Design and Code Generation Concepts for Statechart Diagrams of the UML v1.1 in Concurrent Environments

Pierre Metz
Department of Maths and Computing, Cork Institute of Technology / Department of Computing, University of Applied Sciences, Darmstadt.

Follow this and additional works at: https://sword.cit.ie/allthe

Part of the Software Engineering Commons, and the Systems Architecture Commons

Recommended Citation

P. Metz

Master of Science
1999
"Design and Code Generation Concepts for Statechart Diagrams of the UML v1.1 in Concurrent Environments"

P. Metz

M.Sc.

1999
"Design and Code Generation Concepts for Statechart Diagrams of the UML v1.1 in Concurrent Environments"

Pierre Metz, Dipl.-Inform. (FH)

Master of Science

Department of Mathematics and Computing
Cork Institute of Technology

Department of Computing
University of Applied Sciences, Darmstadt

Supervisors:
John O´Brien
Dr. Wolfgang Weber

Submitted to the National Council for Educational Awards,
May 1999
Pierre Metz

"Design and Code Generation Concepts for Statechart Diagrams of the UML v1.1 in Concurrent Environments"

Abstract

This thesis deals with code generation concepts for the statechart diagrams of the UML v1.1. It explores implementation approaches for mapping statechart diagrams to code. It also focuses on modelling rules for consistent design of dynamic object behaviour.

Most of the given object-oriented (OO) methods have statechart diagrams as their centre. It is proposed to use statechart diagrams to model the internal behaviour of objects and lifecycles, respectively. Generally, statechart diagrams are used in the area of analysis and design of reactive systems such as realtime- and embedded-systems. This thesis introduces different existing implementation patterns and shows how they can be applied. Common features and differences of the implementation concepts are shown; advantages and disadvantages in the context of criteria such as runtime performance, memory requirements, simplicity and extensibility are presented. The author also introduces an alternative implementation approach which compensates for the identified disadvantages. This approach covers the complete syntax of UML, including parallel processing of state activity.

This thesis further deals with concurrency aspects of object-oriented software. It introduces mechanisms for transparent interaction of concurrent objects. In particular, an implementation concept for event queueing is developed. The UML offers to postpone events for deferred handling when the object takes a state for which the event is not defined. These patterns serve as an extension for the presented implementation concepts in the context of concurrent environments.

The introduced implementation patterns are considered to serve for the realization of code generators for CASE-tools but may also be applied manually. In object-oriented software development, code generation facilities are meaningful and play a very important role as they are improving productivity and produce correct and uniform code. The use of advanced code generators helps saving time for programming which can be spent on analysis and design.

Code generation requires consistent and correct models. However, with respect to consistency constraints, the UML semantics are not strict enough. Following the foundations of the UML semantics only, inconsistent statechart diagrams may occur. The author fills in this gap by defining suitable modelling rules and consistency constraints.
Acknowledgements

After my graduation in March 1997 at the University of Applied Science in Darmstadt, Germany, the completion of my master thesis represents the second great success of my academic career. I have expanded my knowledge about computer science, object-orientation in particular, and I have become much more experienced in conceptual thinking and doing research. I want to thank the people who made all this possible.

At first, I wish to express my greatest thanks to my parents who have always been supportive of me, both morally and financially over years. The successful completion of my master thesis would not have been so easy without their help. Therefore, I dedicate this thesis to my dearest parents.

I would like to add special thanks to the following friends and colleagues of mine:

Jörg Frey for all his hints and knowledgeable discussions about object-orientation, state semantics of objects in particular. Also thanks to Kai-Uwe Schäfer, my second companion concerning interchanging ideas and visions about object technology. I joyfully remember our countless sessions and talks.
Thanks to Annika Wille who made her knowledge about OLE technology available for me. I was able to use her work as a basis for the implementation part of my thesis.
Thanks to all employees and associates of Wibas GmbH in Darmstadt. When working with Wibas, there is always a lot of valuable input in the field of object-oriented concepts and methodologies.
Thanks for all the fun and the work we have done together to all postgraduate students and members of the Lab for Distributed Applications of the University of Applied Sciences, Darmstadt. It was a great time indeed.

Last but not least, my sincerest gratitude goes to John O'Brien, my supervisor at the CIT and Wolfgang Weber, my supervisor at the University of Applied Sciences, Darmstadt. I am very thankful for their cooperation and help. Both John O'Brien and Wolfgang Weber have always supported and encouraged me a lot and "are responsible" for my personal improvement in doing research as mentioned above. It was a true pleasure for me to work with them.
Figure List

Chapter 1

figure 1: lifecycle model of object-oriented software development [Oestereich 97] 3
figure 2: object-oriented development phases as views on the system 3
figure 3: links between the chapters 9

Chapter 2

figure 4: lifecycle of a worker 12
figure 5: class structure Shlaer/Mellor 13
figure 6: state transition following Shlaer/Mellor 15
figure 7: structure of the design pattern 'State' 19
figure 8: state transition following the State pattern 19
figure 9: structure of the design pattern 'State Table' 22
figure 10: state transition following the State Table pattern 23

Chapter 3

figure 11: statechart diagram of a compact disc player 33
figure 12: statechart diagram of a compact disc player with an automatic transition 34
figure 13: declaration of class CD_player 35
figure 14: UML syntax for state transitions 37
figure 15: example for transitions with conditions, actions and message-sending 37
figure 16: example for a statechart diagram with entry- exit- and do-activities 41
figure 17: sample statechart diagram with nested states and history markers 50
figure 18: example for concurrent substates 62
figure 19: concurrent substates with only one simple state 63
figure 20: transition from an initial state with an external event 65
figure 21: automatic transition from the initial state 66
figure 22: automatic transition to a final state 68
figure 23: transition with an external event to a final state 69
figure 24: example for the second modelling rule for statechart diagrams 77
Appendices

figure 49: extension of the class model in figure 30

figure 50: simulation of a virtual constructor

figure 51: self-defined exception class hierarchy

figure 52: Use Case Diagram

figure 53: GUI Design of the Use Case 'Generate Code'

figure 54: System Notification

figure 55: System Architecture Design

figure 56: Analysis Class Diagram

figure 57: Scenario of Visiting the System Structure following the design pattern 'Visitor'

figure 58: Visitor Registry Extension of the Scenario in figure 57

figure 59: Class Hierarchy

figure 60: Extension of the Basic Class Diagram (Part I)

figure 61: Extension of the Basic Class Diagram (Part II)
Modelling Constraint List

Modelling Constraint 1: Interruptable Activities ................................................. 48
Modelling Constraint 2: Parameters of Events ...................................................... 71
Modelling Constraint 3: Conditions of Outgoing Transitions ............................. 72
Modelling Constraint 4: Automatic Transitions .................................................... 72
Modelling Constraint 5: Automatic Transitions and State Regions ....................... 73
Modelling Constraint 6: Avoiding Exceptions During Transitions ....................... 74
Modelling Constraint 7: Automatic Transitions and Initial and Final States .......... 77
Modelling Constraint 8: Facades for Architectural Packages of Objects ............. 126
1 Introduction

1.1 Goals, Objectives and Foundations

The aim of this thesis is to explore implementation approaches for mapping statechart diagrams of the UML v1.1 to code. Furthermore, a set of modelling rules for guaranteeing consistent and correct statechart diagrams are developed. Since UML does not provide any direction on how to apply the models in practice and, concerning the metamodel, even lacks strict consistency constraints, this is a necessary analysis step for proposed implementation using UML. Based on the implementation concepts and the modelling rules, a prototype of a code generator is implemented.

The UML is a graphical notation for object-oriented software development. It was developed by Grady Booch, Ivar Jacobson and James Rumbaugh of Rational Software Corporation, Santa Clara, USA. The authors unified the notations of their own methods. The UML was proposed to be adopted by the OMG as a standard for object-oriented modelling language. The OMG (Object Management Group) is an international consortium of industrial and academic partners. Its task is to create standards for object technology. The OMG confirmed the UML as a standard notation in the middle of 1997 in the version 1.0.

The UML offers only notation; it does not impose a method. The developer is free to combine the diagrams of the UML according to the metamodel to cope with the domain requirements. As an example, it is possible to assign statechart diagrams and activity diagrams to classes, operations and use cases.

Statechart diagrams are a means of modelling the internal dynamic behaviour of objects. Applied to a class definition, a statechart diagram shows the influence of events and messages on the objects of that class over time. It shows what states an object has, to what events and messages the object responds and how states are changed as a result of received events and messages. Moreover, a
statechart diagram defines the concrete action that is executed when an event or a message is received and accepted. A statechart diagram represents the lifecycle of an object from creation to destruction. An object is born, passes several states and dies.

Statechart diagrams have evolved from David Harel's Statecharts. Harel introduced Statecharts as a formal language for the formal specification of finite state machines. He originally intended to use them for process flows. Several authors of books about object-oriented methods like Rumbaugh and Shlaer/Mellor soon adopted and modified Harel's Statecharts for state-dependent object behaviour specification.

1.2 Background - Views On Object-Oriented Systems

Today, one of the glaring needs of the software industry is quality improvement. In literature and on conferences, the term 'software quality' is discussed passionately. Books about software quality fill libraries. Areas of software quality improvement are widespread. It concerns requirements analysis, modelling of software, implementation and project management. Object-oriented software, in particular, requires different project management, a different way of abstraction, new modelling notations and a new way of programming.

As in traditional software development, object oriented software development can be distinguished in phases as there are requirements analysis, object-oriented analysis (OOA), object-oriented design (OOD) and implementation.

Object-oriented development is incremental and iterative; that is the phases are passed sequentially but may be repeated several times adding more detail (see figure 1). An incremental and iterative process is suitable for object oriented development with its conceptual foundations such as classes including inheritance semantics, objects, properties, operations and associations comes near to human perception of the real world. Object-oriented modelling notation adopts this perception offering suitable modelling elements and diagrams. These diagrams and model elements remain the
same within the development process, they are invariant. There is no change of notation and concepts when stepping from analysis to design.

Using traditional methods, there is a greater gap between analysis and design models and the structure of the real world. As an example, Structured Analysis (SA, DeMarco) offers data flow diagrams which may be refined by Structure Charts (Constantine) in the Structured Design (SD). The resulting code encompasses functions and variables, which is conceptually far away from the real world-structure. Regarding object-orientation, entities of the real world are obviously better recognized at the modelling level as well as in the code.

![Lifecycle model of object-oriented software development](image1)

*figure 1: lifecycle model of object-oriented software development [Oestereich 97]*

![Object-oriented development phases as views on the system](image2)

*figure 2: object-oriented development phases as views on the system*
In order to map a certain part of the real world (domain) to a software system, a model has to be created. During requirements analysis, the domain requirements are abstracted from the real world. Abstraction means reduction of information (see figure 2). Pieces of information that are required to capture the domain requirements are preserved. The rest is not relevant and can be ignored. The results of the requirements analysis must be accurately captured and transferred into a precise notation (OOA). The corresponding work products are the use case model and the analysis model. The analysis model captures the static aspects (components and their relationships) and dynamic aspects (interaction between the components and their internal behaviour) that meet the problem requirements.

In OOD, information is added to the analysis model. The model is expanded with implementation concepts. This may be concurrency, design patterns, determining the distribution of software components etc. The result of OOD is a model of greater detail saying that there are additional classes and relationships including dynamic behaviour specification. It shows how the system will be implemented. Both the analysis model and the design model are completely independent of any programming language or environment.

Implementation is the last step of development. The OOD model is translated into a particular programming language.

The different phases of object-oriented development may be considered as different views on the software system. We understand a view as a certain level of abstraction, such as OOA and OOD. The OOA model is a subset of OOD; the OOD model bears additional detail. In figure 1 the links between the particular views are shown.

In order to switch between the views, we must be able to derive one from another. This requires consistency of the model and the product, and semantically correct mapping rules also for the elements of the underlying notation. A particular type of diagram is used to refine the information of another one; model elements are linked to each other. This is the first step towards powerful
consistency checkers and documentation facilities and thus to improvement of software quality and software development in general.

The method OOSE (Object Oriented Software Engineering) of Jacobson [Jacobson 92] explains the margin between analysis and design models. The commercially available CASE-tool 'Objectory' supports OOSE and thus the distinction between OOA and OOD.

The idea of consistent mapping between software views can be extended to the translating from OOD to implementation and reverse. Once the resulting model is described very precisely (OOD), code generation is possible. Code generation facilities produce high quality code because it is uniform and correct. Advanced code generation minimizes implementation mistakes and increases documentation as well as readability of code. Moreover, time for implementation and testing is reduced which is of high interest for the project management. This has direct impact on the project calculations concerning costs and time. Another apparent benefit for software quality is that the saved time can, and should, be spent on OOA and OOD. Code generation facilities of CASE-tools is a very important and necessary support for maximizing productivity of the software industry (compare [Douglass 98]).

Ideally, code should be isomorphic to the design model. Consistency is also necessary for matching OOD work products with the implementation. Modifications on either side should be reflected by the other. Choices within the programming domain become design decisions in the model and vice versa. The opposite action of code generation is called reverse engineering. This means creating a model from the source code. The combination of code generation and reverse engineering is called round-trip engineering. Round-trip engineering requires derivation rules and mapping constraints for keeping OOD models and code consistent.

With code generation facilities, development time is transferred from implementation to analysis and design. Programming now happens "at the modelling level". Implementation is no longer performed using programming languages only but, rather, by language-independent modelling notations. This leads to a more detailed and consistent model of the domain. However, there is a
price to pay. Problems are shifted: code generation requires highly detailed and consistent models. This being the case, modelling is more critical than before. In addition to programming skills, software engineers must have an in-depth knowledge of the OO paradigm, development methods and notations. They must know about well-proven modelling concepts and patterns. Education becomes highly important. Also, team-working and inter-personal communication skills are required. A good software system can only be built by teams, consisting of both software engineers and domain experts.

Another interesting consequence of considering views on OO-software is the interchange of model data. If we can interpret a model and code as being "the same" because of consistent mapping rules, the term "reuse" is expanded to models. If a model or a part of a model can be exchanged between CASE-tools, it can be reused. Considering code generation and round-trip engineering, this implies implicit reuse of code.

The Interchange of models requires a standardized notation for object-oriented modelling. Since the UML-metamodel was adopted as a standard notation for object-oriented modelling by the OMG in 1997, this goal has been achieved. The standardization proposal included a UML CDIF-metamodel based on the EIA/CDIF\(^1\) interim standard (see [Flatscher 98]). This facilitates to interchange UML model data from CASE-tools of different vendors.

### 1.3 Structure of the Thesis

For reading this thesis, the reader should be familiar with the full specification of statechart diagram syntax and semantics defined by the UML v1.1 ([UML 97c], [UML 97d]). Knowledge of common object-oriented methods such as OOSE (Object Oriented Software Engineering,

---

\(^1\) In 1987 the "CASE Data Interchange Format" (CDIF) committee was founded within the American Electronic Industries Association (EIA) to create a set of standards allowing the exchange of model data between different CASE-tools. The interchangeability is ensured by the definition of the CDIF meta-metamodel on which particular CDIF metamodels, such as the UML CDIF-metamodel or metamodels of other methodologies, are based.
Chapter 1

Introduction

I. Jacobson) and OOSA (Object Oriented Systems Analysis, Shlaer/Mellor) is helpful. All figures throughout this thesis are based upon UML notation v1.1.

The following paragraphs give a short summary for every chapter. The links between the particular chapters are shown graphically in figure 3.

Chapter 2: Existing Implementation Approaches

Chapter 2 examines existing approaches for implementing statechart diagrams. Based on a small example, each implementation approach is explained. The structure as well as the collaboration of the components of the particular concept are shown. All concepts propose how to represent states, events and transitions at the design stage of object-oriented software as well as in code. They explain how a transition is made from one state to another by the input of an external event.

Chapter 2 closes with a discussion of identified problems with the presented implementation techniques. The comparison covers issues including runtime performance, memory requirements, simplicity and extensibility.

Chapter 3: Alternative Implementation Approach

Based on the findings of chapter 2, chapter 3 introduces an additional implementation approach for statechart diagrams which cannot be found in literature. This alternative approach aims at addressing the problems identified in chapter 2. As neither existing implementation approach offers ideas for the full statechart diagram specification of the UML v1.1, the alternative approach fills in these gaps. Each element of UML's statechart diagram notation is explained briefly before an implementation proposal is introduced.
Chapter 4: Code Generation Concepts for Multithreading

The implementation approaches introduced in chapter 2 and chapter 3 present basic concepts of mapping events, states and transitions to classes, attributes and operations. These concepts assume the presence of a Single-Task Operating System. Consequences of possible concurrency are not considered.

Chapter 4 deals with issues like multithreading and synchronization requirements. It shows different ways of introducing multithreading to object-oriented software and what consequences these issues have on the internal consistency of objects regarding state semantics. It proposes implementation strategies for protection against such effects. Furthermore, it deals with interaction of concurrent objects. In the presence of concurrency, inter-object communication can take place by exchanging either synchronous or asynchronous messages.

The implementation strategies of chapter 4 can be applied to any basic implementation approach for statechart diagrams presented in chapter 2 and 3. Those code generation concepts are made multithread safe. Further, chapter 4 serves as the basis for chapter 5 which develops an event queueing mechanism for concurrent objects.

Chapter 5: Queueing of Events

Object message-interchanges cause events. State semantics of objects say that some of these events may only be accepted in particular states of an object. There may be states in which certain messages and events cannot be responded to. In this case, there are two possibilities to dealing with these events. First, the event can simply be ignored. Second, the event may be saved for deferred handling when the object arrives in a state for which the event is defined. Chapter 5 extends the concepts of chapter 4 by adding event queueing capabilities. It is shown how the queueing of events can be integrated in the interaction patterns of concurrent objects. These extensions require a slight but necessary adaption of the basic implementation technique introduced in chapter 3. These consequences are explained in detail.
Chapter 6: Code Generation Concepts for Multiprocessing

The issues introduced in chapter 4 address concurrent object interaction only in case of a multithread-application that runs within one process. Chapter 6 expands the problems of inter-object communication in a multi-process application. These issues are only discussed briefly as this is beyond the scope of this thesis. Chapter 6 serves as a completion of concepts for object interaction in the presence of concurrency; concrete problems that arise are intentionally left open.

Chapter 7: Outlook and Further Research

Chapter 7 proposes ideas for further research based on the findings of this thesis. These ideas address implementation concepts as well as logical concepts.

Suggestions for the expansion of the code generation concepts and the code generation prototype itself are outlined. Furthermore, ideas for the possible improvement of OO methods, in particular in the areas of OOA and OOD, are outlined. The discussion addresses the issues of state-semantics identification and the modelling of the internal behaviour of objects and their interaction including priority-concepts for messages and events.

*figure 3: links between the chapters*
2 Existing Implementation Approaches

2.1 Shlaer/Mellor [SM 92]

The method OOSA (Object Oriented System Analysis) defined by Shlaer/Mellor focuses on the analysis and design of realtime problems. It puts emphasis on concurrency, synchronous vs. asynchronous communication and timer control. Dealing with software architecture layers is part of the method.

OOSA defines an evolutionary development process consisting mainly of the following three steps:

The Information Model, the State Model and the Process Model.

The information model describes objects and classes together with their relationships and attributes. That is, compared with the UML, a class diagram.

The State Model concerns the behaviour of objects and relationships over time. Each object and relationship has a lifecycle which defines its dynamic behaviour. Shlaer/Mellor use statechart diagrams as well as State Transition Tables (see section 2.1.1) for formalizing lifecycles. Entities like objects and relationships communicate with each other by means of events.

All of the processing required by the domain problem is contained in the State Model. The Process Model refines all action that is defined in the State Model by an enhanced form of traditional data flow diagrams. The Process Model aims at implementing methods and operations at an abstract level. Compared with the UML v1.1, the Process Model comes near to Activity Diagrams.

2 The original term for the description of dynamic behaviour used by Shlaer/Mellor is “State Transition Diagram”. As this thesis is based on the UML v1.1, I will use the term “Statechart Diagram” throughout this thesis in order to reduce “terminology terror”.
2.1.1 State Transition Tables

In order to be able to prove the consistency and completeness of statechart diagrams more easily, Shlaer/Mellor recommend modelling a state transition table after having modelled the lifecycle of a class.

A state transition table is a matrix. It is modelled for each statechart diagram. The columns represent events while the rows represent the states of the statechart diagram. A cell of the matrix specifies the reaction of an object to the occurrence of an event in the particular state.

A cell can have one of the following predefined entries:

- a number:
  Following Shlaer/Mellor, states as well as events must have a distinct number as well as a name. The state numbers are used for table entries. A state number represents the target state of a transition that is fired if the corresponding event occurs.

- "event ignored":
  The event does not have any impact on the object. The object remains in the current state and does not execute any state activity; that is the event is ignored and discarded.

- "can't happen":
  "can't happen" is entered if the occurrence of an event is not possible with respect to the real world. It helps the modeller to achieve a better understanding and clarity of his model. For detailed explanation footnotes for "can't happen"-entries are recommended. If an event occurs in a state for which a "can't happen" entry exists, this is considered as an error of the application logic. The implementation of the statechart diagram may throw an exception then. This serves as a "runtime debugging" facility for the software engineer.
Example:

![Lifecycle of a worker diagram]

**Figure 4: Lifecycle of a worker**

State transition table:

<table>
<thead>
<tr>
<th></th>
<th>1: engagement</th>
<th>2: disengagement</th>
<th>3: take a holiday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. unemployed</td>
<td>2</td>
<td><em>can’t happen</em>¹</td>
<td><em>can’t happen</em>²</td>
</tr>
<tr>
<td>2. employed</td>
<td><em>event ignored</em></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3. on holiday</td>
<td><em>event ignored</em></td>
<td>1</td>
<td><em>can’t happen</em>³</td>
</tr>
</tbody>
</table>

*¹: A worker cannot be disengaged if unemployed
*²: A worker must be employed if requesting a holiday
*³: Holiday is already taken

2.1.2 Implementation

Shlaer/Mellor introduce the term *Active Class* as a class of the problem domain modelled at the analysis level that has a non-trivial lifecycle. This means, its behaviour is more complex than just creation and destruction. A class with a trivial lifecycle is called a *Passive Class*. In the example in figure 5, the class worker is an active class.

Each active class defines public operations for the events that can be received. This results from the interaction identified by the software engineer when coming up with an OOA model. If an event is sent to an object, the corresponding operation is called.

For the implementation of statechart diagrams, Shlaer/Mellor derive the following elements (see figure 5). They define an abstract class *ActiveClass*. It represents a high-level type being able to
perform state transitions. It defines a class operation \texttt{performTransition()}. This operation communicates with a state machine\textsuperscript{3} instance. Further, class Active Class defines an attribute in which the current state is remembered. Each domain class with a non-trivial behaviour inherits from ActiveClass. Instances of this concrete subclass of Active Class is also called "Context Object".

For each cell of the state transition table, which only exists at the modelling level, an instance of class Transition is created. These instances are stored in a list owned by an instance of classStateMachine, which belongs to every Context Object. The only task of a State Machine is to traverse the list of transitions in order to determine whether the Context Object must react to the occurred event. The traversing is initiated after the Context Object has received an event. When traversing, the State Machine calls the operation \texttt{findNewState()} of each Transition instance and submits the current state and the current event number (both pieces of information are delivered by the Context Object). The Transition checks if it is responsible for this combination by

\textsuperscript{3}A common verbal distinction between the specification of internal behaviour and the execution of that behaviour at runtime is statechart diagram and state machine: A statechart diagram is attached to a class and represents the specification of a state machine like a class definition represents the specification of an object instance. Thus a state machine can be interpreted as an instance of a statechart diagram at runtime. Each object of a class having a statechart diagram owns a state machine.
inspecting its attribute values of sourceState and event. The Transition instance returns the value of its attribute targetState which can be one of the following:

- \( > 0 \):
  
  identification of the target state;

- \( 0 \):
  
  indicates an ignored event as defined in the state transition table;

- \( -1 \):
  
  indicates an error. This situation refers to a "can't happen" entry in the state transition table.

