator to express this mode.
Hereby we illustrated the point that we can use previously defined statecharts holding UI Behaviour to create new statecharts, by extending and changing them.

As we illustrated the *extensibility* with the calculator applications, it can be noted that the UI Behaviour that was specified in a statechart once, can be used again in a similar or a different application, assuming that the appropriate adaptations are made. By this

```
switchRomanWithAgeCheck
7 (self roman)
8  ifFalse:[self numbers do:
9            [:each | |button|
10           button := self builder componentAt: each.
11           self makeArabic: button.]
XX          makeBackgroundArabic.]
12 ifTrue:[
1    (self userAge < 11)
2     IfTrue:[Transcript show:
3                'You are not old enough for roman numbers'.
8             self numbers do:
9                  [:each | |button|
10                 button := self builder componentAt: each.
11                 self makeArabic: button.]
XX            makeBackgroundArabic.]
4     ifFalse:[Transcript show:
5                'You are now using roman numbers'.
6             self calculateInRoman.
12            self numbers do:
13                 [:each | |button|
14                button := self builder componentAt: each.
15                self makeRoman: button.]
XX            makeBackgroundRoman.]]
```

Figure 6.3: A part of the advanced calculator application, with mixed application logic and UI logic

we mean that in most cases, the statecharts (or parts of statecharts) holding the UI Behaviour can not be just inserted into other statecharts or reused in other situations, but that changes are necessary. For example a statechart can be extended, but the original UI Behaviour can also be changed in order to fit into a new context (e.g. the events of the transitions in the statechart can be changed to fit in with the context in mind, while the rest of the statechart is kept).
This means that the developer does not need to specify a new statechart from scratch every time, but that it is most likely [2] that there already exists another statechart that can be extended or used in the new statechart.

6.4 Separating UI Behaviour from other concerns

In section 6.2 we described that the UI Behaviour for the advanced calculator was captured in a statechart. By specifying the UI Behaviour for this application in a statechart, we are separating the UI Behaviour from other concerns, which are described in the rest of the application. This is illustrated in figure 6.3, figure 6.4 and figure 6.5.

[2] Since statecharts are widely used (see section 5.1), the chance is high that someone has already drawn a statechart that the developer can (partially) use to specify the wanted UI Behaviour.

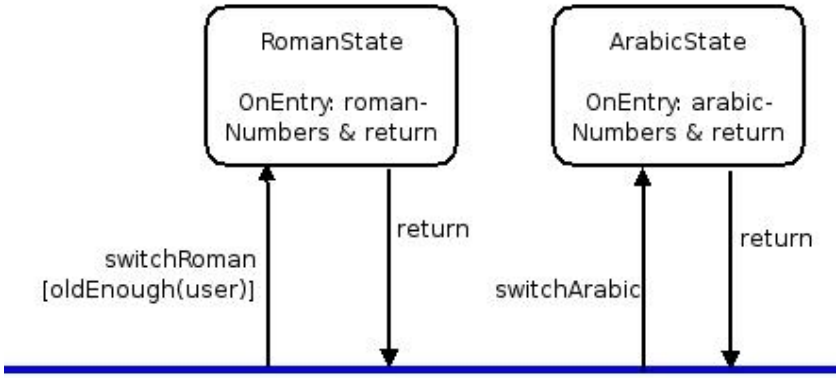

Figure 6.4: A part of the statechart of the calculator, representing the UI Behaviour that was separated

In figure 6.3, we see a code snippet that we remember from chapter 2. This is what a part of the code of the advanced calculator application would look like, if this application was implemented without using our solution. The difficulties with implementing and adapting this code were already mentioned in chapter 2, but we want to stress the entanglement and scattering in this part (in this case: a method) of the calculator application. As we see that the different colors and line-numbers [3] are mixed together in this method, the entanglement of the different concerns is shown, and as lines of code are repeated throughout this method (recognizable by the identical line-numbers), we can see the scattering even within this one method.
It is exactly the code in figure 6.3 that we want to avoid, by separating the UI Behaviour from the other concerns in the calculator application, which is done through the use of our solution, and which is illustrated in the remainder of this section.

In figure 6.4 we see the UI Behaviour for the method in figure 6.3, which has been specified in a statechart. It should be noted that the red, magenta and brown pieces of code out of figure 6.3 (which were described in chapter 2 as the UI Behaviour) are now represented in a statechart, which in fact is a piece of the calculator-statechart as shown in figure 6.2. Hereby we notice that the magenta pieces of code and the brown piece that brings the background in its roman state are captured in the switchRoman transition and in the romanNumbers entry-action. The red pieces of code and the brown pieces that bring the background in its arabic state are captured in the switchArabic transition and the arabicNumbers entry-action.
In figure 6.4, we only see the UI Behaviour for the mentioned method of the calculator, as other concerns are included as little as possible in the statechart. This means that

[3] For an explanation of the colors and line-numbers: see chapter 2.

```
switchRoman
7 (self roman)
12 ifTrue:[
1    (self userAge < 11)
2     ifTrue:[Transcript show:
3                'You are not old enough for roman numbers'.]
4     ifFalse:[Transcript show:
5                'You are now using roman numbers'.
6            self calculateInRoman.]]
```

Figure 6.5: A part of the calculator only holding the application logic, after the separation of the UI Behaviour

we can say that the UI Behaviour was separated from other concerns, and that we can specify and adapt this UI Behaviour separately.
We would also like to note that this also is an example of what was stated at the end of section 6.2, being that specifying UI Behaviour is facilitated when this can be done separately. When we look at the entangled whole in figure 6.3 and the statechart in figure 6.4, we consider the drawing of this statechart to specify the UI Behaviour as an improvement over the inserting of the UI Behaviour code at the appropriate places in the code.
However, now that we specified the UI Behaviour separately, we still need to implement the rest of the application, leaving this UI Behaviour out. This is discussed in the following paragraph and illustrated in figure 6.5.

In figure 6.5 we recognize parts of the code that was shown in figure 6.3. These parts are what remains after separating the UI Behaviour by specifying it in a statechart (as shown in figure 6.4), as they express the application logic. This means that figure 6.5 expresses how the method in 6.3 looks like when our solution is used to create the application and UI, hereby separating the UI Behaviour from the rest of the application.
This shows that there is a *disentanglement of the UI Behaviour* from the other concerns in the application, as the other concerns are still kept together in the application and the UI Behaviour is separated from the application. This is shown for the calculator example, as the UI Behaviour is specified in the statechart depicted in figure 6.4 and the other concerns in the calculator are implemented in the calculator application, shown in figure 6.5.
It should also be noted that it is *facilitated to implement the application*. This is the effect of the disentanglement of the UI Behaviour. As the UI Behaviour concern is separated from the application, the application code has become less entangled and also smaller (since the code for the UI Behaviour is left out), which makes it easier to implement the application. As we consider implementing an application that is smaller and without the need to insert UI Behaviour in the appropriate places an improvement

over implementing an application which has this UI Behaviour entangled with the application, we can say that it is facilitated to implement the application.
Furthermore, the implementation of the application has ruled out errors in the integration of the UI Behaviour, as the developer can no longer wrongly integrate the UI Behaviour code in the rest of the application code (this is done by our system) or accidentally change the UI Behaviour code, as it was specified in the statechart.

As it was illustrated by figures 6.3, 6.4 and 6.5, we can say that by using our solution, we *separated the UI Behaviour from the other concerns in the application*, hereby facilitating as well the implementation of the application as the specification of the UI Behaviour. Hereby we avoid the UI behaviour to be entangled with and scattered in the application, and we enable the UI Behaviour to be adapted separately, without influencing the rest of the application. This is discussed further in section 6.6.
However, as we separated the UI Behaviour from the rest of the application, we want to hold on to this separation as long as possible, to then finally couple the UI Behaviour to the application to ensure that the application can work as if the UI Behaviour was never separated. This is explained in the following section.

6.5 Maintaining separation

As we have specified the UI Behaviour in a statechart and the rest of the application was implemented, the separation between UI behaviour and application is established. This separation is maintained by the further use of our solution, which is shown in this section. As we have already illustrated the language statements, concept network, statemachine and aspects thoroughly in section 5.2, we will not repeat this here for the advanced calculator. However, the language statements, the concept network and the generated aspects for the advanced calculator can be found in the appendix.

First, the developer should transform the created statechart into language statements (as described in section 5.2.6). This means that the developer should enter the language statements representing the created statechart in the Smalltalk transcript, and execute them.
It should be noted that this is the last manual step in our solution, as after this step, the supporting system takes over, hereby avoiding the overhead for the developer of making more manual steps than necessary. It was already mentioned that we would also like the transformation of the statechart to the language statements to be automated, but that this is not yet the case, and therefor the transformation still has to be done manually for the time being.
As we are just transforming the statechart to the corresponding language statements, without adding or removing anything, we can say that the separation of UI Behaviour from the application is still maintained. The only difference is that the statechart is

now expressed in language statements.
This is shortly illustrated by the following language statements for the AFirstOperand-State in the advanced calculator.

```
b := State new: #AFirstOperandState.
b hasOnEntry: 'Operatorbuttons enable'.
b hasOnEntry: 'writeToTicket'.
```

After this last manual step of specifying the UI Behaviour through the language statements, a concept network is created by the supporting system (as explained in section 5.2.7).
As the developer has expressed the statechart in the language statements, these language statements (which are especially created in order to *reduce overhead for the developer as much as possible*) are now used as input for the supporting system. The system will take the language statements and trigger their corresponding CoBro CML statements, which are more complex than the language statements. This is why the developer is asked to represent the statechart in language statements, and not directly in Cobro CML.
For the advanced calculator, this means that we should use the language statements as specified above as input for the system. The system will then create a concept network representing the calculator statechart. This is illustrated with CoBro CML statements, corresponding to the above given language statements. Again, not all concepts are given, but the full list of all the created concepts for the advanced calculator are found in the appendix.

```
(b := Concepts new: #{AFirstOperandState})
    hasPreferredLabel: 'AFirstOperandState'.
    b superconcept: Concepts.State.
    b hasOnEntry: 'Operatorbuttons enable'.
    b hasOnEntry: 'writeToTicket'.
    b save.
```

We can see that the separation of the UI Behaviour from the rest of the application is still maintained, as the UI Behaviour is now captured separately in a concept network, and the rest of the application is still implemented as in the beginning.

As the concept network for the calculator application is established, we want to generate a statemachine out of this concept network, creating a kind of ”operational statechart” for our application (as mentioned in section 5.2.8). The system therefor creates a new statemachine, and adds all the states and transitions that are found in the concept network for the calculator to it. Here we see again that the supporting system is the one building the statemachine, and not the developer. Having to create a statemachine and adding the correct states and transitions, that were extracted from the concept network, would mean a more complex solution and *a overhead for the developer, which we wish*

to avoid.
In this stage, the UI Behaviour is split up into the statemachine and the aspects (which are discussed in the following paragraph), but it is still separated from the rest of the application.

