Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Synthesis of human-readable Statecharts from Sequence Diagrams

in the ROOM Environment

cand. inform. Björn Lüdemann

August 25, 2005

Institute of Computer Science and Applied Mathematics
Real-Time and Embedded Systems Group

Advised by:
Prof. Dr. Reinhard von Hanxleden
Dipl. Inf. Steffen Prochnow
Dipl. Phys. Carsten Ziegenbein[*]

[*]Philips Medical Systems, Hamburg, Germany

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

Due to their intuitive form, sequence diagrams are simple to write and understand and serve as a notation for system requirements and design. Statecharts are widely used for example for modeling of reactive systems and as a base for code generation. Variants of both diagram types are part of the ROOM and UML-RT frameworks that are widely used in industrial software modeling.

This work proposes an approach to the automated synthesis of statecharts from sequence diagrams called statechart composer—SCC. The SCC starts with a possibly non-empty hierarchical statechart that can be manually created. Then the SCC adds behavior for each requirement sequence diagram incrementally. It introduces hierarchy into a statechart to simplify it and/or to increase traceability of requirements. An alpha version of the SCC is implemented as a plug-in for a commercial CASE-tool.

keywords sequence diagram, statechart, synthesis, hierarchy, UML, ROOM, modeling, automation

Contents

List of Figures

1 Introduction

The use of Scenarios, Message Sequence Charts [13] or Sequence Diagrams [14] as notation for system requirements and design has found broad acceptance and is used in software development processes such as the Rational Unified Process (RUP) [16]. Due to their intuitive form, sequence diagrams are simple to write and to understand, which might be a reason for their wide acceptance.

Statecharts [10] are widely used for example for modeling of reactive systems and as a base for code generation. Both sequence diagrams and statecharts are part of the Unified Modeling Language (UML) [14] that is the *de facto* standard in industrial software modeling. In contrast to their wide spread use, the UML standard says little about the formal relation between different views of an UML model. Commercial CASE-tools such as Rational Rose Realtime [27], Rhapsody [28] or Artisan [3] offer no or little automatic support to generate statecharts from sequence diagrams and vice versa.

Different areas make statechart synthesis from sequence diagrams difficult [36]. There is a tendency toward using sequence diagrams informally: Sequence diagrams are interpreted differently by different users [8, 17], thus the semantic is not always clearly defined and it is not always clear how sequence diagrams relate to each other [35]. Furthermore, consistency of sequence diagram requirements has to be defined and ensured [21].

The resulting statechart has to express what the requirement engineer has in her mind. As soon as the statechart includes hierarchical states, the question arises if the behavior is correctly generalized. Another problem is that the statechart is an intermediate result and subject to further refinement. Thus it has to be readable and understandable to the developer. Besides graphic layout [25, 26] hierarchy has to be introduced to guide the reader through the statechart.

Recently much research has been done to investigate the transformation of sequence diagrams into statecharts, for example by Whittle [36], Mäkinen and Systä [22] or Krüger *et al.* [17]. The present thesis describes an algorithm that transforms sequence diagrams to statecharts as an extension of a CASE-tool and its implementation. The method presented in this work follows an incremental approach and can generate hierarchical statecharts.

1.1 Objectives

The approach presented in this thesis, called the *Statechart Composer* (SCC), aims at optimizing software development in terms of quality and time.

The SCC shall accelerate development in automating a step that otherwise would have to be done manually. After applying the SCC there may be still work left in the statechart. For example integrating requirements that are not given in form of sequence diagrams. However, less information has to be entered twice, once in the sequence diagram and a second time in the statechart. The risk of for introducing faults is reduced.

The SCC shall be used in early phases of the development cycle. It shall accept partial specifications and be able to handle non-determinism. That allows another application beside generating implementation statecharts: provide early feedback to the requirement engineer. She can generate a statechart for a part of the system to see how her requirements fit together and where critical areas are. So the SCC helps to detect inconsistencies within requirements, thus preventing faults early in the software life cycle.

Another objective on the SCC is to integrate as smoothly as possible into the existing work flow of a software development apartment. Besides being a plug-in for a CASE-tool, which is based on the UML-RT/ROOM Framework [29], the SCC has to follow an *incremental* approach in generating statecharts: the sequence diagrams shall be integrated one after another into an existing statechart. A manually created statechart may serve as a starting point as well as an automatically created statechart or an empty statechart. This statechart may be structured hierarchically. The developer may also change the resulting statechart arbitrarily and restart the SCC to complete the statechart and thus see if the complete behavior is still implemented as specified in the sequence diagrams. To improve the human readability of the resulting statecharts, hierarchy shall be introduced if needed.

This work presents an approach that reaches these objectives. It also provides an implementation of the basic ideas of this approach.

1.2 Environment

This work has been done in cooperation with a software development team that develops real-time embedded software for medical devices for Philips Medical Systems in Hamburg, Germany. This software is developed according to a customized Rational Unified Process (RUP) [16].

In wide parts of this process a CASE Tool that is based on the ROOM/UML-RT framework is used. The CASE-tool uses the UML 1.4 notation with a real-time extension, *i. e.* basic sequence diagrams (actors, interaction instances, a/synchronous, call, create and destroy messages, local states, local actions, focus of control, coregions) and statecharts without orthogonality that are associated with a class. Active classes, so-called *capsules*, communicate pair-wise via two-directional protocols with priority-ordered message buffers. The semantics is run to completion [29].

While the CASE-tool ensures consistency within the implementation model, the so-called *logical model*, the elements of the design model are coupled rather loosely without even name checks after creation. The CASE-tool provides an API to access model elements, modify the model and to expand the tool itself. The implementation of the SCC is bases on this API.

1.3 Main Issues

Being one of the most influencing authors on statechart synthesis, Whittle [36] identifies three main issues in synthesizing statecharts from sequence diagrams:

1. detect and resolve conflicts in the sequence diagrams;

2. recognize and merge identical or similar behavior from sequence diagrams;

3. construct a highly readable state machine.

The SCC-approach addresses these issues in the following ways:

1. The detection and resolution of conflicts is described in detail by Lischke [21]. The SCC-approach allows some inconsistencies like unconnected states in the specifications to be able to provide early feedback to the designer or to work with intermediate results. To produce results that can serve as a base for code generation, the SCC-approach requires a prior run of a consistency-checking tool [21] to check the sequence diagram specification.

2. We use two ways to recognize and merge identical or similar behavior from sequence diagrams: First, we demand explicitly named local states at the beginning and at the end of every lifeline in a sequence diagram. These states are then mapped to statechart states and the sequence diagrams are merged comparable to the approach of Krüger *et. al* [17] via these states. Second, we use simulation: Starting in the corresponding start state in the statechart, the SCC checks whether the behavior defined by the sequence diagram is already implemented by the statechart. If it is not implemented, only the behavior is added that the statechart was not already capable of.

3. As stated by Whittle [36] hierarchy is a key feature to improve readability. The SCC-approach constructs hierarchical statecharts. After construction a module of the SCC named *Topological Layouter* (TL) further optimizes the statechart. It uses a combination of methods to introduce hierarchy in the statechart as described in Chapter 3. For example the TL uses a partial order on the explicitly named states, similar to the order on the state variables used by Whittle [36]. This order can be entered manually. As stated above the SCC can refine manually provided statecharts that can include hierarchical states as described in Chapter 4.

   Obviously, good graphic layout is another key feature to readability. The capabilities of current CASE tools for graphic layout are not good enough to be of practical use. However, providing graphic layout is beyond the scope of this thesis. For an overview over current approaches on this consider the work of Prochnow and Hanxleden [26]. In the context of automated statechart synthesis automated graphical layout would be desirable. The use of an external layouter like KIEL [32] can lead to a normal form for the SCC results.

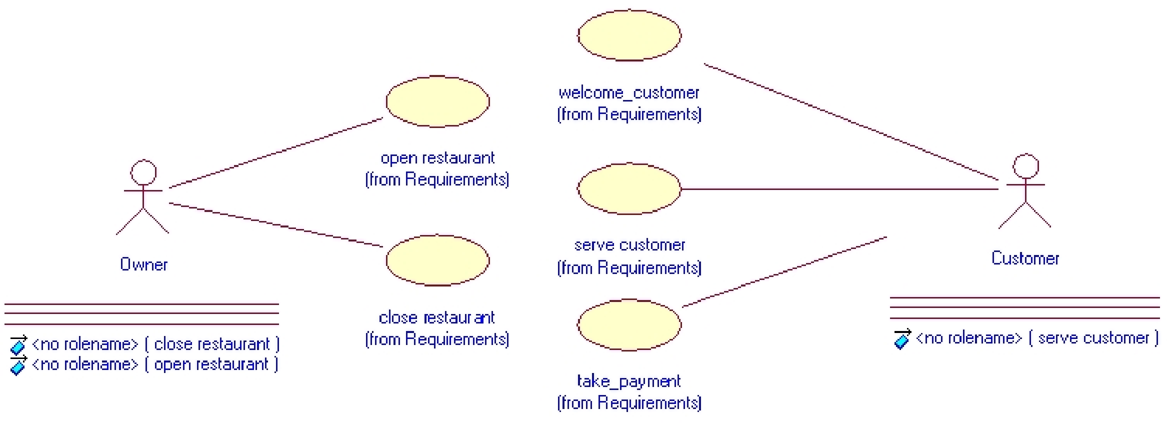


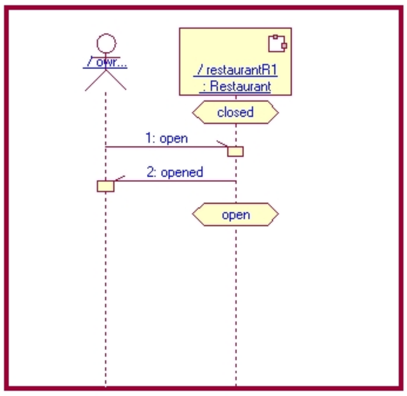
Figure 1.1: Use cases modeled by the restaurant example

1.4 Example

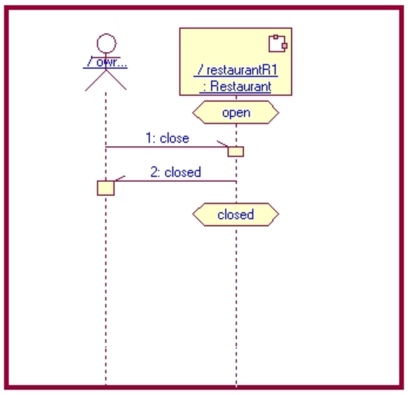
This section introduces a small reactive system that serves as an example in the following chapters. It is a model of a very simple restaurant in which one customer at a time can be served with cocktails. Figure 1.1 gives an overview of all modeled use cases. For each use case one requirement sequence diagram is given as shown by Figure 1.2.

In Figure 1.2(a) the stickman stands for the interaction instance of the actor *owner* and the rectangle labeled *restaurant* for the interaction instance of the capsule *restaurant*. The dashed lines below these icons are the lifelines of the interaction instances. The locations on a lifeline are considered to be ordered in time. *I. e.* the restaurant is in the local state *open* before it receives the asynchronous message *enter* and sends the asynchronous message *show_ table* in return be in the local state *waiting_for_ wink*.

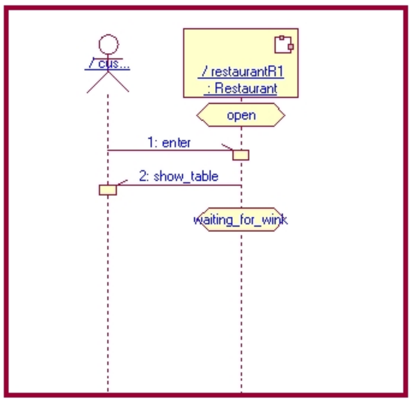
Figure 1.3 shows the *High-Level Message Sequence Chart* (hMSC) [13] that illustrates how the sequence diagrams are connected. The hMSC is given for illustrative purpose only, the SCC-approach does not rely on it to synthesize the statechart. It uses the start and end states on each lifeline to connect the sequence diagrams with each other. Figure 1.4 shows a statechart implementing the behavior as required by the sequence diagrams.



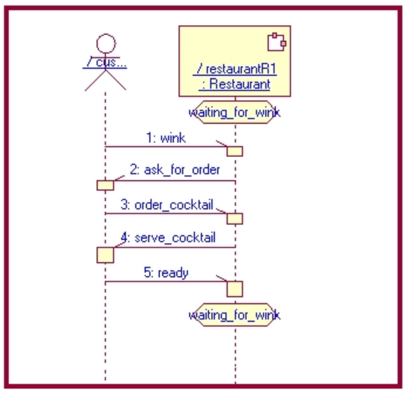
(a) Use case *open restaurant*



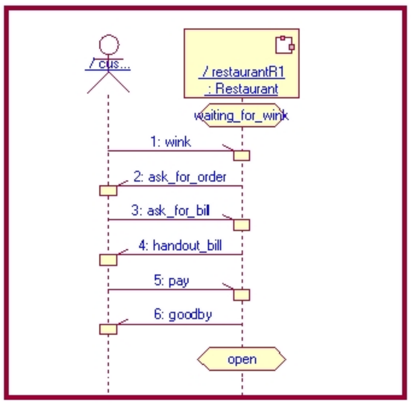
(b) Use case *close restaurant*



(c) Use case *welcome customer*



(d) Use case *serve customer*



(e) Use case *take payment*

Figure 1.2: Sequence diagram requirements for the *restaurant*

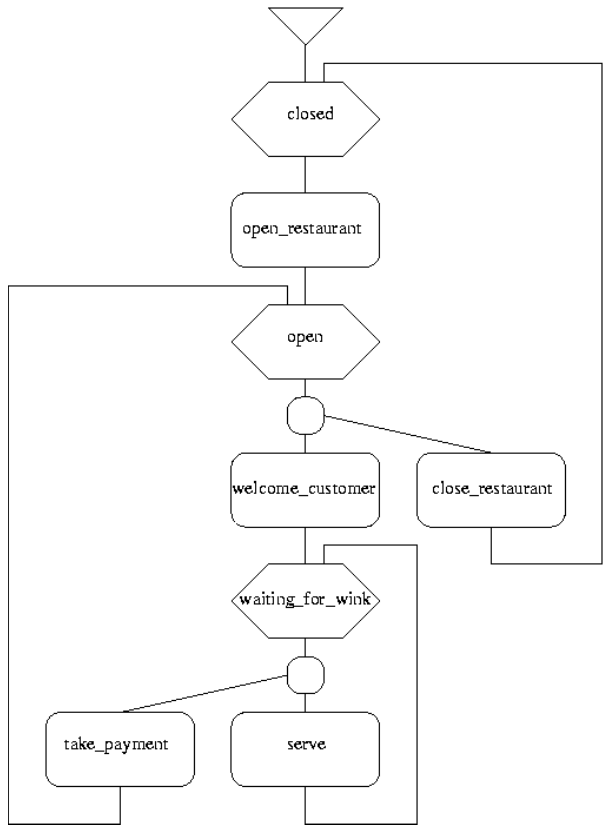


Figure 1.3: Requirement hMSC for the restaurant

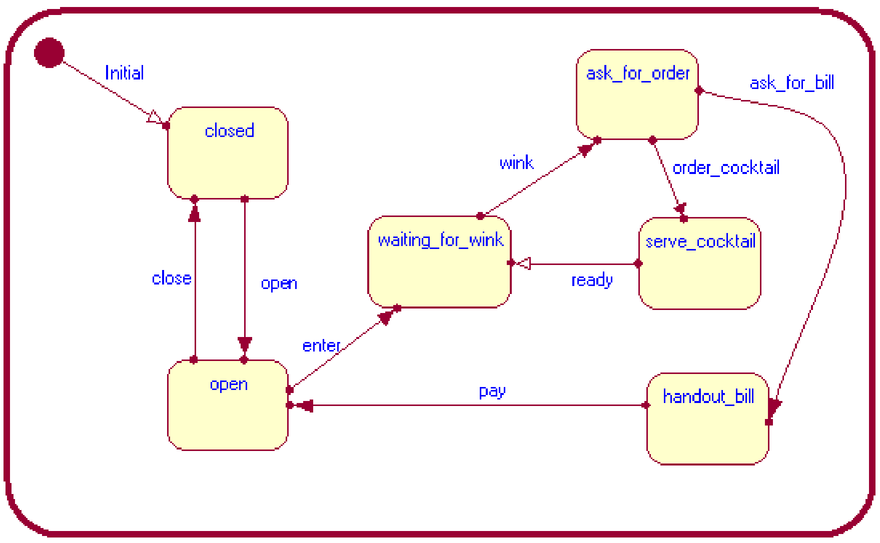


Figure 1.4: Statechart of the *restaurant*

1.5 Overview

This work is divided into six chapters. This introduction is followed by Chapter 2 *Flat Statechart Context*. It introduces a basic framework that is a subset of the ROOM framework and serves as a simplified base for the SCC-algorithm. Further sections detail the incremental approach and the foundation of the SCC-approach, which is based on the work of Krüger *et al.* [17].