As a state machine determines a behaviour template of all Context Objects, the state machine instance is a shared aggregation\(^4\) (see figure 5). A state machine can be referenced by more than one Context Object. The list of transitions is a composite aggregation as it defines the reaction of the state machine to events.

For improving the runtime performance, Shlaer/Mellor suggest to merge the State Machine class and Transition class into one. The resulting class represents a state transition table directly; traversing a list of transitions is not necessary.

*Interaction of the entities shown in figure 5:*

The worker object Jim receives the event *engagement*. The corresponding operation calls the class operation `performTransition()` and passes three parameters: the reference to itself, the reference of its state machine object and the number of the event *engagement*.

\(^4\) For explanation of composite aggregation and shared aggregation see [UML 96] and [UML 97c]. The method OOSA does not distinguish between aggregation and association, so shared aggregation is not defined. Nevertheless, in figure 5 shared aggregation is used for a better understanding: Shlaer/Mellor say that a state machine instance belongs to exactly one active class and defines its behaviour, but is used by all of its object instances. Thus shared aggregation is applicable as this is what Shlaer/Mellor express.
The class operation needs to know Jim's current state. Class operations can only call class operations directly. This is why the reference to Jim is handed in. By dereferencing Jim the class operation receives the desired information. After that, `performTransition()` calls the operation `traverse()` of Jim's state machine by means of the second reference parameter.

Jim's state machine now traverses its list of transition instances and asks them whether they are responsible for the event `engagement` in the current state. The number of the event and the current state have been passed as parameter for the state machine's operation `traverse()`.

Each transition instance compares the current state with its attribute `sourceState` and the event number of engagement with its attribute `event`. If both value pairs are equal, the currently accessed transition instance is responsible for the event. It returns the value of its attribute `targetState` to the state machine. The state machine returns this value to the class operation `performTransition()`.

Figure 6: State transition following Shlaer/Mellor
If the receipt of the event in the current state is not an error and the event is not to be ignored (see explanation of class Transition above), the class operation sets the new state of object Jim by dereferencing it.

The thread of control returns to the event method of Jim. The last step is to execute the reaction to the event engagement which is confirmed now. This happens by calling a corresponding operation of object Jim.
2.2 Design Pattern 'State' [Gamma 96]

The design pattern 'State' of Gamma et al. (see [Gamma 96]) makes it possible to let an object alter its behaviour when its internal state changes.

Design patterns are well-proven solutions for particular design problems. Design patterns address problems that are independent of the application domain and may appear in different design contexts.

Design patterns are formalized pieces of experience of software engineering professionals. They are taken down in a standardized way in order to serve for conservation and interchange of software design experience. Beginners are able to acquire knowledge and skills about building object-oriented software without having to experience the underlying problems themselves. A design pattern has a distinct name, so that software patterns create a "design vocabulary". Further, design patterns facilitate the documentation of design models and source code.

Design patterns do not offer an exact implementation for a problem. Rather, they classify the problem at an abstract level and present a solution consisting of a static and dynamic object-oriented model. Mostly, the model is extended by sample code. Nevertheless, patterns are merely abstract solutions that must be applied to specific requirements. Design patterns are concepts, not implementation frames.

Design patterns emphasize interface orientation, object composition and delegation and avoid static dependencies between source code components like static inheritance.

For further information about patterns see [Gamma 96], [Buschmann 96], [Coplien 95], [Vlissides 96], [Martin 98] and [Fowler 97].

In this section, the design pattern 'State' is presented briefly. The most relevant aspects for understanding the pattern are explained. The complete description of the pattern can be found in [Gamma 96].
2.2.1 Structure and Interaction

The central idea of the state pattern is to have a class for each state defined in the statechart diagram. Each of these classes represents an operational state and defines the reactions to the events that are allowed in that state. They share the same interface by inheriting from an abstract base class State. The interface consists of operations for each external event the object can receive. All of them are abstract.

The object instance, the behaviour of which is defined by the statechart diagram (called Context Object), only knows about the interface of the base class. The current state object held by the Context Object is an instance of one of the concrete subclasses of class State. Each time a state transition occurs, the state object changes.

In figure 7, the structure of the design pattern 'State' is shown following the example of a worker presented in figure 4. The Class Worker offers an interface which is used by any client of a worker object. It includes state-dependent operations (external events) as well as state-independent operations. In our example, the operation goForLunch() is state-independent as this operation does not require a special state. A worker can eat something anytime.

The operations engagement(), disengagement() and take_a_holiday() represent external events modelled in the statechart diagram. If a client invokes these operations on the worker object, the worker object delegates these calls to its state object it currently references. The state object then decides how to react to that event. If it is accepted, the state object performs the state activity of the target state in the statechart diagram. If not, the event is ignored and nothing happens.

A possible state transition following the event engagement() is shown in figure 8. Note that a state transition is represented by the target state instance returned by the current State object. Section 2.2.2 mentions other possible strategies for implementing a state transition.
2.2.2 Implementation

There are some aspects concerning the implementation of the State pattern that are important to mention:

Concrete subclasses of the interface class *State*

The operations of interface class *State* are abstract. There is no default implementation for them. The concrete subclasses define methods for the events which cause a reaction and state transition of the object. The events that are not accepted by the current state object get an operation with an empty body. Alternatively, events may be queued until a corresponding state is reached (see [UML... ](19)
Chapter 2

Existing Implementation Approaches

97d]). This is to be considered as an extension to the State pattern. Chapter 5 proposes detailed concepts for deferred handling of events.

State transitions

A state transition is represented by the exchange of the state object. This can be done either by the Context Object or the state object itself. In the first case, the Context Object must know all of the existing state objects in order to choose one of it as the target state object. In the latter case, there are three possibilities. First, the state instances must have direct access to the properties of the Context Object to set the state object. The principle of encapsulation is given up then. Second, the Context Object must have a corresponding public operation for setting the state object. This is problematic because any other object would be able to change the Context Object's state. The third, and preferable, approach is to let the event methods return the reference to the target State object, which is assigned to the state attribute of the Context Object. This is the approach presented in figure 8.

Properties of the Context Object

The reaction to external events is encapsulated in the state objects. However, the resulting activity needs access to the properties of the Context Object. In the context of an event receipt, instance attributes may be changed or association links may be updated. Gamma et al. [Gamma 96] discuss this topic further.


2.3 Design Pattern ´State Table´ [Douglass 98]

The design pattern ´State Table´ is introduced by Bruce Powel Douglass in [Douglass 98]. His book explains fundamental aspects of real-time systems and shows object-oriented modelling of real-time systems by the use of the UML. In particular, Douglass shows the notation specification of UML's statechart diagrams and presents examples how to use it. Furthermore, it details constraints and requirements for software development, especially with respect to considerations such as correctness, performance and memory size.

Beside the design pattern ´State´ of Gamma et al., he mentions the State Table pattern as a possibility for implementing statechart diagrams.

2.3.1 Structure and Interaction

The State Table pattern is based upon a state table similar to the one proposed by Shlaer/Mellor (see section 2.1). Shlaer/Mellor use a state table at the logical level to improve clarity of the statechart diagram. At the level of implementation, Shlaer/Mellor map the state table to a list of transition instances. Contrary, the ´State Table´ pattern emphasizes a state table as an implementation element.

The structure of the design pattern ´State Table´ is shown in figure 9 following the example of a worker presented in figure 4.

The state table becomes an object consisting of an $n \times m$ array, where $n$ is the number of states and $m$ is the number of transitions. For addressing the array, both states and events are represented as an enumerated type. If the statechart diagram says that an event is valid for a particular state, the corresponding cell contains a single reference to a Transition object. A Transition object represents a transition in the statechart diagram. It may contain a guard condition which must be
evaluated to true before the transition can fire. The abstract class *Transition* serves as the interface specification for the Transition objects (see figure 9).

At the modelling level, the state table matrix is represented by a qualified association from class *StateTable* to class *Transition* (see figure 9).

A State Table instance is owned by a Context Object instance. When receiving an event, the Context Object forwards it to the State Table, which looks up the Transition Object.

Also, there is an object for each state. A State object knows about its name, its *entry-, exit- and do-* activity (for explanation of types of state activity see [Rumbaugh 93], [UML 97c] and section 3.3).

A *Context Object* is an instance of the class for which a statechart diagram is defined. Each Context Object instance has its own State Table object which manages the execution of its state machine.

All necessary methods for state activities and transition actions are implemented within the Context Object. This is done because all activities must be able to manipulate instance and class attributes as well as associated objects. Thus, a State object as well as a Transition Object are responsible for invoking the correct operations. Therefore, State and Transition objects have access to the Context Object (see figure 9).

*figure 9: structure of the design pattern 'State Table'*
The interface of all State objects is the same. Thus, like the design pattern 'State' (see section 2.2)), all concrete State classes inherit from an abstract class State. A State Table instance always references the State instance representing the current state. In case of a state change, the corresponding Transition object returns the new State object to the State Table object.

Interaction of the entities shown in figure 9 (see figure 10):

The worker Jim receives the event engagement. Jim informs its State Table about the event that has occurred by handing in the event ID. The State Table gets the current state ID by interrogating the current State Object. The State Table accesses its matrix in order to find the corresponding Transition object.

![Figure 10: State transition following the State Table pattern](image-url)
Once determined, the Transition object is asked to check the condition, if there is any. If the Transition object confirms its condition, the State Table invokes the exit-activity of the current State object. After that, the State Table initiates the transition actions by calling `doTransition()` on the Transition object. After having processed the transition actions, the Transition object returns the reference to the target State object. The State Table immediately invokes the entry- and do-activity of the new state.

As stated above, all operational action is done by the Context Object as it contains all operations necessary for any state activity and transition actions.

### 2.3.2 Implementation

There are some aspects concerning the implementation of the State Table pattern that are important to mention:

**Concrete subclasses of the interface class State**

The operations of interface class `State` represent entry- exit- and do-activities of a state. They are abstract as there is no default implementation for them. The concrete subclasses define those operations by calling the corresponding state activity method of the Context Object. If there is no state activity defined in the statechart diagram, the method bodies are left empty.

**Ignoring of events**

If an event cannot be accepted in the current state, it is ignored. As an extension to the State Table pattern, events may be queued until a corresponding state is reached (see [UML 97d]). Chapter 5 proposes detailed concepts for deferred handling of events.
2.4 Discussion of the Existing Implementation Concepts

In this section, common features and differences of the implementation concepts of the three last sections are shown. In comparison to each other, advantages and disadvantages are discussed concerning the following criteria:

- Runtime performance
- Memory requirements
- Simplicity
- Extensibility

In the field of Software Engineering, there are more issues that may be considered. As chapter 3 deals with the implementation of logical concepts but not with the evaluation of complete software systems satisfying particular requirements specifications, the inspection of the mentioned quality aspects is sufficient here. Moreover, the conformance with UML is examined.

For reading convenience, the particular design patterns and implementation concepts are referred to by the names of the authors.

Common Features

All three approaches emphasize explicit representation of states. All implementation techniques have a single attribute to remember the current state. Douglass and Gamma et al. propose a pointer attribute to a State object instance, Shlaer/Mellor introduce an attribute of an enumeration type to remember the state. All authors avoid a less explicit state representation on the basis of instance data (attribute values). When checking the state, more than one attribute value would have to be considered which results in extensive condition statements.


**Technical Differences**

Douglass offers to represent state transitions semantics by a state table-lookup. In general, a table-oriented representation of state transitions has the following advantage. Because of the structure of a state table, the criteria for state transitions can be changed and extended more easily. Changing or extending a table means changing data instead of code structure. However, there are the following disadvantages:

- The format of state tables makes state transitions less explicit which is harder to understand.
- It is difficult to apply actions to state transitions to be executed if a transition fires.

Douglass solves the latter problem by introducing object instances for state transitions. A Transition object is referenced by each cell of the state table. A Transition object has the full responsibility for performing some action and determining the target State object. Afterwards, the activity of the target State object can be initiated directly.

Both Shlaer/Mellor and Douglass integrate the state activity into the Context Object. It is not part of the State objects themselves. A State object of Douglass represents a single state of the statechart diagram in terms of being responsible for invoking the correct behaviour of the Context Object. The difference between Shlaer/Mellor and Douglass is that Shlaer/Mellor do not create objects for the states. Following Shlaer/Mellor, the Context Object itself knows what state activity to call based on the target state ID delivered by the Transition object. Furthermore, the correct Transition object is determined by traversing a linear list, whereas Douglass uses a table-lookup.

Both Gamma et al. and Douglass introduce states as objects. The behaviour of the object as a result of an incoming event, specified as state activity at the modelling level, is encapsulated in a single object.
The difference between the two patterns is that the state classes of Gamma et al. share the same interface defined by all possible events that can be received generally by the Context Object. Only those event operations that can be accepted in the particular state have a meaningful implementation. The other event operations are implemented in some other way, i.e. an empty body if ignored or a queueing mechanism. Using the State pattern, there is always a definite implementation for any event.

The state classes of Douglass only define operations for the entry-, the exit- and the do-activity of exactly one state in the statechart diagram.

The State pattern defines state-dependent behaviour for the interface of the Context Object and encapsulates it in different classes, whereas the State Table pattern concentrates on the definition of state transitions.

**UML Conformance**

The method OOSA of Shlaer/Mellor defines its own notation. Shlaer/Mellor had published their ideas years before the first steps towards a unified notation and metamodel for object-oriented modelling were taken. Their statechart diagram notation offers only states with activities, but no actions and message-sendings for transitions. It is not possible to provide actions or message-sendings for transitions. Furthermore, they do not distinguish between subtypes of state activities such as entry-, exit- and do-activities. Actually, the statechart diagram notation of OOSA is a subset of the capabilities of UML's statechart diagrams. Therefore, Shlaer/Mellor do not offer implementation constructs for notation elements such as different kinds of state activities, transitions including actions and message-sending, history markers and event queueing.

Douglass and Gamma et al. propose a design patterns. Since a design pattern is defined as a specification of a generic scheme for a solution for a recurring design problem, it is not expected to describe detailed object-oriented modelling notation. When using the design pattern 'State' of
Gamma et al., a software engineer always has to think himself about the implementation of further concepts.

**Simplicity and Extensibility**

The state classes of Gamma et al. incorporate the reaction of incoming events, which is done by introducing operations for each event. In the case of a complex statechart diagram, the pure State pattern following Gamma et al. causes code redundancy. The result is source code that is hard to maintain. For explanation, consider the following illustration:

When using the State pattern here, we would have at least two state classes \textit{State1} and \textit{State2}. Both implement the operations \texttt{event1()} and \texttt{event2()}. There is an empty body for \texttt{State1\texttt{::}event2()} and \texttt{State2\texttt{::}event1()} because they cannot be accepted there. When \texttt{event1} arrives in \texttt{state1} or \texttt{event2} occurs in \texttt{state2}, both methods \texttt{State1\texttt{::}event2()} and \texttt{State2\texttt{::}event1()} must execute the entry- and do-activity of \texttt{state3}. It is even worse: As \texttt{state3} has a transition without an external event to \texttt{state4}, the whole transition actions including the condition check also becomes part of both event methods.

This problem can be solved if the Strategy design pattern is applied [Gamma 96]. The Strategy pattern establishes classes for methods so that they can be shared by more than one object.
As both Shlaer/Mellor and Douglass incorporate all actions and activities into the Context Object and distinguish between object behaviour and event receipt, there is no code redundancy at all.

The metamodel of the Shlaer/Mellor approach is designed in a strict object-oriented way and thus conceptually well-founded, but it is very complicated indeed.

The following table gives an overview of the impact that changes to the statechart diagram have on the code, depending on the particular implementation approach. The columns represent the name of the author, the rows name a specific modification of the model. The complexity of the particular patterns can be interpreted individually.

<table>
<thead>
<tr>
<th>Additional event (incl. a transition)</th>
<th>Douglass</th>
<th>Shlaer/Mellor</th>
<th>Gamma et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change of state table, new state class</td>
<td>Interface change of the Context Object, new Transition object, adaption of state table</td>
<td>Interface change of the Context Object, extension of factoring process creating Transition instances</td>
<td>Interface change of the Context Object, interface change of all state classes and impl. of the new operation</td>
</tr>
<tr>
<td>New state method for Context Object</td>
<td>Redefinition of a state method of the Context Object</td>
<td>Redefinition of an event method in several concrete state classes (code redundancy)</td>
<td></td>
</tr>
<tr>
<td>Only factoring process is concerned</td>
<td>Redefinition of an event method in one concrete state class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change of factoring process creating Transition instances</td>
<td>Redefinition of an event method in several concrete state classes (code redundancy)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that for each implementation technique, since all of them consist of more than one class, an initialization process is needed for tying the constituent object instances together. A change of the model may also affect the code for the factoring process.

---

5 Mostly, Object Creation Patterns like the 'Abstract Factory Pattern' are used for this kind of requirement. See [Gamma 96] for more information.
Chapter 2

Existing Implementation Approaches

Runtime Performance and Memory Requirements

Today, performance and efficient use of memory is not that critical in comparison to software engineering concepts such as reusability, extensibility and simplicity of models and code. However, when talking about realtime and reactive systems, issues like performance become more important ([Douglass 98]). Embedded systems must often run on a minimum of memory, so that judicious allocation of memory is crucial.

Depending on the complexity of the statechart diagram, Gamma et al. need high memory resources because of having an object for each state. Memory requirements of Douglass are even higher as he introduces objects for transitions as well. Every object instance will demand its own heap space. A step towards minimizing this memory requirements is to combine the State pattern or the State Table pattern with the design pattern 'Flyweight' [Gamma 96]. Applying the Flyweight pattern facilitates the sharing of state objects and transition objects between different Context Object instances of the same type. As long as state and transition objects do not have attributes holding instance-specific data, the Flyweight pattern may be applied.

Also, Shlaer/Mellor need to allocate additional space for their transition instances. The Flyweight pattern may be used here as well.

The worst runtime-performance is revealed by Shlaer/Mellor. Shlaer/Mellor search a linear list of Transition objects with a runtime complexity of \( O(n) \), where \( n \) is the number of the transitions.

The most important advantage concerning runtime performance of Douglass is the use of a state table. Transitions of incoming events are looked up with constant time effort.

On the other hand, Gamma et al. [Gamma 96] argue that sometimes a table-lookup is not as efficient as a virtual function call. The State objects of Gamma et al. fully support the interface of the Context Object. Thus, a reaction to an external event is done by calling the corresponding operation of the current State object.
However, the patterns of Douglass and Gamma et. al produce a larger amount of collaborating objects. Both patterns, in particular the State Table pattern, have many levels of indirection. Thus, there is a lot of dereferencing for performing a state transition and executing state activity, which also has a time overhead.

**Consistency Constraints**

The UML offers wide possibilities for modelling finite state machines as a behavioral description of objects. The notation and semantics for statechart diagrams are clearly defined (see [UML 97c] and [UML 97d]). However, the metamodel is not as restrictive as it should be. The semantics of statechart diagrams should include more constraints and restrictions in order to avoid inconsistent use of the statechart diagram notation.

The aim of the previous sections are implementation proposals for statechart diagrams. They do not include discussion or advice for semantic correctness of a statechart diagram itself. For instance, all external events leading to a particular state must provide as many parameters as a state activity needs. At the same time, a particular external event must have the same signature throughout the statechart diagram.

Section 3.7 describes consistency constraints that should be considered during the development of dynamic behaviour of object-oriented software by the use of UML.
3 Alternative Implementation Approach

This chapter deals with an implementation technique for statechart diagrams defined by the UML v1.1, which is different from the ones presented in chapter 1. This implementation technique was developed by the author of this thesis.

3.1 Events, States and Transitions

In the following implementation concept, states, activities and events are fully integrated into the class definition by matching them to instance methods and attributes.

Events firing transitions are external stimuli of the object. Therefore the events become public methods of the underlying class. Following the implementation approach of Shlaer/Mellor (see section 2.1), each state activity becomes a non-public method as well. The event methods call the corresponding state method depending on the state the object is in.

Why does the state activity not form the body of the event methods?

Possible Code Redundancy

A certain state can be arrived at by different transitions fired by different events. This means, that different events may cause the same reaction of the object. By distinguishing state methods and event methods we avoid redundant method bodies within the class definition.
Example: compact disc player

![Statechart diagram of a compact disc player](image)

The events *pause* and *start* have the same reaction. Both of them cause the CD player to start playing when in state *paused*.

A Certain Event Can Be Accepted in Different States

This means that the same event may cause different reactions of the object. In this case we cannot relate the event to exactly one state activity.

Example:

In figure 11 the event *pause* has a different reaction depending on the current state in which it is received. The reaction is either the activity of state *paused* or the activity of state *playing*.

The next question is how to represent the object's state. It is necessary to identify the object's state; the event methods must be able to identify the state of the object in order to decide what state method to call. We achieve this by providing an attribute *state* for each class having a state diagram. The value of this attribute represents the current state the object is in. We will implement the type of the attribute *state* with an enumeration of state identifiers.
When an object receives a message the corresponding event method is invoked. The event method consults the attribute \textit{state} indicating whether the object can accept the message or not. If the message can be accepted, the event method changes the state by modifying the attribute value and calls the corresponding method of the target state.

\textbf{Automatic Transitions}

Automatic Transitions are transitions that have no external event. These transitions fire as soon as the activity of the source state is completed. An automatic transition is, conceptually, to be considered as a result of an internal event. An internal event is an event which is not provoked by any external stimuli but from the object itself.

As state activities are responsible for firing automatic transitions, we can integrate them into the state methods. The method of the source state calls the method of the target state after having updated the state attribute.

\textit{Example:}

\begin{figure}
\centering
\includegraphics{statechart_diagram.png}
\caption{statechart diagram of a compact disc player with an automatic transition}
\end{figure}

The compact disc player in figure 11 is extended by an automatic transition (see figure 12). The state of the object changes from \textit{playing} to \textit{ready} as soon as the total play time is over. The CD player stops playing then.
The complete implementation of the CD player in figure 12 looks as follows:

```
class CD_player
{
    protected:
        enum {INITIAL, READY, PLAYING, PAUSED, FINAL} stateValue
        stateValue m_state;

    virtual void setState (stateValue newValue)
    {
        if ((newValue >= INITIAL) && (newValue <= FINAL))
            m_state = newValue;
        else
            throw badStateChange; // exception
    }

    public:
        CD_Player (void) {...}; // constructor
        ~CD_Player (void) {...}; // destructor

    virtual void start (void)
    {
        switch (m_state)
        {
            case READY:
                setState (PLAYING);
                do_playing();
                break;

            case PAUSED:
                setState (PLAYING);
                do_playing();
                break;

            default:
                break; // event ignored
        } // switch
    }
```

*figure 13: declaration of class CD_player*
virtual void stop (void) {
    switch (m_state) {
    case PLAYING:
        setState (READY);
        break;

    case PAUSED:
        setState (READY);
        break;

    default:
        break;       // event ignored
    } // switch
};

virtual void pause (void) {
    switch (m_state) {
    case PLAYING:
        setState (PAUSED);
        do_paused();
        break;

    case PAUSED:
        setState (PLAYING);
        do_playing();
        break;

    default:
        break;       // event ignored
    } // switch
};

protected:

virtual void do_playing (void) {
    // read disc
    setState (READY);    // automatic transition to state READY
    do_ready();          // perform state activity
};

virtual void do_paused (void) {
    // hold laser
};
3.2 Conditions and Actions of Transitions

The syntax of transitions offers the execution of sequences of actions and sending of messages [UML 97c, UML 97d]. The actions and the message sending is part of the transition. As a transition represents a change of state, actions and message-sendings are executed immediately after having changed the state but before the execution of the target state activity.