Using the concept network and the hooks that are available in the application, the aspects for the advanced calculator are generated by the system (as discussed in section 5.2.9). Here follows an aspect that is generated for the advanced calculator.

```
Andrew.TestAspects defineAspect: #AStatechartAspect
    superAspect: #{Andrew.Aspect}
    ofEach: #AdvancedCalculator
    instanceVariableNames: ''
    aspectVariableNames: ''
    category: 'AdvancedCalculator'.

before ?jp matching {reception(?jp, #switchRoman)} do
((SOULEvaluator eval:
    'if oldEnough(AdvancedCalculator user)') allResults)
  ifTrue:[AdvancedCalculator statechartInstance switchRoman].
```

As we are using Carma, we have the possibility to use more advanced guards [4], which are not strictly necessary in our solution, but which are useful in some situations. This is the case for the guard condition that is placed on the switchRoman-transition, as seen in figure 6.2, which is also illustrated in the aspect above.
With this guard we want to protect children from getting stuck in the advanced calculator. Children that have not passed the age of 10, have not passed the fourth grade (in which roman numbers are taught in Belgium) yet. Therefor they are not familiar with roman numbers, and we want to prevent these children from using the roman numbers as they have probably clicked the checkbox to switch to roman numbers by accident. Therefor we need a guard condition on the transition ”switchRoman” that rejects to do the transition when the user of the calculator is not older than 10 years old.
It is shown in the aspect that we specify a logic rule ”oldEnough”, which tests if the user of the calculator is old enough to be able to use the roman numbers, and depending on this answer sends the message to the statemachine or not. The code for the logic rule in SOUL is the following.

```
oldEnough(?x)
    if greaterOrEquals(?x, 10)
```

This shows that we can use logic rules to reduce the need to write complicated pieces of Smalltalk code for our guards. As this is just a simple example, the complexity of writing the logic rule or writing the Smalltalk code for this guard will be more or

[4] Through the use of logic programming.

less equal, but as we take a more complicated example, we can show the advantage of using logic rules. Consider for example a situation in which we want to test if a user has a certain age, is female and has a dog. In this case, the Smalltalk code will contain a number of nested if-tests to test these properties, as the logic rule will just contain one other logic rule for each of these properties. Hereby we can see that the more complex the guard gets, the more complex the Smalltalk code will be, and that logic rules can avoid this complexity by using at most one logic rule per tested property.
Furthermore we can use logic rules to specify guards and adapt these guards in our logic repository, without even touching the system. Whenever the calculator is ported to a country where children already learn about roman numbers in the second grade, the only thing the developer has to do is change this one rule in the logic repository, and the UI Behaviour will work in the wanted way, without having to change the statechart, or even the guard condition.
Finally, we can also say that however the use of logic rules may not be strictly necessary (however useful) in our solution, it may become necessary in the future when our solution is put into the context of Deuce, in which other concerns than UI Behaviour are separated.

As up until now the separation between UI Behaviour and application was maintained through the transformations, the time has now come to couple the UI Behaviour to the application by weaving the UI Behaviour in the application in order for the application to work as if the UI Behaviour was never separated.
In this step, the created aspects are woven on the implementation of the calculator application. This means that the appropriate pieces of code (the advices of the aspects for the advanced calculator application) are put in the appropriate places, defined by the pointcuts of the aspects. For example in the model class of the advanced calculator application the method (in this case a hook) with the name ”switchRoman” will now consist of a piece of code that sends the message ”switchRoman” through to the statemachine if the user has the appropriate age, as we defined it in the advice of the corresponding aspect. This means that the advanced calculator application is now ready to be used.

As we now open the advanced calculator application, we first see the basic calculator, as this is the startstate for the advanced calculator statechart. As we push the radiobutton that switches to the advanced calculator, we can see that the user interface for the calculator switches to the advanced mode and the necessary UI Behaviour is executed.
This means that the solution really separates the UI Behaviour concern from other concerns in the calculator application (as was shown throughout this section), as later on the UI Behaviour is coupled to the application in a way that the application can function properly. Hereby, the appropriate UI Behaviour is executed at the appropriate time (when the UI reaches the appropriate state), without the user of the calculator knowing

that the UI Behaviour was ever separated.

6.6 Adaptability

As it was illustrated in this chapter that our solution separates the UI Behaviour from the rest of the application for the advanced calculator, it should still be possible to make changes separately to the UI Behaviour and the application.

Whenever the application should be changed, for instance when some application logic (like the way the sum is calculated in a calculator) is adapted, the changes can just be performed in the application code. Applying changes to the application should be simpler with the UI Behaviour separated, as the application code has become less entangled and also smaller, which makes it *easier to adapt the application* (as mentioned in section 6.4 for the implementation).

Whenever the developer wants to adapt the UI Behaviour, these adaptations can be made on two places.
Firstly, the developer can change the language statements, in which he can adapt the UI Behaviour (like adding a new state the UI for the calculator can be in) by adding some language statements or adapting existing language statements. These language statements should then again be given as input for the supporting system, which will create the adapted UI Behaviour.
Secondly, the changes can be made in CoBro, directly to the concept network for the calculator. As CoBro provides a graphical representation of its concept networks (as shown in section 5.2.7), the developer can also adapt UI Behaviour in this way. In this case, the system will directly get a concept network with which it can create the adapted UI Behaviour.

In both situations *adapting the UI Behaviour can be called simpler* than adapting the UI Behaviour when it was still entangled with the application, as the developer now knows where the changes should be made (in the language statements or in the concept network), and no longer has to scan through the application in search of UI Behaviour. However, it should be noted that after each adaptation of the UI Behaviour, the system should execute the whole process of making a concept network, a statemachine and aspects again, as the changes should have their effect in the application. As mentioned in section 5.3.2, it is thus not possible to apply runtime changes to the UI Behaviour.

Furthermore, adapting the UI Behaviour in the statechart, the language statements or the concept network has no impact on the application, only on the UI Behaviour. This is illustrated with the same example as used in section 6.4.

In figure 6.6 we see that the entry-action for the RomanState has been changed. This, however, is done separated from the application, and has no impact on the appli-

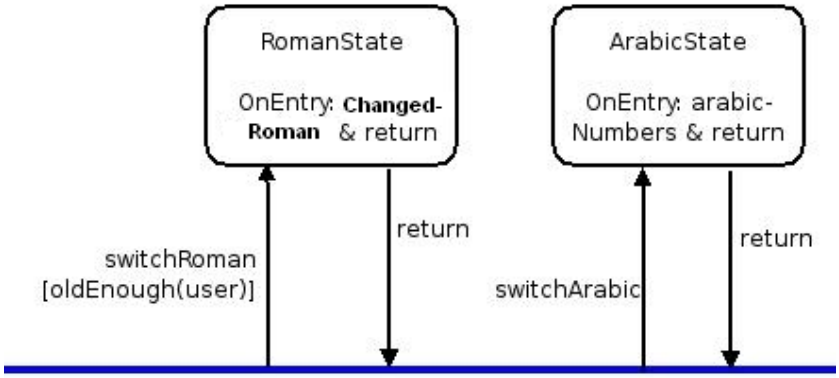

Figure 6.6: A part of the calculator with changed UI Behaviour

cation logic, which was described in figure 6.5.

Now that we have validated our solution through the use of a case study, we can also validate the solution by referring to other research that used the same approach as we did in this solution, or research that is aimed at similar problems as the one that is being tackled in this dissertation. This related research is discussed in the following chapter.

Chapter 7

Related work

In this chapter, the problems and solutions that were discussed in this dissertation are linked to existing research, as this is also a way to validate (the relevance of) our problem and solution, as the other existing research in this field can be aimed at the same goal as was set in this dissertation or use a similar approach. This would mean that we are not alone in trying to solve our stated problem or using a certain approach to solve this problem, which could validate the relevance of this dissertation.
In the beginning of this chapter, we discuss some research which aims at a separation between the entire UI and the application. Hereby, we will see that this separation should not only take place between the UI and the application, but also within the UI, as was stated in chapter 2.
Following this constatation, we discuss some existing research about a separation within the UI. First we take a look at some research which aims at separating the application interactions (which is another UI concern, as explained in the introduction and in chapter 2) from the rest of the application and from the other UI concerns. Thereafter, we consider some research which is aimed at separating UI Behaviour from other concerns, which matches with the goal for this dissertation.

7.1 Separating UI from application

As we have mentioned in chapter 2 and in the introduction of this dissertation, the *UI and the application should be separated* as much as possible [23] [21]. However, a number of possibilities to achieve this separation are available.

In [15], Dobos discusses and investigates a series of kinds of software architectures (such as the Seeheim Model [16], Knowledge-based Front End (KBFE) [17] and MVC [32]) that can be used to achieve such a separation, in order to find out what kind of an architecture a software system should have so that the user interface and the application can be separated from each other. However, Dobos concludes that there is no single answer to that question, as there are *different kind of software architectures that suite*

for different kind of systems. Nevertheless she notices that all of these architectures have some common properties like the division into layers or the important meaning of the application interface [15].
Furthermore, Dobos introduces some solutions that can *support the separation of the user interface and the application* when combined with the appropriate architecture, as she finds that the discussed architectures often need additional help in doing this [15]. Also, she proposes a solution that enables changes on the user interface without affecting the business logic of the application, as again she notices that the mentioned architectures can use some help.

In [8] and [9], Carr (long before Dobos) had already understood that the existing architectures could use some help in separating the UI from the application, and therefor he *created his own method for achieving this separation*. In this method, Carr specifies the complete user interface through the use of Interaction Object Graphs (IOGs), which is an extension of statecharts, to separate the UI from the application.
As Carr examined statecharts (see section 4.2.1) and PetriNets [42] [43] to specify the UI in, he noticed that the mentioned notations use an abstract representation of the UI, which makes it difficult to express the appearance of the UI, but makes them suitable for, for example, expressing the UI Behaviour. Therefor, he created a notation that can handle a concrete representation of the UI, based on statecharts. This way as well the appearance as the behaviour for the UI could be expressed in one notation, which he called Interaction Object Graphs (IOGs) [9].
Nevertheless, Carr's work was not completely finished in [8] and [9], as IOGs are an executable, graphical specification notation, which can be used to specify entire user interfaces [8], but tools that go directly from the diagram to executable code were not yet created. However, if this was the case, an entire UI could be expressed in IOGs, hereby separating the UI from the application, and then using the tools, the coupling between UI and application could be restored.
When we compare the approach that Carr used with the solution that was presented in this dissertation, we should first notice that we are dealing with a *different kind of separation*, as Carr wants to separate the entire UI from the application, as we are only separating one UI concern. Since there could still be entanglement within the UI itself between the different UI concerns, separating the entire UI might still leave entanglement caught in the UI, and therefor we chose to take the path of separating only one concern. This, however, means that we are only solving the entanglement of one concern [1], as Carr sees it bigger and disentangles all the UI concerns from the application.
Nevertheless is Carr also using the same conceptual technique as we are using in our solution, being the separately specifying in a graphical notation (in his case IOGs, in our case statecharts) of the part that needs to be separated, as then later on an exe-

[1] It can be noted that since this work fits into Deuce, which aims for the disentanglement of all UI concerns, we indirectly also aim for a total disentanglement of UI concerns, which goes even further than Carrs work.

cutable equivalent of this graphical notation can be made. Furthermore, it should be noted that as Carr separates the entire UI, he needed a notation that could also describe the appearance of the UI, next to the behaviour, which can be described in statecharts. Therefor he introduced IOGs, which are more powerful than statecharts (at least, they can express what statecharts can express plus the appearance of a UI), but therefor also more complex for the developer to work with, which could be seen as a disadvantage.

Now that these different ways of separating the UI from the application are discussed, we can say that both Dobos in [15] as Carr in [8] and [9] realized that the architectures and solutions to *separate the UI from the application needed something extra*, as they both tried to extend these architectures, whether by adding support or by creating a new system. However, as mentioned in chapter 2, it can be said that it are not the architectures and solutions that need to be improved, but the approach of separating the UI from the application, as we have seen that there can also be *entanglement within the UI* itself [23]. Therefor, we could look at the problem no longer as separating the UI from the application, but as separating one UI concern from all the other concerns, whether application or UI. This is also the approach that is used in this dissertation.

We can see that others also noticed the entanglement within the UI, but that they defined other UI concerns than we did in the introduction and in chapter 2 (e.g. based on the Seeheim model, UI Behaviour is not seen as a separate UI concern) or they applied the disentanglement specifically for their problem. The latter is expressed for example in [49], in which Taylor created a separation as well between the user interface code and the application, as between the user interface code and the underlying windowing systems and toolkits specifically for the Chiron-1 system. This can be considered actually as applying a separation between application interactions (as we call it, see chapter 2) and the rest of the UI onto a specific application, with in this case the application interactions being the interactions with the toolkits and windowing systems.

