Semantic Analysis and Consistency Checking
of UML Sequence Diagrams¹

Aliki Tsiolakis
email: aliki@cs.tu-berlin.de

April 2001

¹Research partially supported by the German Research Council (DFG), project IOSIP as part of “DFG-Schwerpunktprogramm — Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen”.
Abstract

In a UML model, different aspects of a system are covered by different types of diagrams and this bears the risk that an overall system specification becomes inconsistent or incomplete. Hence, it is important to provide means to check the consistency and completeness of a UML model. This problem is addressed in this report by integrating the information specified in class and statechart diagrams into sequence diagrams. The information is represented as constraints attached to certain locations of the object lifelines in the sequence diagram and this allows the identification of gaps and contradictions in the specifications. Furthermore, dependencies between the sequence diagrams of a model can be investigated based on the previous results and represented in use case diagrams. The refined UML diagrams provide the foundation for the next iteration of the specification.
## Contents

1 Introduction 7

2 Object-oriented Modelling using the Unified Modeling Language 11
   2.1 UML Use Case Diagrams 12
   2.2 UML Class Diagrams 15
   2.3 UML Sequence Diagrams 18
   2.4 UML Statechart Diagrams 21
   2.5 UML Object Constraints 24

3 Modelling an Example – The Dynamic Host Configuration Protocol 27
   3.1 DHCP and an example network 27
   3.2 Modelling the Example Network with UML Diagrams 33

4 Analysis and Synthesis of Properties 43
   4.1 Analysing UML Statechart Diagrams 44
      4.1.1 States 46
      4.1.2 Guards of Transitions 47
   4.2 Analysing UML Class Diagrams 50
      4.2.1 Method Specifications 50
      4.2.2 Data Invariants 53
      4.2.3 Multiplicities 55
   4.3 Encapsulating Temporary Inconsistencies 57
   4.4 Synthesising Context Conditions in Sequence Diagrams 58
      4.4.1 Horizontal Synthesis of the Context Conditions 60
      4.4.2 Vertical Synthesis of the Context Conditions 62
      4.4.3 Synthesis of Pre- and Postconditions for Sequence Diagrams 65

5 Formalisation of Sequence Diagrams and their Extensions 69
   5.1 Formalisation of Sequence Diagrams 69
   5.2 Formalisation of Enriched Sequence Diagrams 72
## CONTENTS

6 Dependencies between Sequence Diagrams and their Representation in UML Use Case Diagrams  
   6.1 Refinement of Use Cases .................................................. 79  
   6.2 “Sequence Diagram Inclusion” ............................................ 82  
   6.3 Temporal Order of Sequence Diagrams ................................. 86  

7 Conclusion  

A UML Metamodel
Nothing that is seen, is seen at once in its entirety.

Euclid
Chapter 1

Introduction

Software systems are nowadays widely used in many areas, but developing good software is a difficult task. It can be supported by a software development process that decomposes the development into four main activities: Specification, development, verification/validation, and evolution. Each activity is (roughly speaking) the basis for the next activity. Thus, a detailed specification phase can reduce the effort needed in the development phase as well as in the verification and validation phase. Furthermore, these development processes usually provide an iterative and incremental procedure, i.e., each iteration in the main development process typically involves all four activities and improves and extends the artifacts produced in the last iteration.

Based upon the object-oriented paradigm, several development processes are provided and the **Unified Modeling Language** (UML) has become the de-facto standard as a notation in the specification phase. UML provides different diagram types to model different aspects and views of a system. Hence, the complete model consists of several diagrams to consider the system’s functionality, its static structure, internal behaviour, and the interactions between different model parts. When representing these aspects, this report focuses on UML use case, class, statechart, and sequence diagrams. These are the most typical diagram types used for analysis and design. The analysis is one of the first activities in the specification phase and yields the system requirements. The resulting model focuses on application-domain concepts and does not consider implementation details. These implementation details are considered in a design model at a later stage of the development process. A design model typically contains additional diagram types such as UML implementation diagrams. These diagram types are outside the scope of this report. Therefore, an investigation of a model at such a detailed level is only possible in case that no additional diagram types are used.

UML use case diagrams are used to specify certain aspects of the functional requirements of the system to be modelled. They describe interactions between a user of the system and the system itself. Thus, they help to define the system boundary. Different scenarios associated with a particular function are specified in sequence diagrams. These specify more details about the interactions with the environment (i.e., the users of the system) and between the objects participating in the collaboration. Furthermore, statechart diagrams specify the internal behaviour of each object, i.e., the behaviour depending on the current state of the object and the “incoming request”. Finally, class diagram provide the static structure of the system, i.e., they define the objects, their attributes and operations, and the relations between
the objects. Additionally, object constraints — as there are invariants on classes and pre- and postconditions on their operations — can be considered to refine the specification of the class diagram.

The complete model consists of diagrams of the same type modelling different parts of one aspect and diagrams of different types describing different views, i.e., the information about the system is distributed across all these diagrams. Because of this, the overall specification may be inconsistent or incomplete. The idea of this report is to gather the distributed information about the system by integrating all relevant data in the model’s sequence diagrams. This is achieved by making use of the relationships between the diagram types as defined by the UML standard. The sequence diagrams enriched in such a way allow checking the system model for consistency and completeness. Furthermore, the additional information is used to derive assumptions about the object configuration and the states of the objects before, during, or after each sequence diagram. Based on these properties, it is further possible to specify the dependencies and relations between all sequence diagrams of a model and to represent them in use case diagrams by refining certain use cases and by adding new relationships.

The aim of this report is to describe an approach how to integrate the different information and how to check the sequence diagrams that contain the additional information. Furthermore, these enrichments that are represented as constraints attached to certain points in the sequence diagrams are used to generate pre- and postconditions and to investigate the dependencies of a collection of sequence diagrams. Obviously, retrieving information from certain diagrams, integrating the data in one diagram type, and checking the system for consistency and completeness are only possible at the end of each iteration. Thus, the refinement of the use case diagram to cover the dependencies resulting from the previous investigation is the beginning of a new iteration step. This step is then based on an enhanced model.

The algorithm described can be summarised as follows:

1. Analysis of statechart and class diagrams to determine how to retrieve necessary information and how to integrate this information in sequence diagrams as constraints;
2. Synthesis of these context conditions (i.e., combinations of constraints) by checking for consistency and completeness;
3. Generation of pre- and postconditions for each sequence diagram; and
4. Investigation of the dependencies of a collection of sequence diagrams and their representation in use case diagrams.

Additionally, a formalisation based on the formalism of Life Sequence Charts (LSCs) is given to describe the results of the approach. On one hand, the formalisation should help to define the locations of the attached constraints more precisely, and on the other hand, it should be a general point of contact to other work considering for example verification or model checking.