Transitions may have a condition. A condition is a boolean expression. If an event occurs and fires a transition, a state change only happens if the condition is evaluated to true. The state change is dependent on the condition. A condition of a transition is also called a "guard" ([UML 97a]). There is always a maximum of one condition.

The complete syntax of transitions in statechart diagrams is shown in figure 14. All elements are optional.

\[ \text{event1(param1, param2, ...)} \]
\[ \text{[condition1]} \]
\[ \text{/ action1, action2; ...} \]
\[ \text{^ obj1.message1, obj2.message2; ...} \]

\[ \text{state1} \]
\[ \text{state2} \]

---

\[ \text{event1 (param1, param2)} \]
\[ \text{[condition1]} \]
\[ \text{/ action1} \]
\[ \text{^ Obj1.message1()} \]

\[ \text{do / def,} \]

---

\[ \text{event2 (param1, param2)} \]
\[ \text{[condition2]} \]
\[ \text{/ action2, action3} \]
\[ \text{^ Obj2.message2()} \]

\[ \text{event2 (param3, param4)} \]
\[ \text{[condition3]} \]

---

\[ \text{state3} \]
\[ \text{state1} \]
\[ \text{state2} \]
\[ \text{state4} \]

---

**figure 14: UML syntax for state transitions**

**figure 15: example for transitions with conditions, actions and message-sending**
We will now consider the implementation of conditions, actions and message-sending for the example in figure 15:

class AnyObject
{
    // ...

protected:
typedef enum {INITIAL, STATE1, STATE2, STATE3, ..., FINAL} stateValue;
    stateValue m_state;

public:
    virtual void event1 (<type1> param1, <type1> param2)
    {
        switch (m_state)
        {
            case STATE1:
                if (condition1)
                {
                    setState (STATE3);
                    action1;
                    obj1->message1();
                    do_state3 (param1, param2);
                }
                break;

            case STATE2:
                if (condition2)
                {
                    setState (STATE3);
                    action2;
                    action3;
                    obj2->message2();
                    do_state3 (param1, param2);
                }
                break;

            case STATE3:
                if (condition3)
                {
                    setState (STATE4);
                    // no do-method call because no do-
                    // activity defined for state4
                }
                break;

            default:
                break; // event ignored
        } // switch
    } // event1

    virtual void event2 (<type1> param1, <type2> param2)
    {
        switch (m_state)
        {
            case STATE2:
                if (condition2)
                {
                    setState (STATE3);
                    action2;
                    action3;
                    obj2->message2();
                    do_state3 (param1, param2);
                }
                break;

            case STATE3:
                if (condition3)
                {
                    setState (STATE4);
                    // no do-method call because no do-
                    // activity defined for state4
                }
                break;

            default:
                break; // event ignored

        } // switch
    } // event2

    virtual void event3 (<type1> param1, <type2> param2)
    {
        switch (m_state)
        {
            case STATE3:
                if (condition3)
                {
                    setState (STATE4);
                    // no do-method call because no do-
                    // activity defined for state4
                }
                break;

            default:
                break; // event ignored
        } // switch
    } // event3
default:
    brak;    // event ignored
};     // event1

virtual void do_state3 (<type> arg_a, <type> arg_b)
{
    d;
    e;
    f;
};

};     // class
3.3 Entry, Exit and Do-Activities

This section briefly explains what entry-, exit- and do-activities are. We also consider how they can be implemented. A more detailed introduction and explanation of entry-, exit- and do-activities can be found in [Rumbaugh 93].

The do-activity of a state is a sequence of statements that determine the qualitative reaction to all events that cause a transition to this state.

If several transitions with same target state share a set of identical actions and message sending, this set can be integrated as entry-activity for a greater clarity of the statechart diagram. When entering a state, first the entry-activity and then the do-activity is performed.

As entry-activities result from transition actions and transitions are atomic, entry-activities are, also, atomic and non-interruptable. On the contrary, do-activities are not. When the do-activity of the current state is not completed and a valid event occurs, then the do-activity is interrupted and a transition is fired. If the do-activity is completed and there is no event or automatic transition, the object remains in the current state.

Exit-activities represent an identical set of actions of the outgoing transitions of a state. Exit-activities are also atomic and are executed after an event has occurred but before the outgoing transition fires.

Entry- and do-activities are both executed in the context of the new state. This is obvious for entry-activities because they are transferred transition action sequences that are performed after the transition has fired.

Exit-activities are performed in the context of the old state, before the transition has fired.
As an example, consider the following figure 16.

```
figure 16: example for a statechart diagram with entry- exit- and do-activities
```

Now consider the partial implementation for figure 16:

```cpp
class AnyObject {
public:
    typedef enum {INITIAL, STATE1, STATE2, STATE3, STATE4, ..., FINAL} stateValue;

protected:
    stateValue m_state;

public:
    virtual void event2 (<type1> param1, <type2> param2) {
        switch (m_state) {

            case STATE2:
                if (condition2)
                {
                    exit_state2();
                    setState (STATE3);
                    action2;
                    action3;
                    obj2->message2();
                    entry_state3 (param1, param2);
                    do_state3 (param1, param2);
                }
                break;
```
case STATE3:
    if (condition3)
    {
        exit_state3();
        setState (STATE4);
    }
    break;
    
    default:
    break;   // event ignored
    }
    // switch
);    // event2

virtual void event1 (<type1> param1, <type2> param2)
{
    switch (m_state)
    {
    case STATE1:
        if (condition1)
        {
            // no exit-activity defined for state1
            setState (STATE3);
            action1;
            obj1->message1();
            entry_state3 (param1, param2);
            do_state3 (param1, param2);
        }
        break;
    default:
    break;  // event ignored
    }
    // switch
);  // event1

protected:
virtual void entry_state3 (<type1> arg_a, <type2> arg_b)
{
    a; b; c;
};

virtual void exit_state3 (<type1> arg_a, <type2> arg_b)
{
    g; h;
};

virtual void do_state3 (<type1> arg_a, <type2> arg_b)
{
    d; e; f;
};

// the entry-, exit- and do-methods of state2 are omitted here.
);  // class
Chapter 3

3.4 Interruptable Activities

As outlined in section 3.3, do-activities are interruptable. At the level of implementation, this can be achieved by creating a thread for the current do-activity, which is killed when an event occurs before completion. For this implementation concept the presence of a Multi-Tasking operating system or Multi-Processor system is required.

An object is in exactly one state at any time, therefore there is always only one active "do-thread". Thus, it is suitable to have a member attribute in which the handle of the current do-thread is stored.

The creation and deletion of threads and their handling (creation, killing etc.) is platform-dependent. For this reason, a code generator may generate non-public methods for creating, killing and resuming a thread. The software developer can fill in the method body with API Application Programming Interface) operation-calls of a specific multitasking operating system. When porting the code from one environment to another, the software engineer has to change the code at only one place. Assuming a single process environment, the default implementation for killing and resuming is empty. The creation method would simply invoke the state method directly.

On the other hand, a default implementation for creating a thread cannot easily be found. The reason are parameters delivered by the event which are needed by the state activities. Depending on the particular event, these parameters may differ in number and type. As the thread-creation routine invokes the do-activity of the state, the event method passes the parameter to the thread-creation routine which in turn hands them over to the state method. Thus, the thread-creation routine would have to accept parameters dynamically. This is a problem because, in C++, we cannot implement generic operation invocations with variable arguments.
We can solve this problem by establishing a class `ParamStore` which stores all parameters that are delivered by any event in the statechart diagram. It is generated by the code generator. The class `ParamStore` has got a member attribute for each parameter including set- and get-operations. An object of class `ParamStore` is created by the constructor of the class for which the statechart diagram is defined. Each time an event method is called, it stores the current parameter values into the parameter store object. After that, the thread-creation routine creates a thread for the do-activity of the target state. When running, the do-thread retrieves these parameter values from the parameter store object. It uses local variables that are created on the stack of the thread.

Consider the following code showing the extended methods, `event2()` and `event1()`, of the example in figure 16.

```cpp
class AnyObject {
    protected:
        HANDLE m_threadHandle; // member attribute which stores the handle of the thread returned by the API operation call for creating a thread.
        ParamStore* m_paramStore;

    public:
        AnyObject (void) // constructor
            { m_paramStore = new ParamStore; }
};
```
virtual void event2 (<type1> param1, <type2> param2)
{
    switch (m_state)
    {
        case STATE2:
            if (condition2)
            {
                // killing the running do-thread
                this->kill (m_threadHandle);
                exit_state2();
                setState (STATE3);
                action2;
                action3;
                obj2->message2();
                entry_state3 (param1, param2);
                // provide the parameter values
                m_paramStore->setParam1 (param1)
                m_paramStore->setParam2 (param2);

                // create the thread
                m_threadHandle = createThread (do_state3());
            }
            break;

        case STATE3:
            if (condition3)
            {
                // killing the running do-thread
                this->kill (m_threadHandle);
                exit_state3();
                setState (STATE4);

                // there is no thread creation because state4
                // does not define do-activity
            }
            break;

        default:
            break;    // event ignored
    }    // switch
};    // event2

virtual void event1 (<type1> param1, <type2> param2)
{
    switch (m_state)
    {
        case STATE1:
            if (condition1)
            {
                // no thread to be killed because state1
                // has no do-activity
                // no exit-activity defined for state1
                setState (STATE3);
                action1;
                obj1->message1();
                entry_state3 (param1, param2);
            }
Alternative Implementation Approach

The following code shows the implementation of the related class ParamStore.

```cpp
class ParamStore
{
    protected:
        <type1> m_param1;
        <type2> m_param2;

    public:
        ParamStore (void); // constructor
        virtual void setParam1 (<type1> arg);
        virtual void setParam2 (<type2> arg);
        virtual <type1> getParam1 (void) const;
        virtual <type2> getParam2 (void) const;
}; // class
```

An automatic transition leaves the current state when the do-activity has been executed. Consider the transition leading from state3 to state4 in figure 16 as an automatic transition. In this case, the current do-thread is not killed because it is the current do-thread which fires the automatic transition. Note that an automatic transition cannot carry any parameters as only external events can have parameters.

The current and the new do-thread may overlap each other in existing. Anyway, there will be no collision between the two threads because the creation of the new do-thread is the last action that
takes place within the old do-thread context. The code of the state method \texttt{do\_state3()} would be as follows:

\begin{verbatim}
virtual void do_state3 (void)
{
    // parameter values are fetched from the parameter store to be used
    // within the current do-thread
    <type1> param1 = m_paramStore->getParam1();
    <type2> param2 = m_paramStore->getParam2();
    d; e; f;

    if (condition3)
    {
        // no killing of the thread because the current method
        // represents the do-thread
        exit_state3();
        setState (STATE4);
        entry_state4();

        // create the thread
        m_threadHandle = createThread (do_state4());
    }
    // The thread terminates now
}
\end{verbatim}

Introducing multithreading within an object immediately raises the question for guaranteeing exclusive reading and setting of the state attribute. Without any protection mechanism the object may behave unpredictably because of resulting race conditions. Consider that the current do-thread wants to initiate an automatic transition flow and at the same time an event method is called from outside. At that moment, two different threads address the same data. Chapter 4 refers to this kind of problem.

Basically, creating and killing threads is always a very sensitive issue. Normally, when designing programs, one should avoid killing threads explicitly. For this reason, we propose the first modelling rule for statechart diagrams:
Modelling Constraint 1: Interruptable Activities

"The software designer must carefully consider what state activity, generally, is supposed to be interruptable. Such activities should be modelled by do-activities. Otherwise, they should be integrated in an entry block as entry-activities are atomic."

We make use of entry-activities, because the UML v1.1 does not provide notation for non-interruptable do-activities. Another solution would be to introduce a self-defined stereotype. We decide on the first option because creating additional stereotypes to the UML core should be avoided where possible to reduce complexity.

---

6 The UML offers to extend its metamodel by user-defined stereotypes. The intention is to make the adaptation of the UML to special process requirements possible. See [UML 97c], [UML 97d] and [UML 97e] for more.
3.5 Nested States

The do-activity of a state can be specified by another statechart diagram. A state with an enclosed statechart diagram is called a “state region”. The states of the enclosed statechart diagram are the nested states of the state region. Rumbaugh ([Rumbaugh 93]) speaks of “state generalization”. State regions are “normal” states and thus can have entry- and exit-activity, but no do-activity. Any outgoing transition at the level of the super state (the enclosing state/the state region) is valid for all substates (the nested states). For incoming transitions there are three possibilities:

1. A transition leading to the super state is equivalent to a transition to the substate indicated by an initial state (small filled circle).

2. A transition leading to a shallow history marker of a state region (an ‘H’ in a small circle) makes the object resume the last active substate before the last outgoing transition fired. (Introduced in UML v0.8 [UM 95]). Any necessary entry-activity is performed. If the state region has never been entered before, it behaves as (1).

3. A deep history marker (an ‘H*’ in a small circle) complies to (2), but remembers the state the object last had at any depth within the state region if there are nested substate regions. Any necessary entry-activity is performed. (Introduced in UML v1.1)

Note:

The definition given for a shallow history marker does not completely conform to the UML v1.1 ([UML 97a]). It says that if the state region containing this kind of marker has never been entered before, the object takes the state to which the (only) outgoing transition of the shallow history leads.
This is not meaningful. We advise modelling an initial state for each state region for conceptual exactness. There should be only one substate acting as the "starting point".

Example:

Consider that the object is in state4 when event e6 occurs (see figure 17). The next state is state3. After that e5 occurs so that the next active state is state5. In this situation, the occurrence of e2 will result in state3, whereas e3 causes a transition back to state4.

figure 17: sample statechart diagram with nested states and history markers
3.5.1 Implementation of Nested States

In this section, we consider the implementation of nested states without history markers. There are two possibilities to implement nested states. First, we can establish a member attribute "stateRegion" as an enumeration type of all state region names. Consider the following code for the event methods e5, e6, e1, e4 and e8:

```cpp
class AnyObject
{
public:
    typedef enum {REGION1, REGION2, MAIN} stateRegionValue;
    typedef enum {STATE1, STATE2, STATE3, STATE4, STATE5, FINAL_REGION2} stateValue;
protected:
    stateRegionValue m_stateRegion;
    // ...

public:
    AnyObject (void) // constructor
    {
        m_stateRegion = MAIN;
    }

virtual void e6 (void)
    {
        switch (m_state)
        {
            case STATE4:
                setState (STATE3);
                setSateRegion (REGION2);
                entry_region2();
                break;
            
            default:
                break; // event ignored
        } // switch
    }
};
```
virtual void e8 (void) {
    switch (m_stateRegion) {
    case REGION2:
        if (condition_8)
            {
            exit_region2();
            setState (STATE4);
            setStateRegion (REGION1);
            break;
            default:
            break;   // event ignored
        }   // switch
    }
}

virtual void e4 (void) {
    switch (m_state) {
    case STATE2:
        setState (FINAL_REGION2);
        break;
    default:
    break;   // event ignored
    }   // switch
}

virtual void e5 (void) {
    switch (m_stateRegion) {
    case REGION2:
        exit_region2();
        exit_region1();
        setState (STATE5);
        setStateRegion (MAIN);
        break;
    case REGION1:
        exit_region1();
        setState (STATE5);
        setStateRegion (MAIN);
        break;
    default:
    break;   // event ignored
    }   // switch
}
virtual void el (void)
{
    switch (m_state)
    {
        case STATE5:
            setState (STATE3);
            entry_region1();
            entry_region2();
            break;

        default:
            break; // event ignored
    } // switch
};

protected:
virtual void entry_region1()
{
    a;
};

virtual void exit_region1()
{
    d;
};

virtual void entry_region2()
{
    b;
};

virtual void exit_region2()
{
    c;
}; // class

The final state in state region region2 gets its own state value. The reason is that, once the final state is reached, the object waits for event e8 and e5 but must not react to event e7. Event e7 is supposed to be valid in state state2 only.

This solution makes the code slightly complicated because of all the setStateRegion() operations. We prefer not to establish a state region attribute but to establish a boolean method that determines, whether the value of m_state belongs to the set of substates of that state region or not. This is less efficient concerning runtime, but software engineering issues of clarity and readability are considered to be more important.
When applying the second possibility of implementation, the above sample code for the example in figure 17 changes as follows:

```cpp
class AnyObject {

public:
virtual void e6 (void) {
    switch (m_state) {
        case STATE4:
            setState (STATE3);
            entry_region2();
            break;

        default:
            break;  // event ignored
    }  // switch
};

virtual void e8 (void) {
    if (isRegion2()) {
        if (condition_e8) {
            exit_region2();
            setState (STATE4);
        }
    }
};

virtual void e5 (void) {
    if (isRegion1()) {
        if (isRegion2())
            exit_region2();
        exit_region1();
        setState (STATE5);
    }
};

protected:
virtual bool isRegion2 (void) {
    return ((STATE1 == m_state) ||
             (STATE2 == m_state) ||
             (STATE3 == m_state) ||
             (FINAL_REGION2 == m_state));
};
```
virtual bool isRegion1 (void)
{
    return ( STATE4 == m_state ) ||
            isRegion2();
}

However, the second way of implementation is not sufficient. What we have not considered yet is
that any substate may have an exit-activity which has to be performed in advance of the exit-
activity of the surrounding state region.

Consider that all substates in figure 17 have exit-activities. When event e8 or e5 occurs, we must
know the particular substate the object is in. For this reason, we create a method
leaving_region<name>() . The state is checked by an ordinary C++ switch-statement. If the
current state is found, its exit-method is executed. This is the signal to execute the exit activity of
the surrounding state region.

If there are nested state regions, the method of the outer state region calls the leaving-method of
the inner state region first. If the current state is part of the inner state region, the exit-activity of the
outer state region is also performed. If the current state is part of the outer state region, only the
exit-activity of the outer state region is executed. To achieve this, the leaving-method returns true if
the current state is contained.

With this implementation, the correct order of performing exit activities is guaranteed.

Example:

virtual bool leaving_region2 (void)
{
    bool result = false;
    switch (m_state)
    {
        case STATE1:  exit_statel();
                       result = true;
                       break;
        case STATE2:  exit_state2();
                       result = true;
                       break;
    
Example:
Looking at our example in figure 17 the code for the event methods of e5 and e8 change as follows:

```cpp
virtual void e8 (void)
{
    if (condition_e8)
    {
        if (leaving_region2())
            setState (STATE4);
    }
}
```

```cpp
virtual void e5 (void)
{
    if (leaving_region1())
        setState (STATE5);
}
```
3.5.2 Implementation of History Markers

For remembering the last active state of a state region the UML offers to model history markers. For each history marker, there is a member attribute. These attributes are initialized with value \textit{INITIAL}, because, at first, they are undefined and behave as the initial state of a state region. When a transition to a history marker fires, the state attribute is set to the value of the history marker attribute. Each time the object changes its state, the history marker attributes are updated.

As stated in the previous section, on entering a history marker all nested entry-activities must be executed. In the case of a shallow history state, these are only the entry-activities of the corresponding state region and the one of the atomic target state within the region. As the target state represented by history markers is evaluated dynamically, we cannot foresee which do-activity has to be processed. Thus, a code generator cannot determine entry- and do-method invocations statically. This problem is solved by introducing an \texttt{enter()}-method. This method first sets the object's state attribute to the value of the history marker and checks, whether the target state is contained by a state region and invokes the corresponding entry-methods. Finally, the do-activity of the target state is determined by means of a C++ switch-statement. The event methods initiating a transition to a history state call the corresponding \texttt{enter()}-method.

Consider the following sample code for the example in figure 17. Further, consider \texttt{state5} and \texttt{state3} in figure 17 to have entry-, exit- and do-activities.

```cpp
class AnyObject
{
protected:
    stateValue m_shallowHistReg1;
    stateValue m_deepHistReg1;
};
```
public:
AnyObject (void) // constructor
{
    m_shallowHistRegl = INITIAL;
    m_deppHistRegl = INITIAL;
};

protected:
virtual bool isUndefinedHistory (stateValue historyMarker)
{
    return (INITIAL == historyMarker);
};

virtual void enter_shallowHistRegionl (void)
{
    if (isUndefinedHistory(m_shallowHistRegl))
    {
        setState(STATE3);
    }
    else
    {
        setState (m_shallowHistRegl);
    }
    // in either case, update the deep history marker
    setDeepHistRegl (m_state);

    // a shallow history demands only the entry-activity of the
    // surrounding state region to be performed. If it was undefined
    // before, the object takes state3. Thus, the execution of
    // entry_region2() is also needed.

    if (isRegion1())
    {
        entry_region1();
    }

    if (isRegion2())
    {
        entry_region2();
    }

    // perform corresponding do-activity
    switch (m_state)
    {
        case STATE3:    entry_state3();
                        do_state3();
                        break;
        case STATE4:    entry_state4();
                        do_state4();
                        break;
        default:        break;  // nothing
    }  // switch
};
virtual void enter_DeepHistRegion1 (void)
{
    // the object's state must take the value of the deep history
    if (isUndefinedHistory (m_deepHistRegl))
    {
        setState (STATE3);
        setDeepHistRegl (m_state);
    }
    else
    {
        setState (m_deepHistRegl);
    }

    // also update the shallow history marker, if deep history
    // points to state region1
    if (isRegion1() && (! IsRegion2()))
        setShallowHistRegl (m_deepHistRegl);

    // a deep history demands all nested entry-activities to be
    // performed
    if (isRegion1())
    {
        entry_region1();
    }
    if (isRegion2())
    {
        entry_region2();
    }

    // perform corresponding do-activity
    switch (m_state)
    {
    case STATE1:
        entry_statel();
        do_statel();
        break;
    case STATE2:
        entry_state2();
        do_state2();
        break;
    case STATE3:
        entry_state3();
        do_state3();
        break;
    case STATE4:
        entry_state4();
        do_state4();
        break;
    default:
        break; // nothing
    }
    // switch
}
public:
virtual void e1 (void)
{
    switch (m_state)
    {
    case STATE5:
        exit_state5();
        setState (STATE3);
        setDeepHistReg1 (m_state);
        entry_region1();
        entry_region2();
        entry_state3();
        do_state3()
        break;
        default:
        break;    // event ignored
    }
    // switch
};

virtual void e2 (void)
{
    switch (m_state)
    {
    case STATE5:
        exit_state5();
        enter_DeepHistRegion1();
        break;
        default:
        break;    // event ignored
    }
    // switch
};

virtual void e3 (void)
{
    switch (m_state)
    {
    case STATE5:
        exit_state5();
        enter_shallowHistRegion1();
        break;
        default:
        break;    // event ignored
    }
    // switch
};
virtual void e5 (void)
{
    if (leaving_region1())
    {
        setState (STATE5);
        setDeepHistRegl (m_state);
        entry_state5();
        do_state5();
    }
}

virtual void e6 (void)
{
    switch (m_state)
    {
    case STATE4:
        setState (STATE3);
        setDeepHistRegl (m_state);
        entry_stateRegion2();
        entry_state3();
        do_state3();
        break;
    default:
        break; // event ignored
    }
    // switch
}

virtual void e7 (void)
{
    switch (m_state)
    {
    case STATE2:
        setState (STATE3);
        setDeepHistRegl (m_state);
        // the shallow history marker of region1 is un
        // concerned
        entry_state3();
        do_state3();
        break;
    default:
        break; // event ignored
    }
    // switch
}

virtual void e8 (void)
{
    if (leaving_region2())
    {
        setState (STATE4);
        setDeepHistRegl (m_state);
        setShallowHistRegl (m_state);
    }
}; // class
3.5.3 Concurrent Substates

As introduced in section 3.5, a state may be expanded into substates. Moreover, a state may contain two or more substate regions that are concurrent to each other. Such a state region is also called a "concurrent substate". A concurrent substate contains an ordinary statechart diagram. All concurrent substates are processed concurrently once the surrounding state region is entered. Within a state region, concurrent substates have "and-semantics", whereas non-concurrent substates follow "or-semantics".