In other research however, the same UI concerns as we used throughout this dissertation were distinguished and the research became *focussed on separating one certain UI concern* (e.g. application interactions or UI Behaviour). Furthermore, we also wanted to find more general ways of applying the separation within the UI, and not application-specific, as is the case in [49]. The research that answers to these specifications is discussed in the following section.

7.2 Separating one UI concern from other concerns

Over the years a number of methods have been used to specify specific parts of user interfaces. These include grammars [47], algebraic specifications [24], task description languages [30], transition diagrams [51], statecharts [29] [52], and interface representation graphs [46]. However that there are alternatives available, statecharts are still

commonly used and even considered as ”best practice” for specifying the behaviour of a UI (as mentioned in section 5.1). This is also shown in the remainder of this section, as all the research in the remainder of this section uses statecharts (or an extension of them) to specify one part of the user interface, being the UI concern that needs to be separated from the other concerns. This shows that the approach we used in this dissertation of using statecharts for the specification of UI Behaviour is also used by other researchers, who also considered the statechart-approach as the best approach.

7.2.1 Disentangling application interactions

The research in this section specifically handles only one part of the UI, being the *application interactions*. As we want to show that there is research at hand that be compared to ours that also *disentangles one particular UI concern*, and then more particularly a UI concern that is the same as one of the three UI concerns that we specified in chapter 2, the research in this section does exactly that for the application interactions, hereby using statecharts.

In [36] and [37], Nogueira de Lucena describes a solution for the disentanglement of application interactions that can be compared to the solution that was presented in this dissertation for the disentangling of UI Behaviour.
The proposed solution and approach in Nogueira de Lucena's research is based on Seeheim's logical user interface model [16], which divides a user interface into three layers, as can be seen in figure 7.1.
The presentation layer is responsible for the external presentation (the appearance) of the user interface. This layer defines how an interactive system appears and is viewed by the user. It performs low-level input/output processing (lexical aspects).
The dialogue control layer (which we call *application interactions*) receives input from its surrounding layers and determines how the conversation evolves based on this input. This layer defines as well which sequences of such input are to be considered or not (syntactic aspects). The dialogue (control) layer has to keep the current state of the user interface and has to have control over this state, since the dialogue evolution depends on it. It accepts input from the lexical layer and transforms it into commands and data on the one hand, and receives input from the application, which requires data and is supplying responses to user requests on the other hand.
The last layer (the application interface model) represents the functionality (semantics) of an interactive application [37].

As we mentioned above, the Seeheim model does not take UI Behaviour into account as a separate part (it is mixed in with the presentation layer and the dialog control layer), but this does not get in the way of the fact that it does specify application interactions as we specified it in chapter 2. Since Nogueira de Lucena only handles the application interactions in his research, we can say that his research handles one par-

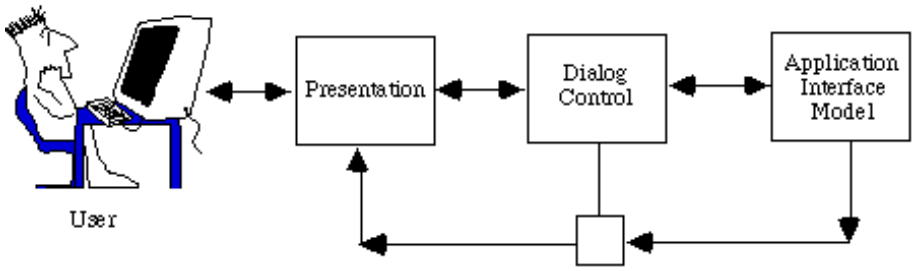

Figure 7.1: The Seeheim Model

ticular UI concern (the application interactions), as we described it in chapter 2.

As the UI concern that needs to be disentangled in Nogueira de Lucena's research is determined, we can now have a look at his solution, and compare this solution to ours.
In his solution, the dialogue control (this is the second layer of the Seeheim Model, which we call application interactions) is expressed in terms of a statechart. This statechart is then converted into tables handled by a run-time system. The execution of this run-time system driven by those tables is then equivalent to performing the behaviour specified by the corresponding statechart.
When the run-time system is executed, the resulting system calls specific functions in the application whenever user interactions take place that are mapped to the corresponding events by the presentation component. This means that the appropriate functions in the application are called whenever an event takes place in the user interface. The application is seen as a set of subroutines which can be invoked by an action of the user.

The solution that is proposed by Nogueira de Lucena is closely related to our solution, as almost the same approaches are used as in this dissertation. The biggest difference naturally lies in which part of the application that is disentangled. As we have seen, we disentangle the UI Behaviour from the rest of the application, while this research disentangles the application interactions (which is the layer in which the UI communicates with the application). Furthermore, as the Seeheim Model is used, UI Behaviour is not considered as a separate part, which is crucial for our solution.
Nevertheless, there are *much resemblances* to be noticed between the proposed solution in this research and our solution: the use of statecharts to express the wanted concern, the use of a run-time system (which is comparable to our statemachine), etc.
Nevertheless, it should be noted that instead of giving the specified statechart to a system and letting the system step through the solution, a manual approach is used here, which forces the developer to transform the statechart into tables that need to be writ-

ten in C. It is also not very clear how the developer should map the statechart to the tables. However, since the solution is implemented in C and not in Smalltalk, like our solution, Nogueira de Lucena does not has the luxury of representing the statecharts in objects, and is therefor forced by his language choice to represent it in tables. Therefor, we can say that this an issue that is bounded to the programming language.
Furthermore, the run-time system is not integrated with the application, so the developer has to manually make use of this run-time system while implementing the application. This means that the developer needs to explicitly call the run-time system when he/she wants to use it, and only then the system will become active. This can be seen as a manual equivalent of the weaving of aspects that is used in our solution.
Seen through the eyes of Nogueira de Lucena, he could notice that his solution will be much more performant than ours, as it is written in C (which is more performant than Smalltalk) and since there only needs to be one transformation done from the statechart to the tables, as we have several transformation. Also it could be mentioned that in our solution we manually map the statechart to language statements, which is the same as mapping the statechart to a table as in Nogueira de Lucena's solution. However, we mentioned before that we are working on avoiding this manual transformation in our solution.

In [38], the same Nogueira de Lucena reflects about the use of statecharts to capture Human-Computer Interface (HCI) behaviour, and uses the framework that was built in [36] and [37] to capture this behaviour as a benchmark for these reflections. Hereby possible improvements for the statecharts to make them more adequate for this specific usage are investigated, which then can be used to improve the solution in [36] and [37].

As we have seen in this section that it is possible to disentangle a certain part of the UI by the use of statecharts, we would now like to show that the same can be done specifically for UI Behaviour. In the next section, we will have a look at research concerning UI Behaviour, as this is also where our focus lies in this dissertation.

7.2.2 Disentangling UI Behaviour

In this section, we specifically focus on *research aiming at the disentanglement of UI Behaviour*, as this is also the focus of this dissertation. Hereby, we present research that is tightly connected to the research in this dissertation, as the same problem is stated and this problem is approached to create a similar solution. The research done at McGill University is such research, which we explain thoroughly.

As researchers at McGill University focus on the modeling and simulation of the behaviour of a system, in particular of Graphical User Interfaces (GUIs), existing formalisms for this purpose, such as Statecharts and DEVS [6] (Discrete EVent Systems specification), are used as the foundation for their research. This research can be split

up into 4 parts: DCharts [11] [48], SVM, SCC [18] and Atom^3 [3].

As mentioned above, existing formalisms are the foundation for their research. As McGill researchers decided that neither statecharts nor DEVS would be sufficient for conducting the research, a solution came at hand with DCharts. DCharts are based on statecharts and DEVS, but have extended statecharts to make them more rigorous and expressive [11]. Many of the DCharts constructs can be found in statecharts and the semantics of those common constructs is the same in both formalisms. Even more, the syntax of DCharts is a superset of the statecharts syntax, as well as the semantics of DCharts is a superset of the semantics of statecharts.
It is also possible to transform DCharts into statecharts and DEVS and the other way around. In the work of Borland [4], a way to transform statecharts into DEVS is described and hence it is proven that DEVS has at least the same expressive power as statecharts (actually, DEVS is even more expressive than statecharts). Further research pointed out that DCharts are at least as powerful as statecharts and DEVS. Therefor the transformation from statecharts or DEVS to DCharts is trivial [4], since anything that is possible in statecharts or DEVS is also possible in DCharts. The transformation of DCharts to statecharts or DEVS on the other hand, is not that trivial. As DCharts are more powerful, it is only possible to transform non-recursive DCharts to statecharts with variables or DEVS.
This is discussed since it is a benefit for the solution of McGill University to express their UI concern in a formalism that can handle as well original Dcharts as statecharts and DEVS, as they can be transformed into a Dchart. This means that we could also use the statecharts that were drawn for our solution, transform them into a Dchart and give them to the McGill system.
Furthermore, DCharts are an interesting step in improving statecharts and developing better models, and they are more general than the statecharts that we use in our solution, as they can express more than statecharts can. However, as mentioned in [18], ”Dcharts are more complicated than statecharts”, which can be seen as the pay-off for more expressivity and generality. Nevertheless we can see that, as in our solution, the UI concern that needs to be separated is expressed through the use of a statechart-based formalism (statecharts themselves or Dcharts).

The Statechart Virtual Machine (SVM) is a project that was originally a statecharts simulator implemented in Python, but now it supports the complete DCharts semantics and the textual syntax, including the syntactic extensions. This means that it can be given a DChart, and it will simulate the flow and behaviour captured in the DChart. SVM also allows the user to raise events for the event handler of SVM from actual code and run an application using SVM with functions called as actions in the output event of transitions, or on entering or exiting a state as is the case in statecharts. SVM has multiple sub-projects. One of its sub-projects, SCC (StateChart Compiler), aims at a tool that synthesizes source code from DCharts models. This is discussed further in a dedicated SCC part.

Although SVM can be used as a stand-alone simulator that does not depend on any other modeling and simulation tool, it can be seamlessly integrated with AToM^3 (see dedicated AToM^3-part). A plugin for AToM^3 generates DCharts model descriptions from its graphical representation in AToM^3. This SVM plugin adds a DCharts meta-model to AToM^3.
In the matter of speed and space efficiency, SVM sacrifices space and speed to achieve functionality and extensibility. Here, space refers to the memory space required for a simulation. Because SVM is a simulator, speed and space usage is not the most important. However, it must provide a suitable experimental platform for a complete DCharts syntax and semantics. It must also be extensible so that new features can be easily added to the simulator, as DCharts are improved over time.

The StateChart Compiler (SCC) is a command-line tool to synthesize executable code from DCharts models. It optimizes the models and produces efficient code, which is independent of the SVM simulator. When SCC is started, the user can give it a model description, for which SCC will generate code. This code is written to a file with the same name as the model description (with its extension changed according to the target language). In the matter of speed and space efficiency, SCC sacrifices space, modularity and functionality for speed. The purpose of code synthesis is to produce highly efficient code that can be used in practical applications. Hence, speed becomes the most important factor. SCC guarantees high performance for most of the implemented features, but sacrifices the features that are not practical, or warns the users about the implemented but inefficient ones.

The parts of the McGill system that have the greatest similarities with the work presented in this dissertation, except from the using of statecharts, are the SVM and SCC. More specifically, in this dissertation, we present a combination of those two: a tool that simulates the flow of a statechart, and a tool that generates code for it. This is achieved by using one system, which has the concept network, the statemachine and the aspects integrated in it. This hybrid approach allows to take care of the state-based UI Behaviour, and at the same time create code for the application to use. But the advantage of this hybrid approach is that we can minimize the shortcomings of both sides. Where the SVM and SCC were forced to sacrifice space and speed for expressiveness and functionality, or the other way around, the hybrid approach chooses the middle path, and thereby averages out the made sacrifices.