Chapter 3 *Hierarchical Statechart Context* describes how the SCC-approach handles hierarchical statecharts in incremental synthesis. Furthermore it proposes six approaches to introduce hierarchy into a possibly flat statechart.

Chapter 4 *Implementation* shows how the plug-in for a CASE-tool is realized. Besides describing the general approach it details the *simulation* of a sequence diagram on a statechart, the *integration* of a sequence diagram into a statechart and the hierarchy-handling.

Chapter 5 describes the *Related Work*. It summarizes the papers that had the strongest influence on this thesis. For each paper it states how this thesis was influenced and which parts of the approach were not suited for the objectives of this thesis or the present environment.

Chapter 6 *Conclusion* states in how far the objectives are fulfilled and what can be the next steps in theory and implementation.

2 Statechart Synthesis

Section 2.1 of this chapter introduces a *basic framework* that serves as a foundation for the SCC-algorithm. Section 2.2 describes the SCC-approach in a flat statechart context and includes an example. Section 2.3 details the relation of the SCC-approach and the work of Krüger *et al.* [17]. Section 2.4 describes the extensions of the SCC-approach that allow it to be applied in an hierarchical context.

2.1 Basic Framework

As described in Chapter 1 the SCC has to work in the ROOM framework. This section introduces a framework, called *basic framework* in the following. It defines the subset of the ROOM framework [29] on which the SCC operates. Here is a short overview over the basic framework according to Lee [19]:

1. Ontology
   Components are *capsules*. Each capsule has a statechart describing its behavior. Each capsule has its own thread of control. Sequence diagrams show scenarios of possible inter-capsule behavior. In sequence diagrams only the following syntactic elements appear: interaction instance, lifeline, local states, asynchronous messages. On a location on a lifeline can thus either be a local state or a message. A sequence diagram has a start and an end state on each lifeline. This states provide a way to connect the scenarios given by sequence diagrams with each other.

2. Epistemology
   The highest structure is a *model* that knows its capsules, the *protocols* they speak and all sequence diagrams. Capsules and the associated ports know each other. Ports know each other if they are connected with *bindings*.

3. Protocols
   Capsules communicate only by two directional asynchronous message passing via channels called *bindings*. Bindings are realizations of protocols. A *Port* is the interface of a capsule to a binding. The messages sent over the bindings are buffered in *mailboxes* and handled based on the *run-to-completion semantics* [29]. The basic framework allows no message overtaking. The environment has to guarantee this. Solving the problems that would arise otherwise [29], is beyond the scope of this thesis.

4. Lexicon
   The messages passed over the bindings contain no data.

2.1.1 Basic Framework Properties

The basic framework has the following properties:

1. The only actions that are possible in a statechart are send actions on the transitions of the statechart.

   The basic framework can be realized in a ROOM-compatible framework. In the ROOM framework *target language code*, *e. g.* C++, is used to express message sending. Beside that arbitrary code may be part of the action of a transition. That allows flexibility in programming, but complicates statecharts analysis. For example, nested procedure calls or communication outside of the framework like accessing shared memory can appear. This is outside the scope of this work.

2. Every send action that occurs on a lifeline in a sequence diagram is triggered by a receive action.

   In the *run-to-completion semantics* [29] every action a system takes is a reaction on some input, *i. e.* it is *causal* according to the definition of Huizing and Gerth [12]. In this context messages have to be triggered from outside a capsule. Due to the start state at the beginning of a lifeline the triggering message appears in the same sequence diagram as the send action. This property includes that no two states follow on each other without a receive action in-between.

2.2 Incremental Statechart Synthesis

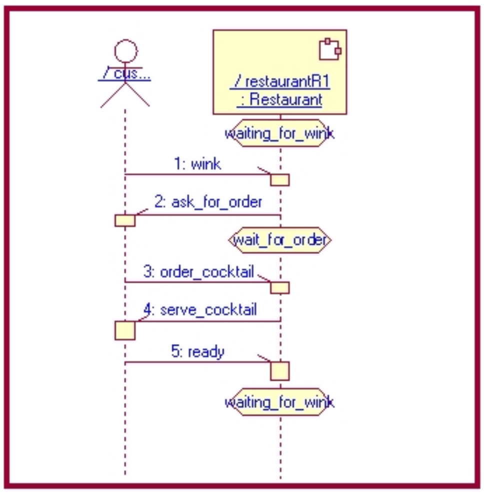
This section details the SCC-approach in the context of flat statecharts. Due to the first basic framework restriction, each sequence diagram has a start and an end state on each lifeline. States in sequence diagrams are referred to in the following as *sd-states*. Analogously states in statecharts are referred to as *sc-states*.

Definition (Corresponding States) Let ι be an injective mapping from the sd-states into the sc-states. If not defined otherwise, ι is the identity in state names. A state $s1$ is called *corresponding* to a state $s2$ iff

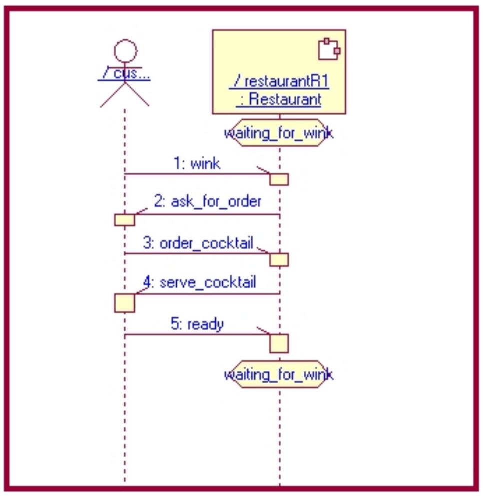
- $s1$ is a sd-state and $s2$ is a sc-state or vice-versa and
- $\iota(S1) = S2$ or $\iota^{-1}(S1) = S2$. $*$

Definition (Projection) A projection of a capsule c on a sequence diagram sd is the part of sd consisting of c, its lifeline l, the states on l and the messages connected to l. The projections of the example sequence diagrams on *restaurant* are shown in Figure 2.2. $*$

Definition (Implicit/Explicit States) Local states that are drawn into a sequence diagram are also referred to as *explicit states*. *Implicit states* are those states that are not drawn as local states on a sequence diagram, but that exist nevertheless. An implicit state is on a lifeline after each incoming message where no explicit state exists yet. $*$



(a) Explicit state *wait_ for_ order*



(b) Implicit state instead of *wait_ for_ order*

Figure 2.1: Sequence diagram *serve_ customer*

In the *run-to-completion semantics* [29], transitions are only taken, when a triggering message is present. Instantaneous transitions that contain only send actions are not possible. That means every change of state is triggered by an incoming message and every incoming message has to be assumed to change the state. In this thesis a self-transition is considered to be a special case of state change. Outgoing messages and local states do not change state in the statechart.

Example (Implicit/Explicit States) Figure 2.1 shows sequence diagrams that are identically with the exception of a local state: In Figure 2.1(a) it is explicitly named *wait_ for_ order*. In Figure 2.1(b) it is not visible. $*$

Definition (Implements) Let c be a capsule with a statechart sc and sd be a sequence diagram. c *implements* sd iff there exists a path in sc that realizes the sequence, *i. e.*:

- in sc exist states s_{start} and s_{end} corresponding to the start and end states on the lifeline l of c in sd and
- there exists a path $p_c^{sd} = (s_{start} \rightarrow^{t_1} s_1 \rightarrow^{t_2} \ \ldots \ s_{k-1} \rightarrow^{t_k} s_{end})$ in sc, where
  - t_i are transitions and s_i are states, such that
  - t_i is triggered by the *i-th* incoming message on l
  - all outgoing messages on t_i lie on l after the *i-th* incoming message and before the next incoming message or state and
  - for all local states s appearing on l exists a state with the name of the target of the transition that is triggered by the last incoming message before s in sc.

In the following the path p_c^{sd} is referred to as the path *corresponding* to the projection of sd on c. $*$

Example (Implements) Consider the leftmost projection of restaurant in Figure 2.2 and the statechart in Figure 2.3. In this case l is the lifeline of the restaurant, $s_{start} = closed$, $s_{end} = open$, ι is the identity in state names, $k = 1$ and $p_{restaurant}^{open\_restaurant} = (closed \rightarrow^{open} open)$. Thus, the statechart implements the sequence diagram of which Figure 2.2 shows a projection. $*$

Algorithm (SCC) The SCC-approach ensures that all sequence diagrams are implemented by the statechart. The diagrams are integrated incrementally, one after another, into the statechart. There are two possible cases to start with.

1. an empty statechart

2. an non-empty statechart

In the first case the path that is required for the statechart to implement the first sequence diagram is drawn: The SCC introduces one state per implicit or explicit state on the lifeline and one transition per incoming message into the statechart. The transition sends all subsequent outgoing messages until the next incoming message on this transition. Note that this path is not reachable. To connect it with the initial state, the initial states have to be sterotyped as *initial* by the developer so that the SCC adds an initial transition. An example for this is given by the leftmost projection of restaurant in Figure 2.2 and the statechart in Figure 2.3.

The second case is the initial situation for every sequence diagram that is integrated after the first one. In order to integrate the sequence diagram into the statechart, the SCC has to

- *test*, if it is already implemented and

- *integrate* the path to ensure implementation. $*$

Algorithm (Test) To *test* whether a statechart of a capsule implements a sequence diagram with the start state *sst* and the end state *est* on the corresponding lifeline, the SCC checks

- if the corresponding start and end states exist in the statechart and

- if they are connected with a path that implements the sequence diagram.

To test the path, the statechart is simulated. Beginning with *sst* the incoming messages defined by the sequence diagram are given as input. If the outgoing messages match the sequences given by the sequence diagram and all explicit states are correctly active, the test is passed. $*$

Algorithm (Integrate) Let *sd* be a sequence diagram, *c* be a capsule, *sc* be the statechart of *c*. The *simple* approach to integrate *sd* into *sc* connects the corresponding start and end states with the path p_c^{sd}.

The *standard* approach simulates the sequence diagram on the statechart as described above. At each divergence from the path p_c^{sd} the needed transition or state is added and the simulation is continued with the new elements. When the end state is reached, *sc* implements *sd*.

In the SCC, a new transition is named after its triggering message. A new state corresponding to an explicit sd-state is named after this sd-state. A new state corresponding to an implicit state is named after the last message sent by an incoming transition. If no message is sent on the transition that leads to the new state it is named after the trigger message. While the naming convention for transitions and states corresponding to explicit states are useful, the naming of states corresponding to implicit states is an open issue. $*$

The SCC-approach extends the behavior of the statechart. All behavior shown by the statechart before applying this algorithm is still shown afterward. If the resulting statechart is viewed as inheriting behavior from the original statechart, *protocol inheritance* as introduced by Aalst *et al.* [4] is preserved.

Example (Integrate) An example is given by Figures 2.2 to 2.7 with the additional input that *closed* is an initial state.

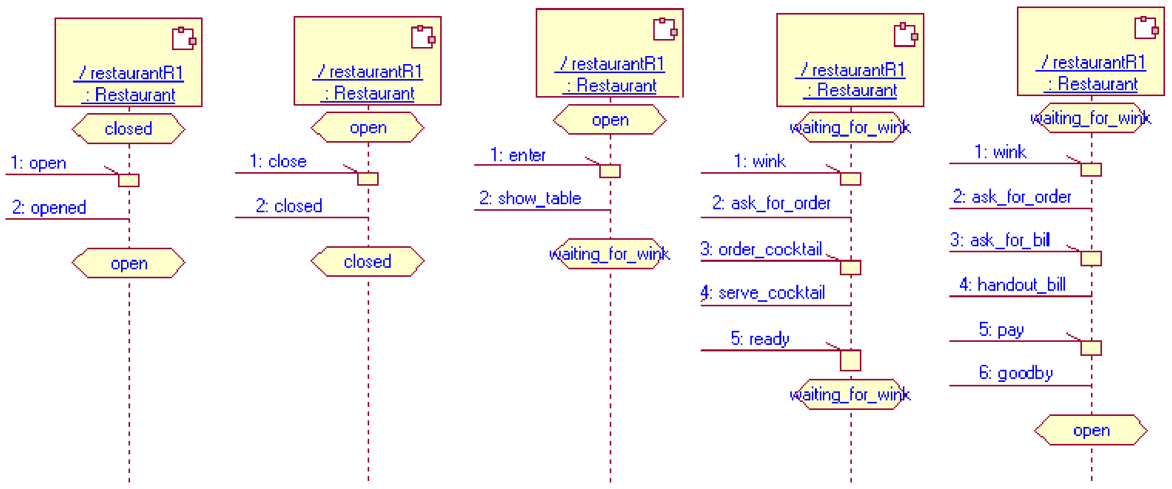


Figure 2.2: Projections of the requirement sequence diagrams shown in Figure 1.2 .

Figure 2.2 shows the projections of the requirement sequence diagrams *open restaurant*, *close restaurant*, *welcome customer*, *serve customer* and *take payment* on the capsule *restaurant*. The following figures show the intermediate results of integrating them successively.

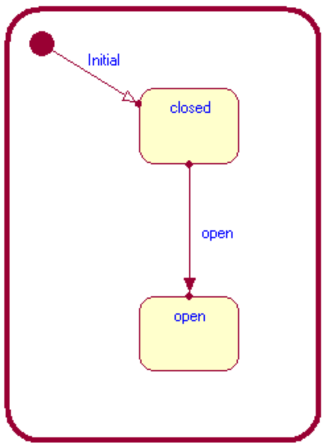


Figure 2.3: Result of integrating *open restaurant* into an empty statechart

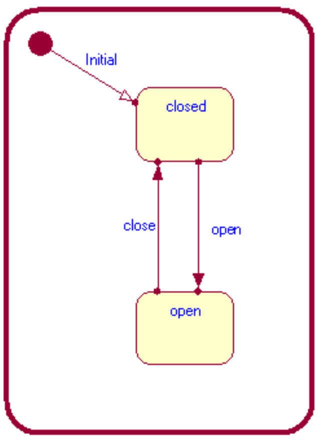


Figure 2.4: Result of integrating *close restaurant*

As shown in Figure 2.3 new states corresponding to the explicit states *open* and *closed* are added to the empty statechart. They are connected with a transition triggered labeled with open. It is triggered by *open* and sends *opened*. Both times it is communicating with the *owner*.

Figure 2.4 illustrates that no new states are needed to introduce the sequence diagram *close restaurant*. The simulation starts at state *open*. The state exists in the statechart, so nothing is to add. Next it tries the trigger *close*. There is no transition to follow, so a new transition, named *close*, is added. *close* is triggered by the message *close* and sends the message *closed* both in communication with the *owner*. Figures 2.5 to 2.7 show the statechart that result accordingly.