Consider the example shown in figure 18. The example shows a sample statechart diagram of a student who takes classes. When entering the state region taking class incomplete, both concurrent substates begin running. The student object accepts events that cause state transitions in both substates such as the events lab done and pass. If the student does not successfully pass the final test, the complete state region is left and the student takes the state failed. The state passed is reached as soon as all concurrent substates have arrived in their final state and the event fail does not occur.

In this thesis, concurrent substates are not concerned because they are not considered as a meaningful modelling element. Concurrent substates should not be used for modelling parallel flows of control in response to received stimuli. There are the following reasons:
1) Conceptually, an object always takes only one state at any time. In the sense of well-defined internal object behaviour, an object should react to only one event at any time as external events can cause state changes.

2) If there is a transition from a single state within a concurrent state leaving the surrounding state region, there is no exact definition what impact this state change has on other concurrent substates with respect to ongoing state activity or recent events. Note that for do-activities of simple states within non-concurrent state regions, there are well-defined semantics (see section 3.3): In the case of an event, the running do-activity is interrupted and a state transition occurs.

Consider the example in figure 18. If event fail occurs, the question is if the student may continue doing his lab or what happens if the student has passed the first lab but the test failure is registered before. Furthermore, how should the running activity of state lab1 be handled?

Consider two concurrent substates with only one simple state. In this case, two concurrent threads of control work on two different do-activities but do not react to any event while processing. Consider the example in figure 19.

![Figure 19: Concurrent substates with only one simple state](image)

A person has got a problem and is looking for a solution. If he begins thinking, he walks around and chews his pencil at the same time. If he becomes tired from walking and chewing, he will
discontinue thinking and go to bed (automatic transition from state region *thinking* to state *fall asleep*). If he gets a sudden idea, he immediately takes it down on a piece of paper (transition with event *good idea* from state region *thinking* to state *take down*). In the first case, both activities are completed when the automatic transition fires. In the latter case, both flows of control are interrupted by an incoming event.

However, a situation in which concurrent substates contain a maximum of one state should be modelled by an activity diagram defined by the UML. An activity diagram is introduced for modelling a procedural flow of control within a method or state activity. In particular, activity diagrams can define the splitting and merging of control within any operation. As a model should always be as simple as possible, we will not add concurrency to a state of a statechart diagram but attach an activity diagram instead.

Considering the statechart diagram in figure 19, there would be a simple state *thinking* while the do-activity is expressed by a corresponding activity diagram.

The semantics of activity diagrams are not further explained as they are outside the scope of this thesis. For more information see [UML 97a] and [UML 97b].
3.6 Initial and Final States

Initial and final states do not only serve as a marking for the beginning and the end of nested states in the surrounding state region; at the highest level, they represent creation and the destruction of an object, respectively.

In the following sections we will examine how transitions from initial and to final states can be modelled and implemented.

3.6.1 Initial State

An initial state represents an object which has just been created. It is the first state the object takes immediately after creation. An initial state cannot have any activities as the object must exist before it can do anything. At the highest level, there is only one initial state symbol in every statechart diagram.

The question is, how can we model transitions from the initial state to any other state. First, a transition from an initial to another state can be driven by an external event.

\[ \text{initiate} \rightarrow \text{state1} \]

\[ \text{do / ...} \]

*figure 20: transition from an initial state with an external event*

In figure 20 the event *initiate* causes a transition to state *state1*. The target state performs some activity. The event method of event *initiate* checks the current state and then switches over to *state1*. As a requirement, the current state must have been set to *INITIAL* before. This is done by the constructor.
In the following, consider the Implementation of the example in figure 20:

```cpp
class AnyObject
{
public:
AnyObject (void) // constructor
{
    setState (INITIAL); // user code
};

virtual void initiate (void)
{
    switch (m_state)
    {
    case INITIAL:
        setState (STATE1);
        do_state1();
        break;

    default:
        break; // event ignored
    }
}; // class
```

A transition leading from the initial state to any other may also be an automatic one. In this case, the object takes the state the initial state points to immediately after being created and initialized (see figure 21).

![State Transition Diagram](image)

*figure 21: automatic transition from the initial state*

If the transition is automatic, there is a restriction concerning the activity of the target state and actions of the transition itself. The creation of objects is considered to be a synchronous class operation. During the execution of the constructor, the object is considered to be non-existent because a constructor belongs to the creation process of an object. When leaving the constructor, the object begins to exist having its own identity: a handle to the creator is returned. For this reason, it is not possible for the object-to-be to receive any message or to react to any event. If an object does not exist, one cannot initiate any behaviour, even the object itself. The constructor can
initialize the state attribute but it cannot perform a state transition including the initiation of state activity. Thus, the target state of an automatic transition from the initial state must not have entry- and do-activities. Exit-activity is allowed. Certainly, messages can be sent to objects from inside the constructor if they have been constructed before.

Note that this restriction results from a conceptual point of view following basic semantics of object-orientation. Many object-oriented programming languages do not prevent the programmer from implementing method calls within the constructor. However, as we are discussing at a logical level, we do not present a model having specific programming language capabilities in mind. We see it in a different way: Implementation is always a mere derivation from logical concepts using a particular programming language and particular implementation concepts.

Actions of transitions should be integrated into the constructor. Generally, an action of a transition could be implemented as an operation call, which in any case is not meaningful. One could argue that atomic statements such as incrementing an integer value may be modelled as actions of the transition because, considering the implementation approach of this chapter, the transition code will become part of the constructor code. Within the constructor, any instance attribute may be set because of a constructor’s initialization requirements. However, we do not model atomic statements as actions of an automatic transition from an initial state for the same reason as stated above: Logically, any modification as well as read-only access is only permitted for a fully instantiated object. If there seems to be the necessity for setting attribute values on the automatic transition flowing out of an initial state, the software engineer should consider this as an initialization of attributes to be transferred to the constructor at the modelling level.

As well, only completely constructed objects may send messages to other objects. Thus, we do not allow an automatic transition from the initial state to perform message-sendings.
The following skeletal code shows the implementation of the example in figure 21:

class AnyObject
{
    public:
    AnyObject (void) // constructor
    {
        setState (INITIAL); // user code
        setState (STATE1);
    }
}; // class

3.6.2 Final State

A final state represents an object which has been destructed. If a final state is reached, the object does not exist any longer. At the highest level (that means outside of all state regions), there may be not more than one initial state symbol in a statechart diagram.

Modelling transitions to a final state is less complicated. Generally, there is no restriction for either the source state or the transition itself. The source state can have any activity and the transition may have actions and message-sendings.

If the transition is fired by an external event, there is an ordinary event method as introduced in section 3.1 (see figure 23). Note that this event cannot carry any parameters as a final state does not have any state activity that could use these parameters.

If the transition leading to the final state is automatic, it should have a condition that must be evaluated to true before the transition can fire (see figure 22). If there is neither an external event nor a condition, the object will be destructed as soon as the do-activity of source state of the automatic transition is completed.

figure 22: automatic transition to a final state
This is not mandatory. It is not forbidden to have an automatic transition without a condition flowing to the final state. However, the software engineer should be aware of what he models in order to avoid unintentional effects.

When the object reaches a final state, it stops existing. Therefore, when implementing transitions to final states, the object must be destroyed. This means that the event method or, in the case of an automatic transition, the particular do-activity, respectively, deletes the object as soon as the state attribute is set to FINAL.

Consider the following implementation showing sample code for the examples in figure 22 and figure 23.

```cpp
class AnyObject
{
    public:
    virtual ~AnyObject (void) // destructor
    {
        // user code
    }
    protected:
    virtual void do_state2 (void)
    {
        // do / ...
        if (cond1) // automatic transition
        {
            setState (FINAL);
            delete this; // the destructor is called
        }
    ...
}; // class
```
class AnyObject
{
    public:
    virtual ~AnyObject (void)
    {
        // user code
    
    virtual void destruct (void)
    {
        switch (m_state)
        {
        case STATE2:
            setState (FINAL);
            delete this;
            break;

        default: // event ignored
        }
    }
}; // class
3.7 Consistency Constraints

In this section, we record several rules for modelling the internal behaviour of objects by statechart diagrams. These rules are an addition to the common statechart diagram semantics defined by the UML [UML 97d]. It is recommended to follow these rules in order to keep statechart diagrams consistent and to make full use of code generation facilities.

Modelling Constraint 1 ("Interruptable Activities") is located in section 3.4. It suggests a piece of modelling advice for guaranteeing exclusive reading and setting of the state attribute in the case of multithreading within an object resulting from realizing interruptable do-activities.

**Modelling Constraint 2: Parameters of Events**

Events can have parameters to provide necessary input for the state activity. A transition is identified with its condition, its actions and message sendings as well as its source and target state. An event is identified with its name and formal parameters. Thus all transitions with the same event (the same event name) must carry the same number of formal parameters with the same types. The names of the formal parameters must not be necessarily the same. However, it is recommended to use the same parameter identifiers for the sake of readability of the model.

For each state, it is important to check two things: First, all incoming transitions must have at least as many parameters as the state activity demands. Some of the events may offer more parameters, but this is not a problem because the state activity uses only the parameters it needs. Note, in this connection, that automatic transitions cannot offer any parameters.

Second, the names of the formal parameters provided by an incoming transition must conform to the names of the formal parameters of all other incoming transitions. The reason is that, once a
state is entered, the information about the occurred event is lost. The state activity must be able to access all parameters by name independent of the event that led to it.

In the case of a transition to a state region the above two points require special consideration, since the external event of that transition must serve all parameters for the nested entry-activities.

Following these rules, there will be no ambiguity for a code generator when producing code for event methods as well as for the access to parameters within the state methods.

**Modelling Constraint 3: Conditions of Outgoing Transitions**

The conditions (if any) of all outgoing transitions that have the same event must be disjoint; otherwise, the state machines will not behave in a deterministic way. Any state transition fired by event 1 in the following illustration behaves deterministically:

```
state 1
  event 1 [a]
    event 1 [not a and not b]
  event 1 [not a and b]
```

**Modelling Constraint 4: Automatic Transitions**

It makes no sense to have two or more unconditional automatic transitions from a single state, because there is no deterministic evaluation of a target state. The following diagram illustrates the problem:
Also, the existence of an outgoing automatic transition without a condition lacks sense, if there is at least one outgoing transition fired by an external event (see the illustration below).

Furthermore, a state must not have an automatic unconditional transition leading to itself as infinite recursive execution of all state activity would be the result. As shown in the following illustration the transition of \textit{event1} will never be fired.

---

**Modelling Constraint 5: Automatic Transitions and State Regions**

If there is a state region without a final state in it, there must not be an automatic transition without a condition.

The situation in the illustration below is not allowed: As soon as the state region \textit{1} is entered and state \textit{3} becomes the active state (and the activity has completed its execution), the automatic transition fires at once. There is no chance to reach state \textit{1} or state \textit{2} anytime.

---

Following the terminology of the UML metamodel, a statechart diagram is attached to a class and represents the specification of a state machine like a class represents the specification of an object. Thus a state machine can be interpreted as an instance of a statechart diagram at runtime. Each object of a class having a statechart diagram owns a state machine.

7
An automatic transition including a condition is ok. It is up to the modeller to establish a meaningful condition.

In accordance with the UML v1.1 the statechart diagram in the illustration below is correct. Event e6 leads to the final state of the state region. The unconditional automatic transition fires as soon as the final state is reached.

---

**Modelling Constraint 6: Avoiding Exceptions During Transitions**

The fact that transitions can have an action sequence and message-sendings forces us to reflect on exceptions that can be raised. This problem becomes more complex if, at the implementation level,
one of the actions becomes an operation call. This modelling constraint addresses consequences for the code generator.

An exception violates the atomicity of a transition and thus endangers the internal consistency of the object. In the case of an exception, the transition is not fully fired and the object ends in an inconsistent state. Attributes may have been changed and messages to other objects may have been sent since the occurrence of the corresponding event. Association links to other objects may, also, have been deleted or established.

Thus, there is a requirement for a rollback mechanism (similar to transactions of database operations) for all actions and message-sendings of a transition that have been performed before an exception was thrown. A rollback only concerns a certain set of actions and operations that must be withdrawn, independent of any point in time.

Consider the following skeletal version of code for the method event1() of figure 16:

```c++
virtual void event1 (<type1> param1, <type2> param2)
{
    try
    {
        switch (m_state)
        {
            case STATE1:
            
            // no thread to be killed because state1
            // has no do-activity
            // no exit-activity defined for state1

            setState (STATE3);
            action1;
            obj1->message1();
            entry_state3 (param1, param2);
            // provide the parameter values
            m_paramStore->setParam1 (param1)
            m_paramStore->setParam2 (param2);
            
            // create the thread
            m_threadHandle = createThread (do_state3());
        
        break;
    
```
Concerning the issue of established or deleted associations and the altering of attribute values, there is a possibility of implementing such a rollback mechanism: The design pattern "Memento" of Gamma et al. (see [Gamma 96]) could be used to restore the status of the object before the transition has fired. The only adaptation of the pattern is that the object stores its Memento objects itself.

However, there is still a problem: States, attributes and association links of other objects may be manipulated by message-sendings. A local rollback mechanism is not sufficient here. We cannot know how many messages of the transition have been sent and what impact these messages already had on other objects and the whole system. It is very difficult to establish a rollback function throughout the system.

This problem goes far beyond the scope of this thesis. However, we can avoid this problem by proposing a second modelling rule for statechart diagrams. This rule does not increase the complexity of the statechart diagram and leaves the responsibility for the named problem on the software modeller:

"A software engineer should avoid applying message-sendings and actions to a transition by bringing up an additional state. The entry block of that state includes all actions and message-sendings. An automatic transition fires from the new state to the original target state."

76
The actions performed by the transition are now captured by the entry-activity of the new state (see figure 24) which are atomic. Since the software engineer is responsible for the consistency of his model, he must be aware of any errors and exceptions that can arise; therefore he should implement exception handling for each statement of the entry code.

**Modelling Constraint 7: Automatic Transitions and Initial and Final States**

There are additional consistency constraints when modelling transitions from initial and to final states. See section 3.6 for details.
3.8 Discussion of the Alternative Implementation Concept

In this section, the alternative implementation concept is compared to the implementation approaches presented in chapter 2. It makes a compromise on simplicity of the resulting code and performance.

Common Features and Technical Differences

Following the Shlaer/Mellor approach, the alternative implementation technique for statechart diagrams introduces a method for all events as well as for the state activities. A state is explicitly represented by an enumeration type and is stored in one instance attribute.

The presented implementation technique does without any additional object creation. There is only one object to be created, which is an instance of a class owning a statechart diagram. Both the State pattern and the State Table pattern consist of more than one component.

UML Conformance and Consistency Constraints

All UML elements for statechart diagrams are addressed and discussed; implementation concepts are proposed. As the UML metamodel specification is not strict enough, advice for modelling statechart diagrams by a set of modelling rules is given. Following these rules guarantees consistent and correct lifecycles of objects.

In [MW 98] the question of inheritance of statechart diagrams is examined. This paper addresses the question how statechart diagrams can be specialized at the subclass-level. The paper offers a set of rules on how to modify a statechart diagram of the superclass at the level of the subclass and introduces the concept of "reaction conformity" of subtypes. It is shown that these rules conform to common object-oriented inheritance semantics of types and classes.
Simplicity and Extensibility

According to Shlaer/Mellor, the presented implementation concepts separates external event methods from state activity methods. Thus, there is no code redundancy at all. All state logic is concentrated in only one class definition. With knowledge about the alternative implementation concept, the code can easily be followed. There is a maximum of readability as well as maintainability for the software engineer.

The following table gives an overview of the impact that changes to the statechart diagram have on the code when using the alternative implementation approach.

<table>
<thead>
<tr>
<th>Pierre Metz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additional event (incl. a transition)</td>
</tr>
<tr>
<td>Interface change of the Context Object</td>
</tr>
<tr>
<td>Additional state</td>
</tr>
<tr>
<td>New state method for the Context Object</td>
</tr>
<tr>
<td>Additional transition without an external event</td>
</tr>
<tr>
<td>Redefinition of a state method</td>
</tr>
<tr>
<td>Redirect a transition arrow</td>
</tr>
<tr>
<td>Redefinition of an event method</td>
</tr>
<tr>
<td>Redirect a transition arrow without an external event</td>
</tr>
<tr>
<td>Redefinition of a state method</td>
</tr>
</tbody>
</table>

Performance and Memory Requirements

The alternative implementation technique does not introduce objects for states or transitions. There are no additional object instances; there is an instance of the Context Object instance only. Therefore, a minimum of memory is needed. Memory has to be allocated for one object only.

A reaction to an external event is addressed by calling the corresponding operation of the object. This maps directly to a virtual function call. There is a certain performance penalty as the current state is checked by a switch-statement. However, there is no runtime cost for indirection as all state logic is concentrated in one class.
Douglass (see section 2.4) needs a constant time effort for looking up a transition because of the use of a state table. Contrary, Gamma et al. [Gamma 96] say that sometimes a virtual function call is more efficient than a table-lookup.
4 Code Generation Concepts for Multithreading

The approaches to mapping states and events to methods and attributes, described in chapter 2 and chapter 3, work fine for single-processor operating systems that do not support multitasking. As there is only one thread of control at any point, all operation calls can only be performed sequentially. The code generation concepts presented so far have to be extended to be multi-thread safe.

We will first examine different ways of introducing threads to object oriented software. Each way is presented briefly in the following sections:

The first possibility is to have vertical thread organization within a system supporting external interfaces (e.g., for graphical user interfaces, external software systems, hardware elements etc.). That means an application has several threads, each of these threads is responsible for the processing of one use case. If an actor invokes a use case, the corresponding thread controls the scenario behind it. This is the case if there is a multi-user application, for instance. Two actors may use different functions of the system at the same time (see figure 25).

If necessary, concurrency can be increased by having more than one thread within a scenario. This is the case when creating an additional thread for asynchronous operation invocations by the main-
thread of the scenario. Asynchronous communication means that the sender immediately proceeds with its activity after having sent an asynchronous message. It does not wait for the operation to be completed.

The main thread of the scenario branches into two concurrent flows of control. Each time an object calls an asynchronous operation of another object, an additional thread is created performing the activity defined by the operation implementation of the receiver (see figure 26). It terminates when the operation is completed. After having initiated a thread, the main thread of the scenario proceeds.

Applying a thread for synchronous operations is not meaningful because the sender waits for the completion of the operation before he continues. If a synchronous operation gets its own thread, the sender would have to be blocked anyway. Hence synchronous operation invocation is performed sequentially.

A third philosophy for introducing concurrency to object-oriented software is the following: A thread is established for a single object or a group of objects as a logical unit (Object Thread). Following this approach, we do not create threads dependent on operation calls but let objects exist in parallel. This is done by introducing a thread to a single object. There is only one thread working on an object at any point. An object receives operation call requests and serves them (see figure...
27). Its thread is responsible for the invocation of operations on the object as a result of an incoming request.

![Figure 27: Object Thread](image)

Generally, when thinking about multi-threading, the software engineer has to be aware of potential concurrency conflicts.

In figure 25 and figure 26, there are two threads flowing through object ObjC. More than one thread working on the same object may cause internal inconsistencies if sensitive data is not protected against multi-thread impact.

The multi-thread strategy presented in figure 27 is not affected by this problem because there is only one thread working on an object at any time. Parallel operation invocation is not possible here.

The object always maintains a consistent condition.

Regarding the code generation concepts of chapter 3, this problem concerns state transitions including the reading and setting of the state attribute. The access to the state attribute and transitions of an object form a critical section because it must not be interruptable. If there is no protection for all this, objects may end in an inconsistent state.

In figure 25 the two objects ObjD and ObjB, each of them belonging to a different scenario thread, hold a reference to ObjC. Consider ObjD calling an event method of ObjC. If there is a context switch immediately after having read the state attribute but before setting it, ObjB calls another event method of ObjC, considers the wrong state attribute value and may also perform a transition. ObjB has violated the atomicity of the transition invoked by ObjD.
In figure 26, the problem is the same. Two threads working on ObjC and may also cause a collision.

Of course, in the presence of concurrency, there is a consistency problem regarding the modification and reading of ordinary instance and class attributes as well. This concerns the domain code written by the software engineer. Thus, keeping the access to instance and class attributes consistent is up to the software developer; it is not the responsibility of a code generator. Therefore it is not considered further in this work.

The following three sections offer solutions for multi-thread safety concerning the three mentioned multi-thread strategies. Advantages and disadvantages of the implementation strategies are discussed.

As multi-thread safety of object-oriented software, in general, is still an open and complex problem, the issues outlined here do not claim to be considered as a silver bullet. They just serve as an adaptation of the basic implementation concepts (see chapter 2 as well as chapter 3) to concurrency problems.
4.1 Thread for Each Use Case Scenario

A solution for the synchronization problems of the first kind of thread design are semaphores that make the manipulation of the state attribute and a transition mutually exclusive by "locking" the particular code fragment. This section shows how the use of semaphores as a basic synchronization mechanism for concurrent threads of control is integrated into the alternative code generation concept (see chapter 3).

A transition forms a critical section that is protected by a semaphore. The acquisition and release of the semaphore is placed at the beginning and end, respectively, of the source code that represents a transition.

Each instance of the class owning a statechart diagram is supplied with its own semaphore object. The semaphore object, having an initial count value of 1 for mutual exclusion, is created within the constructor of the class.

Due to our basic implementation concepts for statechart diagrams in chapter 3, only event methods manipulate the state attribute. They are responsible for signaling and waiting for the mutex semaphore.

As an enhancement we generate operations for semaphore handling (acquisition and release), which are called from within the event methods. The software engineer does not need to adapt all event methods but only has to fill in the specific code of the operating system into the semaphore operations after code generation. The default implementation of these operations is empty.
Consider the following sample code for the event method of event1 in figure 15:

class AnyObject
{
    // ...

protected:
HANDLE mySemaphore; // member attribute which stores the
// handle of the thread returned by the
// API operation call for creating a
// mutex semaphore.

virtual void acquireSema (void)
{
    // specific API call of the underlying operating system
}

virtual void releaseSema (void)
{
    // specific API call of the underlying operating system
}

public:
virtual void event1 (<type1> param1, <type1> param2)
{
    this->acquireSema();
    switch (m_state)
    {
    case STATE1:
        if (condition1)
        {
            setState (STATE3);
            action1;
            obj1->message1();
            entry_state3 (param1, param2);
            this->releaseSema();
            do_state3 (param1, param2);
        }
        break;

    default: ; // event ignored
    } // switch

} // event1

// ...