AToM^3 is a graphical modeling and meta-modeling environment. It is able to meta-model the syntax of many formalisms, as well as generate dedicated visual environments for the model design in those formalisms. The semantics of some of those formalisms, such as PetriNets, is usually modeled with graph grammars. In this way, AToM^3 can also be used as a simulation or execution environment, with graph grammars that transform the model from one state to another. A DCharts meta-model is built in AToM^3, which defines a subset of the DCharts syntax. The semantics of DCharts is

implemented in SVM, which can be loaded in AToM^3 as a simulation engine. It makes it possible to simulate DCharts models and at the same time highlight the current states and enabled transitions in the AToM^3 visual environment.
Atom^3 itself may not be useful to examine further, since this dissertation is not directly aimed at constructing models or meta-models. Nevertheless, this is the foundation for the McGill solution, as all pieces of the solution (Dcharts, SVM, SCC) are placed in the Atom^3-environment, which makes the whole system work. In our solution, we do not use a specialized environment, but just use the Smalltalk-environment.

Chapter 8

Conclusion

In this chapter, we first provide the reader with a short overview of this dissertation, whereafter we describe some ideas for improvements to our solution and further research based on the research in this dissertation. To conclude this chapter and thereby also this dissertation, we give a short conclusion, hereby considering our experiences during and after the writing of this dissertation.

8.1 Overview

Like applications, user interfaces need to show the ability to adapt, as some changes to the application logic cause the need for changes in the UI as well. A problem with applying such changes to the UI is that the UI logic often is entangled with and scattered throughout the underlying application logic, which makes adapting the UI and the application logic cumbersome for the developer (as described in chapter 2). As we want to avoid the issues related to this entanglement and scattering (hereby helping the developer in making adaptations), the UI logic and the application logic are separated as much as possible.
However, not only UI logic and application logic should be separated as much as possible. Different concerns, which also suffer from entanglement and scattering are also found within the UI itself. These concerns, which we call UI concerns (chapter 1), are not only entangled with and scattered throughout the application logic, they are also entangled with and scattered throughout each other. This makes matters even worse for the developer when adapting (a part of) such a UI concern. Therefor, these UI concerns must not alone be separated from the application logic, but also from each other as much as possible. We call this applying separation of concerns onto UIs (chapter 2). In this dissertation, we focussed on separating one of these UI concerns from the rest of the application, being the UI Behaviour, which is described as the behaviour of a widget, with an impact on the UI.

In order to achieve such a separation, we have introduced a solution for the disentangle-

ment of state-based UI Behaviour. This separation of the state-based UI Behaviour is meant to help the developer to cope with the complexity of adapting and implementing an application, its UI and its UI Behaviour, as the developer is not forced to deal with the problems coupled to the entanglement and scattering of the UI Behaviour. To aid the developer further, we constructed a solution that is supported by a system, which was used by the developer to automate some parts of this separation.

The first step towards such a solution was specifying and defining what was understood to be state-based UI Behaviour in this dissertation. The defining of the state-based UI Behaviour was then again made possible by an earlier general study of possible user interface relations, which provided us with an overview of these relations, as we later specified which relations to focus on. This led to defining state-based UI Behaviour as the kind of UI Behaviour that enables the execution of UI Behaviour corresponding to the state the UI is in.
With the state-based UI Behaviour defined, the solution mentioned in this dissertation was explained on a conceptual level (as we have seen in section 5.1), which provided us with a summary of separate components that needed to be taken into account in order for the entire stated problem to be solved, regardless of the environment or language it was implemented in. These components were the following.

First, we represented the UI Behaviour in a way that it was separated from other concerns in our application. This was achieved through the use of statecharts, as the UI Behaviour for a particular UI was expressed in a statechart, in which none of the other concerns, nor parts of the application were expressed.

Secondly, we created an application-level equivalent for this representation of the UI Behaviour, as we wanted an executable version of the UI Behaviour (which is described in the statechart) to be used in our application. For this reason, the statechart holding the UI Behaviour was transformed to a number of application-level statements, during which the separation between UI Behaviour and other concerns was maintained.

Finally, as we wanted to use the UI Behaviour in the application, we also coupled this UI Behaviour to our application, so that the appropriate UI Behaviour was addressed at the appropriate time. This was achieved through the use of aspects which held information about the UI Behaviour. These aspects were then woven on the application on the appropriate places, which allowed the application to call on the UI Behaviour whenever it was needed.

We integrated these three separate components into both a conceptual solution and a practical implementation of this solution, which was based on statecharts, CoBro and Carma (chapter 4.2). In this practical implementation, support for following our solution for separating the UI Behaviour was provided to the developer, as it automatically stepped through the remaining part of the presented solution when a fixed number of

first steps was performed by the developer and the appropriate input was given. It was also used later on in the validation of our approach.

Furthermore it was shown that using our solution disentangled state-based UI Behaviour from other concerns in the application and its UI as much as possible, and left the developer with an operational, normal functioning application, which could be implemented without the troubles of entangled UI Behaviour. Furthermore, by using our solution, the application and the corresponding UI Behaviour became more adaptable and extensibility of the UI Behaviour was supported. The solution also ensured that the developer was supported as much as possible.

We have also validated the functioning of the solution and its supporting system through the use of an experiment, being the advanced calculator case study. This experiment was deliberately constructed in such a way that it could be used to test the benefits (section 5.3.1) that were claimed to be coupled to the solution, and especially to validate the disentanglement of the UI Behaviour. Because of this, the experiment contained state-based UI Behaviour which was entangled with and scattered over an application and its UI, along with the possibility to extend a previously drawn statechart.
By running this experiment, it was shown that the given solution provided the wanted results (a creation of a working application with the state-based UI Behaviour disentangled from it) and that the benefits were recognized during or after the execution of our solution.

Furthermore, the relevance of the conducted research and the trail of thought used in this dissertation was also positioned with regard to the work of other researchers that was relevant within the context of this dissertation.

8.2 Future research

As we have shown throughout this dissertation that we made a supporting system for the disentanglement of UI Behaviour through the use of building blocks like statecharts, CoBro and Carma, we also noticed that further research and work related to these building blocks is needed.

For instance, for the statecharts, it is not certain that statecharts can express every possible kind of UI Behaviour that one can think of. As we set boundaries for the use of statecharts in this dissertation by only allowing to express one kind of UI Behaviour (state-based UI Behaviour) in them and not allowing to use more advanced parts of statecharts such as history-states, we limit the use of statecharts to the context of this dissertation. However, we are interested in seeing if these limitations cause problems for developers that need another kind of UI Behaviour and if our system can still be used to disentangle this. Furthermore, we find it interesting to investigate how much

of statecharts is needed to express all kinds of UI Behaviour or even other concerns, or if in particular cases adaptations are useful or even necessary (e.g. by extending them, as is done with Dcharts [11], or replacing them (for example by PetriNets [43])).
As the current solution only handles state-based UI Behaviour, we expect that more and more advanced parts of statecharts will be needed when they are used to express other kinds of UI Behaviour or even other concerns. In some cases, extensions or alternatives for statecharts may be needed. For example when we are trying to disentangle UI Visualization, alternatives such as Interaction Object Graphs (IOGs) [8] [9] or Concurrent Task Trees (CTTs) [34] could be used, as these techniques are aimed more at handling this visualization than statecharts.

Furthermore, we only use Cobro in this dissertation for its concept-centric approach and graphical support. However, CoBro can do more than what we use it for, and thus we should investigate if there are other useful purposes which can be added to our solution, using the power of Cobro. The use of CoBro can consequently also be seen in the same way as a benefit, as it leaves the path open for more improvements, which can make use of the power of CoBro.
A possible improvement by using CoBro is the enabling of runtime changes, as they are not supported for the moment, as mentioned in section 5.3.2. In that section, we proposed a solution for this shortcoming by moving all runtime-related behaviour to CoBro, where it can be edited at runtime. However, further research about the feasibility of this solution and the necessary updates to the supporting system to match this solution is needed.
Furthermore, we want to investigate the possibilities of using CoBro for using domain knowledge in our solution. We would use this domain knowledge to hold important data about the disentangled concern, in the case of UI Behaviour for example to document the meaning of states and transitions, to document the eventual UI Behaviour code that is woven in, or to keep track of which versions of the UI Behaviour are used (e.g. to make sure that incompatible versions are not mixed).

Thirdly, we mention that there is most certainly more research needed in the field of Carma. As we are not using Carma to the fullest in the current solution, we want to evolve to a solution that uses a pure aspect-oriented approach (as this avoids the shortcomings as described in section 5.3.2). However, the impacts of this aspect-oriented approach on the rest of our solution should be further investigated, as it is possible that using the full power of Carma causes trouble for our statemachine or another part of the system. We then also want to investigate how this pure aspect-oriented approach can possibly be used to disentangle other UI concerns, such as application interactions.
Apart from using Carma to improve our solution such that it uses a pure aspect-oriented approach (which is concretely described in section 5.3.2), we think that it is interesting to investigate how Carma can be used further for the separation of other kinds of UI Behaviour or even other concerns. The availability of logic programming in Carma can be possibly used in the separation of, for instance, automatic layout of a UI, like

for example constraint solving is used in Deuce (as mentioned in section 4.2.4).

As this dissertation fits in the research conducted regarding Deuce, we find it interesting to investigate if and how our solution for disentangling the UI Behaviour has an impact on the other UI concerns (visualization and application interactions) and their disentanglement. We also want to look closer into to what extent our solution can be reused conceptually to solve the disentangling of the other UI concerns. We may also question if our solution is still feasible within the Deuce framework, as not alone UI Behaviour is being disentangled in Deuce.

Furthermore, it needs further examining to what extent our solution is scalable to real-life applications. Hereby problems may rise in the field of speed, as our system may slow down certain applications since everything has to pass through one statemachine or it may take too long to initially automatically create the statemachine and aspects and weave those aspects. The first problem can be possibly resolved by multiplying the number of (synchronized) statemachines, the second by initially splitting up the creation of the statemachine and the aspects into smaller parts and creating them later on in an ad hoc way.

Finally, we want to investigate further if certain concerns can be fully separated from other concerns, and in what cases and for what reasons only a partial separation is feasible. This helps us to determine whether a separation is feasible whenever it is tried to separate a certain concern from other concerns, and if it is not, for which concerns the separation holds and for which it doesn't.

8.3 Conclusion

As we have tackled the problem of separating the UI Behaviour from other concerns during the outline of this dissertation, we notice that we still have a long way to go, as providing the developer with a supporting system for separating UI Behaviour leaves him/her in need for a system that gives support for separating other UI concerns too. Hence, this dissertation is not seen as an individual system, but as the beginning of a framework, which can provide support for disentangling all UI concerns, such as is aimed at in Deuce (see section 4.2.4).