The integration and semantic analysis and synthesis are illustrated using an example model that specifies a simple client-server system. This system is based on the Dynamic Host Configuration Protocol (DHCP) — a protocol that is widely used in networks of different size to assign network addresses to clients. The model diagrams are used throughout the report to explain the single steps.
The report is structured as follows. Chapter 2 describes the types of UML diagrams considered in this report and also emphasises — wherever possible — the relationships between the different diagram types as standardised in the UML standard. For this purpose, the UML metamodel is briefly presented in Appendix A. Chapter 4 elaborates on the integration algorithm summarised above. The diagrams that are used throughout the report to illustrate the approach are introduced in Chapter 3 that also contains a short description of DHCP. In Chapter 5, the formalisation of the enriched sequence diagrams based on LSCs is described. Chapter 6 deals with the investigation of the dependencies between a collection of sequence diagrams and their representation in use case diagrams, based on the formalisation of the sequence diagrams. Chapter 7 summarises the results of this report and discusses possible areas of future work.
Chapter 2

Object-oriented Modelling using the Unified Modeling Language

The Unified Modeling Language (UML) is a general purpose visual modelling language. It is used for specification, visualisation, construction, and documentation of software systems. In November 1997, the UML specification version 1.1 has become an OMG standard. The current version (completed in June 1999) is 1.3. Nowadays, it is the de-facto standard for object-oriented modelling because of its wide use in industry and research.

In object-oriented software modelling using UML, different aspects and views of a system are represented by different diagram types: one diagram type shows either the system’s functionality, the static structure, the internal behaviour, the interactions, or implementation facts. Hence, the complete model of a system consists of several diagrams. These are diagrams of the same type modelling different parts of one aspect and diagrams of different types describing different views.

In this report, the most typical diagram types used for analysis and design are considered: UML use case diagrams for the system’s functionality, UML class diagrams for the static structure, UML sequence diagrams for the message interactions, and UML statechart diagrams for the internal behaviour. Other diagram types such as UML collaboration diagrams, UML activity diagrams, and UML implementation diagrams are outside the scope of this report. Object constraints — as there are invariants on classes, pre- and postconditions on their operations, and guards in statechart diagrams — are also considered. For the analysis in Chapter 4, a common understanding of these terms and concepts is necessary. Therefore, an informal description for each of these diagram types and relevant constraints is given in the following sections. The notation and semantics are generally taken from the UML standard ([UML1.3]), [BRJ98] and [SP00].

Although viewpoint models support the separation of concerns, it is also important that the relationships between the different diagrams are made explicit. These relationships are based on the relationships between the different diagram types and their model elements. Wherever an explicit relationship exists it is presented in the following sections.

The architecture of UML is based on a metamodel structure. The UML metamodel is decomposed into several logical packages: the Foundation package, the Behavioral Elements package, and the Model Management package. The package structure helps to reduce the
complexity by grouping metaclasses that show strong cohesion with each other and loose coupling with metaclasses in other packages. The *Foundation package* and the *Behavioral package* are further decomposed into subpackages. The model elements of the different UML diagram types are part of those packages and subpackages. The UML Semantics Guide deals with the package and subpackage structure. A brief description of the metamodel can also be found in Appendix A.

UML has its origin in the notation of previously existing object-oriented methods which has led to similarities to diagrams used in these other methods. UML use case diagrams are similar in appearance to those in OOSE. UML class diagrams are a melting of OMT, Booch and class diagrams of most other object-oriented methods. Sequence diagrams were found in a variety of object-oriented methods named there interaction diagrams, message trace diagrams, event trace diagrams. In general, they date to pre-object days as Message Sequence Charts and timeline diagrams. UML statechart diagrams are substantially based on the statecharts of David Harel with minor modifications.

Furthermore, other concepts have been added in UML that did not previously exist in the major modelling languages, e.g., the extension mechanisms such as stereotypes, constraints, and tagged values ([UML1.3], p. 1-10).

### 2.1 UML Use Case Diagrams

The UML use case diagrams are regarded in two parts of the UML standard ([UML1.3]) — in the Semantics (p. 2-119 - 2-130) and in the Notation part (p. 3-87 - 3-93).

In the UML metamodel, the model elements for use case diagrams are assigned to the *Use Case* package that is a subpackage of the *Behavioral Elements package*. It specifies the concepts used for defining the functionality of an entity such as a system or a subsystem without specifying its internal structure.

A use case diagram shows the relationships among actors and use cases within a system. A use case is used to define the behaviour of an entity, i.e., of a system, a subsystem, or a class. Each use case specifies a sequence of actions — including variants — that the entity can perform interacting with the actors. Use cases define the functionality on different levels of granularity which depend on the entity, e.g., use cases of a system define the functionality the system offers to clients (users, environment, other systems, etc.) and thereby describe the system’s border. This means, use cases are an appropriate means to envisage the system. On the other hand, use cases of classes are more often used on the realisation level. The use cases of classes are mapped onto operations of the classes, since a service of a class is in essence the invocation of the operations of the class. The notation for a use case is an ellipse containing the name of the use case. The boundary of a system, subsystem or class can optionally be represented by a rectangle enclosing the use cases in a use case diagram.

The service of the entity, i.e., the use case, is initiated by a user. Users can represent human users and other systems if the entity is a system, or also other instances that appear inside the system if the entity is a subsystem or a class. These parties outside the entity are modelled by actors in a use case diagram and are represented as “stick man” figures with the names of the actors below. An actor defines a set of roles that users can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.
A possible communication path is modelled by an association between a use case and an actor. The actor communicates with the entity by sending messages to the use case. In this manner, the actor initiates the use case. As a response, the use case performs its service as specified by the sequence of actions. This may include communication with actors (not necessarily only the initiating one) and elements inside the system, but not with other use cases. Actors and use cases communicate by using signals. For communication inside the system (i.e., with other elements) further communication semantics are also allowed. The interaction may continue until the instance has responded to all input and does not expect any further input. The use case continues until the end of the sequence of actions.

As mentioned before, use cases also include possible variants of the sequence of actions (e.g., alternative sequences, exceptional behaviour, error handling, etc.). These variants can be modelled by extending the behaviour of the use case with some additional behaviour defined in an extending use case. The extension is modelled in a use case diagram by an extend relationship between the extension and the base use case. One use case may extend several use cases and one use case may be extended by several use cases (but no cyclic extensions are allowed). The extension is guarded by a condition that must be fulfilled if the extension is to take place. Since the condition may not be fulfilled in some cases, the base use case may not depend on the addition of the extending use case. The relationship references a sequence of extension points in the base use case which define the locations where the additions are to be made. Once a base use case reaches the first location of an extension point, the condition of the relationship is evaluated. If the condition is fulfilled, the sequence obeyed by the use case is extended to include the sequence of the extending use case. The different parts of the extending use case (i.e., the sequence of actions) are inserted at the locations defined by the sequence of extension points in the relationship. If the extension is not triggered at this first location of the (first) extension point, the condition is evaluated at the next referenced location of the extension point. A fulfilled condition results then in the extension. The location references of the extension point can be described in a textual way, with pre- or postconditions or using the name of a state in a state machine ([UML1.3], p. 2-129). There is no detailed description in the UML standard neither how these sequences of actions nor how the different parts of an extending use case are modelled. Also a specification for the locations is missing. A possible notation is given in Section 6.1.