}; // class

Regarding automatic transitions that cause state changes after completion of the current do-activity, the state method of the current state has to acquire and release the semaphore.
virtual void state_m (void)
{
    // ...do_activity as defined in the statechart diagram
    this->acquireSema();
    setState (STATE_n);
    this->releaseSema();
    do_state_n();
};

4.1.1 Discussion of the Locking Approach

Applying mutex semaphores to event methods guarantees atomic transitions. However, there are three important disadvantages:

Regardless of how many threads there are in the system, with the mutex semaphore strategy, methods are always processed sequentially. A method call returns after execution. This follows synchronous communication semantics.

Unfortunately, it is not possible to implement real asynchronous operation calls by which the sender of a message proceeds with some other activity while the receiver executes the corresponding operation.

A big disadvantage is that the semaphore operations are part of the domain code. The domain code is not independent of the synchronization technique. The synchronization should be transparent; the programmer should not be concerned with it.

The locking strategy works fine as long as the threads access an identical code fragment that is running in the same address space. What we cannot cope with is the need to reference an object belonging to a different process.
4.2 Thread Per Method Invocation

The second possibility of applying threads to object oriented software is to create a thread for each method invocation, which means processing more than one operation of an object at the same time. When an operation is completed, the thread terminates. This section addresses about the implementation of that kind of thread design.

First, we transfer each public method of the class (e.g., event methods) to the non-public section of the class definition. Further, we introduce public methods with the same names, the only action of which is to create a thread with the non-public method as the execution code. A naming difference between these two methods can be made by prepending the non-public event method with two underscore symbols.

Since events that are modelled in UML statechart diagrams are asynchronous operation calls (see [UML 97a]), we can apply this technique at least to all event methods. Operations, that are designed to be asynchronous but are not modelled in statechart diagrams can be implemented like this as well.

We can guarantee synchronous operation calls by not applying this implementation concept. Such operation invocations are processed sequentially in the context of the thread of the sender (see method `getObjID()` below).

Consider the following sample code:

```cpp
class AnyObject {
    // ...

    public:
    virtual const String& getObjID (void) const {
        return String("It's Me!");
    }
};
```
public:
  virtual void event1 (<type1> param1, <type1> param2) {
    // API call
    createThread (__event1, param1, param2);
  }

protected:
  virtual void __event1 (<type1> param1, <type1> param2) {
    switch (m_state)
    {
      case STATE1:
        if (condition1)
        {
          setState (STATE3);
          action1;
          obj1->message1();
          entry_state3 (param1, param2);
          do_state3 (param1, param2);
        }
      default: ; // event ignored
    } // switch
  } // switch
  // ...
}; // class

4.2.1 Known Uses

The object request broker product "Orbix" of Iona Technologies [Iona 96] offers to establish a
thread per method invocation of server objects in another address space. Orbix fully supports the
CORBA 2.2 standard for distributed object oriented software. The thread per method invocation
mode itself is not part of the CORBA standard; it is an additional feature provided by Orbix.

4.2.2 Discussion of the Thread Per Method Invocation Approach

Implementing a thread per method invocation offers asynchronous communication between
objects. The sending object can go on with its activity while the particular thread of the receiving
object is running. Also, synchronous invocation of operations is possible.
The problem of consistent states and atomicity of transitions remains. It may happen that two or more asynchronously invoked methods want to modify the state attribute. Using this approach, it is necessary to hold out for critical sections as well and again protect them by semaphores.

The thread operations remain as part of the domain code. This can be avoided by using a proxy (see design pattern 'Proxy' [Gamma 96]) that forwards the operation calls and creates a thread for asynchronous operations first, in the context of which the operation invocation takes place. The use of proxies for establishing transparency and independence from code generation concepts is shown in section 4.3.1. It is not applied here, because we will develop a much more powerful concept in section 4.3.

The problem of referencing objects belonging to different address spaces mentioned in section 4.1.1 is not solved. The communication and information interchange between processes requires special treatment.
4.3 Object Threads

Object Threads as defined in the introduction of chapter 4 may exist for single objects as well as for a composite of objects. If objects exist in parallel, a strategy for dynamic message interchange across thread boundaries is needed. In the following, a message queueing mechanism for asynchronous as well as synchronous messages is developed. This mechanism is called Message Queue Concept and serves as an advanced implementation pattern for transparent Object Thread interaction.

The Message Queue Concept is independent of any basic implementation concept of statechart diagrams. It may be applied to the concepts presented in chapter 2 as well as in chapter 3. Further, it serves as the basis of an event queueing concept for Object Threads presented in chapter 5.

4.3.1 Message Queues For Asynchronous Messages

Each object having its own thread (Object Thread) gets a message queue. The Object Threads interchange information and communicate with each other by placing messages into each other’s message queue (see illustration in figure 28).

*figure 28: outline of the Message Queue Concept for Object Threads*
Every Object Thread takes messages out of its queue sequentially and invokes the corresponding operation. If the queue is empty, the thread is suspended by the operating system. If another message comes in, the thread is resumed.

The synchronization of the access to a particular message queue between the different Object Threads is managed by the message queue itself. How this is achieved is explained in detail in appendix A.1.

The sequential execution of operations is up to the thread of a receiver. As an Object Thread processes incoming requests sequentially. An object executes only one operation at any time. This makes sense because if an object could execute more than one operation (e.g., a thread for each operation invocation), state transitions become a critical section. Establishing a thread for each object would be of no advantage then. In general, an object as a logical entity should not be executing more than one operation. 8

It is meaningful that the insertion of messages into the message queue is not visible to the sending object. The sender should invoke an operation call on the receiver just as if there is only one thread of control. Sending a message to an Object Thread should be completely transparent.

For this reason, we introduce a proxy class 9 for each domain class. Both classes share the same interface. When objects of the domain classes must communicate with each other (by using each other’s message queue), they actually communicate with each other’s proxy object. The sender just uses the interface of object B but does not know what is behind it.

The proxy object transforms the operation call into a representation of a message and stores it into the message queue of the receiver. This action is performed in the context of the Object Thread of the sender.

8 For performance purposes, operations that return a value and do not modify the object may be allowed to be executed concurrently. For automated code generation, such read-only operations must be specially marked in the model (e.g., by stereotypes), which is a question of object-oriented analysis and design methods.

9 The concept of proxies as representatives of objects is shown in [Gamma 96], [Buschmann 96] and [Vlissides 96].
Chapter 4  

Code Generation Concepts for Multithreading

Transparent design of message interchange is meaningful because the software system can be ported from one operating system to another without changing the domain classes. In a single task system, a client references an object of the domain class directly; in a multitasking or distributed system, a client references a proxy. For enable switching between environments, we must provide two types of the domain class: we need a "pure" class that represents the original definition and another one that includes thread properties. Employing this approach we are able to reuse the platform-independent domain classes in other projects.

The design and implementation concept for message queues concerning asynchronous messages is presented by the following example.

*Example:*

We consider a design model with a class *Person* having an association to itself meaning that any person may communicate with another person (figure 29). In this example, class *Person* just offers to go to work in the morning and to go to sleep in the evening on command. The declarative stereotype «thread» is assigned to class *Person* indicating that each instance will become an Object Thread. This stereotype is part of the core UML (see [UML 97b]) and can be identified by a code generator.

![Figure 29: Example of a class "Person"](image)

We apply the Message Queue Concept as shown in figure 30. The classes *ProxyPerson* and *Person* share the same interface by inheriting from the interface class *AbstractPerson*. Class *Person*
represents the pure domain class, whereas class `PersonThread` encompasses thread semantics by inheriting from class `Thread`. `PersonThread` takes the full implementation from class `Person`.

The proxy class implements the operations of `AbstractPerson` in the way that operation calls are transformed into message representations.

The example in figure 31 shows a sample object diagram.

The `Object Thread` pulls messages from the message queue. In order for them to be stored, the messages must have some kind of representation instances that can be identified. For example, messages can be implemented as integer or string values which serve as the elements for the message queue. In order to invoke the corresponding event method, the receiver’s `Object Thread` has to decode these values then.
A more elegant but more complex solution is to model messages as objects. This approach is based on the design pattern 'Visitor' [Gamma 96]:

We introduce a base class EventPerson\(^{10}\) (see figure 32). Additionally, there is a subclass of EventPerson for every operation that is defined in the interface of class Person. Class EventPerson defines an abstract operation operate(). Each subclass implements this abstract operation in the following way: The method operate() expects a reference to a PersonThread as parameter and immediately invokes the operation with the name of the event subclass on it. For a sample scenario consider figure 34.

In figure 32 an extension to the class diagram in figure 30 is shown. The inheritance relationships of PersonThread and ProxyPerson are left out for better clarity of the diagram. The modelling elements already known are marked grey.

\(^{10}\) The reader may wonder why there are events placed in a message queue as he rather expects messages to be inserted. The reason is the following:
A possible interpretation of the term 'external event' is the receipt of a message. An object can receive a message and then accept, ignore or else queue the corresponding event to remember it later when arriving in the correct state.
As the only purpose of event objects is to invoke exactly one operation of the receiver, we actually do not want to establish two identical class hierarchies of message representations named 'EventPerson' and 'MessagePerson' just to have a logical distinction. It is meaningful to use the same instances for representing messages and deferred events. We decided on the naming convention 'event' for our class hierarchy because the term 'message' is not suitable for deferred events but the other way round fits. We will have a closer look at this in chapter 5.
Actually, figure 32 and figure 30 illustrate a simplified, though technically incomplete, example of the Message Queue concept. The class Queue must be template class. Further, it cannot be associated directly. The template parameter must be bound to some type first (see [UML 97c]). The adaption in figure 33 shows the correct model fragment:
Interaction

The sending object calls an operation of the receiver's proxy. The proxy creates a new message instance of the corresponding subclass of `EventPerson` and inserts it into the message queue of the receiver. The receiving Object Thread is resumed (if the message queue was empty before) and obtains the message instance from the queue. It calls the operation `operate()` of the message instance with a reference to itself as parameter. With this parameter the message instance references the Object Thread and invokes the corresponding method.

The sequence diagram in figure 34 shows the request `goToSleep()` based on figure 32.

This approach is more elegant compared to message representations as integers or strings. The advantage is that the class `PersonThread` must know only about the interface of `EventPerson` and `Queue` and does not need to implement any message identifier decoding logic. There is a minimum
of coupling which provides a maximum of extensibility and resuability. Furthermore, it can be completely generated automatically.

The disadvantage is that the resulting model includes event class hierarchies for every class having a statechart diagram. The interaction design is more complex and thus more difficult to understand.
4.3.2 Message Queue Extension for Synchronous Messages

Section 4.3.1 introduces a transparent queueing mechanism for asynchronous messages for Object Threads. This mechanism does not offer synchronous communication semantics; it does not allow to block the sending thread during execution time of an operation of the receiver. In this section, we will extend the Message Queue Concept to handle synchronous messages.

Synchronous communication is defined as the sender of a message to be waiting until the message is processed by the receiver. This refers not only to operations that return a value; operations without return information may also be designated as synchronous.

As an example, we refer to class Person in section 4.3.1. It is extended to return its name, its age and can also be asked to open a door so that someone can enter the room (see figure 35). These are synchronous operations as the sender needs to wait until the request is served. We apply the self-defined stereotypes «syn» and «asyn» to indicate synchronous and asynchronous semantics at the modelling level. These stereotypes are not part of the UML core, but the UML metamodel facilitates meaningful stereotypes for accommodating special OO-methods and personal development process requirements. The latter stereotype is considered as default and may be omitted.

![Diagram of class Person with synchronous operations]

*figure 35: class person with synchronous operations*
In order to add synchronous message invocation, we recall the queue synchronization technique. It says that an Object Thread receives messages by removing them out of a queue. If it is empty, the thread is suspendend.

We will also use this idea here. The class *Proxy* gets a second association to the template class *Queue* in the role of *returnValueSlot* (see figure 36). Any return value or acknowledgement of execution completion is inserted here. Moreover, we extend the event class hierarchy with synchronous events for each synchronous operation. Like asynchronous events, a synchronous event calls the corresponding operation of class *Person*.

Each time a client calls a synchronous operation of a person, the person’s proxy creates the corresponding event instance and passes the return value slot as constructor parameter (see the sequence diagram shown in figure 38). After that, the proxy expects return information and calls the *obtain()* operation of its return value slot. Unless the return information is available, the thread of the sending object is blocked.

*Remark:*

There will be at most one element in the return value slot, so that a container is not really needed. Nevertheless, we will use the queue because we do not want to implement a special information slot additionally. As most container classes allow the specification of a maximum capacity, we will fix the maximum entries of the return queue to one.
The modelling elements already introduced are marked grey.
The example in figure 37 shows a sample object diagram. As well, the elements marked grey are already known:

![Object Diagram](image)

**figure 37: example of an object diagram for the Message Queue Concept with synchronous message interchange**

**Interaction:**

A proxy forwards a particular operation call by creating a corresponding message instance and placing it into the message queue. The receiving Object Thread takes these message instances out of the queue sequentially and invokes the `operate()`-method on them.

A message instance in turn calls the corresponding synchronous operation of the Object Thread instance (see figure 38). When the Object Thread instance has executed the operation, the thread of control returns to the message instance. The message instance takes the return value (if there is any) and puts it into the return value slot.

If there is no return value, the event instance inserts a default value. This is necessary to signal the semaphore of the return value slot. The default value does not represent a return value but an acknowledgement of the execution of the synchronous operation.
The thread of the proxy is resumed immediately and removes the return information. As each operation of the proxy is the counterpart of a person's operations, it knows if there is a return value different from the default value and returns it to the client immediately.

![Diagram](image)

*figure 38: sample sequence diagram for synchronous interaction by the use of message queue*
4.3.3 Parameters of Queued Messages

In the previous sections we have not addressed the problem of carrying parameters to operations of the receiver. The question is how parameters and message object instances can be combined. First, we will examine at how a message and a parameter can be defined at the meta-level (see figure 39).

A message can transport any amount of parameters, in particular zero. A parameter instance belongs to exactly one message, one parameter cannot be shared with two messages. This is expressed by a composite aggregation between the classes message and parameter. A parameter consists of a type (e.g., a name of a person or a reference to an object) and a value. A certain instance of a value cannot be shared by two parameters (composite aggregation between the classes parameter and value).

Even if two parameters of type integer carry the value of 2, there are two instances of class value. A message can have several parameters of type integer, each having its own value instance. In contrast, a type can be shared by any number of parameters, because a type is a meta-specification of an element.

There are two possibilities to let a message object keep arguments for operations. First, we can implement parameters as member attributes for the particular message class. Second, we can
provide a list for the message object, in which parameters can be stored and retrieved. For performance purposes, they should be accessible directly by external key. The example in figure 40 shows the base class `EventPerson` having a qualified association to the class `Parameter`.

When trying to implement this, we realize that we have to create subclasses of class `value` (figure 40) for each type that we need. There would be corresponding subclasses of class `Value`. However, when dealing with primitive data types such as `int`, `float`, `char`, a class would need to be established for every data type provided by the programming languages supported by a code generator.

As this is too complex and of no real advantage, we have decided on the first option. This can easily be done since in the UML parameters of operations are part of the operation declaration syntax. A code generator can inspect the operation declarations and then create a message class for each operation with member attributes conforming to the types of the modelled operation parameters.

![Diagram](image)

* figure 40: parameters stored in a list

If a parameter represents an object reference, the message class gets a pointer attribute with the type of the interface of that object (e.g., class `AbstractPerson`) because at runtime a reference to a proxy instance must be held.

Following this approach, everything that has to do with one message is concentrated in one class then.

---

11 For explanation of composite aggregation and shared aggregation see [UML 96] and [UML 97c]
Example:

The class Person introduced in figure 32 can be asked to pay a bill in a restaurant, shop etc. The parameter attributes are the price that has to be paid and the waiter ID who takes the money.

The proxyPerson object implements the method payBill() and expects both parameters to be handed in. The proxy itself creates a new message instance of class PersonPayBill and passes these parameters to the constructor.

```
EventPerson <- ProxyPerson
«instantiates»

«async»
PersonPayBill

# price : float
# waiter : Person
+ operate()
```

*figure 41: example of parameters as member attributes*

Remark:

In our example there is an object reference as parameter. This is to show that it is not a problem to interchange addresses of objects because we consider threads here and not processes. In particular, it is allowed to submit references to objects that represent threads. In this case synchronization is guaranteed because there is an Object Thread behind it.

Generally, it is not meaningful to transfer object references that do not have a thread of their own to other Object Threads. An examination and explanation of these points is carried out in chapter 6.

In the following section, the implementation for the example in figure 41 is shown for better understanding.
4.3.3.1 Implementation

We take a look at the implementation of the classes shown in Figure 41:

```cpp
class ProxyPerson : public AbstractPerson
{
    // ...

    virtual void payBill (float price, AbstractPerson* waiter)
    {
        m_receiverQueue->insert (new PersonPayBill (price, waiter));
    }
}; // class

class PersonPayBill : public EventPerson
{
    protected:
    const float m_price;
    const abstractPerson* const m_waiter; // this attribute actually references the waiter's proxy!

    public:
    PersonPayBill (const float price, const abstractPerson* waiter)
    : m_price (price), m_waiter (waiter);

    public:
    virtual void operate (AbstractPerson* thePerson)
    {
        thePerson->payBill (m_price, m_waiter);
    }
}; // class
```
4.3.4 Message Queues for Groups of Objects

In the previous sections message queues were considered for single objects. Now we discuss this concept with respect to Object Threads of logical groups of objects. Such object groups may be a composite object, a subsystem etc. The object being the Object Thread of the object group then acts as a facade (compare with design pattern 'Facade' in [Gamma 96]). The flow of control flows through several objects.

Why not package several objects together and provide a thread for it?

If we establish threads for do-activities (see section 3.4) or want to have asynchronous operation invocation, we again have a conflict regarding non-interruptible transitions and state manipulation. If we do not establish threads for the do-activity of an object's state and we do not need asynchronous operation calls, there is no concurrency within the thread and all method calls are executed sequentially.

Consider the following situations:

1.) A subsystem contains three objects, a facade object and two domain objects A and B, both having a state machine\(^2\). Object A references object B. Now the subsystem receives a message. The facade object reacts to this event by invoking an event method of object A and another event method of object B. This happens in the context of the subsystem Object Thread. When having returned from the first call, the do-thread of object A is running. The facade object calls object B and before the corresponding transition is completed, the do-thread of object A references object B and also invokes an event method of B. We realize a collision as discussed in the introduction of section 4.3.
2.) The do-thread of object A, initiated by an event method call of the facade object, calls a method of object B that manipulates an attribute. At the same time, the facade object wants to read this attribute value in order to compute some other data. In this situation, it is unsure whether the facade object gets the new attribute value determined by object A.

We see, applying the concept of message queues to single objects and groups of objects is not the same. When applied to single objects, it avoids the need for synchronization mechanisms. Applied to groups of objects, we still have the problem of threads that flow through the same code and therefore need to implement protection mechanisms.

4.3.5 Related Patterns

The Message Queue Concept is built upon the design patterns 'Visitor' and 'Proxy' introduced by Gamma et al. ([Gamma 96]).

The Message Queue Concept uses queues for interchanging information following the semantics of the well known producer/consumer problem.

William C. Wake, B. Douglas Wake and Edward A. Fox show the general advantages of using queues in low-level interaction structures of application elements. They propose four sorts of "queue-patterns", each addressing a particular aspect that can be found in reactive and interactive systems (see [Vlissides 96]).

Regarding functionality, a pattern similar to the Message Queue Concept is the "Active Object Pattern" introduced by R. Greg Lavender and Douglas C. Schmidt ([Vlissides 96]). The pattern is intended to be used for message dispatching instances within a system that performs nonblocking operations (e.g., gateways).

12 Following the terminology of the UML metamodel, a statechart diagram is attached to a class and represents the specification of a state machine like a class represents the specification of an object
Chapter 4  Code Generation Concepts for Multithreading

Common Features of the Message Queue Concept and the Active Object Pattern

The intention of the Active Object Pattern is to decouple the invocation of an operation from its execution in multi-thread environments.

The Active Object Pattern also constructs messages as objects and uses a queue following the producer/consumer synchronization policy. The transformation of messages into objects is also transparently done by a proxy.

A Message Queue Scheduler retrieves the messages and dispatches them. This functionality refers to the implementation of the execute() -method of the Object Thread (section 4.3.1).

Differences between the Message Queue Concept and the Active Object Pattern:

The Message Queue Concept blocks the client immediately after having sent a synchronous message. It is intended to be used for transparent unidirectional communication between two concurrent objects. Hence the sender is the only owner of the result value.

The Active Object Pattern manages synchronous operations in a different way. Control is returned to the client immediately by returning a handle to a “Future Object”. A Future Object is a synchronization object that provides the return value of the corresponding synchronous operation and has “write-once, read-many” semantics. It blocks any reader until the Message Queue Scheduler has forwarded the result to it. The return value is not exclusively provided for the client. Any object knowing about the handle may obtain the result.

This implementation of synchronous operations enables the client to perform some action before requesting the return value. The waiting time for the completion of the synchronous operation can be used otherwise. Synchronous method call semantics are resolved into two asynchronous messages here.

As a consequence, the Active Object Pattern is not fully transparent to the client.

instance. Thus a state machine can be interpreted as an instance of a statechart diagram at runtime. Each
Finally, the Message Queue Concept offers more implementation details. The Active Object Pattern does not offer great detail about how the dispatch of message instances results in an operation invocation of the Active Object.

### 4.3.6 Known Uses

The multithreaded version of Orbix 1.3 uses the same basic scheme as the Message Queue Concept to submit CORBA remote requests to a server ([Horn 93]): The ORB transfers message representations across a network to a specific server process. A low-level thread existing for each client connection is responsible for receiving the requests. When a request has arrived, the thread enqueues it in the main queue of the server process. Another active thread dequeues the request and performs a call on the server object implementation.

### 4.3.7 Discussion of the Message Queue Concept

Like the Active Object Pattern, the Message Queue Concept is based on concurrent objects. It is not designed to be applied to systems with a sequential flow of control. However, it is also possible to implement message queueing in a similar way for objects in non-concurrent systems. As objects and components in event-centered architectures such as reactive and realtime systems are mostly concurrent and even distributed, a message queueing mechanism for procedural flow of control is not considered.

### Full Transparency

Regarding transparency, the Message Queue Concept reveals three essential achievements which are a very important contribution to reusability and portability of code:
only for single objects but for logical groups of objects. In the case of existence of sub-threads followed by other implementation techniques such as do-activities of statechart diagrams (see section 3.5) or asynchronous operation invocations within an existing Object Thread, there must be protection against multi-thread impact.

Architectural Considerations

The Message Queue Concepts is a powerful and useful technique to establish transparent communication between vertical subsystems (such as the Presentation-Abstraction-Control Architecture Pattern [Buschmann 96]) and horizontal architecture layers (like 2-Tier, 3-Tier or Multi-Tier architectures) if the requirements demand concurrency. In general, the Message Queue Concept can serve as "communication interfaces" between concurrent software elements.

Communication Semantics

The use of proxies for an Object Thread makes synchronous and asynchronous communication explicit. A sender blocks if he sends a synchronous message. Further, a proxy may register with a timer if synchronous communication with timeout semantics is necessary. Furthermore, a transparent interaction logging mechanism for debugging purposes can be applied to the proxies easily.