During the outline of this dissertation, we have seen how state-based UI Behaviour was disentangled with the help of three main components: statecharts, CoBro and Carma, with which our experiences are now discussed.
Through the use of statecharts the developer was able to describe the UI Behaviour in this dissertation in a widely dispersed and commonly used way, hereby also leaving the possibility open to extend or replace these statecharts when needed in a later stage. Nevertheless, some more research about which degree of expressiveness of statecharts

is needed in which situations and about the further use of statecharts in the context of this dissertation needs to be conducted, as mentioned above. However, we found that during this dissertation, statecharts offered the appropriate solutions needed for our problem as well as some extra benefits (e.g. that it is commonly used by many developers), and however that there might be other alternatives possible, statecharts turned out to be an appropriate choice.
As we introduced CoBro into our solution, we found that CoBro had a lot more to offer than what we used of it. CoBro is a powerful tool, of which we are currently only using the graphical side (for the graphical representation of the UI Behaviour) and the concept-centric side (for storing the UI Behaviour), but of which we like to explore more possibilities in the future, as was described in the previous section. As CoBro can be used in a variety of contexts [1], it is possible that the (until now unused) power of CoBro can help us in further developing and improving our solution, or even be useful for solving the disentanglement of other concerns. For this dissertation however, CoBro offered the needed solutions, but also offered a large number of other properties that may be useful (such as the use of domain knowledge or the runtime adaptations) but which are not currently used. Consequently, CoBro was used successfully in our solution, as the needed functionality was provided and room for possible improvements was left open.
By using Carma in our solution, we introduced an artifact that was way to powerful for what it was eventually used, however that a number of plans to use it in a better way were lying on the shelves. Therefor it is important that these plans (such as improving our solution to a pure AOP-approach, as mentioned above) are executed and that further research is conducted about what more Carma has to offer that can be used in our solution or in the disentanglement of other concerns (as mentioned in section 8.2). We also note that extra features of Carma (such as logic programming) can be a stimulant to investigate the use of Carma in certain contexts further, as we mentioned in the previous section. In the context of this dissertation, Carma is currently not done justice, but in the spirit of evolving to an improved solution, we find that given its power, it leaves enough room for making as well AOP-related improvements as other (such as logic programming related) improvements to our solution.

Furthermore we have seen that a separation of state-based UI Behaviour is feasible, as shown in this dissertation and validated in chapter 6. This separation helps the developer with the implementation of the application, its UI and its UI Behaviour, as intended in this dissertation, but (as mentioned above) this still leaves us in need for a disentanglement of other kinds of UI Behaviour and also other UI concerns.

Finally, it is shown throughout this dissertation that the conceptual solution solved the problem that we wished to tackle and the solution has proven to work within the

[1] It was originally created as a tool to facilitate the active use of domain knowledge, but as shown in this dissertation, it can serve other purposes as well.

appropriate context, which was the goal of this dissertation.

Appendix A

Code for the Basic Calculator

A.1 Language Statements

```
a := State new: #StartState.
b := State new: #FirstOperandState.
c := State new: #OperatorState.
d := State new: #SecondOperandState.
e := State new: #NegativeOperandState.
f := State new: #ResultState.

a beginState. f endState.

a hasOnEntry: 'Numberbuttons enable'. a hasOnEntry: 'Operatorbuttons
disable'. a hasOnEntry: 'Equalbutton disable'.

b hasOnEntry: 'Operatorbuttons enable'.

c hasOnEntry: 'Operatorbuttons disable'. c hasOnEntry:
'Numberbuttons enable'.

d hasOnEntry: 'Equalbutton enable'.

f hasOnEntry: 'Numberbuttons disable'. f hasOnEntry:
'Operatorbuttons enable'. f hasOnEntry: 'Equalbutton disable'.

t1 := Transition new: #numberEntered. t1 hasEvent: 'numberEntered'.
a -> t1 -> b. c -> t1 -> d.

t2 := Transition new: #operatorEntered. t2 hasEvent:
'operatorEntered'. b -> t2 -> c. d -> t2 -> c. e -> t2 -> c. f -> t2
```

```
-> c.

t3 := Transition new: #switchNegative. t3 hasEvent:
'switchNegative'. b -> t3 -> e. d -> t3 -> e. f -> t3 -> e.

t4 := Transition new: #clear. t4 hasEvent: 'clear'. b -> t4 -> a. c
-> t4 -> a. d -> t4 -> a. e -> t4 -> a. f -> t4 -> a.

t5 := Transition new: #equals. t5 hasEvent: 'equals'. d -> t5 -> f.

t6 := Transition new: #switchNegDis. t6 hasEvent: 'switchNegative'.
t6 hasGuard: 'EqualbuttonDisabled'. e -> t6 -> b.

t7 := Transition new: #switchNegEn. t7 hasEvent: 'switchNegative'.
t7 hasGuard: 'EqualbuttonEnabled'. e -> t7 -> d.
```

A.2 Concept Network

```
(a := Concepts new: #{StartState})
    hasPreferredLabel: 'StartState'.
    a superconcept: Concepts.BeginState.
    a hasOnEntry: 'Numberbuttons enable'.
    a hasOnEntry: 'Operatorbuttons disable'.
    a hasOnEntry: 'Equalbutton disable'.
    a save.

(b := Concepts new: #{FirstOperandState})
    hasPreferredLabel: 'FirstOperandState'.
    b superconcept: Concepts.State.
    b hasOnEntry: 'Operatorbuttons enable'.
    b save.

(c := Concepts new: #{OperatorState})
    hasPreferredLabel: 'OperatorState'.
    c superconcept: Concepts.State.
    c hasOnEntry: 'Operatorbuttons disable'.
    c hasOnEntry: 'Numberbuttons enable'.
    c save.

(d := Concepts new: #{SecondOperandState})
    hasPreferredLabel: 'SecondOperandState'.
    d superconcept: Concepts.State.
```

```
    d hasOnEntry: 'Equalbutton enable'.
    d save.

(e := Concepts new: #{NegativeOperandState})
    hasPreferredLabel: 'NegativeOperandState'.
    e superconcept: Concepts.State.
    e save.

(f := Concepts new: #{ResultState})
    hasPreferredLabel: 'ResultState'.
    f superconcept: Concepts.EndState.
    f hasOnEntry: 'Numberbuttons disable'.
    f hasOnEntry: 'Operatorbuttons enable'.
    f hasOnEntry: 'Equalbutton disable'.
    f save.

(t11 := Concepts new: #{numberEntered1})
    hasPreferredLabel: 'numberEntered'.
    t11 superconcept: Concepts.hasTransition.
    t11 hasDestination: Concepts.StartState.
    t11 hasSource: Concepts.FirstOperandState.
    t11 hasTransitionEvent: 'numberEntered'.
    t11 save.

(t12 := Concepts new: #{numberEntered2})
    hasPreferredLabel: 'numberEntered'.
    t12 superconcept: Concepts.hasTransition.
    t12 hasDestination: Concepts.OperatorState.
    t12 hasSource: Concepts.SecondOperandState.
    t12 hasTransitionEvent: 'numberEntered'.
    t12 save.

(t21 := Concepts new: #{operatorEntered1})
    hasPreferredLabel: 'operatorEntered'.
    t21 superconcept: Concepts.hasTransition.
    t21 hasDestination: Concepts.FirstOperandState.
    t21 hasSource: Concepts.OperatorState.
    t21 hasTransitionEvent: 'operatorEntered'.
    t21 save.

(t22 := Concepts new: #{operatorEntered2})
    hasPreferredLabel: 'operatorEntered'.
    t22 superconcept: Concepts.hasTransition.
```

```
    t22 hasDestination: Concepts.SecondOperandState.
    t22 hasSource: Concepts.OperatorState.
    t22 hasTransitionEvent: 'operatorEntered'.
    t22 save.

(t23 := Concepts new: #{operatorEntered3})
    hasPreferredLabel: 'operatorEntered'.
    t23 superconcept: Concepts.hasTransition.
    t23 hasDestination: Concepts.NegativeOperandState.
    t23 hasSource: Concepts.OperatorState.
    t23 hasTransitionEvent: 'operatorEntered'.
    t23 save.

(t24 := Concepts new: #{operatorEntered4})
    hasPreferredLabel: 'operatorEntered'.
    t24 superconcept: Concepts.hasTransition.
    t24 hasDestination: Concepts.ResultState.
    t24 hasSource: Concepts.OperatorState.
    t24 hasTransitionEvent: 'operatorEntered'.
    t24 save.

(t31 := Concepts new: #{switchNegative1})
    hasPreferredLabel: 'switchNegative'.
    t31 superconcept: Concepts.hasTransition.
    t31 hasDestination: Concepts.FirstOperandState.
    t31 hasSource: Concepts.NegativeOperandState.
    t31 hasTransitionEvent: 'switchNegative'.
    t31 save.

(t32 := Concepts new: #{switchNegative2})
    hasPreferredLabel: 'switchNegative'.
    t32 superconcept: Concepts.hasTransition.
    t32 hasDestination: Concepts.SecondOperandState.
    t32 hasSource: Concepts.NegativeOperandState.
    t32 hasTransitionEvent: 'switchNegative'.
    t32 save.

(t33 := Concepts new: #{switchNegative3})
    hasPreferredLabel: 'switchNegative'.
    t33 superconcept: Concepts.hasTransition.
    t33 hasDestination: Concepts.ResultState.
    t33 hasSource: Concepts.NegativeOperandState.
    t33 hasTransitionEvent: 'switchNegative'.
```

```
    t33 save.

(t41 := Concepts new: #{clear1})
    hasPreferredLabel: 'clear'.
    t41 superconcept: Concepts.hasTransition.
    t41 hasDestination: Concepts.FirstOperandState.
    t41 hasSource: Concepts.StartState.
    t41 hasTransitionEvent: 'clear'.
    t41 save.

(t42 := Concepts new: #{clear2})
    hasPreferredLabel: 'clear'.
    t42 superconcept: Concepts.hasTransition.
    t42 hasDestination: Concepts.OperatorState.
    t42 hasSource: Concepts.StartState.
    t42 hasTransitionEvent: 'clear'.
    t42 save.

(t43 := Concepts new: #{clear3})
    hasPreferredLabel: 'clear'.
    t43 superconcept: Concepts.hasTransition.
    t43 hasDestination: Concepts.SecondOperandState.
    t43 hasSource: Concepts.StartState.
    t43 hasTransitionEvent: 'clear'.
    t43 save.

(t44 := Concepts new: #{clear4})
    hasPreferredLabel: 'clear'.
    t44 superconcept: Concepts.hasTransition.
    t44 hasDestination: Concepts.NegativeOperandState.
    t44 hasSource: Concepts.StartState.
    t44 hasTransitionEvent: 'clear'.
    t44 save.

(t45 := Concepts new: #{clear5})
    hasPreferredLabel: 'clear'.
    t45 superconcept: Concepts.hasTransition.
    t45 hasDestination: Concepts.ResultState.
    t45 hasSource: Concepts.StartState.
    t45 hasTransitionEvent: 'clear'.
    t45 save.

(t5 := Concepts new: #{equals})
```

```
    hasPreferredLabel: 'equals'.
    t5 superconcept: Concepts.hasTransition.
    t5 hasDestination: Concepts.SecondOperandState.
    t5 hasSource: Concepts.ResultState.
    t5 hasTransitionEvent: 'equals'.
    t5 save.

(t6 := Concepts new: #{switchNegDis})
    hasPreferredLabel: 'switchNegDis'.
    t6 superconcept: Concepts.hasTransition.
    t6 hasDestination: Concepts.NegativeOperandState.
    t6 hasSource: Concepts.FirstOperandState.
    t6 hasTransitionEvent: 'switchNegative'.
    t6 hasTransitionGuard: 'EqualbuttonDisabled'.
    t6 save.

(t7 := Concepts new: #{switchNegEn})
    hasPreferredLabel: 'switchNegEn'.
    t7 superconcept: Concepts.hasTransition.
    t7 hasDestination: Concepts.NegativeOperandState.
    t7 hasSource: Concepts.SecondOperandState.
    t7 hasTransitionEvent: 'switchNegative'.
    t7 hasTransitionGuard: 'EqualbuttonEnabled'.
    t7 save.
```