It is also possible to model an include relationship between two use cases meaning that a base use case contains the behaviour defined in another use case. The included use case represents an encapsulated behaviour that is added at exactly one location of the base use case. The addition takes place when a use cases reaches the location where the behaviour is to be included and performs all the actions described by the included use case. Then the base use case continues according to its original specification. Therefore, although there may be several paths through the included use case (for example due to conditional statements), all of them must end in such a way that the base use case can continue. Since the included

---

1 More precisely, instances of uses cases and instances of actors interact when the services of the entity are used.
2 Use cases never communicate with other use cases of the same entity since each of them individually describes a complete usage of the entity ([UML1.3], p. 2-127).
3 The notation for the extend relationship is a directed association from the extension to the base use case labelled with the keyword "<extend>". Stereotypes are described in the Appendix A on page 91.
4 The notation for the include relationship is a directed association from the base use case to the addition labelled with the keyword "<include>". Stereotypes are described in the Appendix A on page 91.
use case represents encapsulated behaviour and a use case may be included in several other use cases, the behaviour may easily be reused. Furthermore, it is possible that a use case includes several other use cases (but cyclic dependencies are not allowed).

The third kind of relationship between use cases of one entity is a generalisation relation and is described in [UML1.3] on p. 2-128. Generalisation relationships between use cases are not referred to in this report.

Figure 2.1 shows a use case diagram with two actors (Actor1 is depicted with the actor symbol, Actor2 is presented as a rectangle) and five use cases. The use case named Use Case has associations to both actors, the Base Use Case can be initiated only by actor Actor1. The behaviour of the Base Use Case includes the behaviour of the use cases Addition 1 and Addition 2. In case that the condition extension condition of the extension is fulfilled when evaluated at the location reference location, the behaviour of the use case Extension is added to the sequence of actions of the Base Use Case. The location of the extension point and the condition are described with a textual notation.

The relationships to other diagram types are explicitly described in the UML specification. The realisation of a use case in terms of cooperating objects (defined by classes in the entity) can be specified in collaboration or sequence diagrams. They model how objects in the system interact to perform the sequence of actions of the use case (see [UML1.3], p. 2-119, 2-122). As well, a description of a use case can be given in plain text, as well as using operations and methods together with attributes, in activity graphs, by a state machine, or pre and post conditions (see [UML1.3], p. 2-127).
2.2 UML Class Diagrams

UML class diagrams are considered in the UML standard only in the Notation Guide ([UML1.3], p. 3-31 - 3-47) but their model elements are presented in the Semantics Guide as well. The relevant model elements are defined in the Core package (a subpackage of the Foundations package) (see also Appendix A). A brief description of class diagrams that concentrates on the model elements relevant for this report is given in the following. Which model elements are used in a system’s class diagram depends mostly on the level of abstraction and the complexity of the system.

A class diagram shows the static structure of a system’s model, i.e., a collection of system elements, their internal structure, and their relationships to other system elements. For graphical convenience, it is possible to separate the system’s static structure into several class diagrams. The system elements and their relationships are modelled as classes and different kinds of static relations (associations, aggregations, compositions and generalisations). The package concept of UML provides a hierarchical grouping mechanism for classes.

A class is a description of a set of objects with similar structure, behaviour, relations and semantics. Every class has a name that must be unique (at least, within its package in which it is declared). At run-time execution, objects as instances of classes are used. All objects of a class share the same list of attributes and operations, but their attribute values can be different. The class symbol (a rectangle) is superscribed by the class name. If the class’ attribute and operation lists are shown as well, the class symbol contains three compartments. A reference to a class can be given by using its full pathname, e.g., a class declared in a package is referenced as package-name::class-name. The attribute and operation lists of a class are lists of text strings (one for each attribute or operation, respectively).

- An attribute is a structural feature of the class. The domain of the attribute’s value, i.e., the attribute’s type, is a language dependent specification of the implementation type of the attribute. It is not defined by the UML specification. Since an attribute is semantically equivalent to a composition association, it is also possible that other classes of the system’s model are used for attribute types, e.g., if an attribute stores a reference to one object of that class. Optionally, an initial value for the attribute can be specified. The visibility of an attribute specifies whether the attribute can be used by other objects. It is either public (+), protected (#) or private (-). Public attributes are accessible for all other objects with visibility to the object (including the object itself), protected attributes for any descendent, and private attributes can be used only by the object itself. The default syntax of an attribute in the attribute list of a class is:

\[
\text{visibility attribute-name : type-expression = initial-value}
\]
• An operation is a service that an object of the class may be requested to perform. The operation’s result may depend on parameters which are specified in the parameter list of the operation. Alike the list of attributes and operations of a class, it is a comma-separated list of parameter specifications. The domain of the values is a language-dependent definition (see the description of attribute types on page 15). As well, the type of the parameter can be specified (either in, out, or inout). In this report, the parameter types are not considered. To indicate that the operation returns a value, the domain of this value is given as a language-dependent definition. It is omitted if the operation has no return value. The visibility of operations is handled like the visibility of attributes. A stereotype keyword can precede the entire operation string.\(^8\)

The default syntax of an operation in the operation list of a class is:

```
visibility operation-name (parameter-list) : return-type-expression.
```

Each parameter is specified by `name : type-expression`.

A method realising an operation has the same signature as the operation and a body implementing the specification of an operation.

The structural relations between two classes are specified by associations and generalisations. An association between two classes is represented by a line between the classes and an optional association name. Generally, it is possible to distinguish binary associations connecting exactly two classes and n-ary associations, but the latter are not referred to in this report. The UML metamodel separates the association itself and its ends. Binary associations therefore have exactly two association ends. Different properties of the association can be specified by adornments at the association ends: indicators for navigation and aggregation, role name, multiplicity, and visibility.

• If an arrow head is attached to the end of the line, it is indicated that navigation is supported in the given direction. This directed association shows one-way navigability. Bidirectional associations (usually represented just by the line without any arrow heads) show the navigability in both directions. Specifying that an association is navigable is a statement that, given an object at one end, one can easily and directly get to objects at the other end, usually because the source object stores some references to objects of the target ([BRJ98], p. 144). References to the same class (i.e., to objects of the same class or even the same object) are possible.