State Semantics of Objects

The UML v1.1 offers deferred event handling for statechart diagrams [UML 97d]. This means to hold a particular event for later management if it cannot be served in the current state. The Message Queue Concept is flexible enough to be extended for event queueing requirements (see 5). The Message Queue Concept, including event queueing, is applicable to any implementation pattern for statechart diagrams (see chapter 2 and 3).
Inter-Process Communication

Applying the Message Queue Concept requires objects located in the same address space. The message queue strategy does not yet include communication between processes. The problem of referencing objects across different address spaces mentioned in section 4.1.1 remains. The message queue strategy does not include communication between processes. Chapter 6 traces ideas for this problem.

General Considerations

The Message Queue Concept goes far beyond the main issue of this dissertation which is exploring code generation techniques for statechart diagrams. When generating code for message queues, information of statechart diagrams is not sufficient. The information of statechart diagrams and class diagrams influence each other; they collapse into one central concern. When implementing message queues, data of the class model is needed and even adapted. There are also additional classes, associations and operations. The generated code results from the metadata of both class diagrams and statechart diagrams.
5 Queueing of Events

In this chapter we address queueing of events (not messages) as described in the UML v1.1 metamodel ([UML 97d]). We will develop a suitable approach for an event queueing mechanism based on the results of section 4.3.

An object can accept certain events in certain states. If an event, as a result of a received message, cannot be accepted in the current state the object is in, we must think about what to do with it. The simplest answer is to ignore the event, if we are not interested in handling it later. However, if it is necessary to be able to remember events that occur while the object is in the "wrong" state, we must queue them. After having received a message, the instance of the corresponding event is remembered until the object reaches a state in which it can respond to it. When the object has arrived in the correct state, queued events have higher priority compared to new incoming messages.

At the analysis level, it is up to the software engineer to decide in which state a particular event is ignored or queued. When defining dynamic object behaviour by modelling statechart diagrams for a class, the software engineer does not offer a transition for an event if he wants it to be ignored in a particular state. Unfortunately, the UML v1.1 does not provide explicit notation to model queued events (see [UML 97c]). However, the decision on queueing or ignoring events depends on the results of the requirements analysis and the OOA model.

In this chapter, we will develop a suitable implementation pattern for the queueing of events, based on the findings of section 4.3. The lack of notation elements for event queueing is not considered.
Consider the following example:

*Example:* Polling Sensor

When active, a polling sensor is required to read a value each second. The sensor receives a message from a timer in the cycle of milliseconds and counts them (see figure 42). After one second the sensor changes its state to read a value and checks a limiting value. If the limiting value is exceeded, the sensor takes the state for performing error handling (internal event). If exceeded or not, the sensor will change its state to notify all its observers, which have registered with the sensor as being interested in the measurement value.

Finally, the sensor returns to the state in which it counts the timer cycles.

When the sensor is receiving time cycles (state `receiveTimeCycle`) or it is reading a value (state `readValue`), it may be interrupted by the message `deactivate`. The current measurement is discarded then.

![figure 42: example of the lifecycle of a polling sensor](image-url)
Chapter 5

Queueing of Events

The deactivation request must not interrupt the notification of the observers and the error handling, so these activities form an entry block (Modelling Constraint 1 for statechart diagrams).

Also, the sensor can be calibrated. In this example the calibration is linear, so that the sensor gets two member attributes m_offset and m_amplification. After the calibration the sensor must be explicitly reactivated which means the beginning of a new measurement period.

The sensor can also be asked to send the complete data to a database interface for statistics and quality inspection purposes of the tested instrument. The request to save the data can arrive at the sensor at any time. If this happens during a measurement or while updating the error log, the request cannot be served. In this situation, it is not reasonable to completely ignore the event because we definitely want the data to be saved. It is essential here to hold the event until the sensor can react to it.

Also, the deactivation event is to be queued in those states that do not belong to the state region active. The sender does not need to worry whether the deactivation request was valid for the current state or not. The deactivation will definitely happen.

The Message Queue Concept, originally intended to satisfy inter-object communication requirements and thread synchronization constraints for sensitive data and code fragments, proves to be capable of capturing this problem very well. We can extend the Message Queue Concept in the way that the receiver is able to decide which event is to be queued and to hold them for later use.

Consider the Message Queue Concept applied to the polling sensor in figure 43. The areas marked grey represent modelling elements that are introduced in section 4.3.2. The class SensorThread gets another association to class Queue in the role of eventQueue. This queue instance stores events designated to be queued in certain states. Also, class SensorThread has a message queue to receive synchronous and asynchronous requests from other objects.
We now look for a mechanism that indicates whether an event is ignored or has to be queued. The first idea is to let operations return a certain value that must be inspected by the Object Thread. At the second look, this is not possible because this idea clashes with synchronous operations that already have return values. Moreover, the establishment of return information for asynchronous operations seems to be somewhat artificial. Thus, we consequently do not change the formal definition of any operation, neither for asynchronous nor for synchronous ones. In order to have a distinction between queueing and ignoring, we will add exception raising instead.

Each event method (as presented in chapter 3) throws an exception depending on whether the object can react to that event or not. This is either the exception `EventToBeQueued` or `EventIgnored`. The Object Thread catches these exceptions after having taken an event out of the message queue. If to be queued, the event instance is inserted into the event queue. If ignored, the event instance is deleted.
Of course, the ObjectThread code (`execute()`-method) must be adapted to look for a pending event in the event queue before considering the message queue.

If not empty, the Object Thread iterates over the complete event queue and invokes the operation `operate()` of each event. If there is no exception, the event was valid for the current state and it is removed from the list and deleted. Afterwards, the iteration continues.

![Event Queue Diagram](image)

*figure 44: scenario of the event queueing mechanism*

If any of the exceptions `EventToBeQueued` and `EventIgnored` is raised, the event is not removed and the next one is tested. When the end of the event queue is reached, the Object Thread begins inspecting the message queue. As usual, the thread is suspended if the message queue is empty.

This scenario is illustrated by the sequence diagram in figure 44 based on the example of the polling sensor in figure 42.
Implications of this Approach

A receiver throws an exception in order to notify its Object Thread what has happened with a received message. The raising of exceptions is part of the event methods of the receiver. If there is no concurrency and thus no Object Thread, the exceptions are propagated to the sender. In this case, the exceptions should not be left unhandled. Therefore, the code of the client must include exception-catching. This is the price we have to pay for the benefit of transparency.

Normally, the catch block of the client code remains empty. It can be filled manually after code generation to perform some action where needed. If the Message Queue Concept is used, the exception handling of the client is never concerned.

5.1 Discussion of the Event Queueing Concept

The Event Queueing Concept presented in this chapter extends the Message Queue Concept introduced in chapter 4.3. In the presence of concurrency, this combination of implementation patterns covers inter-object communication in general and synchronized state behaviour in particular.

The event queueing concept provides a transparent means of notification through the use of exceptions. The object informs about events that are ignored or queued by raising a corresponding exception. These exceptions are caught by the Object Thread belonging to that object. Based on this information, the object thread queues or deletes the particular event instance.

Using exceptions in C++ can be very expensive in terms of code size and runtime performance. Scott Meyers [Meyers 96] shows the difference between exception handling and invoking virtual functions and explains the cost of exceptions within applications. Meyers advises not to use exceptions for indicating frequent occurrences. Exceptions should primarily be used to make
software robust against non-expected happenings because of resulting performance penalties. However, the approach presented in this chapter puts emphasis on transparency and a minimum of adaption of domain code. Therefore, these disadvantages are intentionally not considered.
6 Code Generation Concepts for Multiprocessing

In order to complete the ideas discussed, it is necessary to look at how the Message Queue Concept can be used for objects that belong to different processes. As the development of detailed inter-process communication (IPC) patterns is outside the scope of this thesis, we look only on some issues briefly in this chapter to create a basis for further research.

Most multitasking operating systems offer services for inter-process communication like signaling, pipes, sockets, middleware products or other types of information channels. Whatever service is used, it can be encapsulated in the proxy classes so that the client code remains unconcerned. Like thread synchronization, the proxies hide the inter-process handling. The execute() -method of the receiving Object Thread\(^\text{13}\) still asks its message queue for input. It should not work directly with IPC mechanisms for the sake of transparency.

How can message objects pass process boundaries?

Many problems of transparency in object oriented software can be solved by establishing an additional level of indirection. This is the idea of the 'Forwarder-Receiver' pattern of Buschmann et. al. ([Buschmann 96]). The Forwarder-Receiver pattern deals with transparent inter-process and network communication. An object in the role of a Forwarder submits information by using a particular IPC mechanism. The Receiver takes this information and decodes it before handing it over to the Object Thread of the server\(^\text{14}\). Information always flows from Forwarder to Receiver.

Each Object Thread that must be able to receive messages from another address space gets a Receiver object. The Receiver object calls the corresponding operation of the proxy of the local
server Object Thread (see figure 45). Of course, the client needs to have a Forwarder as a counterpart. A Forwarder of the client has the same interface as the server Object Thread and acts as a remote proxy.

In the case of synchronous communication, the receiver of the client blocks after having called a synchronous operation of the sender's remote proxy, which forwards it to the server. The remote proxy expects the result placed in its return value slot as usual. The result is placed there by a local return-Value-Receiver which gets it from the return-value-Forwarder of the server. This Forwarder of the server is responsible for returning results to another address space (see figure 46). It is not to be mistaken with a Forwarder with which the server itself (acting as a client) references a second server of another process.

*figure 45: combination of Message Queue Concept and Forwarder-Receiver Pattern, example for asynchronous communication*

---

13 Each process consists of at least one thread.
14 A "client" represents the sender of a message, whereas a "server" represents the receiver of that message.
When using the Forwarder-Receiver pattern, it is necessary that the client which requests a certain service of a server knows where the server is located. This means, the Forwarder of the client must have information about the process the server is in, respectively the physical node of a distributed system on which the server resides.

In order to alleviate this problem, the Broker pattern of Buschmann et al. ([Buschmann 96]) can be applied. A Broker also introduces an additional level of indirection.

Each node possesses a Broker process that knows what servers exist and where to find them. Servers can register and deregister with their local Broker dynamically. A client asks the local Broker for a certain server. If the server is not locally available, the Broker delegates the request to the correct Broker. The client-server communication always flows through the Broker processes.

When using the Broker pattern, the code of a Forwarder is not concerned with looking up a server.

*figure 46: combination of Message Queue Concept and Forwarder-Receiver Pattern, example for synchronous communication*
How can we submit message instances to another process?

Messages are implemented as objects. One idea is to serialize these objects and then submit them to the target process via an IPC mechanism. Parameters for the corresponding operation are not a problem for the serialization because they become instance attributes of the message object anyhow (see section 4.3.3).

If parameters were implemented as aggregate objects, the message instance would have to have an operation that transforms them into values before serialization.

What about serving object references as parameters across different address spaces?

Consider the example outlined in figure 47. It shows two processes. Process 1 consists of three objects, two of them being an Object Thread. In process 2 there is only one object in one thread.

Consider that ObjectA calls an operation of ObjectC in Process2 and hands in a reference to ObjectB. ObjectB is not a thread and thus has no proxy; ObjectA knows ObjectB directly. If ObjectC addresses ObjectB in order to invoke an operation, ObjectB would have to become a thread temporarily having a proxy and a Receiver (following the Forwarder-Receiver pattern). If ObjectC calls a synchronous operation of ObjectB, ObjectB would have to have a Forwarder.
additionally to be able to return a result value. It is not obvious, how the evolution of *ObjectB* from a simple object to an Object Thread can be managed.

If we consider this problem as being solved for a moment and *ObjectB* becomes a thread for a limited time, there still is another problem:

*ObjectA* does not expect *ObjectB* to be a thread. *ObjectA* would have to change its direct reference to *ObjectB* into a reference to the proxy of *ObjectB* temporarily in order to avoid collisions with *ObjectC*.

This sort of requirement is too complex. It cannot reasonably be fulfilled dynamically.

Therefore, we propose a design rule that makes the considerations outlined above unnecessary. It refers to the architectural principle "Single Point of Reference" (see [Fowler 97b]). This is a common concept for building software architecture layers and decoupling software elements:

```
Modelling Constraint 8: Facades for Architectural Packages of Objects
```

"Each group of objects designated as a thread or process gets a facade object\(^{15}\) as an entry point. A single object designated as a thread or process typifies a facade itself. Any object belonging to a thread or process never communicates directly with an object within a different thread or process. Inter-thread and inter-process communication of objects is only managed by exchanging messages from facade to facade. The facade mediates the requests to the correct objects."

---

\(^{15}\) See design pattern 'Facade' of Gamma et al. ([Gamma 96]). See also the 'Whole-Part' pattern of Buschmann et al. ([Buschmann 96]).


Chapter 7

Outlook and Further Research

This thesis deals with implementation techniques for statechart diagrams of the UML v1.1 based on the C++ programming language. These concepts are expanded by ideas for information interchange between concurrent objects, subsystems etc. Also, these expansions are implementation-oriented.

In section 7.1, I want to show, in what way code generation concepts may be further developed and improved.

Section 7.2 addresses logical concepts. It traces ideas for the improvement of object-oriented methods. This concerns issues of object-oriented analysis (OOA) and object-oriented design (OOD) like identification of state semantics or the modelling of internal behaviour of objects and their interaction.

7.1 Further Code Generation Concepts

7.1.1 Inter-process Communication Between Objects

In chapter 4 concepts for inter-thread communication are introduced. Chapter 6 deals briefly with the same topic concerning inter-process communication. The foundations presented there are to be further examined; a detailed concept for applying the Message Queue Concept to inter-process communication and distributed environments should be worked out.

7.1.2 OO Programming Languages and Operating Systems

In this thesis, all sample code and implementation techniques are based on C++. The next step is to extend the ideas to other OO programming languages that all have different syntax and semantics (e.g., Smalltalk).
The concepts of chapter 4 need to use platform-dependent API operations for thread-, process- and semaphore handling. This concerns the chapters 4 and 6 as well as section 3.4. A code generator should offer a choice of programming languages and operating systems. In such a situation, a code generator can directly generate the correct API calls without manual adaptations by the software engineer.

### 7.1.3 Round-Trip-Engineering

Round-Trip-Engineering means the integration of Reverse-Engineering and code generation. Source code and models are always kept consistent. Source code can be generated from models, models are adapted if generated code is modified manually. The code generation concepts worked out in this thesis propose implementation techniques that are very complex. Since a code generator serves as an add-on component for a CASE-tool and is based on a specific repository structure, it is meaningful to develop reverse-engineering concepts to enable full round-trip-engineering.

### 7.2 State Dependency Analysis and Messaging Concepts Within OO Methods

From the logical point of view, a remaining problem concerning OO methods and software development process models is that there is no common agreement on how to find, and document, state dependencies between objects in the OOA, nor, indeed, how to derive statechart diagrams from the results of the OOA. From the model view there is little information about how statechart diagrams can be mapped to other diagrams of the UML while remaining consistent.

Frequently asked questions include "Must each object have a statechart diagram?" or "Where is the necessity for modelling statechart diagrams - why not use several boolean flags in ordinary functions?".

128
One of the most developed OO methods is OOSE\textsuperscript{16} (Object Oriented Software Engineering) of Ivar Jacobson, which is supported by the CASE-tool "Objectory". Jacobson provides an integrated, implementation-oriented method from requirements analysis to implementation with a well-formed distinction of OOA and OOD (see [Jacobson 92]). Nevertheless, this method also lacks the mentioned aspects.

Filling in these gaps demands a review of definitions for relevant terms like message, operation, event and state in order to increase the understanding of it. In literature, these terms are often jumbled. Reviewing these definitions is the basis for potential improvement of OO methods with respect to detecting state dependencies in the OOA and state modelling in the OOD.

7.2.1 States

First, we examine the accepted definition of the term state:

"A state is a class of values of an object's attributes. A state change is determined by the manipulation of attributes as a result of an incoming event. Incoming events may cause a state change. When reaching a state, an object can perform a certain activity. The activity of the target state is considered as the object's reaction to that event."

The existence of a state is dependent on the attributes and their possible values (which are determined by their types). States define equivalence classes of all possible attribute values. Following this definition has the following implication:

Whenever we want to introduce a new state to the object iteratively, we must formally prove that the coherence of the equivalence classes (the states) is not violated. We must reflect on the impact a new state (a new equivalence class of attribute values) will have on the existing equivalence

\textsuperscript{16} Rumbaugh, Jacobson and Booch are about to publish a new book called "The Unified Process" based on
classes. We have to reorganize the sets of attribute values so that the new state determines a subset of the other classes of attribute values.

When adding or removing an attribute, the problem is the same, because the equivalence classes of attribute values change.

This definition does not help to find relevant states for an object. There is no relation to the interaction with other objects. Actually, a software developer does not look at attribute values or the existence of associated objects. During object-oriented software development, he or she designs interaction and deals with messages, events and operations of objects.

A good OO method should give pieces of advice on how to derive states and transition from the OOA and OOD. When developing an object-oriented system, we set up a class diagram to cover the abstraction results which are objects and their relationships. Based on the requirements analysis we then define interaction and scenarios to instantiate each use case. This process is iterative, that means that we adapt and extend the class diagram. At the same time we develop the interaction and scenarios further and add messages that will result in operations for the classes.

The first and only results of the OOA are use cases, classes and scenarios.

For seamlessly proceeding to design and completing analysis, we must be able to map these results to descriptions of dynamic behaviour. That means, we must be able to derive states and operations for classes from inter-object communication. We have to have a close look at the interchanged messages to lay down what messages are state dependent and what messages result in state-independent operations.

In order to achieve this, the proposals of section 7.2.2 should be helpful.
7.2.2 Types of Messages and Events

Types of messages should be classified at an abstract level. This is a necessary step before proposing a method on how to realize state dependencies between objects and to model messages and lifecycles of objects. A software engineer’s task is to inspect the object communication and the operations found in the OOA and to hold out for state dependencies and message types before advancing to design.

Initially, messages can be classified into synchronous and asynchronous messages. In [SM 92] Shlaer/Mellor introduce only these types of messages. They suggest modelling all asynchronous messages in statechart diagrams. A statechart diagram is modelled for a single class.

Concerning synchronous messages, Shlaer/Mellor define operations for them in the class declaration as they interpret synchronous communication only as the setting and reading of attributes. Synchronous communication is supposed to be completely state-independent. Only asynchronous messages are relevant for the internal state of an object.

[MW 96] emphasize the importance of levels of abstraction and extend the approach of Shlaer/Mellor by synchronous operation calls. Still, this is not sufficient. A message can be dependent on an object’s state whether it is asynchronous/synchronous or not. The classification of synchronous and asynchronous seems to be orthogonal to state-dependencies of messages.

The UML v1.1 introduces the notion of “internal transitions”; i.e. a transition followed by the occurrence of a certain event in a certain state. It does not change the object’s state, the target state is identical to the source state. In contrast to ordinary transitions, internal transitions do not cause the state activity of the target state to be performed.

In figure 48, eventX causes an internal transition to fire. Neither the exit-activity nor the entry- and do-activities of state1 are executed. The only reaction of the object is defined by the activity following the slash of eventX. The transition followed by the occurrence of eventY behaves as usual. The exit, entry- and do-activities of state1 are performed in that order.
An internal transition defines a specific reaction of a certain event that is distinct from the regular state activity. An event causing internal transitions in different states is not restricted to always carry out the same activity. The reaction to this event may differ depending on the state. If we want to guarantee that an event has the same reaction but is only accepted in certain states we have to define a constraint at design level. There is no other possibility for semantic classification.

Therefore, the terms event, message and transition must be clearly defined and types of messages should be more finely distinguished in order to make the identification of messages in the analysis possible.

7.2.3 Priorities of Messages and Events

This section does not deal with code generation concepts or patterns. It outlines basic architectural concepts that are useful for concurrent and reactive systems. The following ideas presented have impact on the UML as standard notation for object-oriented software development.

**Priorites for Queued Events (Event Response Priorities)**

Events that cannot be responded to can be queued and remembered. A queued event is an event, which has happened as a result of a received message that cannot be accepted in the current state. After having received it, the instance of the message is gone, but the instance of the corresponding event is held until the object reaches a state in which it can respond to it. Chapter 5 refers to this by presenting event queues.
Chapter 7

Outlook and Further Research

Corresponding to [UML 97c] and [UML 97d] queued events will be re-dispatched in sequential order. The implementation concept introduced in chapter 5 conforms to that.

Additionally, it should be possible to explicitly model priorities for queued events. The software engineer should be able to determine what, and in what order, queued events shall be reactivated. This is of interest as soon as there is more than one pending event in the queue that is valid for the state the object reaches next.

This feature is not offered by the UML, but this is meaningful for designing lifecycles of objects within OO software development.

*Example:*

The polling sensor in figure 42 in chapter 5 accepts amongst others a request for sending the recent data to the database and a recalibration request.

Consider the sensor taking the state `errorHandling`. A recalibration request occurs and afterwards a request for saving the data arrives.

Since both events are to be queued in the current state, the object would first respond to the recalibration request. The data request keeps pending. A certain time may pass until the user submits the amplification and offset values in which the system may potentially crash. In this situation, the data request should be served prior to the recalibration request, independent of the order of receipt.

**Priorities for Dispatched Messages (Message Dispatch Priorities)**

Regarding the communication between concurrent objects or logical groups of objects, it makes sense to have priorities for messages sent between them. An object can send as many messages to other objects as it wants, but in what order they are dispatched in relation to messages sent by other objects should depend on dispatch priorities.
The actions of sending and dispatching messages is separated. A single object is responsible for sending, but not for dispatching a message. The sending object does not know about the dispatch priority, it is completely transparent.

The messages can be collected by a central message dispatcher, which queues and broadcasts them. Furthermore, the dispatcher is responsible for deferring a message if a delay time is attached to it.

This concept can be an alternative to a process priority strategy (including complex hard- and software interrupt mechanisms) of matching concurrent objects and logical groups of objects to prioritized processes that preempt each other:

By applying priorities to messages, the processes and threads can dynamically set their priority relative to that of a received message (the central dispatcher must always have the highest priority) and are awakened when the dispatcher delivers a message.

Further, the messages are inserted into the message queues (not the event queues\(^\text{17}\)) due to their dispatch priorities. Consequently, it is guaranteed that a certain object processes them in the same order that the central message dispatcher submits them.

Note that message dispatch priorities and event response priorities do not contradict each other: An object itself does not know anything about the dispatch priority of the message it has received. It only decides how to handle the occurred event internally.

**Examples:**

- A tactical aircraft has a missile defence system. In the case of an approaching missile, the software system should dispatch and compute the message of the missile warning system to the defensive system before updating the fuel display and calculating the air pressure.
• A feedback control loop of a machine always displays operational data on the GUI. In case of an emergency stop (e.g., the temperature of the motor reaches critical state) the interruption of the process by signaling specific actuators must occur before the writing of a log file and the data storage.

Shlaer/Mellor ([SM 92]) introduce a concept for a central message dispatcher, but not for priorities and time delay.

As mentioned above, in the UML v1.1 [UML 97d] queueing of ignored events is possible but the authors do not assume an event response priority concept.
Appendix A.1 Implementation of Asynchronous Messaging

This section shows a sample implementation for asynchronous message interchange by the use of the Message Queue Concept introduced in section 4.3.1. All examples and class names refer to those introduced in section 4.3.1. All code samples are shown in C++.