A.3 Aspects

```
Andrew.TestAspects defineAspect: #StatechartAspect
    superAspect: #{Andrew.Aspect}
    ofEach: #TestCalculator
    instanceVariableNames: ''
    aspectVariableNames: ''
    category: 'Calculator'.

before ?jp matching {reception(?jp, #numberEntered)} do
TestCalculator statechartInstance numberEntered.

before ?jp matching {reception(?jp, #operatorEntered)} do
TestCalculator statechartInstance operatorEntered.

before ?jp matching {reception(?jp, #switchNegative)} do
TestCalculator statechartInstance switchNegative.
```

```
before ?jp matching {reception(?jp, #clear)} do
TestCalculator statechartInstance clear.

before ?jp matching {reception(?jp, #equals)} do
TestCalculator statechartInstance equals.

before ?jp matching {reception(?jp, #switchNegative)} do
((SOULEvaluator eval: 'if EqualbuttonDisabled') allResults)
    ifTrue:[TestCalculator statechartInstance switchNegDis].

before ?jp matching {reception(?jp, #switchNegative)} do
((SOULEvaluator eval: 'if EqualbuttonEnabled') allResults)
    ifTrue:[TestCalculator statechartInstance switchNegEn].
```

Appendix B

Code for the Advanced Calculator

B.1 Language Statements

```
a := State new: #AStartState.
b := State new: #AFirstOperandState.
c := State new: #AOperatorState.
d := State new: #ASecondOperandState.
e := State new: #ANegativeOperandState.
f := State new: #AResultState.
g := State new: #TicketOnState.
h := State new: #TicketOffState.
i := State new: #ArabicState.
j := State new: #RomanState.

a hasOnEntry: 'Numberbuttons enable'.
a hasOnEntry: 'Operatorbuttons disable'.
a hasOnEntry: 'Equalbutton disable'.
a hasOnEntry: 'AOperatorbuttons visible'.

b hasOnEntry: 'Operatorbuttons enable'.
b hasOnEntry: 'writeToTicket'.

c hasOnEntry: 'Operatorbuttons disable'.
c hasOnEntry: 'Numberbuttons enable'.
c hasOnEntry: 'writeToTicket'.

d hasOnEntry: 'Equalbutton enable'.
d hasOnEntry: 'writeToTicket'.

f hasOnEntry: 'Equalbutton disable'.
```

```
f hasOnEntry: 'Operatorbuttons enable'.
f hasOnEntry: 'Numberbuttons disable'.

g hasOnEntry: 'ticketOn'.
g hasOnEntry: 'return'.

h hasOnEntry: 'ticketOff'.
h hasOnEntry: 'return'.

i hasOnEntry: 'arabicNumbers'.
i hasOnEntry: 'return'.

j hasOnEntry: 'romanNumbers'.
j hasOnEntry: 'return'.

t1 := Transition new: #numberEntered.
t1 hasEvent: 'numberEntered'.
a -> t1 -> b. c -> t1 -> d.

t2 := Transition new: #operatorEntered.
t2 hasEvent: 'operatorEntered'.
b -> t2 -> c. d -> t2 -> c. e -> t2 -> c. f -> t2 -> c.

t3 := Transition new: #switchNegative.
t3 hasEvent: 'switchNegative'.
b -> t3 -> e. d -> t3 -> e. f -> t3 -> e.

t4 := Transition new: #clear.
t4 hasEvent: 'clear'.
b -> t4 -> a. c -> t4 -> a. d -> t4 -> a. e -> t4 -> a. f -> t4 -> a.

t5 := Transition new: #equals.
t5 hasEvent: 'equals'.
d -> t5 -> f.

t6 := Transition new: #switchNegDis.
t6 hasEvent: 'switchNegative'.
t6 hasGuard: 'EqualbuttonDisabled'.
e -> t6 -> b.

t7 := Transition new: #switchNegEn.
t7 hasEvent: 'switchNegative'.
t7 hasGuard: 'EqualbuttonEnabled'.
```

```
e -> t7 -> d.

t8 := Transition new: #advancedOperatorEntered.
t8 hasEvent: 'advancedOperatorEntered'.
b -> t8 -> f. e -> t8 -> f. f -> t8 -> f.

t9 := Transition new: #backspace.
t9 hasEvent: 'backspace'.
b -> t9 -> b. d -> t9 -> d.

t10 := Transition new: #switchOn.
t10 hasEvent: 'switchOn'.
a -> t10 -> g. b -> t10 -> g. c -> t10 -> g. d -> t10 -> g.
e -> t10 -> g. f -> t10 -> g.

t11 := Transition new: #switchOff.
t11 hasEvent: 'switchOn'.
a -> t11 -> h. b -> t11 -> h. c -> t11 -> h. d -> t11 -> h.
e -> t11 -> h. f -> t11 -> h.

t12 := Transition new: #switchRoman.
t12 hasEvent: 'switchRoman'.
t12 hasGuard: 'oldEnough(user)'.
a -> t12 -> j. b -> t12 -> j. c -> t12 -> j. d -> t12 -> j.
e -> t12 -> j. f -> t12 -> j.

t13 := Transition new: #switchArabic.
t13 hasEvent: 'switchArabic'.
a -> t13 -> i. b -> t13 -> i. c -> t13 -> i. d -> t13 -> i.
e -> t13 -> i. f -> t13 -> i.

t14 := Transition new: #return.
t14 hasEvent: 'return'.
g -> t14 -> a. h -> t14 -> a. i -> t14 -> a. j -> t14 -> a.
```