• The aggregation indicator denotes one of the two special forms of associations that show part-of relationships: aggregation and composition. To represent an aggregation, the aggregate end of the association is marked with a diamond. The aggregation is entirely conceptual. Its intended meaning is to distinguish a “whole” from a “part”. ([BRJ98], p. 68) The composition is a stronger form of an aggregation expressing that the part is strongly owned by the composite and may not be part of any other object. This means normally that every part of a composition is destroyed in case the composite is destroyed (i.e., nonexistence of the part without the composite). A composition is represented by a filled diamond.

\(^8\)Stereotypes are described in the Appendix A on page 91.
• By adding a role name at an association end, objects of the class at this association end get a name which is used from objects of the other association end’s class while traversing across the association to these objects. The role indicated by the role name can be played by one object or a set of objects. ([UML1.3], p. 2-22, 3-65)

The role name is a representation of a pseudo-attribute of the source class (the class at the other association end), i.e., it may be used in the same way as an attribute. The following adornments — multiplicity and visibility — are properties of the role of the association.

• A multiplicity attached to an association end specifies the number of objects that may be associated with a single object across the given association. The multiplicity is a range of nonnegative integers (lower-bound .. upper-bound). A lower bound of zero indicates that the given qualifier value may be absent, i.e., that the object does not need a reference link. A lower bound of one enforces the presence of the qualifier value. Table 2.1 shows the possible multiplicity ranges and their meaning. The multiplicity

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n..m</td>
<td>default syntax for a range of values ( n \leq m )</td>
</tr>
<tr>
<td>n = m</td>
<td>exact number of instances ( n = m )</td>
</tr>
<tr>
<td>0..*</td>
<td>general case ( 0..* )</td>
</tr>
<tr>
<td>( n_i ) .. m ( i )</td>
<td>list of multiplicity ranges ( n_i \leq m )</td>
</tr>
<tr>
<td>0..0</td>
<td>no valid value (indicates that no instances could occur)</td>
</tr>
<tr>
<td>0..1</td>
<td>common case, at most one associated object</td>
</tr>
</tbody>
</table>

Table 2.1: Multiplicities attached to association ends

at composite association ends is only allowed to be 1 because the component object strongly belongs to the composite. Unless a list of ranges is not equivalent to one range, it is not discussed in this report. For example, the list of two ranges \( 2..4, 4..* \) can also be represented as one range \( 2..* \), but the list of two ranges \( 2..4, 6..8 \) has no equivalent one-range form. ([UML1.3], p. 2-22, 2-60, 3-68)

• The visibility of the association end can be restricted (from the viewpoint of the class on the other association end) using the keywords public (+), protected (#) and private (-). These visibility indicators restrict the navigability and the use of the role name via link paths. Its meaning is analogous to the use of attributes with the same visibility keyword. ([UML1.3], p. 2-22, [BRJ98], p. 145)

It is also possible that classes are related by a generalisation relationship that is used to show that the properties defined in the superclass (parent) are also present in the subclass (child). The subclass inherits all properties of the superclass, especially all attributes and operations, including all classes contained in the superclass (e.g. by composition or aggregation). However, additional properties (e.g., attributes, operations, and associations) can be specified in the subclasses. Generalisation between classes implies substitutability, i.e., an instance of the subclass may be used whenever an instance of the superclass is expected, but not vice versa. Inheritance cycles are not allowed (in contrast to the possibility of “association cycles”, i.e., associations to the same class). UML allows single as well as multiple inheritance.

---

9 As role names represent pseudo-attributes, there must not be an attribute in the source class with the same name. See the corresponding well-formedness rules in [UML1.3], p. 2-50
Figure 2.2 shows an example class diagram. The class Class1 has attributes and operations with different visibilities, different parameter lists and types and return types. The last line in the operation compartment of Class1 shows an operation with a stereotype. Class3 has an operation stereotyped as a constructor and an operation stereotyped as a destructor. The other classes have no attributes or operations. The class diagram also contains examples for directed and bidirectional associations, composition, and multiple inheritance.

![Class Diagram Example](image)

**Figure 2.2: Example for an UML class diagram**

### Relationships to other diagram types

The relationships to other diagram types are not explicitly described in the UML Notation Guide (Part 5 (Static Structure Diagrams), [UML1.3], p. 3-31 - 3-87) because a description of the static structure of a system does not depend on the description of other aspects. But the model elements used in sequence diagrams and statechart diagrams are partly associated with those of class diagrams and these relations are discussed at the end of Section 2.3 and 2.4, respectively.

#### 2.3 UML Sequence Diagrams

UML sequence diagrams are considered in the Notation Guide of the UML standard ([UML1.3], p. 3-93 - 3-105) and the model elements for interaction diagrams in its Semantics Guide ([UML1.3], p. 2-106 - 2-119). Because UML defines two forms to describe the interaction of a group of objects — sequence diagrams and collaboration diagrams — the descriptions of both diagram types are used (see also UML collaboration diagrams in [UML1.3], p. 3-105 - 3-125).

Both forms of interaction diagrams describe interaction patterns among objects. They are based on the same underlying information, but each form emphasises a particular aspect of the interaction. Sequence diagrams focus on the explicit time sequence of the interactions.
2.3. UML SEQUENCE DIAGRAMS

Unlike collaboration diagrams, they do not show the relationships among the objects playing the different roles in the interaction and their links to each other. So the relation between the sending and the receiving object is not explicitly represented.

A sequence diagram shows an interaction arranged in time sequence. The time is represented by the vertical axis of the sequence diagram (normally time proceeds downward) and the participating objects by the horizontal dimension. The interaction is realised using messages sent from the objects playing the sender role to the objects playing the receiver role.

The elements in interaction diagrams that are participating in an interaction are not objects although objects communicate to accomplish the task of the collaboration. Instead, they are roles describing many possible objects. Each role may be held by a specific object or a set of objects. ([UML1.3], p. 3-59) Nevertheless, these elements are commonly called objects or instances. The representation of the objects by object symbols is similar to classes in class diagrams because objects are instances of classes.

The object symbols in a sequence diagram are superscribed by object-name/role-name: class-name.

If there is only one role to be played by objects of a particular class, the role name may be omitted. UML defines for object symbols in collaboration diagrams that the object name, the role name or the class name may be omitted (see [UML1.3], p. 3-116). For the analysis in this report, at least the class name is expected for an object to match with a class of the class diagram.

The lifeline shows the existence of an object and is represented by a dashed line. If an object is created during the time period of a sequence diagram, the lifeline starts at that point. If an object is destroyed or terminates during this period the lifeline ends with the object termination symbol (X).