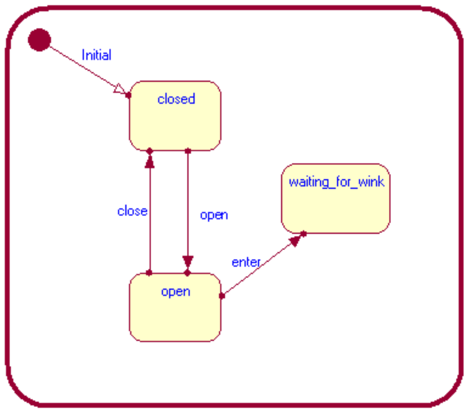


Figure 2.5: Result of integrating *welcome customer*

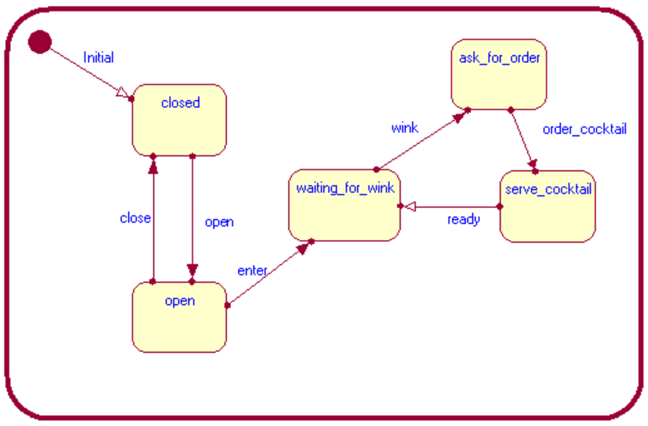


Figure 2.6: Result of integrating *serve customer*

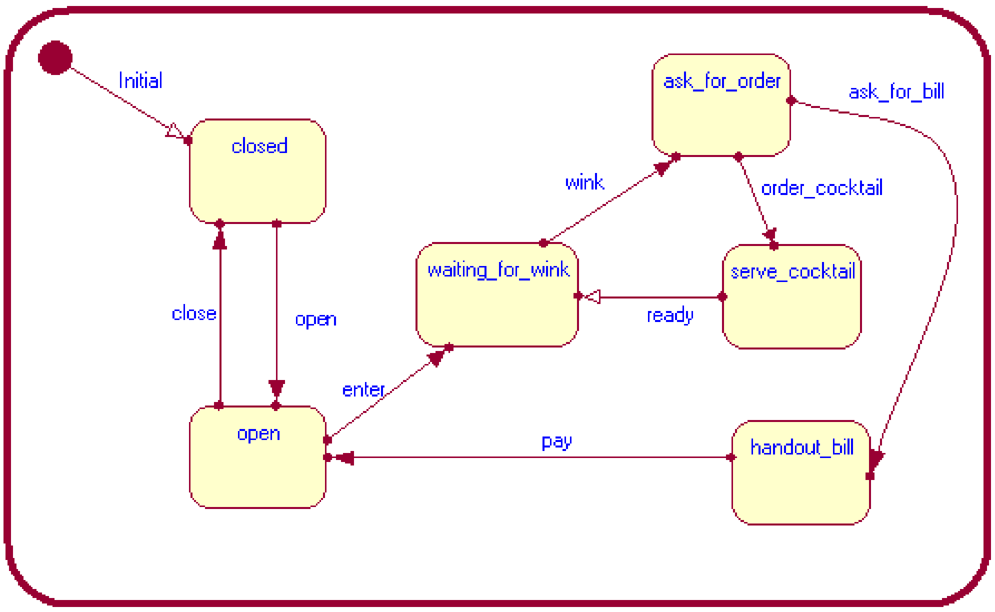


Figure 2.7: Result of integrating all sequence diagrams

2.3 Foundation of the Approach

The SCC-approach is based on the work of Krüger *et al.* [17]. In Chapter 5 a short overview of their approach is given and it is compared with the SCC-approach. This section discusses the common ground and the differences of the two approaches in detail.

Krüger *et al.* [17] use a framework similar to the basic framework. They talk of *components* instead of capsules, *channels* instead of bindings and ports, *conditions* instead of *sd-states* and instead of model they use the term *system* for the highest structure level. In this work a *system* is considered to be the implementation of a model. As in the SCC-approach sequence diagrams and statecharts are potentially non-deterministic. The interpretation of sequence diagrams as specification is the same in both works.

Their algorithm consists of five phases:

1. Projection on one interaction component: This is the same in both approaches.

2. Normalization: split the sequence diagrams until every sequence diagram starts and ends with pre and post states and has a message sequence in-between. That ensures that only explicit states are used to connect paths in the statechart. That is what the SCC does with the *simple* integration as described in Section 2.2.

3. Transformation into a sequence diagram (MSC)-automaton: identify sequence diagram states with statechart states and insert one transition for each sequence diagram.

4. Transformation into an automaton: expand the sequence diagram transitions into normal statechart transitions. This and the third phase are one step in the *simple* integration.

5. Optimization: The former phases introduce one transition per message and state into the statechart. This is done to allow the synthesis algorithm to work with different semantics. In this phase the statechart is minimized. The SCC-approach is tailored to the *run-to-completion semantics* [29]. It produces a statechart with only the needed transitions in one step. When the *standard* integration is used, the number of states in the constructed statechart is already minimized. The reason to use *simple* integration is that this minimization is not wanted.

2.4 Hierarchical Statecharts in Incremental Synthesis

A first approach to incremental statechart synthesis is to abstract from hierarchy and to allow only flat statecharts as described in the previous section. This does not suite practical use. Flat statecharts are way to big and cluttered in describing complex behavior. This section extends the approach to suit the needs of hierarchical statecharts.

The SCC-approach uses

- a mapping ι between sequence diagram states and statechart states and

- an explicit partial order π on sequence diagram states

As introduced in Section 2.2 ι defines which sd-state corresponds to which sc-state. For hierarchical statecharts ι maps sd-states to arbitrarily nested sc-states. Considering the image of the hierarchy relation given by a statechart under ι^{-1} the sd-states are ordered hierarchically, too. This order is made explicit with π. Due to the fact that sequence diagrams are given and statecharts are to be produced, the SCC-approach allows π as an additional input to statechart synthesis. π can be entered manually or be inferred from ι and a given statechart.

Defining π manually as part of the requirements can reduce the amount of sequence diagrams needed to specify a behavior: When the same basic sequence can start in different states, such as an exception handling, normally one sequence diagram has to be drawn for each such start state. When a local state is declared to be superstate of all start states, only a single sequence diagram is needed.

Transitions and states can be added on each layer of the statechart hierarchy. There are four reasonable choices for the layer on which to add a new state:

1. The top layer.

   The top layer is the most conservative choice, no generalization is introduced. On the one hand this choice neglects existing hierarchical structures and clutters the top layer. On the other hand the resulting statechart does exactly what is visible in the sequence diagram. Mostly that is not what the developer intends.

2. The layer of the source state ssl of the last added transition.
   $\iota(sds)$ is such a state, where sds is the start state of a sequence diagram.

   This choice adds all implicit states between an explicit state sds and the next explicit state under the parent state of $\iota(sds)$. This is the appropriate generalization when the last incoming message before the next explicit state changes the general behavior.

3. The layer of the target state tsn of the next transition to add.
   $\iota(sds)$ is such a state, where sds is the end state of a sequence diagram.

   This choice adds all implicit states between an explicit state and the next explicit state sds' under the parent state of $\iota(sds')$. This is the appropriate generalization when the first incoming message after an explicit state changes the general behavior.

4. The layer of the least common ancestor state of ssl and tsn.

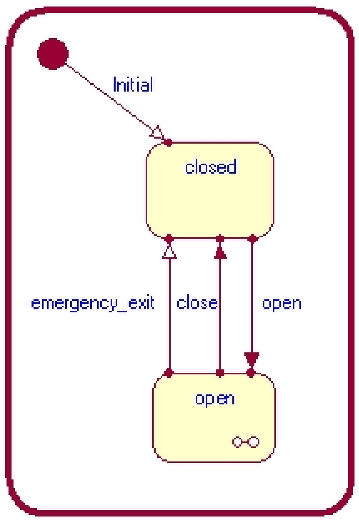
   This choice is a compromise. It is not as conservative as choice 1, but adds the behavior that ssl and tsn share to all implicit states. This is the appropriate generalization when none of the messages between the explicit states changes the general behavior.

Choices 2, 3 and 4 add the behavior of the respective parent states, *i. e.* in case of ssl and tsn having the same parent state choices 2, 3 and 4 produce the same statechart.

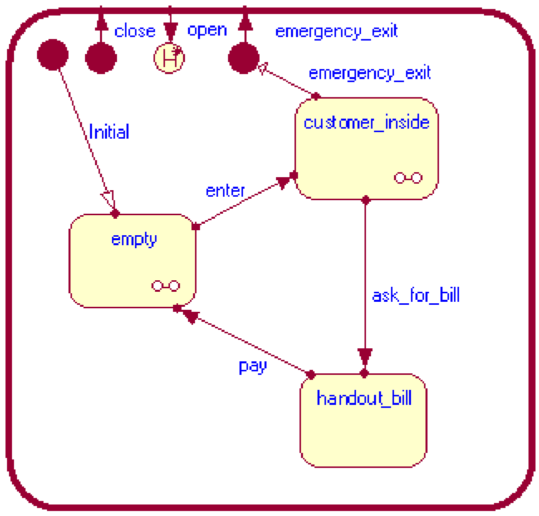
The four choices differ in the degree of generalization of behavior. In every case the specification has to be generalized *appropriately*. What "appropriately" means depends on the development context. Choices 2, 3 and 4 propose possibilities for generalization. A different view on this problem can be found in the work of Mäkinen and Systä [22].

The only way to be sure that the intended degree of generalization is reached is considering additional information. The developer can provide them as part of the requirements in form of π, in form of a strict semantic of sequence diagrams or interactively at runtime of a synthesizer as proposed by Mäkinen and Systä [22].

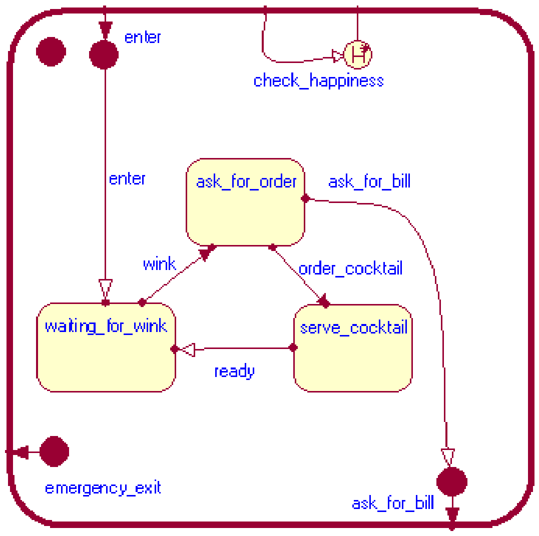
The following pages give a detailed example of the impact of the choice on the synthesized statechart.



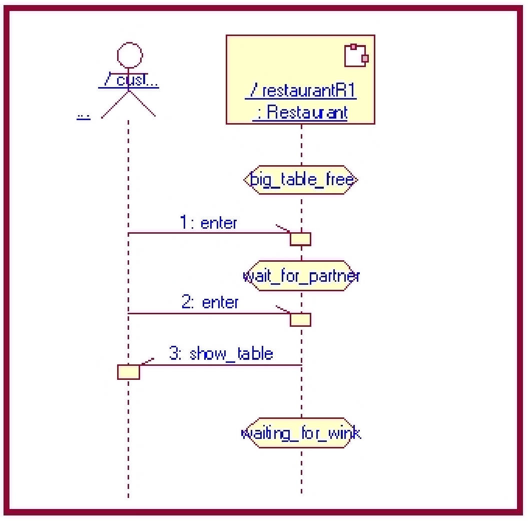
(a) Top state of *restaurant*



(b) Inside of state *open*



(c) Inside of state *customer_ inside*

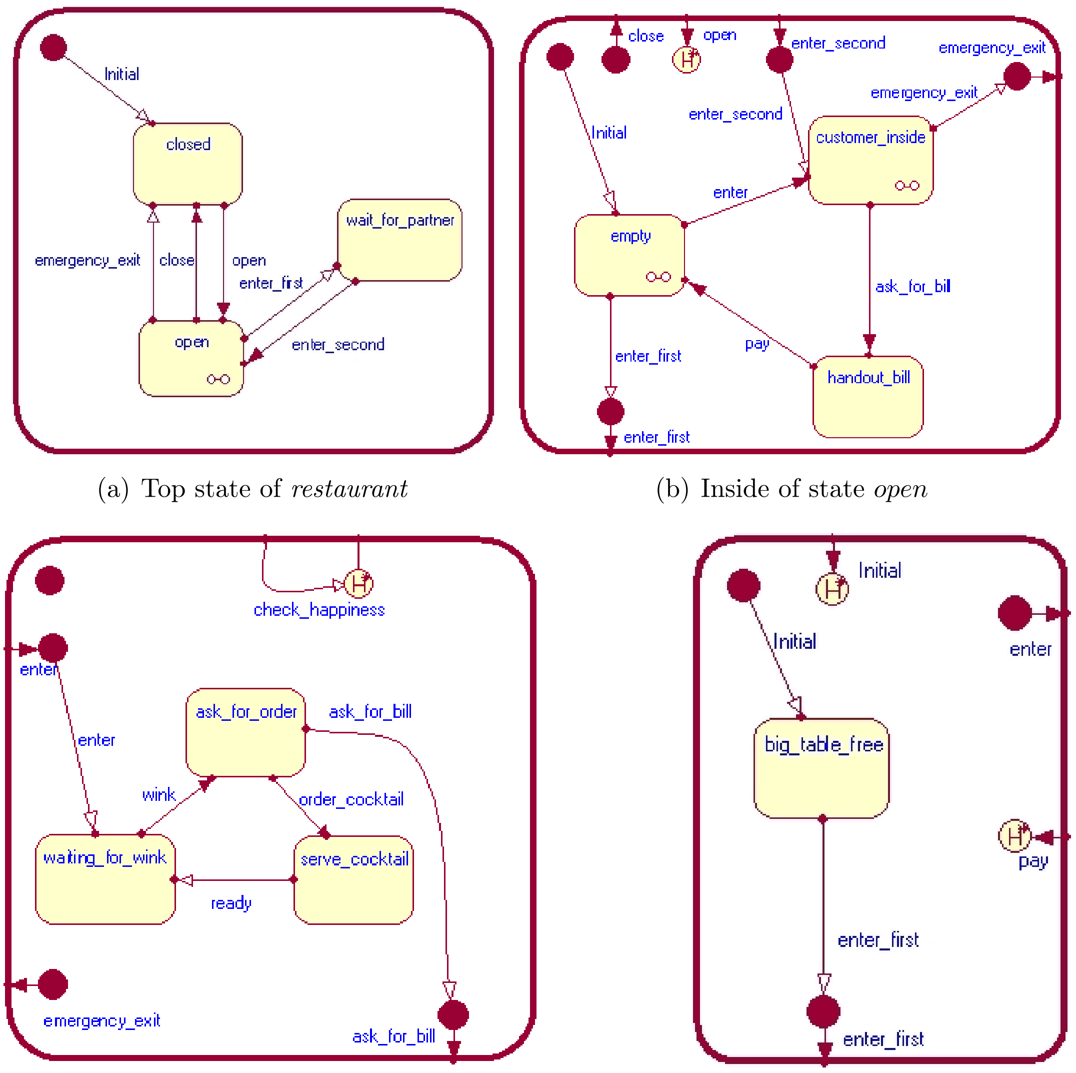


(d) Sequence diagram *welcome_ couple*

Figure 2.8: Extended *restaurant* example

Example (Choices) Consider the extended *restaurant* example as given by Figure 2.8. Two additional use cases are implemented: *check happiness* and *emergency exit*. As long as a customer is sitting at the table and is not yet paying, the owner can check, whether the customer is currently happy or in case of an emergency, the restaurant can be closed.