B.2 Concept Network

```
(a := Concepts new: #{AStartState})
    hasPreferredLabel: 'AStartState'.
    a superconcept: Concepts.State.
    a hasOnEntry: 'Numberbuttons enable'.
    a hasOnEntry: 'Operatorbuttons disable'.
```

```
    a hasOnEntry: 'Equalbutton disable'.
    a hasOnEntry: 'AOperatorbuttons visible'.
    a save.

(b := Concepts new: #{AFirstOperandState})
    hasPreferredLabel: 'AFirstOperandState'.
    b superconcept: Concepts.State.
    b hasOnEntry: 'Operatorbuttons enable'.
    b hasOnEntry: 'writeToTicket'.
    b save.

(c := Concepts new: #{AOperatorState})
    hasPreferredLabel: 'AOperatorState'.
    c superconcept: Concepts.State.
    c hasOnEntry: 'Operatorbuttons disable'.
    c hasOnEntry: 'Numberbuttons enable'.
    c hasOnEntry: 'writeToTicket'.
    c save.

(d := Concepts new: #{ASecondOperandState})
    hasPreferredLabel: 'ASecondOperandState'.
    d superconcept: Concepts.State.
    d hasOnEntry: 'Equalbutton enable'.
    d hasOnEntry: 'writeToTicket'.
    d save.

(e := Concepts new: #{ANegativeOperandState})
    hasPreferredLabel: 'ANegativeOperandState'.
    e superconcept: Concepts.State.
    e save.

(f := Concepts new: #{AResultState})
    hasPreferredLabel: 'AResultState'.
    f superconcept: Concepts.State.
    f hasOnEntry: 'Numberbuttons disable'.
    f hasOnEntry: 'Operatorbuttons enable'.
    f hasOnEntry: 'Equalbutton disable'.
    f save.

(g := Concepts new: #{TicketOnState})
    hasPreferredLabel: 'TicketOnState'.
    g superconcept: Concepts.State.
    g hasOnEntry: 'ticketOn'.
```

```
    g hasOnEntry: 'return'.
    g save.

(h := Concepts new: #{TicketOffState})
    hasPreferredLabel: 'TicketOffState'.
    h superconcept: Concepts.State.
    h hasOnEntry: 'ticketOff'.
    h hasOnEntry: 'return'.
    h save.

(i := Concepts new: #{ArabicState})
    hasPreferredLabel: 'ArabicState'.
    i superconcept: Concepts.State.
    i hasOnEntry: 'arabicNumbers'.
    i hasOnEntry: 'return'.
    i save.

(j := Concepts new: #{RomanState})
    hasPreferredLabel: 'RomanState'.
    j superconcept: Concepts.State.
    j hasOnEntry: 'romanNumbers'.
    j hasOnEntry: 'return'.
    j save.

(t11 := Concepts new: #{numberEntered1})
    hasPreferredLabel: 'numberEntered'.
    t11 superconcept: Concepts.hasTransition.
    t11 hasDestination: Concepts.AStartState.
    t11 hasSource: Concepts.AFirstOperandState.
    t11 hasTransitionEvent: 'numberEntered'.
    t11 save.

(t12 := Concepts new: #{numberEntered2})
    hasPreferredLabel: 'numberEntered'.
    t12 superconcept: Concepts.hasTransition.
    t12 hasDestination: Concepts.AOperatorState.
    t12 hasSource: Concepts.ASecondOperandState.
    t12 hasTransitionEvent: 'numberEntered'.
    t12 save.

(t21 := Concepts new: #{operatorEntered1})
    hasPreferredLabel: 'operatorEntered'.
    t21 superconcept: Concepts.hasTransition.
```

```
    t21 hasDestination: Concepts.AFirstOperandState.
    t21 hasSource: Concepts.AOperatorState.
    t21 hasTransitionEvent: 'operatorEntered'.
    t21 save.

(t22 := Concepts new: #{operatorEntered2})
    hasPreferredLabel: 'operatorEntered'.
    t22 superconcept: Concepts.hasTransition.
    t22 hasDestination: Concepts.ASecondOperandState.
    t22 hasSource: Concepts.AOperatorState.
    t22 hasTransitionEvent: 'operatorEntered'.
    t22 save.

(t23 := Concepts new: #{operatorEntered3})
    hasPreferredLabel: 'operatorEntered'.
    t23 superconcept: Concepts.hasTransition.
    t23 hasDestination: Concepts.ANegativeOperandState.
    t23 hasSource: Concepts.AOperatorState.
    t23 hasTransitionEvent: 'operatorEntered'.
    t23 save.

(t24 := Concepts new: #{operatorEntered4})
    hasPreferredLabel: 'operatorEntered'.
    t24 superconcept: Concepts.hasTransition.
    t24 hasDestination: Concepts.AResultState.
    t24 hasSource: Concepts.AOperatorState.
    t24 hasTransitionEvent: 'operatorEntered'.
    t24 save.

(t31 := Concepts new: #{switchNegative1})
    hasPreferredLabel: 'switchNegative'.
    t31 superconcept: Concepts.hasTransition.
    t31 hasDestination: Concepts.AFirstOperandState.
    t31 hasSource: Concepts.ANegativeOperandState.
    t31 hasTransitionEvent: 'switchNegative'.
    t31 save.

(t32 := Concepts new: #{switchNegative2})
    hasPreferredLabel: 'switchNegative'.
    t32 superconcept: Concepts.hasTransition.
    t32 hasDestination: Concepts.ASecondOperandState.
    t32 hasSource: Concepts.ANegativeOperandState.
    t32 hasTransitionEvent: 'switchNegative'.
```

```
    t32 save.

(t33 := Concepts new: #{switchNegative3})
    hasPreferredLabel: 'switchNegative'.
    t33 superconcept: Concepts.hasTransition.
    t33 hasDestination: Concepts.AResultState.
    t33 hasSource: Concepts.ANegativeOperandState.
    t33 hasTransitionEvent: 'switchNegative'.
    t33 save.

(t41 := Concepts new: #{clear1})
    hasPreferredLabel: 'clear'.
    t41 superconcept: Concepts.hasTransition.
    t41 hasDestination: Concepts.AFirstOperandState.
    t41 hasSource: Concepts.AStartState.
    t41 hasTransitionEvent: 'clear'.
    t41 save.

(t42 := Concepts new: #{clear2})
    hasPreferredLabel: 'clear'.
    t42 superconcept: Concepts.hasTransition.
    t42 hasDestination: Concepts.AOperatorState.
    t42 hasSource: Concepts.AStartState.
    t42 hasTransitionEvent: 'clear'.
    t42 save.

(t43 := Concepts new: #{clear3})
    hasPreferredLabel: 'clear'.
    t43 superconcept: Concepts.hasTransition.
    t43 hasDestination: Concepts.ASecondOperandState.
    t43 hasSource: Concepts.AStartState.
    t43 hasTransitionEvent: 'clear'.
    t43 save.

(t44 := Concepts new: #{clear4})
    hasPreferredLabel: 'clear'.
    t44 superconcept: Concepts.hasTransition.
    t44 hasDestination: Concepts.ANegativeOperandState.
    t44 hasSource: Concepts.AStartState.
    t44 hasTransitionEvent: 'clear'.
    t44 save.

(t45 := Concepts new: #{clear5})
```

```
    hasPreferredLabel: 'clear'.
    t45 superconcept: Concepts.hasTransition.
    t45 hasDestination: Concepts.AResultState.
    t45 hasSource: Concepts.AStartState.
    t45 hasTransitionEvent: 'clear'.
    t45 save.

(t5 := Concepts new: #{equals})
    hasPreferredLabel: 'equals'.
    t5 superconcept: Concepts.hasTransition.
    t5 hasDestination: Concepts.ASecondOperandState.
    t5 hasSource: Concepts.AResultState.
    t5 hasTransitionEvent: 'equals'.
    t5 save.

(t6 := Concepts new: #{switchNegDis})
    hasPreferredLabel: 'switchNegDis'.
    t6 superconcept: Concepts.hasTransition.
    t6 hasDestination: Concepts.ANegativeOperandState.
    t6 hasSource: Concepts.AFirstOperandState.
    t6 hasTransitionEvent: 'switchNegative'.
    t6 hasTransitionGuard: 'EqualbuttonDisabled'.
    t6 save.

(t7 := Concepts new: #{switchNegEn})
    hasPreferredLabel: 'switchNegEn'.
    t7 superconcept: Concepts.hasTransition.
    t7 hasDestination: Concepts.ANegativeOperandState.
    t7 hasSource: Concepts.ASecondOperandState.
    t7 hasTransitionEvent: 'switchNegative'.
    t7 hasTransitionGuard: 'EqualbuttonEnabled'.
    t7 save.

(t81 := Concepts new: #{advancedOperatorEntered1})
    hasPreferredLabel: 'advancedOperatorEntered'.
    t81 superconcept: Concepts.hasTransition.
    t81 hasDestination: Concepts.AFirstOperandState.
    t81 hasSource: Concepts.AResultState.
    t81 hasTransitionEvent: 'advancedOperatorEntered'.
    t81 save.

(t82 := Concepts new: #{advancedOperatorEntered2})
    hasPreferredLabel: 'advancedOperatorEntered'.
```

```
    t82 superconcept: Concepts.hasTransition.
    t82 hasDestination: Concepts.ANegativeOperandState.
    t82 hasSource: Concepts.AResultState.
    t82 hasTransitionEvent: 'advancedOperatorEntered'.
    t82 save.

(t83 := Concepts new: #{advancedOperatorEntered3})
    hasPreferredLabel: 'advancedOperatorEntered'.
    t83 superconcept: Concepts.hasTransition.
    t83 hasDestination: Concepts.AResultState.
    t83 hasSource: Concepts.AResultState.
    t83 hasTransitionEvent: 'advancedOperatorEntered'.
    t83 save.

(t91 := Concepts new: #{backspace1})
    hasPreferredLabel: 'backspace'.
    t91 superconcept: Concepts.hasTransition.
    t91 hasDestination: Concepts.AFirstOperandState.
    t91 hasSource: Concepts.AFirstOperandState.
    t91 hasTransitionEvent: 'backspace'.
    t91 save.

(t92 := Concepts new: #{backspace2})
    hasPreferredLabel: 'backspace'.
    t92 superconcept: Concepts.hasTransition.
    t92 hasDestination: Concepts.ASecondOperandState.
    t92 hasSource: Concepts.ASecondOperandState.
    t92 hasTransitionEvent: 'backspace'.
    t92 save.

(t101 := Concepts new: #{switchOn1})
    hasPreferredLabel: 'switchOn'.
    t101 superconcept: Concepts.hasTransition.
    t101 hasDestination: Concepts.AStartState.
    t101 hasSource: Concepts.TicketOnState.
    t101 hasTransitionEvent: 'switchOn'.
    t101 save.

(t102 := Concepts new: #{switchOn2})
    hasPreferredLabel: 'switchOn'.
    t102 superconcept: Concepts.hasTransition.
    t102 hasDestination: Concepts.AFirstOperandState.
    t102 hasSource: Concepts.TicketOnState.
```

```
    t102 hasTransitionEvent: 'switchOn'.
    t102 save.

(t103 := Concepts new: #{switchOn3})
    hasPreferredLabel: 'switchOn'.
    t103 superconcept: Concepts.hasTransition.
    t103 hasDestination: Concepts.AOperatorState.
    t103 hasSource: Concepts.TicketOnState.
    t103 hasTransitionEvent: 'switchOn'.
    t103 save.

(t104 := Concepts new: #{switchOn4})
    hasPreferredLabel: 'switchOn'.
    t104 superconcept: Concepts.hasTransition.
    t104 hasDestination: Concepts.ASecondOperandState.
    t104 hasSource: Concepts.TicketOnState.
    t104 hasTransitionEvent: 'switchOn'.
    t104 save.

(t105 := Concepts new: #{switchOn5})
    hasPreferredLabel: 'switchOn'.
    t105 superconcept: Concepts.hasTransition.
    t105 hasDestination: Concepts.ANegativeOperandState.
    t105 hasSource: Concepts.TicketOnState.
    t105 hasTransitionEvent: 'switchOn'.
    t105 save.

(t106 := Concepts new: #{switchOn6})
    hasPreferredLabel: 'switchOn'.
    t106 superconcept: Concepts.hasTransition.
    t106 hasDestination: Concepts.AResultState.
    t106 hasSource: Concepts.TicketOnState.
    t106 hasTransitionEvent: 'switchOn'.
    t106 save.

(t111 := Concepts new: #{switchOff1})
    hasPreferredLabel: 'switchOff'.
    t111 superconcept: Concepts.hasTransition.
    t111 hasDestination: Concepts.AStartState.
    t111 hasSource: Concepts.TicketOffState.
    t111 hasTransitionEvent: 'switchOff'.
    t111 save.
```

```
(t112 := Concepts new: #{switchOff2})
    hasPreferredLabel: 'switchOff'.
    t112 superconcept: Concepts.hasTransition.
    t112 hasDestination: Concepts.AFirstOperandState.
    t112 hasSource: Concepts.TicketOffState.
    t112 hasTransitionEvent: 'switchOff'.
    t112 save.

(t113 := Concepts new: #{switchOff3})
    hasPreferredLabel: 'switchOff'.
    t113 superconcept: Concepts.hasTransition.
    t113 hasDestination: Concepts.AOperatorState.
    t113 hasSource: Concepts.TicketOffState.
    t113 hasTransitionEvent: 'switchOff'.
    t113 save.

(t114 := Concepts new: #{switchOff4})
    hasPreferredLabel: 'switchOff'.
    t114 superconcept: Concepts.hasTransition.
    t114 hasDestination: Concepts.ASecondOperandState.
    t114 hasSource: Concepts.TicketOffState.
    t114 hasTransitionEvent: 'switchOff'.
    t114 save.

(t115 := Concepts new: #{switchOff5})
    hasPreferredLabel: 'switchOff'.
    t115 superconcept: Concepts.hasTransition.
    t115 hasDestination: Concepts.ANegativeOperandState.
    t115 hasSource: Concepts.TicketOffState.
    t115 hasTransitionEvent: 'switchOff'.
    t115 save.

(t116 := Concepts new: #{switchOff6})
    hasPreferredLabel: 'switchOff'.
    t116 superconcept: Concepts.hasTransition.
    t116 hasDestination: Concepts.AResultState.
    t116 hasSource: Concepts.TicketOffState.
    t116 hasTransitionEvent: 'switchOff'.
    t116 save.

(t121 := Concepts new: #{switchRoman1})
    hasPreferredLabel: 'switchRoman'.
    t121 superconcept: Concepts.hasTransition.
```

```
    t121 hasDestination: Concepts.AStartState.
    t121 hasSource: Concepts.RomanState.
    t121 hasTransitionEvent: 'switchRoman'.
    t121 hasTransitionGuard: 'oldEnough(user)'.
    t121 save.

(t122 := Concepts new: #{switchRoman2})
    hasPreferredLabel: 'switchRoman'.
    t122 superconcept: Concepts.hasTransition.
    t122 hasDestination: Concepts.AFirstOperandState.
    t122 hasSource: Concepts.RomanState.
    t122 hasTransitionEvent: 'switchRoman'.
    t122 hasTransitionGuard: 'oldEnough(user)'.
    t122 save.

(t123 := Concepts new: #{switchRoman3})
    hasPreferredLabel: 'switchRoman'.
    t123 superconcept: Concepts.hasTransition.
    t123 hasDestination: Concepts.AOperatorState.
    t123 hasSource: Concepts.RomanState.
    t123 hasTransitionEvent: 'switchRoman'.
    t123 hasTransitionGuard: 'oldEnough(user)'.
    t123 save.

(t124 := Concepts new: #{switchRoman4})
    hasPreferredLabel: 'switchRoman'.
    t124 superconcept: Concepts.hasTransition.
    t124 hasDestination: Concepts.ASecondOperandState.
    t124 hasSource: Concepts.RomanState.
    t124 hasTransitionEvent: 'switchRoman'.
    t124 hasTransitionGuard: 'oldEnough(user)'.
    t124 save.

(t125 := Concepts new: #{switchRoman5})
    hasPreferredLabel: 'switchRoman'.
    t125 superconcept: Concepts.hasTransition.
    t125 hasDestination: Concepts.ANegativeOperandState.
    t125 hasSource: Concepts.RomanState.
    t125 hasTransitionEvent: 'switchRoman'.
    t125 hasTransitionGuard: 'oldEnough(user)'.
    t125 save.

(t126 := Concepts new: #{switchRoman6})
```

```
    hasPreferredLabel: 'switchRoman'.
    t126 superconcept: Concepts.hasTransition.
    t126 hasDestination: Concepts.AResultState.
    t126 hasSource: Concepts.RomanState.
    t126 hasTransitionEvent: 'switchRoman'.
    t126 hasTransitionGuard: 'oldEnough(user)'.
    t126 save.

(t131 := Concepts new: #{switchArabic1})
    hasPreferredLabel: 'switchArabic'.
    t131 superconcept: Concepts.hasTransition.
    t131 hasDestination: Concepts.AStartState.
    t131 hasSource: Concepts.ArabicState.
    t131 hasTransitionEvent: 'switchArabic'.
    t131 save.

(t132 := Concepts new: #{switchArabic2})
    hasPreferredLabel: 'switchArabic'.
    t132 superconcept: Concepts.hasTransition.
    t132 hasDestination: Concepts.AFirstOperandState.
    t132 hasSource: Concepts.ArabicState.
    t132 hasTransitionEvent: 'switchArabic'.
    t132 save.

(t133 := Concepts new: #{switchArabic3})
    hasPreferredLabel: 'switchArabic'.
    t133 superconcept: Concepts.hasTransition.
    t133 hasDestination: Concepts.AOperatorState.
    t133 hasSource: Concepts.ArabicState.
    t133 hasTransitionEvent: 'switchArabic'.
    t133 save.

(t134 := Concepts new: #{switchArabic4})
    hasPreferredLabel: 'switchArabic'.
    t134 superconcept: Concepts.hasTransition.
    t134 hasDestination: Concepts.ASecondOperandState.
    t134 hasSource: Concepts.ArabicState.
    t134 hasTransitionEvent: 'switchArabic'.
    t134 save.

(t135 := Concepts new: #{switchArabic5})
    hasPreferredLabel: 'switchArabic'.
    t135 superconcept: Concepts.hasTransition.
```

```
    t135 hasDestination: Concepts.ANegativeOperandState.
    t135 hasSource: Concepts.ArabicState.
    t135 hasTransitionEvent: 'switchArabic'.
    t135 save.

(t136 := Concepts new: #{switchArabic6})
    hasPreferredLabel: 'switchArabic'.
    t136 superconcept: Concepts.hasTransition.
    t136 hasDestination: Concepts.AResultState.
    t136 hasSource: Concepts.ArabicState.
    t136 hasTransitionEvent: 'switchArabic'.
    t136 save.

(t141 := Concepts new: #{return1})
    hasPreferredLabel: 'return'.
    t141 superconcept: Concepts.hasTransition.
    t141 hasDestination: Concepts.TicketOnState.
    t141 hasSource: Concepts.AStartState.
    t141 hasTransitionEvent: 'return'.
    t141 save.

(t142 := Concepts new: #{return2})
    hasPreferredLabel: 'return'.
    t142 superconcept: Concepts.hasTransition.
    t142 hasDestination: Concepts.TicketOffState.
    t142 hasSource: Concepts.AStartState.
    t142 hasTransitionEvent: 'return'.
    t142 save.

(t143 := Concepts new: #{return3})
    hasPreferredLabel: 'return'.
    t143 superconcept: Concepts.hasTransition.
    t143 hasDestination: Concepts.RomanState.
    t143 hasSource: Concepts.AStartState.
    t143 hasTransitionEvent: 'return'.
    t143 save.

(t144 := Concepts new: #{return4})
    hasPreferredLabel: 'return'.
    t144 superconcept: Concepts.hasTransition.
    t144 hasDestination: Concepts.ArabicState.
    t144 hasSource: Concepts.AStartState.
    t144 hasTransitionEvent: 'return'.
```

```
    t144 save.
```