The communication between the objects is denoted by an arrow from the sender’s to the receiver’s lifeline that represents a message. In case of a message from an object to itself, the arrow may start and finish on the same lifeline. The message arrow is labelled by the operation to be invoked. The message label is return-value:=operation-name (argument-list). The arguments in the argument list can be object references, expressions that use return values of previous messages (in the scope of this sequence diagram) and navigation expressions starting from the sender object (i.e., attributes of the sender object, and links from the sender object and paths reachable from them). Variables might be used for the return value. Normally messages are drawn horizontally to indicate that the duration required to send the stimulus is “atomic” (i.e., it is brief compared to the granularity of the interaction and that nothing else can “happen” during the transmission). Otherwise, the message can be drawn in such a way that the arrowhead is below the tail. The latter are not referred to in this report.

Messages can also invoke the constructor of a class or the destructor of an object. A message that initiates the creation of a new object (by invoking the constructor of the object’s class) is represented by an arrow pointing with the arrowhead to the object symbol. The unique identity of the new object is then returned to the creator. This also implies that the objects created by the associated classes are semantically connected (i.e., that links exist between the objects, according to the requirements of the associations.) If the object termination symbol is the target of the arrow, then the arrow is a message that invokes the destructor of the object’s class.
Branching points are shown by multiple arrows leaving a single point, each labelled by a guard condition. If the guard conditions are mutually exclusive they represent conditionality, otherwise concurrency. ([UML1.3], p. 3-103) A special case of concurrent messages is a branching point where all guard conditions are true. It shows messages where the sequence of the sending is undefined and supposed to be done in parallel. The graphical notation of this special case is limited to a few messages at one branching point.

If all messages of a branching point are sent to different objects playing the same role in this collaboration and all messages invoke the same operation with identical arguments, it is assumed that the notation of multiobjects used in collaboration diagrams can be applied to sequence diagrams as well. A multiobject represents a set of objects all playing the same role. A message arrow to a multiobject symbol indicates a message that addresses the entire set, rather than a single object in it. This corresponds to an association with upper bound “many” used to access a set of associated objects. To perform an operation on each element in a set of associated objects requires an iteration to extract links to each individual object and then a message sent to each individual object using the received link. In contrast to collaboration diagrams, there is no possibility in sequence diagrams to show that an object is part of the set. Multiobjects are represented as two rectangles in which the top rectangle is shifted slightly vertically and horizontally to suggest a stack of rectangles (see [UML1.3], p. 3-117).

Figure 2.3 shows an example sequence diagram. The object o2 of Class2 plays the role role2 to communicate with objects of Class1 and Class3. This role conforms to the association roles privateRole and sourceRole in the example class diagram (Fig. 2.2 on page 18). The unnamed object of Class1 plays a role that conforms to the role in the association between Class1 and Class2. The role name of the object o3 of Class3 is omitted, expressing that objects of this class can only play one role. The sequence diagram contains a message between to existing objects, a message from an object to itself, and messages that invoke the constructor or destructor.

![Sequence Diagram Example](image-url)
2.4  UML STATECHART DIAGRAMS

There are several relationships to other diagram types especially to model elements of class diagrams. To send a message from an object to another one, a link — the instance of an association — is used. Therefore, an association between the classes of the objects has to exist in the class diagram that is navigable from the sender’s class to the receiver’s class. Furthermore, the operation to be invoked by the message has to exist in the receiver’s class (e.g. in the list of operations or by inheritance). The receiver object and the operation in the receiver object have to be visible for the sender object. Additionally, the roles that are played by the sender and receiver must conform to the roles of the association that is used for this interaction. This conformance is described in the UML metamodel in the Collaborations package (see for example the well-formedness rules in [UML1.3], p. 2-111).

The last paragraph of the next section (on page 23) will deal with the relationships between UML sequence diagrams and UML statechart diagrams.

2.4  UML Statechart Diagrams

UML statechart diagrams are described in the UML Notation Guide ([UML1.3], p. 3-125 - 3-145) and their model elements are regarded in the UML Semantics Guide ([UML1.3], p. 2-130 - 2-159). The concepts of statecharts are specified in the State Machine package, a subpackage of the Behavioral Elements package. UML statecharts are an object-based variant of Harel statecharts. In this report, only a simple form of UML statecharts is considered leaving out concepts like concurrent and hierarchical states, history and timing events.

The UML statechart diagrams can be used to model the behaviour of individual class instances and also contain the possible interactions between objects by events and actions. A statechart diagram describes possible sequences of states and actions through which the object can proceed during its lifetime as a response to events. This event processing is stepwise and therefore the incoming events are queued until they can be processed.

A statechart diagram contains different states each representing a situation during which some (usually implicit) invariant condition holds and the object optionally performs some action. The invariant may represent a static situation such as an object waiting for some operation to be called. Conceptually, an object remains in a state for an interval of time. When a state is entered as a result of some transition it becomes active. If it is exited as a result of a transition it becomes inactive. A state is depicted as a rectangle with rounded corners and an optional name. The beginning and the end of the sequence of states are marked by the initial state and the final state. The former has at most one outgoing transition and no incoming transitions and implies that the corresponding initial transitions creates the object while the latter has no outgoing transition and implies the termination of the object. The representations are a small solid filled circle (initial state) and a circle surrounding a small solid filled circle (final state).

A transition is a directed relationship between a source state and a target state, i.e., between the state before the transition is triggered and the state that is reached when the transition is taken. The transition indicates that specific actions will be performed when a specified event occurs provided that a certain condition is satisfied. A transition is labelled event (parameter-list) [guard-condition] /action.
A transition is triggered by at most one event that represents the reception of a request to invoke a specific operation of the object. The list of parameters conforms to the operation’s list of parameters. The parameters are available to all actions directly caused by that event.

A guard provides control over the firing of the transition. A guard is represented as a boolean expression and a missing one is always true. It is possible that the boolean expression contains parameters of the event and attributes and links of the object, but in well-formed models guards should be without side effects.

A transition can also effect an optional action or sequence of actions that is performed when the transition fires. It is possible that the action expression contains operations, attributes and links of the corresponding object as well as parameters of the triggering event and other attributes and operations visible in its scope. A semicolon separates the actions in a sequence of actions. The sequence of actions is performed sequentially.

The event that a statechart receives is a result of some action either within the object or in its surrounding environment. If an event is received, it is placed in the event queue that holds incoming event instances until they are dispatched. As mentioned before, the statemachine dequeues and processes one event for each step. A transition is enabled if the source state is active, the trigger of the transition is satisfied by the current event and the guard is evaluated to true. When an event is dispatched, it may result in one or more transitions being enabled for firing. If exactly one transition is enabled, the transition fires. If more than one transition is enabled, one of those conflicting transitions is chosen by the statemachine. If no transition is enabled, the event is discarded and the step is complete. A fired transition is completed when all actions have been completed.