The sequence diagram given in Figure 2.8(d) defines a behavior that has to be implemented in the *restaurant* statechart: a couple shall be able to come in and be served together. π defines the start state *big_ table_ free* to be a substate of empty.



(a) Top state of *restaurant*

(b) Inside of state *open*

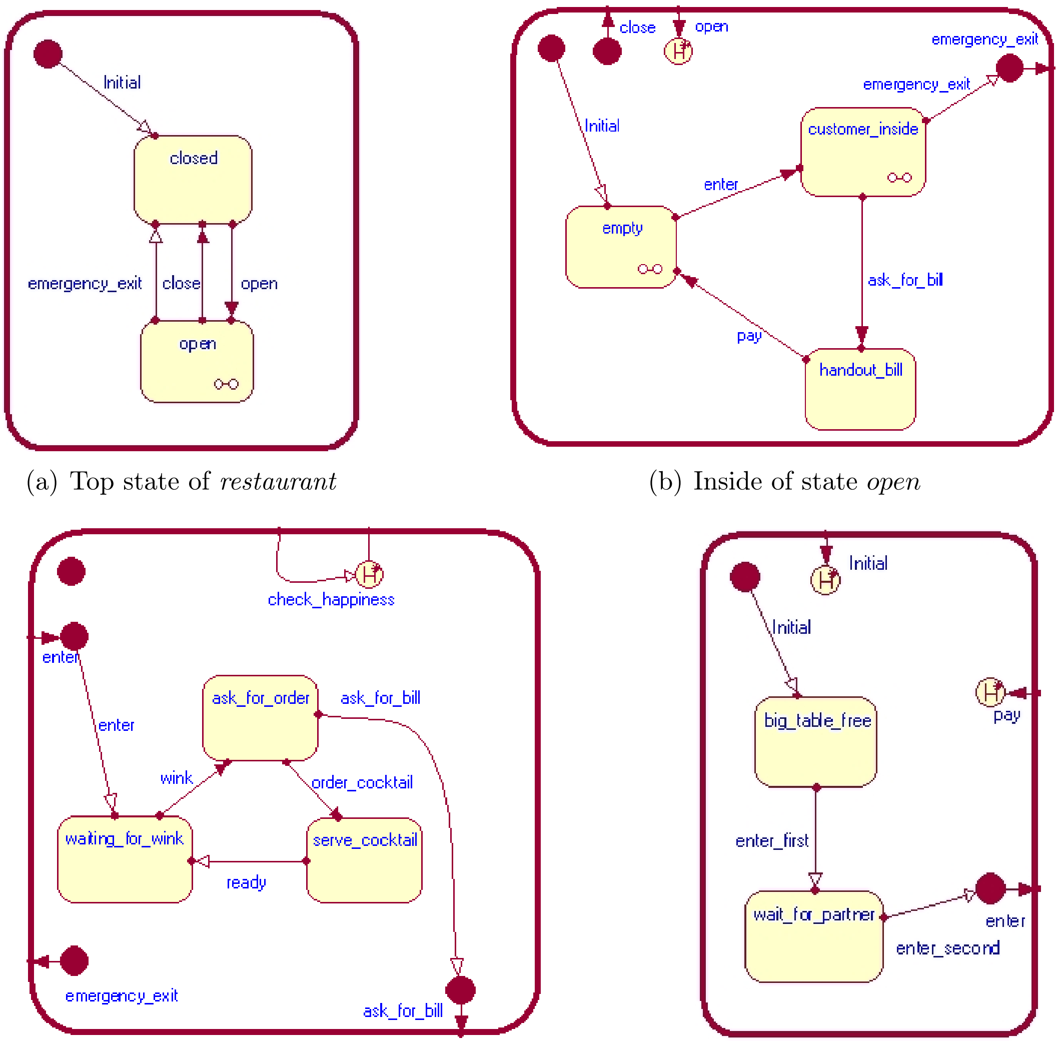
(c) Inside of state *customer_ inside*

(d) Inside of state *empty*

Figure 2.9: *restaurant* statechart resulting from choice 1

Figure 2.9 illustrates choice 1. The new state *wait_ for_ partner* is added on the top layer and the transitions from *big_ table_ free* and to *waiting_ for_ wink* are drawn as inter-level transitions. While *wait_ for_ partner* is the active state, the restaurant only reacts on the incoming message that indicates that the partner enters the *restaurant*. That means neither *emergency_ exit* is possible nor can the *restaurant* be closed. This is not likely to be the behavior the requirement engineer had in mind.

Nevertheless, if the algorithm which introduces the state *wait_ for_ partner* has no further information like π, this is a good choice, because no behavior is added to the statechart that is not explicitly specified in a sequence diagram.



(a) Top state of *restaurant*

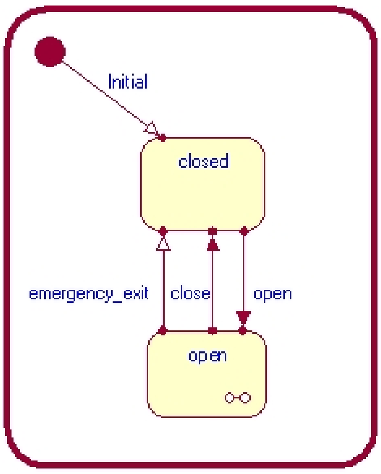
(b) Inside of state *open*

(c) Inside of state *customer_inside*

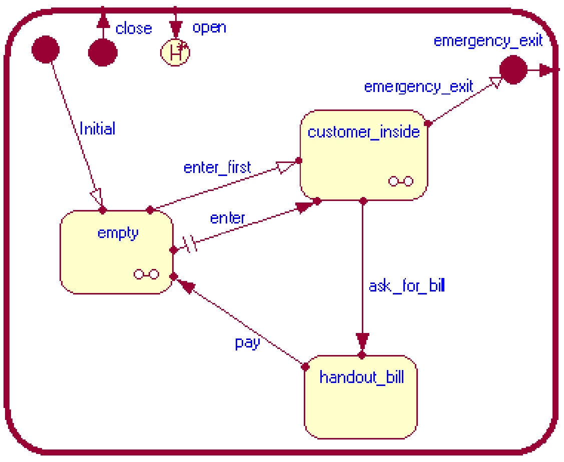
(d) Inside of state *empty*

Figure 2.10: *restaurant* statechart resulting from choice 2

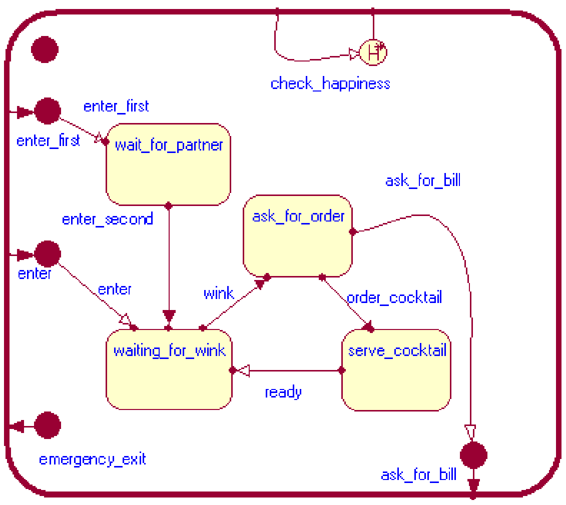
Figure 2.10 illustrates choice 2. The new state *waiting_for_partner* is added on the same layer as the start state of the sequence diagram *big_table_free*. So it inherits all behavior shown by *empty*. While *waiting_for_partner* is the active state, the restaurant reacts not only on the second *enter*, but allows the *restaurant* be closed, too. Still it is not possible to react on an *emergency_exit*. Following this choice behavior is introduced that is not explicitly specified.



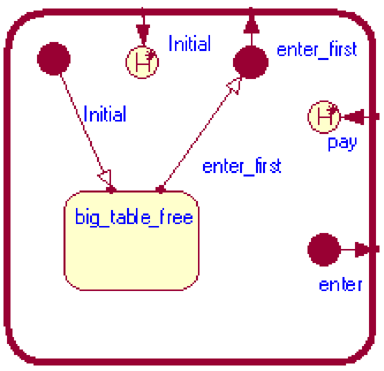
(a) Top state of *restaurant*



(b) Inside of state *open*



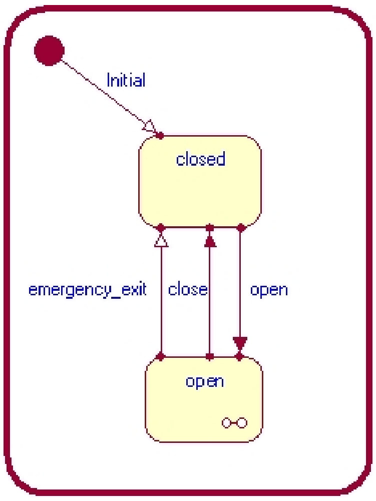
(c) Inside of state *customer_ inside*



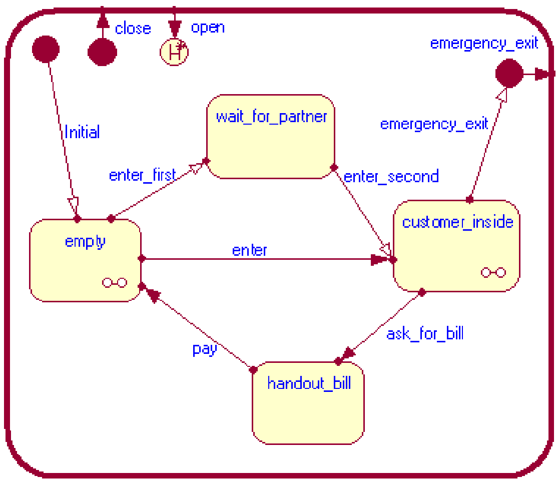
(d) Inside of state *empty*

Figure 2.11: *restaurant* statechart resulting from choice 3

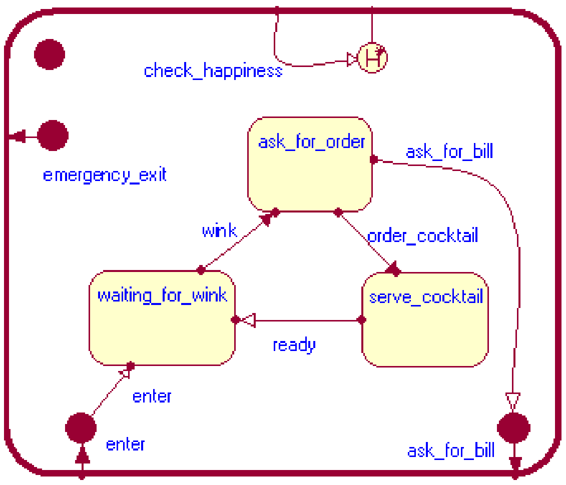
Figure 2.11 illustrates choice 3. The new state *waiting_ for_ partner* is added on the same layer as the end state of the sequence diagram *waiting_ for_ wink*. So it inherits all behavior shown by *customer_ inside*. While *waiting_ for_ partner* is the active state, the restaurant reacts on the second *enter*, *close*, *check_ happiness* and *emergency_ exit*. Even if this might seem the most likely intention in this example, the problem is the same as in choice 2: not explicitly specified behavior is introduced.



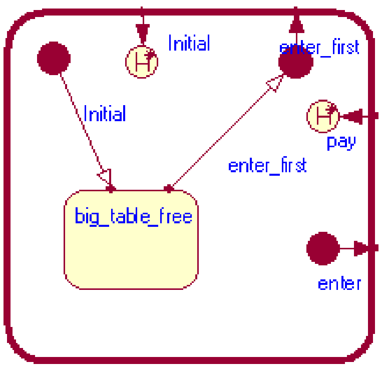
(a) Top state of *restaurant*



(b) Inside of state *open*



(c) Inside of state *customer_inside*



(d) Inside of state *empty*

Figure 2.12: *restaurant* statechart resulting from choice 4

Figure 2.12 illustrates choice 4. The new state *waiting_for_partner* is added on the layer of the least common ancestor of *big_table_free* and *waiting_for_wink*. So it inherits all behavior shown by *open*. While *waiting_for_partner* is the active state, the restaurant reacts on the second *enter* and *close*.

Choosing the fourth possibility means introducing not explicitly specified behavior, too. However, it could be an acceptable choice under some circumstances. ∗

3 Introduction of Hierarchy

This chapter discusses six approaches to introduce hierarchy into a statechart. Each of them can be automated and aims at making a statechart more readable.

Two features that make a statechart human-readable are:

- simplicity and
- traceability.

On a high hierarchical layer of a statechart only that behavior common to groups of states is shown. On nested layers only the refinement of that behavior is visible. So both views are free of the information shown by the respective other view. Thus a strategy to make a statechart simpler, is to identify common behavior and express it on a higher level.

To make specifications traceable in a statechart, the statechart has to reflect the structure of the specifications. In the given context the specifications are sequence diagrams organized in use cases.

For both categories means of introducing hierarchy in a flat state machine are discussed in the Section 3.1 and 3.2, respectively.

3.1 Simplicity

Hierarchy is useful in statecharts because it reduces the number of transitions and the number of states that are visible at a time. It helps to reduce redundant information in the chart and improves the readability. This section explains two ways to reduce the number of transitions without changing the behavior defined by the statechart. Both can be applied after having constructed the statechart with the SCC-approach.

3.1.1 Common Outgoing Transitions

When several states have a transition with the same trigger, action and target, a super state for these states can be introduced: one transition with the same trigger, action and target originating from the new state is added and the other transitions are removed. Depending on the visualization of the statechart, one down side of this simplification is that the labels of the transitions are displayed more than once.

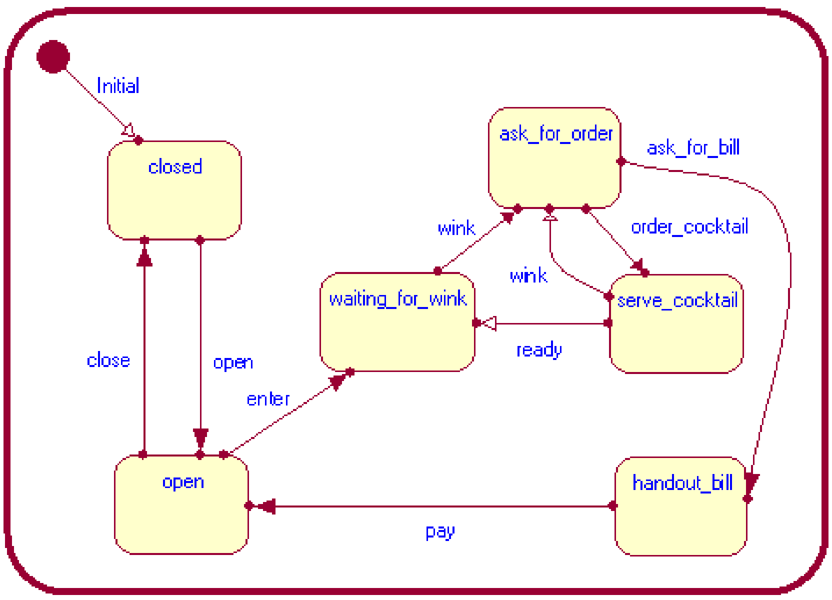
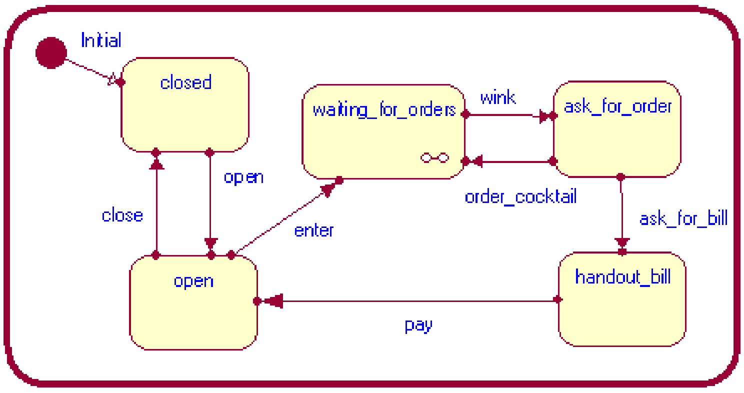
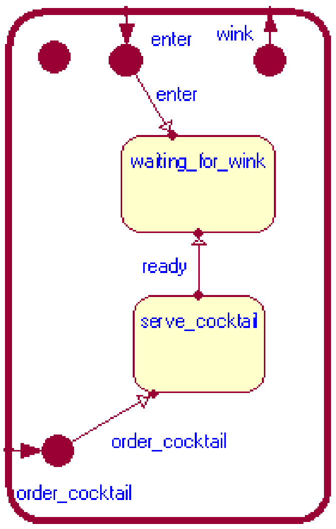


Figure 3.1: Statechart for *restaurant* with extra transition *wink* from *serve_ cocktail* to *ask_ for_ order*



(a) Superstate indroduced—only one *wink* transition needed



(b) Inside of state *waiting_for_ order*

Figure 3.2: *restaurant* statechart with new superstate *waiting_for_ order*

Example (Common Outgoing Transitions) Figure 3.1 shows an extended statechart for the *restaurant*. *waiting_for_wink* and *serve_cocktail* here both have a transition triggered by the event *wink* that causes the no action and ends at *ask_for_order*. That makes it possible to introduce a new superstate *waiting_for_orders* for *waiting_for_wink* and *serve_cocktail* and let the *wink* transition originate from it as shown in Figure 3.2(a). Figure 3.2(b) shows the inside of the new state: the transition *wink* is moved to start from the new parent.