B.3 Aspects

```
Andrew.TestAspects defineAspect: #CalculatorAspect
    superAspect: #{Andrew.Aspect}
    ofEach: #AdvancedCalculator
    instanceVariableNames: ''
    aspectVariableNames: ''
    category: 'Calculator'.

before ?jp matching {reception(?jp, #numberEntered)} do
AdvancedCalculator statemachineInstance numberEntered.

before ?jp matching {reception(?jp, #operatorEntered)} do
AdvancedCalculator statemachineInstance operatorEntered.

before ?jp matching {reception(?jp, #switchNegative)} do
AdvancedCalculator statemachineInstance switchNegative.

before ?jp matching {reception(?jp, #clear)} do
AdvancedCalculator statemachineInstance clear.

before ?jp matching {reception(?jp, #equals)} do
AdvancedCalculator statemachineInstance equals.

before ?jp matching {reception(?jp, #switchNegative)} do
((SOULEvaluator eval: 'if EqualbuttonDisabled') allResults)
    ifTrue:[AdvancedCalculator statemachineInstance switchNegDis].

before ?jp matching {reception(?jp, #switchNegative)} do
((SOULEvaluator eval: 'if EqualbuttonEnabled') allResults)
    ifTrue:[AdvancedCalculator statemachineInstance switchNegEn].

before ?jp matching {reception(?jp, #advancedOperatorEntered)} do
AdvancedCalculator statemachineInstance advancedOperatorEntered.

before ?jp matching {reception(?jp, #backspace)} do
AdvancedCalculator statemachineInstance backspace.

before ?jp matching {reception(?jp, #switchOn)} do
AdvancedCalculator statemachineInstance switchOn.
```

```
before ?jp matching {reception(?jp, #switchOff)} do
AdvancedCalculator statemachineInstance switchOff.

before ?jp matching {reception(?jp, #switchArabic)} do
AdvancedCalculator statemachineInstance switchArabic.

before ?jp matching {reception(?jp, #return)} do
AdvancedCalculator statemachineInstance return.

before ?jp matching {reception(?jp, #switchRoman)} do
((SOULEvaluator eval: 'if oldEnough(AdvancedCalculator user)')
allResults)
    ifTrue:[AdvancedCalculator statemachineInstance switchRoman].
```

Bibliography

[1] D Anderson, B O'Byrne, (2003). *Lean interaction design and implementation: using statecharts with feature driven development*, Proceedings of ForUse 2003.

[2] AOP on Wikipedia, *http://en.wikipedia.org/wiki/Aspect-oriented_programming*.

[3] Atom, *http://atom3.cs.mcgill.ca/*.

[4] S Borland, (2003). *Transforming statechart models to DEVS*, Master's thesis, School of Computer Science, McGill University, Montral, Canada, August 2003.

[5] J Brichau, T D'Hondt, (2005). *Aspect-Oriented Software Development (AOSD) - An Introduction.*

[6] A Concepcion, B Zeigler, (1988). *DEVS Formalism: A Framework for Hierarchical Model Development*, IEEE Transactions on Software Engineering, vol. 14, no. 2, pp. 228-241, Feb. 1988.

[7] Carma, *http://prog.vub.ac.be/ kgybels/Research/AOP.html*

[8] D Carr, (1995). *A compact graphical representation of user interface interaction objects*, University of Maryland, Doctorate's Thesis Dissertation, 1995.

[9] D Carr, (1997). *Interaction Object Graphs: An Executable Graphical Notation for Specifying User Interfaces*, Formal Method for Computer Human Interaction, 1997.

[10] J Carroll, D Long, (1989). *Theory of Finite Automata with an Introduction to Formal Languages*, Prentice Hall, Englewood Cliffs, 1989.

[11] Dcharts and Atom. *http://moncs.cs.mcgill.ca/people/astewa5/report.dtml* .

[12] D Deridder, T D'Hondt, (2005). *A concept-centric approach to software evolution - enabling open adaptive software development*, Ontologies as Software Engineering Artefacts, Vancouver, Canada, 2005. OOPSLA.

[13] D Deridder, (2006). *A Concept-Centric Environment for Software Evolution in an Agile context*, PhD Dissertation, VUB, Brussels, 2006.

[14] E Dijkstra, (1976). *A Discipline of programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.

[15] H Dobos, (2002). *Separable User Interface and Interaction Controls*, Master Thesis, University of Jyvaskyla, Novembre 2002.

[16] E Edmonds, (1992). *The Emergence of the Separable User Interface*, The Separable User Interface, Academic Press, pp. 5-18, 1992.

[17] E Edmonds, E McDaid, (1992). *An Architecture for Knowledge-based Front Ends*, The Separable User Interface, Academic Press, pp. 263-269, 1992.

[18] H Feng, (2004). *Dcharts, a Formalism for Modeling and Simulation Based Design of Reactive Software Systems*, Masters Thesis Dissertation, McGill University, Montreal, Canada.

[19] R Filman, T Elrad, S Clarke, (2005). *Engineering aspect-oriented systems*, Aspect-Oriented Software Development, p. 379-406, Addison-Wesley, 2005.

[20] M Fowler, (2003). *UML Distilled 3rd Edition*, Addison-Wesley, ISBN 0-321-19368-7.

[21] M Fowler, (2001). *Separating user interface code*, IEEE software, March/April 2001, p. 96-97, 2001.

[22] A Gill, (1962). *Introduction to the Theory of Finite-state Machines*, McGraw-Hill, 1962.

[23] S Goderis, (2005). *High-Level Declarative User Interfaces*, OOPSLA Companion Proceedings 2005.

[24] J Guttag, J Horning, (1980). *Formal Specification as a Design Tool*, Proceedings of the 7th Symposium on Programming Languages, 1980, pp. 251-261.

[25] K Gybels, (2001). *Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure*, Licentiate's Thesis Dissertation, VUB, Brussels.

[26] K Gybels, (2003). *SOUL and Smalltalk - Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis*, Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.

[27] K Gybels, (2002). *Using a logic language to express cross-cutting through dynamic joinpoints*, Proceedings of the Second German Workshop on Aspect-Oriented Software Development, Technical Report IAI-TR-2002-1 Universitt Bonn, 2002.

[28] D Harel, (1987). *Statecharts: A visual Formalism for Complex Systems*, Science of Computer Programming, 8(3), June 1987.

[29] D Harel, (1988). *On Visual Formalism*, Communications of the ACM, 31(5), May 1988.

[30] H Hartson, P Gray, (1992). *Temporal Aspects of Tasks in the User Action Notation*, Human-Computer Interaction, Volume 7, 1992, Lawrence Erlbaum Associates, Inc. pp. 1-45.

[31] I Horrocks, (1999). *Constructing the User Interface with Statecharts*, Addison-Wesley, ISBN `0-201-34278-2`.

[32] G Krasner, S Pope, (1988). *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk -80*, Journal Of Object-Oriented Programming (JOOP), August/September 1988.

[33] K Moens, (2005). *A Declarative Meta Programming Approach to Enable Runtime User Interface Adaptation*, Licentiate's Thesis Dissertation, VUB, Brussels, 2005.

[34] G Mori, F Paterno, C Santoro, (2002). *CTTE: Support for Developing and Analyzing Task Models for Interactive System Design*, IEEE Transactions on Software Engineering, August 2002, pp.797-813.

[35] B Myers, (1991). *Separating application code from toolkits : Eliminating the spaghetti of call-backs*, UIST91, 1991.

[36] F Nogueira de Lucena, H Liesenberg, (1994). *A Statechart Engine to Support Implementations of Complex Behaviour.*

[37] F Nogueira de Lucena, H Liesenberg, (1993). *Programming Dialogue Control of User Interfaces Using Statecharts.*

[38] F Nogueira de Lucena, H Liesenberg, (1993). *Reflections on Using Statecharts to capture Human-Computer Interface Behaviour.*

[39] Object Management Group, Inc. , (2003). *OMG Unified Modeling Language Specification*, version 1.5, formal/03-03-01, March 2003.

[40] D Parnas, (1972). *On the criteria to be used in decomposing systems into modules*, Communications of the ACM, v.15 n.12, p.1053-1058, Dec. 1972.

[41] F Paterno, (2000). *Model-Based Design and Evaluation of Interactive Applications*, Springer, 2000.

[42] PetriNets, *http://pdv.cs.tu-berlin.de/ azi/petri.html.*

[43] J Peterson, (1981). *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, New Yersey, 1981, 290 p.

[44] Prog Artifacts, *http://prog.vub.ac.be/progsite/Projects.html*

[45] Redmon-Pyle, Moore, (1995). *Graphical User Interface Design and Evaluation*, Prentice-Hall, 1995.

[46] C Rouff, (1991). *Specification and Rapid Prototyping of User Interfaces*, University of Southern California Ph.D., 1991, 219 pages.

[47] B Shneiderman, (1982). *Multiparty Grammars and Related Features for Defining Interactive Systems*, IEEE Transactions on Systems, Man, and Cybernetics, 12(2):148-154, March/April 1982.

[48] C Sun, (2003). *Statecharts based GUI design*, School of Computer Science, McGill University, Montreal, Canada.

[49] R Taylor, G Johnson, (1993). *Separations of Concerns in the Chiron-1 User Interface Development and Management System*, Proceedings of InterChi, april 1993.

[50] K van de Berg, J Conejero, R Chitchyan, (2005). *AOSD Ontology 1.0: Public Ontology of Aspect Orientation*, Report of the EU Network of Excellence on AOSD, 2005.

[51] A Wasserman, (1985). *Extending State Transition Diagrams for the Specification of Human-Computer Interaction*, IEEE Transactions on Software Engineering, vol. SE-11(8), August 1985, pp. 699-713.

[52] P Wellner, (1989). *Statemaster: A UIMS Based on Statecharts for Prototyping and Target Implementation Notation for Specification*, Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems, 1989, pp. 177-182.

[53] Whatis definition on Statemachine, *http://whatis.techtarget.com/definition*