It is also possible that a transition has no explicit trigger, although it may have a guard defined. When all transition actions and entry actions in the currently active state are completed, this completion transition will be taken automatically before any other event of the event queue is dispatched. If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions, otherwise the behaviour of the object is non-deterministic.

A run-to-completion step is completed if the transition enabled by the event has fired and a possible sequence of completion transitions has been taken, too. For the next step another event is dispatched from the event queue.

Figure 2.4 shows a UML statechart diagram for an object of class Class1 (see the example class diagram, Fig. 2.2 on page 18). The statechart diagram contains all forms of transitions discussed in this report including completion transitions.

\(^{10}\)In the UML Semantics Guide different events are considered: “call events” (invocation of a specific operation), “change events” (raised implicitly), “signal events” (represent the reception of a particular asynchronous signal) and “time events” (model the expiration of specific deadline). This report only deals with call events because time, timing constraints, and signals are not considered for the other diagram types, either.

\(^{11}\)The firing priorities and the transition selection algorithm are described in [UML1.3], p. 2-153. In statecharts with concurrent substates sets of parallel/non conflicting transitions can fire simultaneously within one step.

\(^{12}\)For these completion transitions a completion event is generated as an implicit trigger. The completion event is then dispatched before any other queued events. Therefore, the completion transition fires before any other event of the event queue is dispatched.
2.4. UML STATECHART DIAGRAMS

Figure 2.4: Example for an UML statechart diagram

Relationships to other diagram types

UML statechart diagrams are related primarily to sequence diagrams and class diagrams. A class diagram defines the attributes, operations and relations for a class of objects. This static structure is the basis for inter- and intra-object behaviour (modelled in sequence and statechart diagrams).

In statechart diagrams, events invoking operations of the particular object have to conform to existing and visible operations in this object. The actions that are performed when a transition fires have to conform to operations (in the same object or in the called object) in the same way.

In sequence diagrams, the messages between objects conform to these events and actions. The sending of a message generates an action. This action causes the reception of an event in the receiver object.

The statechart diagram explains the different results of sending a message — and therefore the different behaviours of an object. The behaviour — and thus the discarding of events or firing of different transition — depends on the current state of the object.

The UML metamodel describes the relationships between all the diagrams discussed in this report (see also Appendix A). The model elements of use case diagrams, sequence diagrams and statechart diagrams are defined in the Behavioral Elements package. The Use Case package (the basis for use case diagrams), the Collaborations package (the basis for sequence diagrams) and the State Machine package (the basis for statechart diagrams) are all subpackages of the Common Behavior package — a subpackage of the Behavioral Elements package.

In these packages, model elements from the Core package — the basis of the class diagram — are used. An interesting example is the metaclass Action and its relations as shown in Figure “Common Behavior – Actions” ([UML1.3], Fig. 2-15, p. 2-88), Figure “Common Behavior – Instances and Links” ([UML1.3], Fig. 2-16, p. 2-89), Figure “Collaborations” ([UML1.3], Fig. 2-17, p. 2-107) and Figure “State Machines” ([UML1.3], Fig. 2-21, p. 2-131). The metaclass Action is related to the metaclasses Operation, Object (possible via other metaclasses), Message, State and Transition.
2.5 UML Object Constraints

A UML diagram, such as a class diagram, is typically not detailed enough to address all the relevant aspects of a specification. Among other things, there is a need to describe additional constraints about the objects in the model that specify conditions and propositions that must be maintained as true. In this report, constraints for classes and their attributes and operations — data invariants and pre- and postconditions — and constraints for multiplicities attached to associations — multiplicity invariants — are considered. The UML standard contains a predefined (formal) language to describe constraints — the Object Constraint Language (OCL) ([UML1.3], p. 7-1 - 7-50). Using this constraint language, it is possible to describe constraint expressions as text strings but not only in natural language.

Furthermore, there are also other diagram types that contain constraining expressions, e.g., guards in statechart diagrams. Moreover, the UML specification allows to attach constraints to all model elements. This includes the special constraints data invariants, pre- and postconditions, multiplicity invariants, and guards (see for example [UML1.3], p. 2-30). They are the most common ones and therefore investigated in this report. This section deals with their representation as OCL expressions and their representation in UML diagrams.

There are two ways to attach constraints to diagrams or their model elements: either by placing the constraint strings near the model element (e.g., the class and its operations) ([UML1.3], p. 3-25), or by a separate constraints part with context declarations and the OCL expression of these contexts ([UML1.3], p. 7-5). In this report, the latter will be used to avoid overloading the diagrams with too many model elements.

OCL provides a number of predefined data types like Boolean, Integer, String and Set and operations on these data types to be used in the constraints for attributes, parameters and return values. A complete list can be found in [UML1.3] (p. 7-28 - 7-50). Besides these predefined types, all classes in the model can be used as types for OCL expressions. To denote their elements the attributes and operations of the corresponding classes can be used. A further possibility provided by OCL is to navigate along the associations of a class using the role names defined in the class diagram. As well, the context declaration may specify parameters that can be used in the OCL expressions (e.g., if the context is an operation of a class).

There are also certain reserved keywords to denote special parts of the constraint. The following table lists the keywords used in this report.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>context declaration for the constraint expression</td>
</tr>
<tr>
<td>inv</td>
<td>invariant for the corresponding context class</td>
</tr>
<tr>
<td>pre</td>
<td>precondition for the corresponding context operation</td>
</tr>
<tr>
<td>post</td>
<td>postcondition for the corresponding context operation</td>
</tr>
<tr>
<td>self</td>
<td>reference to the corresponding context object itself</td>
</tr>
<tr>
<td>result</td>
<td>result of the corresponding context operation</td>
</tr>
<tr>
<td>@pre</td>
<td>reference to the value of an attribute before the application of an operation (used only in postconditions)</td>
</tr>
</tbody>
</table>
Constraints concerning operations of classes are pre- and postconditions to specify the effects of their invocation. A constraint with pre- and postcondition for one of the operations of Class1 (see Figure 2.2) can be written for example as:

```plaintext
class context Class1::protectedOperation (param : Integer) : Integer
  pre : param > 0
  post : result = self.privateAttribute * param
```

Constraints concerning the values of attributes of a class are data invariants that specify conditions that must be maintained as true at any time in the life of an object. A data invariant for the objects of Class4 (see Figure 2.2) for example can be written as:

```plaintext
class context Class4
  inv: (self.n > 0) and (self.n < 100)
```

Constraints concerning the multiplicity adornments of association ends are multiplicity invariants that specify a multiplicity condition that must be maintained as true at any time in an object configuration. In contrast to data invariants that define additional properties of the system, these multiplicity invariants are just an equivalent notation for multiplicity adornments of association ends. The upper bound * cannot be represented as a multiplicity invariant because it is not possible to represent an unlimited number of elements with an integer. Moreover, only multiplicity adornments at reachable association ends are represented as multiplicity invariants (i.e., the multiplicities at the source end of a directed association are not represented as multiplicity invariants since these objects cannot be addressed by any other object using this association from the target end).