The transition labels *enter* and *order_cocktail* each appear twice as transition labels and once as label for a junction point. ∗

3.1.2 Common Self-Transitions

Several states that have a self-transition with the same trigger/action pair can be grouped under one super state. A self-transition with the trigger/action pair targeting to history has to be added to the super state and the other self-transitions have to be removed.

Note that in a statechart dialect that uses entry and exit actions the behavior of the statechart with the superstate is generally not the same as in the original statechart.

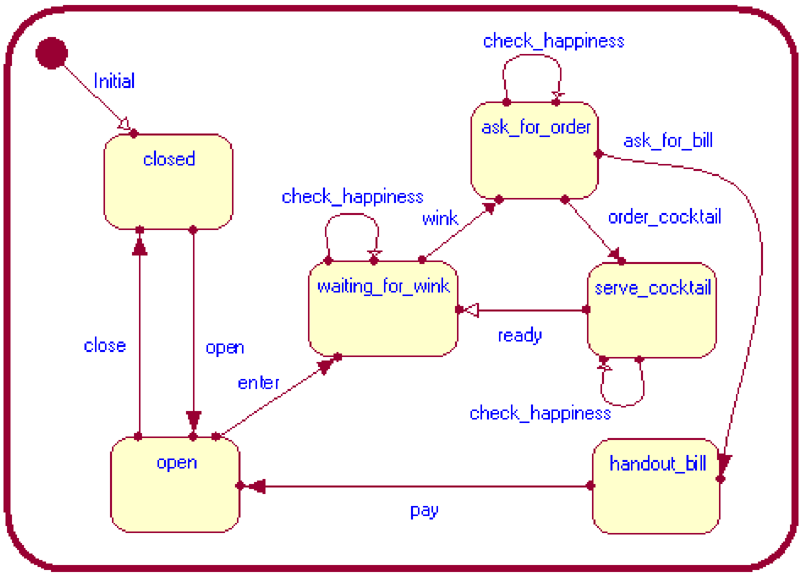
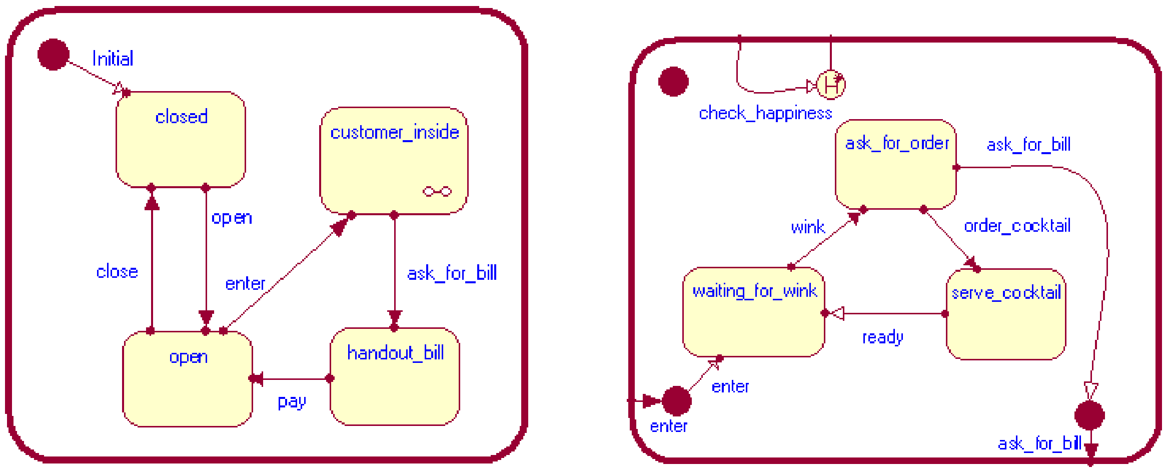


Figure 3.3: Statechart for restaurant with extra *check_happiness* transitions

Example (Common Self-Transitions) Figure 3.3 shows an extended statechart for the *restaurant*. *waiting_for_wink*, *serve_cocktail* and *ask_for_order* here all have an extra self-transition *check_happiness*. That allows to introduce a new superstate *customer_inside* for *waiting_for_wink*, *serve_cocktail* and *ask_for_order* as shown in Figure 3.4(a). Figure 3.4(b) shows the inside of the new state: The *check_happiness* self-transitions on *waiting_for_wink*, *serve_cocktail* and *ask_for_order* are removed. A new *check_happiness* self-transition is added that sources from the new parent and targets the parent's deep history connector. ∗



(a) Superstate *customer_inside* added (b) Inside of state *customer_inside*

Figure 3.4: *restaurant* statechart with new superstate *customer_inside*

3.1.3 Generalize Behavior

The generalization of behavior is the most important use of hierarchy. A result of introducing hierarchy in that way is a smaller number of transition and a thus simpler and more readable statechart.

This section describes approaches to group states of a flat statechart in a way that generalizes behavior. This can lead to the addition of behavior that is not explicitly defined in the sequence diagrams, but by π.

If a state $s1$ is bigger than state $s2$ according to π, $\iota(s2)$ is placed within $\iota(s1)$. $\iota(s2)$ inherits the behavior of $\iota(s1)$.

In the basic framework, as in the ROOM framework [29], transition trigger priority goes from deep nesting to shallow nesting: When two transitions $t1$ and $t2$ have the same trigger, $t1$ sources from sc-state $s1$ and $t2$ sources from sc-state $s2$, $t2$ is followed. In terms of inheritance $t2$ overwrites $t1$. That behavior conforms to a sequence diagram that causes $t2$ to source from $s2$.

If π is given before applying the SCC-approach, this kind of hierarchy is added at construction time as described above. However, it can also be introduced in a flat statechart afterward.

Example (Generalize Behavior) Consider Figures 3.5, 3.6 and

$$\begin{aligned}\pi = \{&(open, waiting\_for\_wink),\\ &(open, ask\_for\_order),\\ &(open, serve\_cocktail),\\ &(open, handout\_bill)\}.\end{aligned}$$

Figure 3.5 shows the flat statechart of the *restaurant*. As π defines *open* to be a parent state for *waiting_for_wink*, *ask_for_order*, *serve_cocktail* and *handout_bill*, these states are aggregated under a super state *open* in Figure 3.6. Still, a substate *open* has to be kept within the new superstate. This is the state in which the *restaurant* waits until a customer enters it.

Note that the *restaurant* now may be closed with customers locked inside. This is behavior that is not defined in form of sequence diagrams, but with the information entered through π. $*$

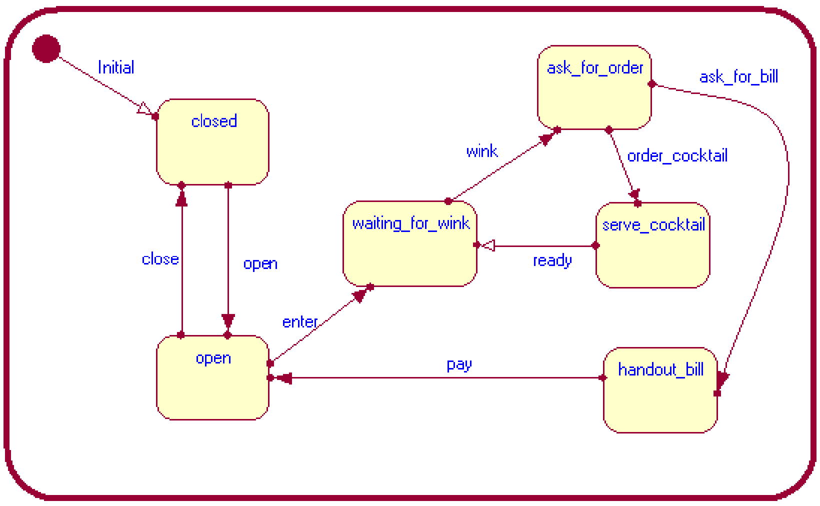
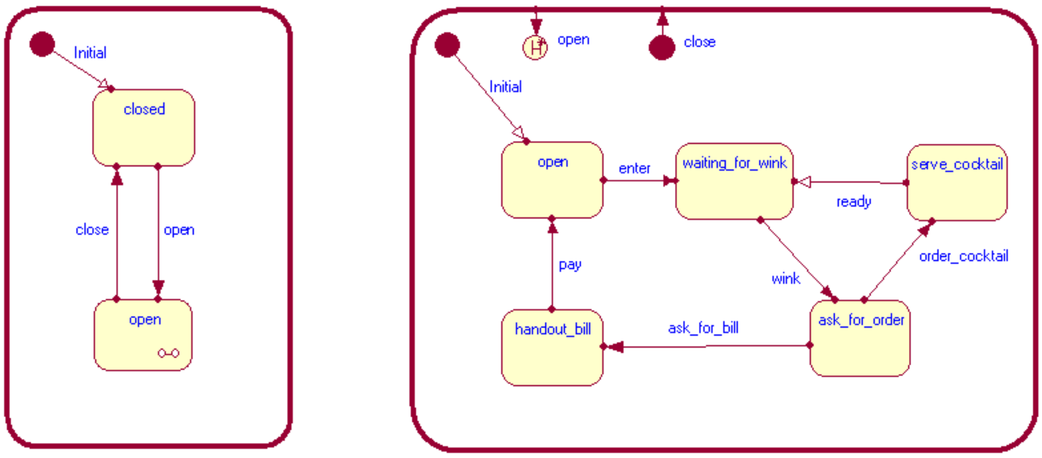


Figure 3.5: Flat *restaurant* statechart



(a) Top state (b) Inside of state *open*

Figure 3.6: *restaurant* statechart after applying π to introduce hierarchy

3.1.4 Group States

Another possibility to introduce hierarchy is the simple grouping of states. When the methods mentioned in the last two sections have been applied and the statechart is still hard to understand because too many states are on one layer the following approach can be used.

Due to their loop-like structure strongly connected components provide a natural grouping approach: The active state can be part of the strongly connected component for an arbitrary time. One possibility to group states like this is to apply Tarjans Algorithm [31] on a level of a statechart. When a super state is introduced for each maximal strongly connected component, the resulting level is acyclic. That does not change levels on which completely cyclic behavior is defined - just one new super state could be added. If different maximal strongly connected components are on one layer such a grouping simplifies to recognize the overall flow of behavior and makes acyclic behavior obvious.

Example (Maximum Strongly Connected Components) Figure 3.7 shows the *restaurant* statechart that results when the use case *take_payment* is not specified. Figures 3.8(a) to 3.8(c) show the result of applying this method to the statechart shown in Figure 3.7. The resulting acyclic statechart shows that the state *empty* cannot be reached again after leaving it. Note that the names for the new superstates are not generated automatically. To generate meaningful names for new superstates is even harder than generating names for new states inferred from the sequence diagrams as detailed in Section 2.2. *

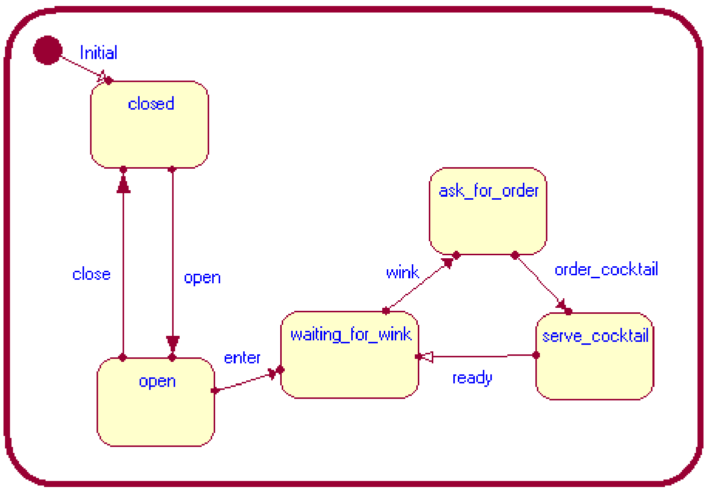
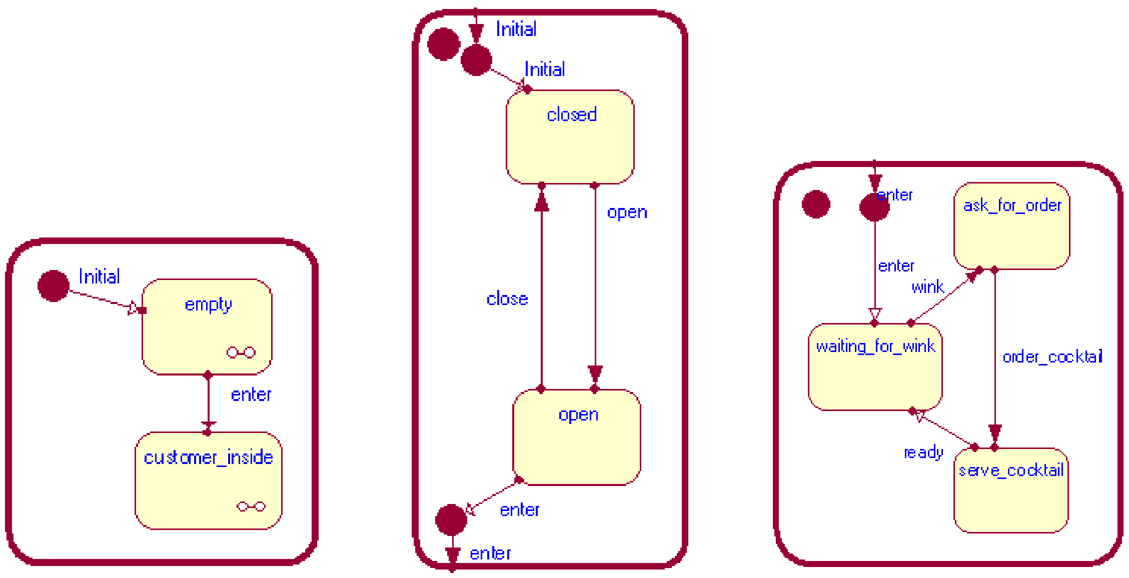


Figure 3.7: Top level of *restaurant* with no possibility to leave



(a) Superstates on the top layer

(b) Inside of *empty*

(c) Inside of *customer_ inside*

Figure 3.8: *restaurant* with states grouped to maximum strongly connected components

3.2 Traceability

This section describes how the structure of the requirements can be reflected in a statechart. This provides a view that makes it easy to find the place in a statechart that corresponds to a specific part of requirements. The importance of traceability is described by Bordeleau [7]. He proposes patterns to enhance traceability of requirements in the statecharts.