Implementation of the Classes AbstractPerson, ProxyPerson and PersonThread

The class AbstractPerson is an abstract class that defines the interface of class Person. The implementation is shown below.

```
//-----------------------------------------------
// class ABSTRACT PERSON
//-----------------------------------------------

class AbstractPerson // abstract class
{
    public: // abstract class
        AbstractPerson (void); // constructor

        virtual void goToSleep (void) = 0;
        virtual void goToWork (void) = 0;
    };
    // class
```

*figure 49: extension of the class model in figure 30*
In figure 30 the classes ProxyPerson and PersonThread both have an association to class Queue which is redundant. For this reason, we create a new class ElementWithQueue as a generalization of ProxyPerson and PersonThread (see figure 49).

```cpp
class ElementWithQueue {
    public:
        typedef Queue<EventPerson*> PersonMessageQueue;

    protected:
        PersonMessageQueue* const m_messageQueue;

    public:
        ElementWithQueue (const MessageQueue* queue) : m_messageQueue (queue);
}
```

The class PersonThread inherits from class Thread. Most integrated development environments (IDE) offer source code components that encapsulate thread handling operations, synchronization facilities and termination handling. The class Thread offers an abstract operation (which we call `execute()` here) that represents the code to be executed by the thread. It is to be defined by PersonThread.

A thread may be terminated by by calling `terminate()`, which sets the boolean attribute `m_terminated` to true. This attribute indicates whether `terminate()` has been called or not. If terminated, the thread will call its termination handler routine (which also has to be defined by class ProxyPerson). After termination, the memory of the object representing the thread must be deallocated.

```cpp
PersonThread::execute (void) {
    PersonEvent* event;

    while (! m_terminated) {
        event = m_messageQueue->obtainMessage();
        event->operate (this);
        delete event;
    }
}
```
This is what the proxy does when it is deleted. We designate the proxy instance as responsible for termination and deletion of its Object Thread. For this reason, there is an association between class `ProxyPerson` and `PersonThread` (see figure 49).

An instance of class `ProxyPerson` creates message instances and inserts them into its queue inherited from `ElementWithQueue`. The class `ProxyPerson` is implemented as follows:

```cpp
//------------------------------------------------------------------------------
// class PROXY PERSON
//------------------------------------------------------------------------------

class ProxyPerson : public AbstractPerson,
                      public ElementWithQueue
{

    protected:
    const PersonThread* const m_person;

    // constructor
    public:
    ProxyPerson (const PersonThread* thePerson,
                 const ElementWithQueue::PersonMessageQueue* theQueue
                 ) : ElementWithQueue (theQueue),
                 m_person (thePerson);

    virtual ~ProxyPerson (void) // destructor
    {
        m_person->terminate();
        delete m_person;

        // iterate the message queue and delete all items
        delete m_messageQueue;
    }

    virtual void goToWork (void)
    {
        // place a message instance into the queue
        m_messageQueue->insert (new PersonGoToWork);
    }

    virtual void goToSleep (void)
    {
        // place a message instance into the queue
        m_messageQueue->insert (new PersonGoToSleep);
    }

}; // class
Remark:

In many integrated development environments the inheritance of thread properties demands single inheritance. The only superclass must be the thread class. In this case, we use delegation instead of multiple inheritance. Class PersonThread gets an attribute for referencing a message queue directly and aggregates an object of class Person.

Implementation of the Event Class Hierarchy

The event class hierarchy is easy to implement. The operate()-methods expect an object that supports the interface of AbstractPerson to call a specific operation. Consider the following implementation:

```cpp
//------------------------------
// class EVENT PERSON
//------------------------------

class EventPerson // abstract class
{
    public:
    EventPerson (void);

    virtual void operate (AbstractPerson* thePerson) = 0;
}; // class

//------------------------------
// class PERSON GO TO SLEEP
//------------------------------

class PersonGoToSleep : public EventPerson
{
    public:
    PersonGoToSleep (void); // constructor

    virtual void operate (AbstractPerson* thePerson)
    {
        thePerson->goToSleep();
    }
}; // class
```
The implementation of class PersonGoToWork is not shown here. It refers to the implementation of class PersonGoToSleep.

Implementation of the Template Class Queue

Finally, we will examine the implementation of the generic class Queue that serves as the type of the message queue instances. It is implemented as follows:

```cpp
#include <FIFOList.h>

template <typename T>
class Queue
{
public:
    typedef FIFOList<T> FIFOList;

protected:
    static int HANDLE s_SEMA_INITIAL_VALUE; // class attribute
    HANDLE m_semaphore;
    FIFOList* m_Queue;

public:
    Queue(void)
    {
        m_Queue = new FIFOList;
        m_semaphore = createSemaphore (SEMA_INITIAL_VALUE);
    }

    virtual Queue* insert (T newElem)
    {
        m_Queue->insert (newElem);
        m_semaphore.release();
        return (this);
    }

    virtual T obtain (void)
    {
        m_semaphore.acquire();
        return (m_Queue->removeFirst());
    }
}; // class Queue
```
The idea behind this implementation follows the known producer/consumer synchronization problem:

The template class `Queue` serves as an adapter for a container class that guarantees FIFO order. It has a semaphore. The semaphore value indicates the amount of elements that are stored in `m_queue`. After being created, the instance of class `Queue` has no entries. Thus, the initial value of the semaphore is 0.

If a thread wants to remove an element out of the queue, it decrements the semaphore value. If the queue remains non-empty afterwards (the semaphore value remains greater than zero), the thread receives an element. If the queue remains empty (the semaphore value has become less than or equal to zero), the thread is blocked by the operating system.

If a thread enqueues an element, it increases the semaphore value afterwards. If there is another thread waiting, it is resumed receiving an element.

Of course, there is a synchronization constraint related to the container class itself as a shared resource: The container `FIFOList` must be multi-thread class itself. If a thread puts a message into the queue, then it must be ensured that a second thread does not dequeue or enqueue another message at the same time. This may happen if there is at least one element in the list. The internal structure of the container must always be consistent.
Appendix A.2 Implementation of the Transparent Creation of Object Threads

A question not answered yet is how a client gets a reference to an instance of ProxyPerson instead of class Person. How are the client, the proxy and the Object Thread tied together?

There are three possible approaches:

Class Operation for the Creation of All Necessary Instances

The code generator generates a protected constructor for class PersonThread and provides a class operation createPerson(), which creates a new instance of PersonThread and a corresponding proxy as well as a message queue instance (see the sample code below). This message queue instance is registered with the proxy instance and the person instance. This happens by passing the queue as a constructor parameter. After that, it resumes the thread of person. Finally, it returns the reference to the proxy.

```
//-----------------------------------------------
//  class PERSON THREAD
//-----------------------------------------------

class PersonThread : public ElementWithQueue,
    public Person,
    public Thread
{

    protected: // constructor
    Person (void) (...);

    public:
    virtual static abstractPerson* createPerson (void)
    {
        MessageQueue* messageQueue = new MessageQueue;

        PersonThread* personThread = new PersonThread (messageQueue);
        ProxyPerson* proxy = new ProxyPerson (person, messageQueue);

        personThread->resume();

        return (proxy);
    }

```
Implications resulting from this approach:

- First, the client code is not completely independent of the Message Queue Concept as we have said in section 4.3.1. The programmer of the client code has to use `PersonThread::createPerson()` instead of creation by the `new` operator. Furthermore, the client's attribute holding the reference to the proxy must be of type `AbstractPerson` instead of `Person`. The latter is not a big problem because these dependencies are minimal. It is a simple and acceptable effort to change the attribute declaration and the person creation statements of the client code when switching from the Message Queue Concept to another.

- Second, we are not able to create stack allocated instances of `PersonThread` which is perfectly all right. This restriction goes along with the fact that we have designated instances of the class `Person` as being a thread. It is a design decision to apply thread semantics to objects saying that an object is intended to exist in parallel to others and thus must be independent of the scope of the method in which it is created. Generally, establishing threads that operate on stack allocated objects lacks sense.

**Type Definition for Class ProxyPerson**

The second implementation approach for instantiating a `PersonThread` including its proxy increases transparency and reduces the client code dependency mentioned above. Following the first approach, a proxy and a person thread are instantiated by calling the class operation `createPerson()` of class `PersonThread`, which returns a reference to a proxy. The attribute of the client code holding that reference is declared as `AbstractPerson`. 

protected:
virtual void execute (void) {...};       // see Appendix A.1

};      // class
We can do without a class operation that creates the necessary instances if the constructor of ProxyPerson is made responsible for it. As a proxy associates an instance of PersonThread anyway, it can take over this task. The client attribute must be declared as type of ProxyPerson.

The class ProxyPerson is implemented as follows:

```cpp
// class PROXY PERSON

class ProxyPerson : public AbstractPerson,
                    public ElementWithQueue
{
  protected:
    const PersonThread* const m_person;

  public:
    virtual ~ProxyPerson (void) {...}; // destructor (impl. see above)

    ProxyPerson (const MessageQueue* queue) // constructor
    {
      m_messageQueue = new MessageQueue;
      m_person = new PersonThread (m_messageQueue);
      Person->resume();
    }

    virtual void goToWork (void) {...};
    virtual void goToSleep (void) {...};
}; // class
```

The client attribute does not need to be of type ProxyPerson directly if we add a type definition with name Person and type ProxyPerson. This increases the independency of the client code. Whenever a client instantiates an attribute of type Person, it will actually get an instance of ProxyPerson with a person thread standing behind it.

If the type definition is left out, an ordinary non-thread instance of class Person is created. By this method, we can easily switch from a Multi-Tasking environment to a Single-Tasking environment respectively to another implementation strategy, respectively.
Appendix A

Sample Implementation

//-------------------------------------------------------------
//                     sample client code
//-------------------------------------------------------------

#include "Person.h"
#include "ProxyPerson.h"

// when switching to single task environment, make a comment out of
// the following type definition:

typedef ProxyPerson Person;

class AnyClient
{
   protected:
   Person* myPerson;

   public:
   AnyClient (void) // constructor
   {
      // ...
      myPerson = new Person; // a person thread with its proxy
      // are created !
      // ...
   }

   // ...
}; // class AnyClient

The advantages of this approach are obvious:

- We do not need to call a class operation whenever we want to instantiate a person. The C++
  creation statement (operator new) is sufficient. This kind of transparency maximizes reusability
  and decoupling of software modules.

- We can switch from Multi-Tasking environments to single process environments simply by
  leaving out the type definition statement in the client code.

- This approach also prevents a thread object being allocated on the stack. It is allowed to
  instantiate aProxyPerson (via typedef Person) as a non-pointer variable within the scope
  of a method since the destructor of ProxyPerson terminates the thread and deletes the
object standing behind it. However, the software engineer has to make his mind up if this is something he wants.

**Using a “Virtual Constructor”**

In the second implementation approach, switching from a person-thread object to an ordinary person object is accomplished by leaving out the type definition statement and recompiling the client code.

The implementation approach presented here serves as an enhancement for the second one. The programmer should be able to decide himself whether he wants to have an object instantiated as a thread at one point, and to have a non-concurrent object of the same class at another point. The decision should be made when instantiating the class.

We can achieve this by making class `AbstractPerson` a concrete class. It gets an additional member attribute of type pointer to `AbstractPerson`, which can hold a reference to an instance of one of the subclasses of `AbstractPerson` (see figure 50).

![Diagram of virtual constructor](image)

*figure 50: simulation of a virtual constructor*

This is either `ProxyPerson` or `Person`. The corresponding instance is created within the constructor of `AbstractPerson`. The information about what instance has to be created is
Appendix A

Sample Implementation

passed as constructor parameter. The interface operations of AbstractPerson are implemented by forwarding the operation call to the referenced object.

Following this approach, the client code actually creates an instance of AbstractPerson, which acts as an envelope for either a Person instance or a PersonThread instance. We also establish a type definition with name Person and type AbstractPerson here.

We have used the virtual constructor idiom of James Coplien (see [Coplien 92]) here, which is a specialization of the handle-body idiom. The definition of a virtual constructor says that it determines what class to instantiate (the class the constructor belongs to or one of its subclasses) depending on some condition. Virtual constructors are not supported in C++ directly.

The constructor of AbstractPerson acts as a virtual constructor whenever AbstractPerson is instantiated.

This implementation approach reveals the following advantages:

• In the programming domain, the programmer of the client code can decide whether he wants to instantiate a person object living in its own thread or not. Furthermore, documentation of the source code is improved. One can see immediately, which object is instantiated as a thread and which is not.

• By the use of a type definition, the attributes of the client code may still be of type Person instead of AbstractPerson.

• Flexibility is improved in terms of being dynamically able to instantiate an object as a thread or not.
There are the following disadvantages:

- This implementation is not as transparent as the second approach. The programmer of the client code must know about the constructor interface of class `AbstractPerson` because he needs to hand in a corresponding parameter value.

- The additional indirection by having an instance of `AbstractPerson` incurs a performance penalty.

The class `AbstractPerson` simulating a virtual constructor is implemented as follows:

```cpp
// An instance of AbstractPerson acts as a virtual constructor for // an instance of ProxyPerson.

public:
typedef Queue<PersonEvent*> MessageQueue;

protected:
AbstractPerson* const m_person;

public: // constructor
AbstractPerson (bool CREATE_AS_THREAD) // no default-constructor!
{
    if (CREATE_AS_THREAD)
    {
        MessageQueue* queue = new MessageQueue;

        PersonThread* personThread = new PersonThread (queue);
        m_person = new ProxyPerson (personThread, queue);
        person->resume();
    }
    else
    {
        m_person = new Person;
    }
}
```
virtual ~AbstractPerson(void) // destructor
{
    // if m_person is a proxy object, the thread object is
    // terminated and deleted within the destructor of the proxy
    delete m_person;
};

// forwarding the operation calls
virtual void goToWork(void) { m_person->goToWork(); }  
virtual void goToSleep(void) { m_person->goToSleep(); }
};  // class

As we do not need a class operation for creating PersonThread and PersonProxy instances
introduced in the first implementation approach any longer, the code for class PersonThread
looks like this:

//---class PERSON THREAD
//---

class PersonThread : public Person,
public ElementWithQueue,
public Thread
{

    public:  // constructor
    PersonThread (MessageQueue* theQueue)
        : ElementWithQueue (theQueue);

    protected:
    virtual void execute (void) {...};  // see Appendix A.1
};  // class

The implementation of ProxyPerson is identical to the one introduced in appendix A.1.
Nevertheless, the code is repeated here for better illustration.

//---class PROXY PERSON
//---

class ProxyPerson : public AbstractPerson,
public ElementWithQueue
{

    protected:
    const PersonThread* const m_person;

149
public:
~ProxyPerson (void); // destructor (impl. see above)

// constructor
ProxyPerson (ElementWithQueue::PersonMessageQueue* queue,
       const Person* thePerson)
   : ElementWithQueue (queue),
     m_person (thePerson)
{...};

virtual void goToWork (void) {...};
virtual void goToSleep (void) {...};

}; // class

//---------------
// sample client code
//---------------

#include "AbstractPerson.h"

typedef AbstractPerson Person;

class AnyClient
{
    protected:
        Person* m_myThreadPerson;
        Person* m_myNormalPerson;

public:
    AnyClient (void)
    {
        bool THREAD = true;
        bool NO_THREAD = false;

        // a person thread instance with a proxy instance is created
        m_myThreadPerson = new Person (THREAD);

        // an non-concurrent person instance is created !
        m_myNormalPerson = new Person (NO_THREAD);

        m_myNormalPerson->goToWork();
        m_myThreadPerson->goToWork();

        // ...
    }

}; // class AnyClient
Appendix A.3 Implementation of Synchronous Messaging

This section shows a sample implementation for synchronous message interchange by the use of the Message Queue Concept introduced in section 4.3.2. The ideas for synchronous messaging serve as an extension to the implementation of asynchronous messaging for the Message Queue Concept in section A.1. All examples and class names refer to those introduced in section 4.3.2. All code samples are shown in C++.

Instances of synchronous messages are created by the class ProxyPerson. The reference to the return value slot, in which the event instance places return information, is handed in as constructor parameter.

The template argument for the template class Queue is a pointer to void; it represents pointers. The reason is that a return value slot must be able to store references to objects as well as common data types like floats or integers etc.

Dynamic casting is used here to establish type conformance when ProxyPerson retrieves values out of the queue (see sample code for ProxyPerson below). In case of primitive data types, the values are dereferenced by ProxyPerson before being returned to the client.

The default value element indicating that a synchronous operation does not have a return value is implemented by a constant class attribute of the type pointer to integer. It serves as an acknowledgement for the completion of the operation. Its value is set to 0.

It is possible to transfer pointers because we consider the Message Queue Concept for threads belonging to the same process. Of course, this implementation concept will not work regarding threads in different address spaces (see chapter 6).
//---------------------------------------------------------------
// class EVENT PERSON SYNCHRONOUS
//---------------------------------------------------------------

class EventPersonSynchronous : public EventPerson
{
    protected:
    // Queue into which the return value delivered by the other
    // thread is placed.
    // There is only one return value in it at any time. The sending
    // thread blocks until the return value is fully determined.
    ProxyPerson::PersonMessageQueue* const m_returnValueSlot;

    // default value for synchronous void operations (class
    // attribute)
    static const int* s_DEFAULT_INFO;

    public:
    EventPersonSynchronous (ProxyPerson::PersonMessageQueue* theReturnValueSlot)
    {
        m_returnValueSlot = theReturnValueSlot;
    }
    // class

    // initialization of the const class attribute
    EventPersonSynchronous::s_DEFAULT_INFO = new int(0);

    //---------------------------------------------------------------
    // class EVENT PERSON GET AGE
    //---------------------------------------------------------------

class PersonGetAge : public EventPersonSynchronous
{
    public:
    PersonGetAge (ProxyPerson::PersonMessageQueue* theReturnValueSlot)
        : EventPersonSynchronous (theReturnValueSlot);

    virtual void operate (AbstractPerson* thePerson)
    {
        unsigned int result = thePerson->getAge();
        m_returnValueSlot->insert (new int(result));
    }
    // class
class PersonGetName : public EventPersonSynchronous
{
    public:
    PersonGetName (ProxyPerson::PersonMessageQueue* theReturnValueSlot)
        : EventPersonSynchronous (theReturnValueSlot);

    virtual void operate (AbstractPerson* thePerson)
    {
        const char* result = thePerson->getName();
        m_returnValueSlot->insert (result);
    }
};  // class

class PersonOpenDoor : public EventPersonSynchronous
{
    public:
    PersonOpenDoor (ProxyPerson::PersonMessageQueue* theReturnValueSlot)
        : EventPersonSynchronous (theReturnValueSlot);

    virtual void operate (AbstractPerson* thePerson)
    {
        thePerson->openDoor();
        m_returnValueSlot->insert (s_DEFAULT_INFO);
    }
};  // class

The class ProxyPerson defines the following methods:

class ProxyPerson : public AbstractPerson
{

    public:
    typedef Queue<void*> PersonMessageQueue;

    // ...

    PersonMessageQueue;
public:
virtual const char* getName (void) const
{
    // casting away the constness (explanation see below)
    ProxyPerson* myThis = const_cast<ProxyPerson*> (this);

    myThis->m_messageQueue->
        insert (new PersonGetName (m_returnValueSlot));

    return (dynamic_cast<const char*>(
                m_returnValueSlot->obtain()));
}

virtual unsigned int getAge (void) const
{
    // casting away the constness (explanation see below)
    ProxyPerson* myThis = const_cast<ProxyPerson*> (this);

    myThis->m_messageQueue->
        insert (new PersonAge (m_returnValueSlot));

    return *(dynamic_cast<unsigned int*>(
                m_returnValueSlot->obtain()));
}

virtual void openDoor (void) const
// synchronous operation without return value
{
    // casting away the constness (explanation see below)
    ProxyPerson* myThis = const_cast<ProxyPerson*> (this);

    myThis->m_messageQueue->
        insert (new PersonOpenDoor (m_returnValueSlot));

    m_returnValueSlot->obtain();
}; // class

Physical and logical constness

The methods ProxyPerson::getName(), ProxyPerson::getAge() and ProxyPerson::openDoor() are const methods because their semantics say that they just return a value and do not modify the object instance. However, the concrete implementation necessarily calls the insert method of the receiver queue, which indeed is a modifying method. Following C++ semantics, calling modifying operations or assignments is not allowed within const methods. Therefore, the synchronous operations of Proxy cannot be declared as const methods.
A solution for this problem is given by James Coplien [Coplien 92], who contrasts and discusses "logical constness" and "physical constness". Coplien proposes an idiom that uses casting in order to let an operation be const but makes assignments or calls of non-const methods possible, if reasonable.
Appendix A.4 Implementation of Event Queueing

This section shows detailed implementation concepts for the event queueing mechanism introduced in chapter 5. All examples and class names refer to those introduced in section 4.3.1. The event queueing mechanism can be added to the implementation of the Message Queue Concept, including both synchronous and asynchronous message interchange. See sections 4.3.1, 4.3.2, A.1 and A.3 for more details. All code samples are shown in C++.

The implementation of event queueing includes exception raising. For this reason, we create a convenient exception class hierarchy first (see figure 51).