Multiplicity invariants for the objects of Class1 and objects of Class4 (see Figure 2.2) can be written as:

```plaintext
class context Class1
  inv: self.component->size = 5

class context Class4
  inv: self.base->size = 1
```

---

13 An object configuration is the set of all existing objects of a system and their links at run time.
Chapter 3

Modelling an Example – The Dynamic Host Configuration Protocol

3.1 DHCP and an example network

The presentation of the UML diagrams in the previous chapter gives a first impression of how to model a system although the classes, attributes, operations and their behaviour are very abstract yet. Throughout the rest of this report, a single case study will be used. The Dynamic Host Configuration Protocol (DHCP) has been chosen for this purpose because the standardised protocol can be reduced to an easily understandable example — represented in a few diagrams.

The Dynamic Host Configuration Protocol (DHCP) as standardised by the IETF\(^1\) is a protocol that is widely used in networks of different size — at networks of universities as well as in many enterprises. For a host in a network to communicate with other hosts in the network (e.g., to have access to the Internet), numerous initialisation and configuration parameters have to be set by the user or the system administrator. These parameters include its unique network address, a netmask, and addresses of necessary servers and services. To avoid repeated manual configuration, the mechanism of DHCP can be used to provide initialisation and configuration parameters to hosts that can be dynamically configured. A DHCP client host requests the parameters from a DHCP server for the correct setting of its network parameters. This service is extremely helpful in various scenarios: for example, if a host (e.g., a notebook computer) is used in different networks and therefore needs different parameters for each network or if all hosts of a network have to be reconfigured due to changes in the network configuration.

The most important parameter is the unique network address — the IP address.\(^2\) Because the number of unique network addresses is limited (254 network addresses per subnet) it might not

\(^1\)The current standard from the IETF ([IETF]) is RFC 2131 ([RFC2131]).

\(^2\) A valid IP address consists of 32 bits, usually represented as four integer parts separated by dots. Each part is an 8-bit value (i.e., the range of this integers is 0 to 255). In a typically corporate network, the first 24 bits (i.e., the first three parts) denote the subnet to which the host is attached (e.g., “130.149.25”) and the last 8 bits (i.e., the last part) are a unique host identifier in this subnet. The host identifiers 0 and 255 are reserved as the network and broadcast address and cannot be assigned to any host. For example, “130.149.25.0” is the network address of the subnet and the network address “130.149.25.255” is used to address all hosts in this network.
even be possible to give a static network address to each client. Therefore, DHCP also provides a mechanism to share a pool of IP addresses and each client obtains a network address for a limited period of time. It allows to automatically reuse an address that is no longer needed by the client to which it was assigned. This is referred to as dynamic configuration. Examples of client hosts using this dynamic configuration are notebook computers used at home or at work, or hosts that are connected to the network for a short period of time via a modem (and therefore can share the pool of IP addresses for the dial-in ports). But DHCP also simplifies managing a network with static address assignments where addresses are manually assigned by the network administrator. Instead of configuring each host individually, this can be done based on a single (central) data base. If a permanent IP address is assigned to a client in the data base, the DHCP server will automatically allocate this IP address for the client. This is often referred to as static configuration.

The example used in this report will concentrate on dynamic configuration, since, in a particular network, not all of these mechanisms have to be used. It is not even the intention to model all of them. Moreover, the model contains only the necessary structure and behaviour of DHCP servers, DHCP clients and their communication protocol to avoid very big diagrams. Modelling a simplified DHCP is sufficient for the purpose of this report (and further details would only distract from its subject).

Three DHCP clients that have to be dynamically configured and two DHCP servers are part of the example network. The DHCP servers have been configured by the system administrator in a way that they can allocate exactly one IP address each, i.e., their pool of IP addresses is a single element set. This is the smallest possible example configuration. These IP addresses are all unique in their subnet. The subnet — used in this example for both servers — is 130.149.25 and the IP address in the pool of IP addresses is 130.149.25.20 for server s1 and 130.149.25.21 for server s2. See Figure 3.1 for a visualisation of this network.

For a connection of a (client) host to the network, at least its network address has to be configured. For this purpose, it initiates the configuration process and the DHCP server
3.1. DHCP AND AN EXAMPLE NETWORK

allocates one network address from its pool of IP addresses to the client host and delivers it using DHCP. After obtaining this address (and the other parameters that are not referred to in this report) the client should be able to exchange data with any other host in the Internet (or at least with any other host in its local network if the network is not connected to the Internet). If the client is disconnected from the network, it releases its network address thus allowing reuse of the IP address.

The DHCP clients and the DHCP servers have to follow the DHCP protocol. The protocol described in the following is used for the examples in this report, but it is only a subset of the original protocol — a few procedures and messages are missing. See [RFC2131] for the complete description of the protocol. There are also other assumptions about the network that conflict with “reality”, e.g., no loss of messages — but all theses simplifications are “acceptable” for the purpose of this report.

1. To locate available DHCP servers, a client broadcasts a DHCP_DISCOVER message.

2. Each server may respond with a DHCP_OFFER that includes an available network address. The servers need not reserve the offered IP address.³

3. The client receives one or more of these offers and chooses one server from which to request the configuration parameters. To inform the servers, it broadcasts a DHCP_REQUEST message that includes the chosen parameters and the identification of the server. If the client receives no DHCP_OFFER message it times out and retransmits its first message.

4. All servers receive the client’s broadcast and those servers not selected by the client use the message as notification that the client has declined their respective offers. The server selected by the client stores the binding in its database of allocated network addresses and sends an acknowledgement: a DHCP_ACK message. This completes the initialisation process. If the requested IP address is not available any more (has been allocated to another client), the server sends a DHCP_NAK message to the client. The client has then to start again.

5. A client receiving an acknowledgement message can use the requested network address until it chooses to relinquish its lease by sending a DHCP_RELEASE message.

The scenario in Figure 3.2 shows the communication between a client and the DHCP servers from the viewpoint of the client if the configuration process is successful.

³But [RFC2131] (p. 13) comments that “the protocol will work more efficiently if the server avoids allocating the offered network address to a client”.
CHAPTER 3. MODELLING AN EXAMPLE

Figure 3.2: Communication between client \textit{c1} and the DHCP servers \textit{s1} and \textit{s2} showing a successful configuration of \textit{c1}.
Assuming that also the user of another client (e.g., client c2) has successfully requested the connection of its workstation to the network, there is no server that can offer any network address. Therefore, the servers cannot send an offer to a request of a third client (e.g., client c3). Figure 3.3 shows a client that has to time out in order to resend its DHCP_DISCOVER later on.