3.2.1 Use Case and Sequence Diagram Structure

Software development processes like the RUP [16] involve among others the following steps:

1. Textual requirements are expressed in form of use cases illustrated by use case diagrams. Together with the use cases their possible orders of execution can be given.

2. The use cases are detailed by sequence diagrams. One basic flow and possibly several alternative flows are each described by one corresponding sequence diagram.

When these structures are transferred to a statechart, it is similar to a higher order sequence chart (hMSC) [13]. It illustrates how the sequence diagrams fit together.

The following can be performed to group the states of a flat statechart in a hMSC like style:

- All states that mark a start or end state of a sequence diagram are left untouched on the uppermost layer.
- All states between them are grouped together under one super state per sequence diagram. This state carries the name of the sequence diagram.

Example (hMSC style) Figure 3.9 shows the *restaurant* statechart as it is generated with the *simple* integration approach as described in Section 2.2. The state *ask_for_order* appears twice, once with the slightly changed name *ask_for_order_1*.

Figure 3.10 shows the result of transferring the sequence diagram structure to the statechart, *i. e.* one superstate per sequence diagram is introduced. The states introduced for the sequence diagrams *open_restaurant*, *close_restaurant* and *welcome_customer* serve illustrative purpose only. The transitions leaving them have to be instantaneous, because no state change takes place in the respective sequence diagrams.

All other new superstates are refined by the behavior specified by the respective sequence diagrams. In Figure 3.9 the states *ask_for_order* and *ask_for_order_1* appear due to the *simple* integration approach. This is needed, because the state *ask_for_order* is part of both states representing the sequence diagrams *serve_customer* and *take_payment*. In Figure 3.10 *ask_for_order* and *serve_cocktail* are substates of *serve_customer* and *ask_for_order_1* and *handout_bill* are substates of *take_payment*.

The statechart is non-deterministic: when *waiting_for_order* is the active state, it is not clear which transition to follow, the one leading into *SD_serve_customer* or the one leading into *SD_take_payment*: both are triggered by *wink*. $*$

A similar topology results, when one state per use case is introduced and each use case has exactly one start and one end state. It is the same in the restaurant example. Differences appear when more than one sequence diagram per use case is part of the specification, then the hMSC-style could be applied after the use case structure is introduced in the statechart.

There can be more structure in the sequence diagram specification than start, end states and sequence diagram names. In a way similar to the hMSC like style notations like *sequence diagram reference* and *operators* as used in UML 2.0 [23] can be transferred into the statechart.

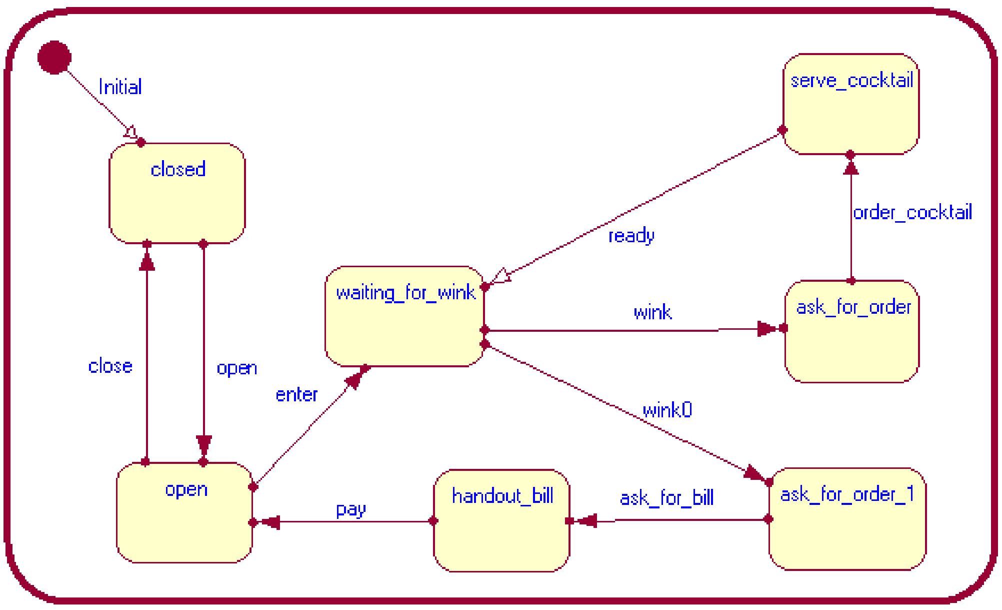


Figure 3.9: Flat restaurant statechart without simulation gained optimization

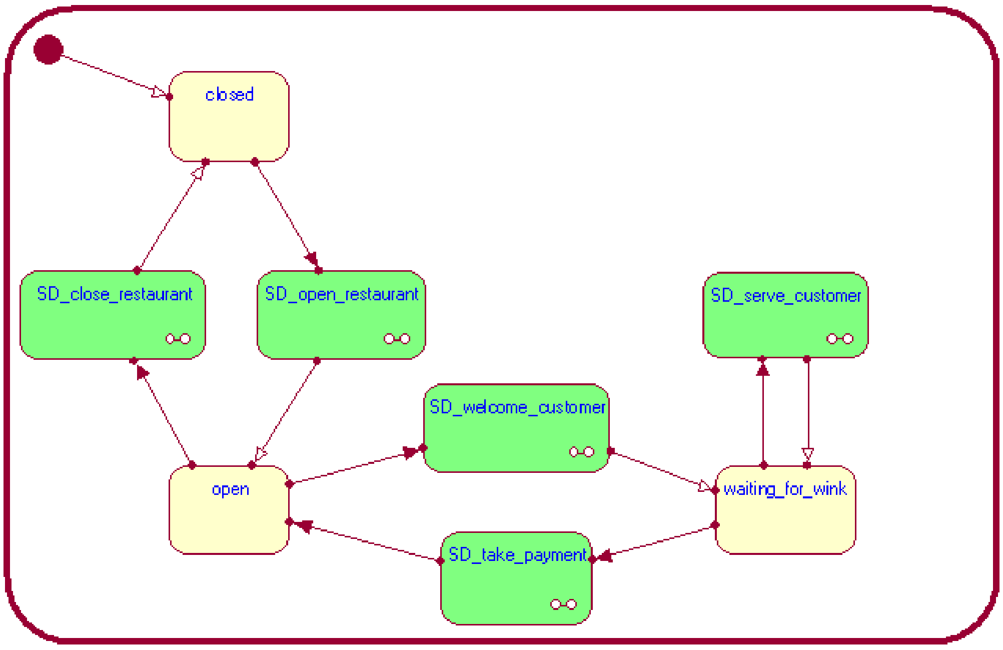


Figure 3.10: States grouped in a hMSC like style

3.2.2 Component Refinement Structure

In early development phases of the RT-TROOP modeling process [7], sequence diagrams for components are defined. During the *specification MSC modeling phase* sequence diagrams are *refined*, *i. e.* the sequence diagrams are extended to *decomposed components*. One possibility to transfer this structure into a statechart is explained by the following example.

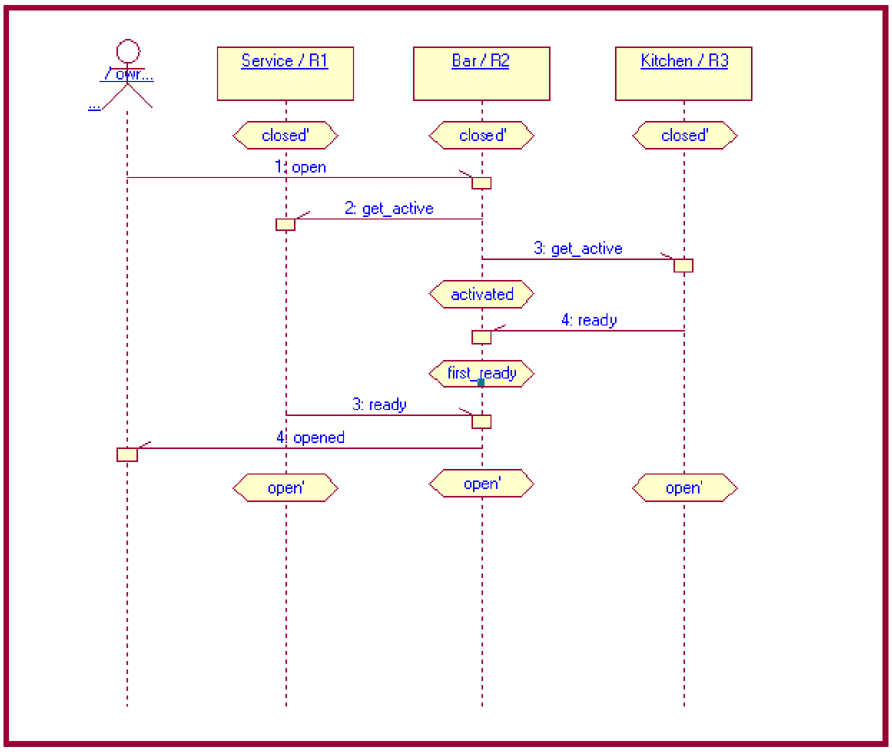


Figure 3.11: Refined sequence diagram for *open_restaurant* as shown in Figure 3.12

Example (Refinement) For the *restaurant* example this could be a decomposition of the *restaurant* capsule into the capsules *bar*, *kitchen* and *service*. Figures 3.11 shows the refined sequence diagram for the use case *open_restaurant*. Figure 3.13 shows the statechart for the *bar* that implements only this use case. Figure 3.14 illustrates a method to introduce hierarchy. The communication that has before been defined is considered to be more abstract. Hiding the newly specified internal communication in a superstate provides a more abstract view on the statechart. The possibility for internal communication can be visualized with an *local action* in the more abstract specification of *restaurant* as shown in Figure 3.12. ∗

Another way to make use of this relationship between sequences diagrams is to group all states under a superstate that contribute to a local action or a local state on the more abstract layer.

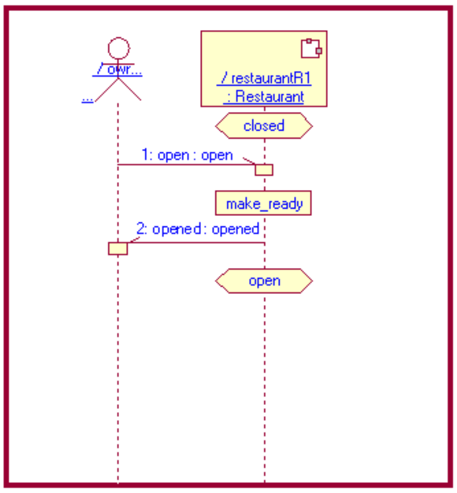


Figure 3.12: Local action visualizing internal communication

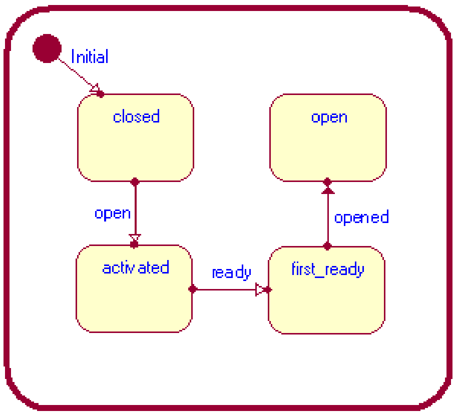
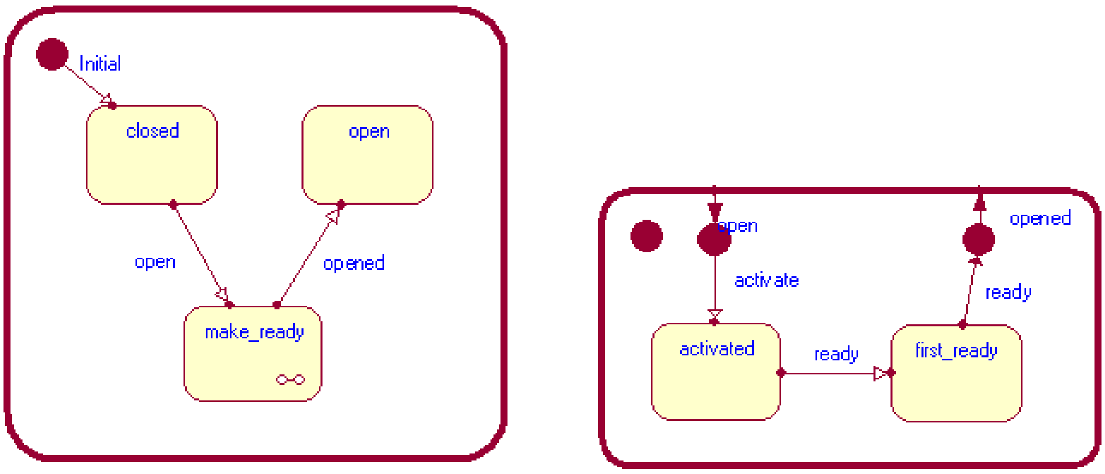


Figure 3.13: Flat *Bar* statechart



(a) Superstate for refined communication (b) Inside of *make_ ready*

Figure 3.14: *bar* with superstate *make_ ready*

4 Implementation

This thesis provides a tool that implements the theory presented in Chapters 2 and 3. This chapter describes the functionality and structure of this tool: the statechart composer (SCC).

Section 4.1 shows the general approach of the implementation. Section 4.2 describes the simulation of a sequence diagram on a statechart. Section 4.3 shows how the results of the simulation are used to integrate the sequence diagram into the statechart. Section 4.4 describes the introduction of hierarchy into a statechart with two examples.

4.1 General Approach

The algorithm shown in figure 4.1 takes a model `theModel` and one capsule `theCapsule` with an possibly empty statechart `theSC` as input.

```
0  main:
1   sds = collectSDs(theModel, theCapsule)
2   for each sd in sds
3      if not implements(theSC, sd) then
4         integrateIn(sd, theSC)
5      end if
6   end for
7   layoutTopological(theSC)
```

Figure 4.1: Pseudocode illustrating the general approach in the procedure `main`

In line 1 all sequence diagrams in `theModel` are collected in *sds* that include `theCapsule`. Then, in lines 2-6, for every sequence diagram `sd` in `sds` the algorithm checks (line 3) whether it is implemented in `theSC`. For this test `theSD` is simulated on `theSC` as described in the next section. If `sd` is implemented, nothing is done. Otherwise (line 4) the algorithm integrates `sd` in `theSC`.

So at line 7 `theCS` shows all behavior that is defined in `sds`, possibly more, if `theSC` was not empty at the start of `main`. In line 7 the call of `layoutTopological` optimizes `theSC` in introducing hierarchy.

`CollectSDs` (line 1) is an uncritical function that visits all sequence diagrams of `theModel`, checks, if the current `sd` includes an interaction instance of the capsule and adds it to the collection if needed. It returns the collected sequence diagrams. The interesting functions are `implements`, `integrateIn` and `layoutTopological`. The following sections 4.2, 4.3 and 4.4 describe their functionality.

4.2 Simulation

This section describes how the simulation works. Lets start with a closer look on the `implements` test that is called in `main`.

```
0 implements anSC anSD:
1  startStateSD = getStartState(getCorrespondingLifeline(anSD, anSC) )
2  startStateSC = getStartStateSC(startStateSD, anSC)
3  if startStateSD is nothing or startStateSC is nothing then
4    return false
5  else
6    return simulate(startStateSD, startStateSC)
7  end if
```

Figure 4.2: Pseudocode illustrating the `implements` test

`Implements` checks if `anSD` is implemented by `anSC`. At the first lines (1 and 2) three functions are called to set the start states for simulation. `getCorrespondingLifeline` returns the lifeline of the interaction instance that is associated with `theCapsule`, which is represented by its statechart `anSC`. `getStartState` returns the local state that is the first element on the lifeline as ensured by the basic framework, see Section 2.1.