```cpp
class MessageQueueException : stdException
{
    public:
    MessageQueueException (const char* info = "")
        : stdException (info);
};

class EventToBeQueued : public MessageQueueException
{
    public:
    EventToBeQueued (const char* info = "")
        : MessageQueueException (info);
};

class EventIgnored : public MessageQueueException
{
    public:
    MessageQueueException (const char* info = "")
        : MessageQueueException (info);
};
```

![figure 51: self-defined exception class hierarchy](image)
The implementation of the execute method of the Object Thread is expanded. Consider the following example of class Person in section 4.3.2:

```cpp
Person::execute (void) {
    PersonEvent* event;  // initialization with false, because after the creation of the
    QueueIterator iter (m_eventQueue);  // object no message can have been received before, thus there are
    // no events pending.
    bool checkQueuedEvents = false;

    while (! threadTerminated) {
        iter.setFirst();  // set to first item.
        while (checkQueuedEvents && iter.isValid()) {
            try {
                event = iter.item();
                event->operate (this);
                // if no exception occurs, the following
                // statements are performed
                m_eventQueue->remove (event);
                delete event;
                iter.setFirst();  // set to first item
            }
            catch (const MessageQueueException& e) {
                // Either the exception 'eventToBeQueued' or
                // 'eventIgnored' is raised after having called
                // 'operate()' of the event instance. In either
                // case, the event must stay in the event queue
                // then.
                iter.setNext();  // advance to next item
            }
        }  // while checkQueuedEvents

        // inspecting the message queue. Thread is blocked if empty.
        event = m_messageQueue->getFirst();
        m_messageQueue->remove (event);
        try {
            event->operate (this);
            // if no exception occurs, the following statements
            // are executed
```

157
delete event;
checkQueuedEvents = true;
}

catch (const EventToBeQueued& e)
{
    // this exception occurred after having called
    // 'operate()' of the event instance.
    // remember the event. It is already dequeued in the
    // message queue.
    m_eventQueue->insert (event);
    checkQueuedEvents = false;
}

catch (const eventIgnored& e)
{
    // this exception occurred after having called
    // 'operate()' of the event instance
    // object can be destroyed
    delete event;
    checkQueuedEvents = false;
}
} // while (true)
}; // execute

The event acceptance methods introduced in chapter 3 have to throw exceptions. We look at the
implementation by taking the event sendToDatabase() of the polling sensor in figure 42 as an
example:

void Sensor::sendToDatabase (void) throw (MessageQueueException)
{
    switch (m_state)
    {
    case RECEIVE_TIME_CYCLE:
        setState (SEND_TO_DATABASE);
        do_SendDataToDatabase ();
        break;

    default:
        throw EventToBeQueued;
        break;
    }
} // switch
Appendix A

Sample Implementation

void Sensor::do_SendDataToDatabase (void)
{
    // submit data
    // automatic transition
    setState (RECEIVE_TIME_CYCLE);
    do_receiveTimeCycle();
};

The event activate is only accepted within the states INACTIVE and CALIBRATION. It does not need to be considered in all other states:

void Sensor::activate (void) throw (MessageQueueException)
{
    switch (m_state)
    {
    case INACTIVE:
        setState (RECEIVE_TIME_CYCLE);
        m_count = 0;
        do_receiveTimeCycle();
        break;
    case CALIBRATION:
        setState (RECEIVE_TIME_CYCLE);
        m_count = 0;
        do_receiveTimeCycle();
        break;
    default:
        throw EventIgnored;
        break;
    }
    // switch
};
Appendices B.1 to B.6 inclusive give details on the design of the code generator prototype. They neither include a summary of code files, nor a detailed description of the complete internal structure of the classes in terms of attributes and non-public operations. The code includes comments and explanations of functionality implemented by the particular class.

The following sections present the work products of requirements analysis, object-oriented analysis and design (OOA, OOD) of the code generator software. Static and dynamic models together with verbal explanations make the reader familiar with the decisions made during the development of the prototype.

**Appendix B.1 Code Generator Prototype - Objectives and Technical Resources**

The prototype of the code generator software is implemented in C++ by the use of the Inprise C++ Builder 3 IDE. The software was developed using the OO method OOSE (Object Oriented Software Engineering) of Jacobson (see [Jacobson 92]). The work products of the use case analysis, OOA and OOD are modelled in UML using the CASE-tool ‘Innovator v6.0’ of MID company, Germany.

The code generator prototype is supposed to serve as an add-on facility for the CASE-tool ‘Select Enterprise’ by Select Software Tools. It supports the implementation strategy developed in chapter 3. The implementation approaches of chapter 2 are not concerned. Code generation is done for the programming language C++ only. Before code generation, the generator checks the consistency of the statechart diagram according to the modelling rules defined in section 3.7.

Select Enterprise offers an OLE interface (Object Linking and Embedding) for accessing its repository data. The repository structure of Select Enterprise is designed in an object-oriented way.
Appendix B

Technical Implementation

All elements stored in the repository are objects providing links to each other. This structure can be traversed using special commands submitted to Select Enterprise via OLE interface. Objects of the repository can be asked for information about themselves. Furthermore, objects can be added to the repository, manipulated and deleted. All information about the OLE commands as well as the object-oriented repository structure is published in [Select 97b].

Appendix B.2 Use Case Diagram

The use case diagram in figure 52 shows the identified use cases and actors. There is an actor *software engineer* initiating the only use case *generate code*. Any software engineer may communicate with the system via the dialog form shown in figure 53.

Description of the use case 'generate code':

The code generator offers a list of all existing projects to choose from. A project name represents a certain subset of the Select Enterprise repository that contains all model data of a particular software system. If a project is selected by a software engineer, the code generator immediately shows all classes within that project in the list box 'Classes in Projects'. The software engineer may select one or more classes from that list. This is done by transferring the particular entries to the list box 'Chosen Classes'. Finally, a service has to be selected by using the radio buttons of the group box 'Service'. A service represents a certain functionality the code generator offers for the selected classes. This is either a consistency check or code generation. When modelling statechart diagrams, the code generator may be used for consistency checking only. Code generation usually is of interest at a later design stage or even after completion of design. By pushing the 'Start' button, the code generator performs the service chosen before. When the current service has been completed,
the code generator informs the software engineer by opening a message box shown in figure 54. The software engineer may now choose another project, other classes or initiate another service.

![Figure 52: Use Case Diagram](image)

![Figure 53: GUI Design of the Use Case 'Generate Code'](image)
Appendix B.3 Architectural Design

The system architecture is designed as a 2-tier architecture. It consists of a GUI layer and a system core which covers the system logic.

The GUI layer includes only one class, that is the dialog form for the use case 'generate code' (see section B.2). The GUI layer is designed following the Document-View concept (see design pattern 'Model-View-Controller', [Buschmann 96]). There is no controller concept such as the Model-View-Controller concept (MVC) since the system knows only one actor and offers only one use case which has a very simple functionality. Compared to the increased complexity, applying the MVC concept is of no advantage for the code generator software.

The system core encompasses the complete logic necessary for consistency checking and code generation. The class SystemCore acts as a facade (see design pattern 'Facade', [Gamma 96]). To provide loose coupling, the class SystemCore implements the interface ClickButtonObserver. The use case dialog communicates with the system core via this interface. For detailed information about the semantics of the design pattern 'Observer' see [Gamma 96]. The dialog form informs the system core about the service (code generation or consistency check) the software engineer requests by submitting a corresponding parameter value. This conforms to the Push-Model of the Observer pattern. After being notified, the system core asks the dialog form for the chosen class names of the selected project to be serviced. The system core knows about the dialog form by an unidirectional association (see figure 55).
The class \textit{OLEConnection} establishes the connection to the CASE-tool Select Enterprise via Select OLE interface (Object Linking and Embedding). There is only one instance of that class in the system (see design pattern 'Singleton', [Gamma 96]).

The OLEConnection is used by the dialog form as well as by the system core. Both the dialog form and the system core ask for particular pieces of information they need for processing: The dialog form needs information about existing projects and the classes defined in these projects (see section B.2). For this reason, the OLEConnection offers the operations \texttt{getAllProjects()} and \texttt{getClassesOfProject()}. The dialog form immediately informs the OLEConnection about the project choice the software engineer made by calling the operation \texttt{setProject()}. The OLEConnection then activates the corresponding project repository.

The system core needs to reference the OLEConnection for demanding an object structure representing each selected class including its statechart diagram information (for detailed explanation of this object structure see section B.2.). The object structure is built by the OLEConnection based on the information obtained from Select repository. Beforehand, the OLEConnection must have been informed about the class names of the current project selected by the software engineer. This is done by iteratively invoking the operation \texttt{getClassStructure()} with handing in the particular class name as a parameter value (detailed information about the system dynamics can be found in section B.5).
Appendix B.4  Analysis Class Model

This section informs about the static analysis model of the system core. The corresponding class diagram is shown in figure 56.

Classes Class and OLEConnection

The class OLEConnection is responsible for extracting statechart diagram information out of the repository of the CASE-tool Select Enterprise. The extracted information is transformed into an object structure representing a self-defined metamodel of a statechart diagram. This structure includes a class with its events, states and transitions. The transitions additionally know their
condition, action list, message-sendings and parameters. The OLEConnection builds this structure at runtime on demand of the system core (see section B.3).

The system core possesses a list in which class objects are stored. These are the class objects delivered by the OLEConnection (see above). When the GUI notifies the system core, the system core obtains the class objects from the OLEConnection, fills the list and iterates over the list initiating code generation or consistency checks. Section B.5 shows how this is done exactly.

The class `Class` knows all its states and events that it can receive. The associations to the classes `Event` and `ComponentState` provide the necessary object connections.

**Class ComponentState**

A state instance may have `entry-`, `exit-` and `do-`-activities. An activity is represented as an instance of class `Activity`. A state instance may connect activity instances by the association defined between the two classes.

Generally, a state may be rectified by nested statechart diagram ([UML 97a], [UML 97b]). In the analysis model, this fact is not concerned. See section B.6 for design extensions of the OOA class diagram.

**Classes Transition, Condition, Action and Message**

A transition of a statechart diagram is represented by its own object. A transition instance consists of a set of action statements, a set of messages to be sent when the transition fires as well as a condition. These properties are optional. Actions, messages and the condition become an object as well defined by the classes `Action`, `Condition` and `Message`.

A transition may either represents an automatic transition or an event-driven transition. This depends on whether there is an event instance registered in the role 'externalEvent' or not.
A state instance knows about all transitions that are leading to it. This is necessary for consistency checking: A state can be asked to examine whether all incoming transitions provide the same formal parameters (see section 3.7). Furthermore, a state instance has a reference to each automatic transition that leaves this state on completion of the do-activity (see 3.1). Thus, class State must have a connection to the corresponding transition instances.

A transition instance itself knows its source and target state as it is responsible for processing the exit-activity of the source state and the entry- as well as the do-activity of the target state. The class model shows two "bidirectional" associations between the classes State and Transition. Note that an association with navigation arrows in both directions does not imply real bidirectional semantics. It is only a simplified graphical shortcut for two unidirectional associations. In this case, there are two unidirectional associations from Transition to State and two unidirectional associations from State to Transition.

Classes Event and Parameter

The Class Event has an association to class Transition. The associated transition instances represent all transitions leaving any state driven by a specific event. Note that a state may have more than one outgoing transition fired by the same event. These transitions must have disjoint conditions then (see section 3.7).

Additionally, a code generator needs to know for what states a particular event is valid. Therefore, class Event gets an additional association to class State in the role 'sourceStates'. The associated state objects represent the states in which the particular event can be responded to. This provides redundant information as the source state objects of an event are a subset of its firing transition objects. Anyhow, this intentional redundancy simplifies the code generation functionality.

An event, as well as a message that will result in an event for the receiver, may carry parameters. This is optional. Therefore, both classes need to have an association to class Parameter. Parameter instances represent formal parameters. Following class generalization policies, this association is
transferred to a superclass `ElementWithParameter`. Thus, the association only needs to be modelled once as it is inherited by the classes `Event` and `Message`.

**Factoring Process of the Object Instances**

In the previous paragraphs the connections between the different object types are described. Object instances must be tied together before code generation or consistency checking can take place. For this purpose, every class having an association to another implements public registration operations. As mentioned above, the object structure is created by the OLEConnection at runtime for each class the software engineer has selected (see section B.3). The OLEConnection traverses the CASE-tool repository and creates an object for each piece of information found. If necessary, it registers the created objects with other ones. This is the way the object structure encompassing classes and their statechart diagrams is constructed and submitted to the system core.

The object creation scheme used by the OLEConnection follows the principles of the design pattern 'Builder' (see [Gamma 96]). The Builder pattern is one of the Object Creation Patterns described by Gamma et al.
Each class possesses operations for associated objects. These are suppressed here.

class "Message" as well as "Event" can have parameters. The corresponding association is thus generalized.

Figure 56: Analysis Class Diagram
Appendix B.5  Dynamic Specification of the Use Cases

The use case defined in figure 52 is specified by two sequence diagrams presented in figure 57 and figure 58. The sequence diagrams define the interaction of the system components when initiating the use case.

The class diagram in figure 56 reveals that objects of type Class serve as the entry point for the complete object structure behind it. The system core iterates over its list of class instances in order to initiate code generation or consistency checking. A design approach could be as follows: Since every element of the structure is supposed to support these services, we may add corresponding operations to each. The system core would only need to invoke the code generation or consistency check operation while iterating. For example, when generating code, a class instance writes its code to an output stream and then call the code generation operation of its associated state and event instances sequentially passing over the stream object.

The code generator prototype generates C++ code following the implementation concept of chapter 3. Consider that the code generator software is extended for generating code for any other programming language such as Java or Smalltalk. Further, consider that the implementation strategies presented in chapter 2 are to be added. Moreover, additional modelling rules and consistency constraints may be developed that also must be supported by the code generator. Obviously, the integration of code generation and consistency check operations into the elements of the system core structure causes a frequent change of the analysis model concerning class interfaces and implementation. When extending the code generator software, a lot of recompiling and linking will be the result. Also, the interfaces of the particular classes will become larger over time.

It is difficult to argue that this kind of software design supports maintainability and extensibility, which generally are the primary goals of object-oriented software development. In order to avoid
these disadvantages, the system core structure must be decoupled from the logic of code generation and consistency checking. The system structure should only store basic information passively. The operations working on the structure should be completely independent. For achieving this, we apply the design pattern ‘Visitor’ to our model. The Visitor pattern avoids the mentioned problems (for detailed information, refer to [Gamma 96]). The sequence diagram in figure 57 shows how the Visitor pattern works. The consequences of applying the Visitor pattern for the OOA class diagram (additional classes, associations etc.) in figure 56 are explained in section B.6.

The basic idea of the Visitor pattern is to encapsulate operations for a certain purpose in one class. Such a visitor object is passed through an object structure. For each element of the structure the visitor offers its own operation. When a structure element receives the visitor object (the visitor object “visits” a structure element), it invokes its operation on the visitor and provides a pointer to itself. The visitor in turn references the structure element and operates on it. After that, the structure element submits the visitor to its successors. An implication of this pattern is that the structure elements must offer a sufficient interface for the visitor to work. A visitor possibly requires to have access to internal data of the structure elements.

Consider the scenario in figure 57. After being notified by the use case dialog, the system core creates a new visitor instance and hands it over to the class instances. The visitor object may be a C++ visitor, a Java visitor or a visitor for checking a particular consistency constraint etc.
Notifying the System Core

GenerateCode
UseCaseDialog

CodeGenerator
SystemCore

theRegistry
VisitorRegistry

VisitorList
LinkedList

aClassObject
Class

notify (serviceID)

getChosenClasses

getVisitors (serviceID)

gelElement

receiveVisitor (theDequeuedVisitor)

Figure 58: Visitor Registry Extension of the Scenario in figure 57

173
At present, the system core is made responsible for creating visitor instances. Therefore, it must know about all visitor types that are defined. Furthermore, the system core must be able to map the visitor types to the information about what service was chosen on the GUI. The creation of visitor objects depends on whether code generation or consistency checking is wanted. Considering possible extensions of the code generator as mentioned above, the class SystemCore would have to be reimplemented. The use of the Visitor pattern reduces the number of classes to be adapted to one. However, there still is a frequent change of at least one class of the core model.

This design approach can be further improved. The dependencies between the class SystemCore and visitor types can be eliminated by introducing a registry concept. All visitors that may be instantiated are stored in a visitor registry and are assigned to a key representing the services that can be selected on the GUI. The system core instance knows about the registry and requests a list of visitors connected to the chosen service delivered by the use case dialog. Thus, the class SystemCore is only dependent on the interface of the visitor classes but not on their types. We avoid the encoding of visitor creation and assignment to notification hints of the GUI within elements of the analysis model. The system core instance simply obtains a list of visitor instances and sends them into the structure one by one.

The initialization of the visitor registry is done by a factory (see design pattern 'Abstract Factory', [Gamma 96]). The factory creates all necessary visitor instances, attaches a service ID and puts them into the registry. In case of a system extension, the only action is to implement new visitor classes and to adapt the factory code. The code of the analysis model is totally left unconcerned. The changes of the dynamic behaviour is reflected by the sequence diagram in figure 58.
Appendix B.6 Design Class Model

In order to apply the design ideas of section B.5, the analysis class model has to be adapted. Furthermore, several adaptations are made for improving the design. This section explains all design decisions based on the analysis model.

Design of Visitor Receipt

A visitor traverses the object structure of the analysis class model introduced in figure 56. For traversing, each element of the structure must be able to accept a visitor and has to maintain a list of successors. Therefore, the classes of the object structure inherit from class VisitorReceiver. The Class VisitorReceiver defines the operations receiveVisitor(), submitVisitor() and registerSuccessor(). The latter operation expects an object of type VisitorReceiver and puts it into the successor list defined by an association from class VisitorReceiver to itself (see figure 59).

A successor has always a type of one of the analysis classes. Registration of successors happens at the same time the object structure is built: Each time the OLEConnection provides an object representing a piece of information of the statechart diagram (see section B.3), another object inserts it into its successor list.

A visitor instance is not generally submitted to the successors of a receiver. The visitor can decide itself whether it wants to be submitted to any successor. Therefore, the class VisitorReceiver offers the public operation submitVisitor(). If the visitor calls this operation, the VisitorReceiver iterates over its list of successors each time calling receiveVisitor().

For instance, a visitor visits instances of type Class in order to produce class frames. After that, the visitor calls the submitVisitor() operation of that class instance in order to create methods
for all connected states and events. The state and event instances are stored in the successor list of the class instance.

Depending on the particular implementation technique for statechart diagrams and programming language, a visitor must determine itself to what special kind of successors it wants to be passed next. It is not always meaningful to generally pass the visitor to all successors. For example, when visiting an event, a visitor must print the names of the formal parameters of an event method before it pulls information about the states the event may be received in. Therefore, all classes of the analysis model offer public operations for traversing each association to other objects. In particular, these are operations for setting an internal cursor to the first element, for advancing to the next element, for checking if the last element is reached and for obtaining the current element (see figure 60 and figure 61). When a visitor wants to submit itself while traversing a particular object connection, it invokes the operation `receiveVisitor()` on each element which is inherited from class `VisitorReceiver`.

In order to let a visitor operate on visited objects, all visited elements must offer a sufficient interface for the visitor to use. For this reason, all objects belonging to the object structure of the analysis model must provide necessary access to their attributes (e.g., names of states, events etc.) additionally. Objects of the analysis model are supposed to be passive in terms of carrying data and maintaining connections to other objects only.

**Design of the Visitor Registry**

As stated in section B.5, the visitor instances are stored in a registry for decoupling the visitor creation mechanism and the system core structure. The system core references the visitor registry which was filled by a factory during system initialization (figure 60). The registry consists of visitor instances associated by a shared aggregation (for explanation, see [UML 97a]). It is a qualified aggregation because the system core requests a list of visitor instances by an external key.
provided by the use case dialog (see section B.5). The key represents the service the software engineer has selected. The qualifier handed in by the system core must serve at least one visitor instance. It may also return a list of visitors to be submitted to the structure (see section B.5).

As the code generator prototype only supports code generation for C++ (see section B.1), the only programming language subclass of the abstract class *Visitor* is *CPPVisitor*.

**Composite States**

Due to the definition of statechart diagrams of the UML, states may contain substates (see [UML 97c], and [UML 97d] and section 3.5). The code generator design considers only one type of state; all state instances share the same interface. However, some states must have a containment relationship to other states. A solution for this kind of design problem is offered by the design pattern 'Composite'. This pattern introduces a base class (*Component*) which defines a specific interface, and two subclasses *Composite* and *Leaf*. Both subclasses implement the interface by inheriting from the abstract class *Component*. The class *Composite* additionally has an aggregation relationship to the *Component* interface. Thus, this aggregation may hold *Composite* instances as well as *Leaf* instances at runtime. See [Gamma 96] for more information about the Composite pattern.

The Composite pattern is used here for modelling nested states. Therefore, the base type of state instances is renamed *ComponentState* (see figure 59).

In statechart diagrams, a transition leading to a state region actually follows the initial states through the recursively nested states until the first atomic state is reached. Any necessary entry activities must be performed.

The class *ComponentState* offers an operation `findFirstSubstate()` which recursively searches for the first atomic substate within the state composite (see figure 60). The state composite

---

18 This refers to the prototype documented. It realizes the Alternative Implementation Technique introduced in chapter 3. For realizing the State pattern introduced in section 2.2, the visitor would behave in a different
implements *internal iterators* (Gamma et al. contrast external and internal iteration concepts for the Composite pattern, see [Gamma 96]). Once the first atomic substate is found, the path is fixed by the cursors of the internal iterators.

In order to generate code for the necessary entry-activity calls, an instance of class *PrintEntryTraceVisitor* (see figure 59) is sent along the path through the state composite. The Visitor pattern briefly explained in section B.5 is also used within the state composite. Such a visitor can be sent into the state composite by calling the the operation *handleVisitor()* defined in class *ComponentState* (see figure 60).

An instance of class *CompositeState* first calls back the visitor. Second, the visitor is passed to the current element of the internal substate iterator. An instance of class *State* calls back the visitor if it is not an initial state and stops forwarding the visitor. If it is an initial state, the visitor is submitted along the connected automatic transition instances to the first simple state the initial state points to. Each time the *PrintEntryTraceVisitor* is called, it prints the corresponding entry-activity invocation of the calling state.

**States and Activities**

A state instance may have *entry-, exit- and do-activities*. An activity is represented as an instance of class *Activity*. As there are only three types of activities, the association between *ComponentState* and *Activity* is made qualified (see figure 60). The keywords “entry”, “exit” and “do” serve as the qualifier values for direct access to associated objects. The multiplicity of the qualified association remains ‘many’ because the CASE-tool Select Enterprise allows a state to bear more than one statement assigned to one of the keywords. The set of statements sharing the same keyword represents the corresponding state activity.
The visitor registry is filled by a factory (design pattern "Abstract Factory") during system initialization.

Figure 60: Extension of the Basic Class Diagram (Part I)
Each class possesses register-operations for associated objects. These are suppressed here.

figure 61: Extension of the Basic Class Diagram (Part II)
## References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>“The Unified Modelling Language - Technical Aspects and Applications“</td>
</tr>
</tbody>
</table>
References

[Fowler 97b] M. Fowler
"Analysis Patterns"
1997 Addison Wesley

[Gamma 96] E. Gamma, R. Helm, R. Johnson, J. Vlissides,
"Design Patterns - Elements Of Reusable Object-Oriented Software"
1996 Addison Wesley

[Harel 97] D. Harel
"Executable Object Modeling With Statecharts"
IEEE Computer, 07/97

[Hitz 92] M. Hitz
"C++ Grundlagen und Programmierung"
("C++ Foundations And Programming")
1992 Springer Verlag, Vienna

[Horn 93] C. Horn
"The Orbix Architecture"
Technical Report
1993 IONA Technologies

[Iona 96] IONA Technologies
Orbix 2.3 Reference Guide
1996, IONA Technologies

"Object-Oriented Software Engineering - A Use Case Driven Approach"
1992 Addison-Wesley

[Kathuria 97] R. Kathuria
"Improved Modeling And Design Using Assimilation And Property
Modeling"
Journal Of Object-Oriented Programming
January 1997, Vol. 9, No. 8
SIGS Publications

"Pattern Languages Of Program Design 3"
1998 Addison-Wesley

"Object Oriented Software Construction"
1988 Prentice-Hall International Inc.
References

[Meyers 98] S.Meyers
"Effective C++" – Second Edition
1996 Addison-Wesley

[Meyers 96] S.Meyers
"More Effective C++"
1996 Addison-Wesley

"Modelling Synchronous and Asynchronous Communication Between Objects"
lecture at the working group „fundamentals of object-oriented modelling“ (GROOM)
German Association Of Computer Science
1996 Feb., University of Munich

"Reuse of Models and Diagrams of The UML and Implementation Concepts Regarding Dynamic Modelling"
published in
"The Unified Modelling Language - Technical Aspects and Applications"
1998 Physica-Verlag, Germany

[Oestereich 97] B.Oesterreich
"Objektorientierte Systementwicklung mit der Unified Modeling Language"
("Object-Oriented System Development With The UML")
1997 Oldenburg Verlag

"Objektorientiertes Modellieren und Entwerfen"
("Object-Oriented Modelling And Design")

[Select 97a] Select Software Tools Inc.
Select Enterprise, Volume 1
1997, Select Software Tools Inc.

[Select 97b] Select Software Tools Inc.
Select Enterprise, Volume 2
1997, Select Software Tools Inc.

"Object Lifecycles - Modeling The World In States"
1992 Prentice Hall
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Version</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>[UM 95]</td>
<td>G.Booch, J.Rumbaugh</td>
<td>&quot;Unified Method for Object-Oriented Development&quot;</td>
<td>Documentation Set Version 0.8</td>
<td>1995 Rational Software Corporation</td>
</tr>
</tbody>
</table>
[Vlissides 96] J.Vlissides, J.Coplien, N.L.Kerth
"Pattern Languages Of Program Design 2"
1996 Addison-Wesley