`getStartStateSC` searches the statechart `anSC` for a state with a name *corresponding* to the name of the `startStateSD`. If such a state exists, it is returned. Otherwise nothing is returned and the if-statement in line 3 leads to line 4 where false is returned, because one of the arguments for `simulate` is not set. Note that in the second case the new state is not connected with the initial state and thus not reachable. That is a desired behavior to allow early feedback in the requirement or in the design phase. A closed result is only produced if the specifications are closed. As introduced in Chapter 2 *corresponding* means having an identical name.

Figure 4.3 describes the simulation. The function `simulate` returns a boolean `success` indicating whether the simulation successfully reached the end of the sequence diagram or not. `simulate` is called recursively in line 15. With each recursion one `(SDLocation, SCState)` pair is checked. Generally, a location on a lifeline can be a sd-state, a receive message or a send message. The function `simulate` is not called with `SDLocation` being a send message.

If such a *simulation step* is successful, the next pair is checked. In case of a non-deterministic choice, a backtracking is implemented via depth first search with the loop enclosing lines 6 to 18. This search terminates, because it is limited by the number of locations on the lifeline. For the result it is only important whether a way exists and not whether the simulation takes the shortest way, so a breadth first search is not needed here.

A simulation step corresponds to the interpretation of one transition in the statechart: If `SDLocation` is a local state, it has to correspond to the `SCState`. This is checked in line 2. If the states do not correspond to each other, `simulate` returns `false` (line 23).

In line 3 the `SDLocation` corresponds to `SCState` or is a receive message. A receive

```
0  simulate SDLocation, SCState:
1   success = false
2   if not (isState(SDLocation) and not correspondsTo(SDLocation, SCState)) then
3     if exits (getNextTriggerMessage(SDLocation) then
4       triggerMessage = getNextTriggerMessage(SDLocation)
5       sendMessages = getNextSendMessages(triggerMessage)
6       transitions = getTransitions(SCState, triggerMessage, sendMessages)
7       ready = false
8       while not ready do                'Backtrack point
9         if isEmpty(transitions) then
10          ready = true
11        else
12          transition = takeFirst(transitions)
13          nextSCState = getTarget(transition)
14          nextSDLocation = getNextNonSendSDLocation(SDLocation)
15          success = simulate(nextSDLocation, nextSCState)
16          ready = success
17        end if
18      end while
19    else
20       success = true
21    end if
22  end if
23  return success
```

Figure 4.3: Pseudocode illustrating the `simulation`

message means a change of state for the statechart as detailed in Chapter 2. Directly before each location that is a receive message `simulate` infers an implicit state, if no explicit state is given instead. In the case of an implicit state no check for correspondence is needed, because the state is not required explicitly.

If no further incoming message is required by the sequence diagram (line 3), the simulation is successful: All receive messages and respective send messages are handled and all explicit states have a correspondence in the statechart. Otherwise, the next receive message is set as `triggerMessage` in line 4.

In line 5 all messages that are sent after the `triggerMessage` and before the next non-send location are collected. In line 6 `getTransitions` collects all transitions originating from `SCState` that are triggered by the `triggerMessage` and send out exactly the messages given by `sendMessages`.

If the statechart is deterministic, at most one triggered transition is found. If none is found the simulation finds `transitions` empty in line 9 and the default value for `success`, `false` remains set. The loop is left (line 10 and 8) and the boolean `false` is returned in line 23. However, non-determinism is allowed so the number of transition matching the requirement may be arbitrary large. So the simulation takes the first transition (line 12) and keeps the transitions left to be followed in `transition`.

Now the next simulation step is prepared: In line 13 the `nextSCState` is set to be the source of the transition in the next simulation step. The function `getNextNonSend-`

`SDLocation` called in line 14 returns either a receive message or a sd-state, whatever of them follows the handled send messages.

If the recursive call is successful line 15 sets `success` to `true` and the backtracking while loop terminates to allow `simulate` to return `success`. Otherwise, the next transition in `transitions` is checked. If the set is empty, the `transitions` of the calling recursive instance of `simulate` are checked. If all such backtrack-points are handled in this way, the simulation ends.

During a run of `simulate` the following information is stored additionally:

- The last state to which a valid transition leads in simulation. This state is called *last valid state* and is not set, if no state was valid.
- The first input message that triggers no transition or a transition that does not lead to the expected state or a transition that does not cause the expected messages to be sent. If the simulation has not found the start state of the corresponding lifeline in `theSC`, this is the first message after the start state. Note that the basic framework ensures that this is an incoming message. This message is called *first invalid input message*. If the simulation is successful, it is not set.

Consider the main procedure shown in Figure 4.1 again. Line 4 is reached, when the simulation for the statechart `theSC` and the sequence diagram `sd` is not successful. `theSC` has to be extended to implement `sd`. This is done by the procedure `integrateIn`. The next section describes the integration of the behavior given by a sequence diagram into a statechart.

4.3 Integration

This section describes the integration of the behavior specified by a sequence diagram `theSD` into a statechart `theSC`. A first approach is to integrate the scenario over the start and end states. The *standard* SCC integration produces smaller statecharts by utilizing the result of a simulation as shown in Figure 4.4.

The procedure `integrateIn` adds behavior to `theSC` until it implements `theSD`. This is tested by the while-loop enclosing lines 2 to 14. The `implements` test in line 2 sets the `lastValidState` and `firstInvalidInputMessage` that are used in lines 3 and 4 to set the `SDLocation` representing an input message and `sourceState` representing an sc-state. If no such state exists (line 5), this is the first iteration of the while-loop and in line 6 a new state is added to `theSC` that bears the name of the start state in `theSD`.

Line 8 calls `getNextSCState` to set the `targetState` for the transition-to-add. `get-NextSCState` checks if an explicit state is required by `theSD`. If not, it infers the next implicit state. If no corresponding state exists in `theSC` it creates a new state.

Line 9 adds a transition with the name of `theSDLocation` as label to `theSC`. The next two lines set the trigger to be the message given by the `SDLocation` and the send actions defined by `theSD`.

Then line 2 tests again if `theSC` implements `theSD`. The procedure terminates when this test is successful.

```
integrateIn theSD, theSC:
 while not implements(theSC, theSD)
    SDLocation = getfirstInvalidInputMessage(theSD, theSC)
    sourceState = getLastValidState(theSD, theSC)
    if not exists sourceState
       sourceState = addState(theSC, getStartStateName(theSD))
    end if
    targetState = getNextSCState(sourceState, SDLocation)
    transition = addTransition(getName(SDLocation), sourceState, targetState)
    setTrigger(transition, SDLocation)
    setSendActions(transition, getNextSendMessages(SDLocation))
 end while
```

Figure 4.4: Pseudocode illustrating the procedure `integrateIn`

4.4 Hierarchy

The SCC relies on the assumption that no state name appears more than once in a statechart. Otherwise the identity in state names may produce wrong results, for example in attaching a branch to a wrong state. Currently the simplest implementation is used to choose the level on which to add a new state: choice 2 as introduced in Section 2.4. Every state is added under the same parent as the state from which the transition to add originates.

The Toplogical Layouter TL is implemented as part of the SCC. It is run after statechart generation to optimize the statechart. Two exemplary approaches to introduce hierarchy are implemented as described in Chapter 3. First the TL introduces generalization as described by π and afterward it applies Tarjans algorithm [31]. The generalization is handled first, because it may change the behavior defined by the statechart. The new superstates introduced for strongly connected components give only visual structure.

```
0  layoutTopological parentState
1   groupWithPartialOrder parentState
2   groupCommonSCCs parentState
3   childStates = getChildStates(parentState)
4   while exists(childStates) do
5      childState = takeFirst(childStates)
6      layoutTopological(childState)
7   end while
```

Figure 4.5: Pseudocode illustrating the procedure `layoutTopological`

The procedure `layoutTopological` is called for the top state of a statechart. In line 1 and 2 the procedures to introduce hierarchy into the child states are called. Other such procedures would be added here. In the following backtracking is used to call `layoutTopological` for each `childState`.

4.4.1 Partial Order

The partial order is represented by a text-file of the form shown in Figure 4.6. This example defines *open* as a parent state of *waiting_for_ wink*, *ask_for_ order*, *handout_ bill* and *serve_ cocktail*.

```
open > waiting_for_wink
open > ask_for_order
open > handout_bill
open > serve_cocktail
```

Figure 4.6: Example content of the text file representing π

This file has to be entered manually in the current version. Later versions may also add changes automatically. For example, if only $open > waiting\_for\_wink$ is entered manually before starting the SCC, the other three relation-pairs can be added as soon as the respective states are added to the statechart. For this example states have to be added according to choice 2, 3 or 4 as detailed in Section 2.4.

The CASE-tool provides a functionality `introduceSuperstate` to introduce a superstate. It takes a set of states that lie on a common layer as input and generates a new state on that layer. It places the states in the set within the new state. Each transition that is connected with one of the new substates is simply transformed into an inter-level transition.

```
0  groupWithPartialOrder oldParent
1   readPartialOrder
2   childStates = oldParent.getChildStates
3   for stateID = 1 to numberOf(childStates)
4      theState = childStates[stateID]
5      if not isMinimal theState then
6         for innerID = stateID to numberOf(childStates)
7            innerState = childStates[innerID]
8            if theState > innerState then
9               visitedStates.Add innerState
10              newSubStates = findNewSubstates(theState, innerState, visitedStates)
11              newSuperState = createSuperState newSubStates
12           end if
13        end for
14     end if
15  end for
```

Figure 4.7: Pseudocode illustrating the procedure `groupWithPartialOrder`

The procedure `groupWithPartialOrder` introduces hierarchy into the statechart as defined by the text file representing π. In line 1 this file is parsed and an object representing the partial order is created. In the following lines a state is searched that shall be a superstate. This is done by checking all `childStates` of the `oldParent` (line 3) if they are not minimal with respect to π (line 5).

If `theState` is bigger with respect to π than `theInnerState` (line 8), the `innerState` has to be a substate of `theState`. Otherwise, if it is bigger, it will be handled in a later iteration of the for-loop enclosing lines 3 to 16 and then `theState` will be found as a substate-to-be for it.

In Line 9 `innerState` is added to the `visitedStates`. The states are marked and passed by reference to `findNewSubstates` to be visited only once by `findNewSubstates`. `findNewSubstates` gathers the other new substates around `innerState`. Therefore it follows recursively the transitions originating from and leading to `innerState` as shown in Figure 4.8. In line 12 the `newSuperState` is created. Its children are the `newSubStates` as gathered by `findNewSubstates`.

```
0  findNewSubstates newParent firstChild visitedStates
1   newChildStates = new StateSet
2   newChildStates.Add firstChild
3   for each transition in getIncomingTransitions(firstChild)
4      theNeighborState = getSource(transition)
5      if newParent > theNeighborState and _
6       not existsIn(visitedStates, theNeighborState) then
7         visitedStates.Add theNeighborState
8         newStates = findNewSubstates(newParent, theNeighborState, visitedStates)
9         newChildStates = join(newChildStates, newStates)
10     end if
11  end for
12  for each transition in getOutgoingTransitions(firstChild)
13     if newParent > theNeighborState and _
14      not existsIn(visitedStates, theNeighborState) then
15        theNeighborState = getTarget(transition)
16        visitedStates.Add theNeighborState
17        newStates = findNewSubstates(newParent, theNeighborState, visitedStates)
18        newChildStates = join(newChildStates, newStates)
19     end if
20  end for
21
22  return newChildStates
```

Figure 4.8: Pseudocode illustrating the function `findNewSubstates`

`findNewSubstates` returns all states that are connected with the `firstChild` that are also less than the `newParent` with respect to π. It recursively checks all transitions originating from and leading to the `firstChild`. `newChildStates` is the set to store the return value (line 1). It is passed by reference to the recursive calls of `findNewSubstates`. In line 2 the `firstChild` is added to this set of `newChildStates`.

The for-loop enclosing lines 3 to 11 visits all states that are connected via transitions leading to the `firstChild`: If `theNeighborState` is smaller than the `newParent` (line 5) and it has not been visited before (line 6) it is added to the `visitedStates` (line 7) and handled with an recursive call of `findNewSubstates`. The `newStates` that have been found in this way are added to the `newChildStates` in line 18. In lines 12 to 20 the transitions originating from the `newChild` are visited in an according way.

4.4.2 Strongly Connected Components

The procedure `groupCommonSCCs` introduces a superstate for each maximum strongly connected component of a statechart with the parent state `oldParent` as shown in Figure 4.9.

```
0 groupCommonSCCs oldParent
1  theSCCs = getStronglyConnectedComponents(oldParent)
2  while not isEmpty(theSCCs) do
3     scc = takeFirst(theSCCs)
4     introduceSuperstate(scc)
5  end while
```

Figure 4.9: Pseudocode illustrating the procedure `groupCommonSCCs`

The procedure `getStronglyConnectedComponents` (line 1) takes a parent state as input and returns a set `theSCCs` of sets of states. Each of the sets of states in `theSCCs` defines a maximum strongly connected component. In each iteration the loop enclosing lines 2 to 5 takes one strongly connected component `scc` out of `theSCCs` (line 3). Line 4 introduces a superstate with the procedure `introduceSuperstate` as described above. The loop and the procedure `getStronglyConnectedComponents` terminate (line 2) as soon as all strongly connected components are consumed in that way.

5 Related Work

The generation and generalization of behavior from examples is a rather old discipline rooted in reverse engineering [5]. It has found application in forward engineering as well and can be used to reduce the amount of redundant information entered by humans as an early step in automated code generation. This chapter describes a selection of approaches to synthesis of statecharts from sequence diagrams and their impact on this work.

5.1 Model-Structure Based Synthesis

The approaches to statechart synthesis described in this section perform the synthesis directly on the model-structure or use a very similar intermediate structure. They interpret a projection of a sequence diagram on one capsule as a statechart fragment.

Krüger *et al.* [17] use the basic notation of sequence diagrams and a synchronous message-passing environment. Their algorithm projects each sequence diagram on the needed interaction instance. The resulting statechart fragments consist of one transition per incoming or outgoing message or local state. The sequence diagrams are connected via start and end states that are required to be given for each component. Afterward the resulting flat statechart is minimized. This approach is the base for the SCC-approach as detailed in Chapter 2.

Bordeleau [7] uses patterns to perform the transformation manually. His approach starts with *Use Case Maps* (UCM) and uses scenarios as an intermediate stage before defining one statechart for each component. This thesis transforms some of the patterns he presents into algorithmic approaches as detailed in Section 3. He claims automatic translation leads to artifacts that are too hard to understand to be of practical use. This thesis understands this as a challenge and goes another step toward human readable auto generated code.

Leue *et al.* [20] use *high level message sequence charts* (hMSCs) to express the relation between the sequence diagrams. They assume concurrent processes that are each "in exactly one bMSC[1] at any given point of time" as the SCC does. They assume a static, flat process structure, absence of timer events and consider local/global decisions, naming conflicts, deadlock situations and message overtaking. They present two algorithms, one for "maximum traceability" and one for "maximum progress". The maximum progress algorithm produces flat statecharts with a minimized set of states. The maximum traceability algorithm maps the structure of the hMSC/ bMSC relation to the statechart,

[1]basic message sequence chart

thus introducing one hierarchical level. Their maximum traceability approach is similar to the SCC-approach. This method to introduce hierarchy is discussed in Section 3.2.

Uchitel *et al.* [35] provide a framework with formal semantics in which existing approaches can be integrated. It is based on *Labeled Transition Systems* (LTSs), which are used to represent both sequence diagram projections and statecharts. Further work [34] discusses merging of partial behavior and the impact of architecture on statechart synthesis [33].

Even though the possibility to use the *Labeled Transition System Analyzer* (LTSA) is a benefit of their framework, the SCC-approach does not use it: It does not directly allow incremental approaches and would have complicated implementation.

According to the classification of Uchitel *et al.* [35] the SCC-approach uses a *design-oriented* semantics.

5.2 State-Variable Based Synthesis

The two approaches described in this section use an additional constraint language to enhance sequence diagrams by conditions on state variables. These conditions are used to infer states for the statechart. An ordering on the variables provides a means to introduce hierarchy.

Somé *et al.* [30] allow scenarios to make use of timers and to be combined overlapping, sequential, alternative and parallel. They use component variables to express conditions in sequence diagrams and to introduce hierarchy in the generated sequence diagrams. This concept is also used by Whittle and Schumann [36]. The result is a timed automaton in which non-determinism is allowed. The algorithm introduces hierarchy and proposes so called *synthetic* states and transitions to generalize the behavior of the automaton. The SCC-approach does not use timers yet and makes no use of state variables. The partial order on the explicitly named states the SCC uses to introduce hierarchy follows the approach proposed here.

Whittle and Schumann [36] identify three main issues for statechart synthesis as already stated in the introduction. They propose to use message pre and post conditions as a basis for statechart synthesis. In doing this they use the *object constraint language* OCL [23]. These conditions are expressed in terms of state variables. Between each pair of messages a state is added that is identified by its *state vector*, *i.e.* the vector containing the current values of the state variables. The states in the resulting statechart are merged when they have “similar” state vectors. A partial order on the state variables allows the introduction of hierarchy.

The SCC-approach does not use OCL constraints in sequence diagrams. This would put an additional burden on the designer. The start and end states the SCC uses are closer to normal system design: the designer has only to define in which state a system is before a sequence diagram applies.

5.3 Formal-Language Based Synthesis

The following approaches use the close relation of formal languages and automata for synthesis. Sequence diagrams are interpreted as words of a language that the statechart to compose has to accept.

Mäkinen and Systä [22] model the synthesis process as a language inference system and use Angluins [2] framework of the Minimal Adequate Teacher MAS. MAS overcomes the weakness of over-generalization that was part of their previous work SCED [15]. MAS is a tool that interacts with the developer in asking whether a path through a state machine shall be allowed and whether a conjectured statechart is acceptable. The additional information that is entered in this way allows MAS to generate a statechart that generalizes behavior according to the demands of the designer. The heart of MAS is an intermediate data structure between sequence diagrams and statechart, called *observation table*. Given an observation table, a statechart can be generated. This table is constructed from an arbitrary number of sequence diagrams or one statechart and an arbitrary number of sequence diagrams. The observation table is a compact notation of all words that are accepted by the statechart and can be enhanced incrementally. So it is possible to enhance an existing statechart with additional behavior based on new requirements in form of sequence diagrams.

The SCC-approach does not use direct questions to the developer during statechart generation. It makes generalization explicit via a partial order as detailed in Chapter 3. The approach of Mäkinen and Systä [22] is one of the few incremental approaches. Instead of the intermediate structure observation table the SCC uses simulation and integrates new information directly into the statechart. The SCC is able to handle hierarchical structures in manually altered statecharts.

Alur *et al.* [1] use a formal language definition of MSCs and automata. They use closure conditions to express realizability and the absence of deadlocks in MSCs and generate a flat automaton of the given MSCs. Their work also detects and shows implied scenarios that lead to a more complete specification and show the designer possible failures in their specification. The consistency check of the sequence diagram specifications is done in [21]. The present work assumes a correct specification. For the synthesis algorithm no formal language is needed.

Latronico and Koopman [18], as well as Alur *et al.* [1], make use of a formal grammar. They use compiler theory to generate deterministic statecharts. They identify three attributes of embedded systems: multiple initial condition, same user action invoking different system responses and timing sensitivity. They state that the combination of the three attributes leads to a need for further information to generate correct and deterministic statecharts. They make use of hMSCs and address sequential, conditional, iterative and concurrent execution of sequence diagrams. The use of hMSCs is an interesting aspect addressed by this work and may be included in later versions of the SCC. However, the SCC must be able to handle incomplete specifications and non-determinism, so that the benefits of the approach of [18] are rather small for it.

Okazaki *et al.* [24] use a subset of concurrent regular expressions to formulate their approach. The MSCs are translated into *concurrent regular expressions* and checked for consistency in-between. They assume a synchronous communication and interleaved execution in case of orthogonality inside of a statechart. As detailed in Section 1.2, the framework the SCC-approach operates in uses asynchronous communication and the consistency check is handled separately [21].

5.4 Synthesis of Statecharts from LSCs

Live Sequence Charts (LSCs) as introduced by [8] are a dialect of MSCs that also inspired the UML 2 notation of sequence diagrams.

The translation algorithm proposed by Harel and Kugler [11] does not scale well enough to be of practical use. Furthermore it relies on the *perfect information hypothesis*: every component knows all events from all other components. This hypothesis is not compatible with information encapsulating approaches such as the ROOM notation used in UML-RT.

The algorithm presented by Bontemps and Heymans [6] does not allow every set of sequence diagrams to be synthesized that can possibly be synthesized, but runs faster and does not rely on the perfect information hypothesis.

Kugler *et al.* [9] lately proposed a similar approach to handle his LSCs. As earlier [11] the resulting statecharts are flat, but concurrency is introduced. Due to the added notation in LSCs of "hot"—globally quantified—sections, the synthesis is considerably harder than with only normal—existentially quantified—sequence diagrams.

We decided against using the hot semantics because it clutters the sequence diagrams and makes them harder to understand. Ease of use is the key feature of sequence diagrams. However, the LSC semantic has more expressive power than the normal sequence diagram [8]. It may therefore be integrated in later versions of the SCC.

6 Conclusion

The previous chapters have discussed the environment and objective of this thesis (Chapter 1), an incremental approach to statechart synthesis from sequence diagrams (Chapter 2), means to simplify a statechart and to improve traceability in introducing hierarchy (Chapter 3), the implementation of the SCC-approach (Chapter 4) and its relations to other currently followed approaches (Chapter 5). This section puts these pieces together, discusses in how far the objectives are reached and what must be done in the future to improve implementation and theory.

6.1 Classification of the SCC-Approach

The SCC-approach performs statechart synthesis based on the model structure and interprets a projection of a sequence diagram as a statechart fragment. This statechart fragments are connected via start and end states as as done *e. g.* in the work of Krüger *et al.* [17].

In contrast to the other model structure based synthesis approaches the SCC is incremental. This allows direct reuse of existing structures. Besides conserving the former behavior the SCC preserves the hierarchical layout of the existing parts and changes the graphical layout only slightly. The preserved gestalt of the statechart has two effects: it reduces the time a developer needs to understand the changed statechart and it preserves the secondary notation [25].

The approaches based on state variables can introduce hierarchy in the synthesized statecharts. The partial order π on the states provides an alternative. On the one hand, the SCC-approach avoids the additional burden on the requirement engineer that state variables bring with them. On the other hand, the SCC approach cannot yet handle generalization as exactly as desired when new states are added to an hierarchical statechart. While the approach of Mäkinen and Systä [22] uses massive developer interaction at synthesis time to specify generalization the SCC avoids that. This allows fast feedback at a time, at which it is not yet clear how the behavior shall be generalized.

Besides statechart synthesis, the SCC can introduce hierarchy into an existing statechart. This simplifies the statechart and/or structures of the statechart in a way that makes it easier to trace the requirements. While simplifications like state-reduction in flat statecharts are done by most other approaches at synthesis time, hierarchy and traceability are supported by few [7, 20] of them.

6.2 Assessment of Results

The objectives for the SCC as listed in Section 1.1 are

- optimize development in terms of quality and time,
- automate manual work and reduce the input of redundant information,
- allow partial specifications and non-determinism,
- integrate smoothly in the existing work flow of an industrial software development team and
- introduce hierarchy into a statechart.

First experiences with the SCC-implementation indicate that it optimizes development in terms of quality and time. The instantaneous feedback in the requirement phase turns out to be useful: A run of the SCC after a change in a requirement sequence diagram makes critical areas visible. For example it shows whether the requirements are closed and whether the sequence diagrams express the idea of a statechart the requirement engineer has in mind.

The use of the SCC in the requirement phase can make this phase longer. Most of the additional time is needed to improve the requirements, not to use the SCC. However, further evaluation of the SCC is needed to gain reliable results. For such an evaluation the SCC has to be extended to handle more than the basic framework, see Section 6.3, to be adopted in a case study.

Further evaluation is also needed to find out in how far the SCC is suited to the generation of implementation statecharts. At the moment it is not clear how much work is left for the developer in the statechart and how much of the work flow is automated by the SCC. When the SCC is used less information has to be entered twice and the risk of introducing faults is reduced. However, this depends on a more complete implementation of the SCC.

As discussed in Section 2.4 this thesis can provide no optimal solution for the question of choosing the right parent for a state-to-add.

The SCC-approach handles non-determinism with a backtracking strategy and accepts partial requirements. However, both aspects are only partly implemented.

The SCC integrates into the work flow used of the software development team at Philips Medical Systems. The introduction of local start and end states on each lifeline is little extra work, because local states are already used in such a manner. The amount of work is less than it would have been when another method like state variables as proposed by Whittle [36] would have been used to connect the sequence diagrams.

After being graphically laid out the resulting statecharts are quite readable and hierarchy can be introduced. Again, further evaluation is needed.

6.3 Next Steps for Implementation

The current implementation of the SCC-approach includes only the basic ideas. Generally two paths can be followed for further implementation:

- complete the SCC for industrial use;
- prototype more ideas.

To complete the SCC for industrial use, simulation and integration have to be extended to work with all syntax elements in sequence diagrams and statecharts. For example target language code and history connectors have to be treated. Another important issue for practical use is graphical layout. The CASE-tools layout capabilities strongly suggest the use of external or manual layout. A layout tool such as KIEL [32] is not yet connected with the CASE-tool. As discussed in the next section, naming conventions are an unsolved issue.

All ideas presented in this work that are not yet implemented could be prototyped: Perhaps most interesting is the possibility to insert new states according to all four choices detailed in Section 2.4. Aside from that the different approaches to introduce hierarchy into a statechart can be added.

Beside that other applications of the modules used in the SCC are possible: The simulation of sequence diagrams on statecharts could be extended to a stand-alone operation or to perform coverage analysis. In simulating each specification diagram on a given statechart and marking the visited states and transitions the parts of a statechart that are not specified with sequence diagrams become visible.

6.4 Next Targets for Theory

The most important open question in automated statechart generation is the naming of new states. This problem arises with both new superstates as well as new normal states. The state names are important to understand a statechart. Similar to other *secondary notation* such as graphical layout, inappropriate state names can complicate the understanding of a statechart [25].

As proposed in other approaches [22, 1] incomplete specifications may lead to implied scenarios not obviously visible to the developer. Integrating techniques to detect such scenarios and to propose appropriate generalizations for the generated statecharts would complete the SCC-approach.

The ability to create statecharts from specifications that demand concurrent behavior would widen the field of use of the SCC. In the ROOM framework orthogonality is no syntax element for statecharts, so sub-capsules have to be introduced.

An approach that can automatically generate sub-capsules first has to identify whether a sub-capsule is needed in the given situation. For some simple concurrency problems, self-transitions to history on the parent state can be used. If a sub-capsule has to be used, the generator has to take care of communication with the parent capsule.

The application of design patterns could optimize this and other cases. If design patterns are part of the development policy, the approach to statechart synthesis has to be capable of generating statecharts that are compliant to these patterns.

The order in which sequence diagrams are integrated into a statechart is another field for improvement. For example, when basic flows are integrated first, and then hierarchy according to the use case structure is introduced and afterward alternative flows are added, the traceability might be enhanced.

The SCC-approach provides a good foundation for these extensions.

Bibliography

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of Message Sequence Charts. In *IEEE Transactions on Software Engineering*, volume 29, pages 632–633, 2003.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[3] Artisan Software. Available from World Wide Web: `http://www.artisansw.com/`.

[4] Twan Basten and Wil M. P. van der Aalst. Inheritance of Behavior. In *Computing Science Report*, volume 17, Eindhoven University of Technology, Eindhoven, 1999.

[5] Alan W. Biermann and Ramachandran Krishnaswamy. Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, 1976.

[6] Yves Bontemps and Patrick Heymans. As Fast As Sound (Lightweight Formal Scenario Synthesis and Verification). In H. Giese and I. Krueger, editors, *third International Workshop on Scenarios and State Machines: algorithms, models and tools, workshop co-located with ICSE'04*, pages 27–34, Edinburgh, 2004. IEE.

[7] Francis Bordeleau. *Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines.* PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, 1999.

[8] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[9] Hillel Kugler David Harel and Amir Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In H.-J. Kreowski et. al., editor, *Formal Methods in Software and System Modeling*, volume 3393, pages 309–324. Lecture Notes in Computer Science, Springer-Verlag, 2005.

[10] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[11] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *CIAA '00: Revised Papers from the 5th International Conference on Implementation and Application of Automata*, pages 1–33, London, UK, 2001. Springer-Verlag.

[12] Cornelis Huizing and Rob Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314. Springer-Verlag, 1992.

[13] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC99). Geneva, 1999.

[14] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process.* Addison-Wesley, 1999.

[15] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi. SCED: A tool for dynamic modeling of object systems. Technical Report A-1996-4, Department of Computer Science, University of Tampere, 1996.

[16] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison Wesley, 2003.

[17] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[18] Elizabeth Latronico and Philip Koopman. Representing Embedded System Sequence Diagrams As A Formal Language. *UML 2001, Toronto Ontario*, pages 302–316, October 2001.

[19] Edward A. Lee. What's Ahead for Embedded Software? *Computer*, 33(9):18–26, September 2000.

[20] Stefan Leue, Lars Mehrmann, and Mohammad Reza. Synthesizing ROOM Models from Message Sequence Chart Specifications. In *13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, 1998.

[21] Andrea Lischke. Consistency Check of Sequence Diagrams. TU Cottbus, To be published in October, 2005.

[22] Erkki Mäkinen and Tarja Systä. MAS—An Interactive Synthesizer to Support Behavioral Modelling in UML. In *ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24, 2001.

[23] Object Management Group. Available from World Wide Web: `http://www.omg.org/`.

[24] Mitsutaka Okazaki, Toshiaki Aoki, and Takuya Katayama. Formalizing sequence diagrams and state machines using Concurrent Regular Expression. *Proceedings of 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 74–79.

[25] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[26] Steffen Prochnow and Reinhard von Hanxleden. Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technischer Bericht 0406, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, June 2004.

[27] Rational Rose Realtime. IBM. Available from World Wide Web: `http://www.rational.com/rosert`.

[28] Rhapsody. I-Logix, Inc. Available from World Wide Web: `http://www.ilogix.com/products/`.

[29] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[30] Stephane Somé, Rachida Dssouli, and Jean Vaucher. From Scenarios to Timed Automata: Building Specifications from Users Requirements. In *1995 Asia-Pacific Software Engineering Conference*, pages 48–57, 1995.

[31] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160.

[32] The KIEL Project. Project Homepage, 2004. Available from World Wide Web: `http://www.informatik.uni-kiel.de/~rt-kiel/`. Kiel Integrated Environment for Layout.

[33] Sebastian Uchitel, Robert Chatley, Jeff Kramer, and Jeff Magee. System Architecture: the Context for Scenario-based Model Synthesis. In *ACM International Symposium on Foundations of Software Engineering*, pages 33–42, New York, NY, USA, 2004. ACM Press.

[34] Sebastian Uchitel and Marsha Chechik. Merging Partial Behavioural Models. In *ACM International Symposium on Foundations of Software Engineering*, pages 43–52, New York, NY, USA, 2004. ACM Press.

[35] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[36] Jon Whittle and Johann Schumann. Generating Statechart Designs From Scenarios. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 314–323, New York, NY, USA, 2000. ACM Press.