![Diagram showing the network configuration and client interactions.](image)

Figure 3.3: Unsuccessful request of client c3 because it receives no answer. The client remains unconfigured (and retries later).

If a client is disconnected from the network, it sends a DHCP_RELEASE message to its server and the server removes the binding from its database. To carry on with the example network, if c2 is disconnected, client c3 can now connect to the network because server s2 can offer it an available network address.

Configuration requests can be processed by the servers “in parallel”, i.e., a server can either offer different network addresses each time or even the same to different clients. In the latter case, it is possible that one of the clients may receive a DHCP_NAK message and has to restart its configuration process (assuming that two clients have requested the same configuration). Figure 3.4 visualises a situation where a second configuration process (of c2) started and completed while the first one (initiated by c3) is not completed yet. Client c3 receives a negative acknowledgement and has to broadcast a DHCP_DISCOVER message again.
Figure 3.4: A server $s_2$ offers an available network address to two requesting clients (first to $c_3$ and then to $c_2$). The server responds with a positive acknowledgement message to the second DHCP_REQUEST message (sent by $c_2$) and with a negative acknowledgement to the first one (sent by $c_3$).
3.2 Modelling the Example Network with UML Diagrams

The previous section has described the protocol and the functionality of DHCP servers and DHCP clients and has informally visualised some interesting scenarios. This section considers the representation of such a network and the DHCP protocol functionality with UML diagrams. The UML diagram types considered in this report are use case diagrams, class diagrams and object constraints, sequence diagrams, and statechart diagrams. They are described in Chapter 2.

UML Use Case Diagram

![Use Case Diagram](image_url)

Figure 3.5: UML Use Case Diagram for the DHCP model

The UML use case diagram of Figure 3.5 shows the functionality of the network with respect to the DHCP protocol. A system administrator configures hosts in the network in order to be a DHCP client or a DHCP server (see use cases Initiate new DHCP server or Initiate new DHCP client). A user can initiate the connection or disconnection of its host (a DHCP client) to or from the network. This is modelled by the use cases Connect to network and Disconnect from network. The system administrator and the user are represented as actors. The use case diagrams represent an early modelling phase and as the knowledge about the network increases they can be more detailed after further analysis and design.
UML Class Diagram and Object Constraints

Class diagram

Figure 3.6: UML Class Diagram for the DHCP model

The class diagram of Figure 3.6 shows the system elements of the example: The DHCP servers, the DHCP clients, the IP addresses and the network itself are represented as classes.

The class **DHCP_Client** represents the set of DHCP clients that can connect to and disconnect from the network (using the public operations `connect` and `disconnect`). Either connected or disconnected (or either configured or unconfigured) a client is always part of a network (see the composition relation between the classes *Network* and *DHCP_Client*). A DHCP client can be activated by the create operation `initiate`. For the DHCP protocol mechanism, it can receive DHCP_OFFER (`offer`), DHCP_ACK (`ack`), and DHCP_NAK (`nak`) messages. The private attributes `myIPAddress` and `myServer` of each DHCP client store the configured IP address and the corresponding server, respectively.

The class **DHCP_Server** represents the set of DHCP servers that are part of a network (modelled by the composition relation between the classes *Network* and *DHCP_Server*). A DHCP server can be activated using the constructor operation `initiate` that also configures the necessary parameters of a DHCP server: the lower range of the block of IP addresses, the upper range of this block, and the corresponding subnet of the network addresses. These parameters are stored in the private attributes `ipBlock_lower`, `ipBlock_upper`, and `subnet`. The number of elements in the range of network addresses is cached in the private attribute `numberOfIPAddresses`. For example, a server initiated with the subnet “130.149.25” and the range of IP
addresses in this subnet from “20” to “22” can assign the three IP addresses “130.149.25.20”, “130.149.25.21”, and “130.149.25.22”. The server creates these network addresses during its initialisation phase (during the execution of the constructor initiate). Each DHCP server can generally react to the reception of DHCP_DISCOVER (discover), DHCP_REQUEST (request), and DHCP_RELEASE (release) messages. The private predicate hasFreeIP checks the availability of unbound IP addresses in the block of IP addresses and the private operation getFreeIP returns one of these addresses (supposed that there is an unbound IP address). The private attribute numberOfClients saves the number of clients, i.e., it caches also the number of assigned IP addresses.

The class IPAddress represents each IP address that is an element of the server’s block of IP addresses (note that the assumption is that the sets of IP addresses of different servers do not overlap). This relation is modelled by the composition relation between class DHCP_Server and IPAddress. The objects are created by invocation of the constructor operation (create (ip)) (the parameter ip is assigned to the private attribute ipAddress). Each object can be either bound or unbound as indicated by the private boolean attribute bound (the default is “false”, i.e., unbound). The value of this attribute can be interrogated by the operation getBinding and changed by the operations bindIPAddress and unbindIPAddress. Apart from the predefined stereotype <<create>>, the stereotypes <<unicast>> and <<broadcast>> are used to indicate that a message invoking this operation addresses one object or a set of objects (a multiobject), respectively. For broadcast messages from a client to the servers the directed association broadcast is used, for unicast messages the bidirectional association unicast is used.

Object Constraints

The class diagram of Figure 3.6 is further refined by constraints attached to the classes and their attributes and operations. These constraints are given in formal textual notation using the OCL for expressions. These constraints include existing invariants for the classes and their attributes and pre- and postconditions for their operations.

Class DHCP_Client

There are no data invariants for the objects of class DHCP_Client but several multiplicity invariants. The DHCP client can have at most one link to a particular server (first invariant) and is part of exactly one network (last invariant). These multiplicities are represented in the class diagram already, but can also be given as OCL constraints as follows.

```ocl
class DHCP_Client
  inv: self.aServer->size <= 1
  inv: self.network->size = 1

context DHCP_Client::ack (requestedIP : IPAddress, server :_DHCP_Server)
  post: (self.myIPAddress = requestedIP) and (self.myServer = server)
```

In this form they are used in the analysis discussed in Chapter 4. The DHCP client may also know other servers via the association allServers, but its multiplicity cannot be represented as a multiplicity invariant (see Section 2.5, p. 25).

The postcondition of the operation ack specifies that the parameters of the operation are assigned to the private attributes myIPAddress and myServer, respectively.

```ocl
class DHCP_Client

context DHCP_Client::ack (requestedIP : IPAddress, server :_DHCP_Server)
  post: (self.myIPAddress = requestedIP) and (self.myServer = server)
```
CHAPTER 3. MODELLING AN EXAMPLE

Class DHCP_Server

The invariants of the class DHCP_Server specify that the range of the IP block is valid for IP addresses in general and that the upper bound of the range is greater than or equal to the lower bound. The overall number of available IP addresses (that conforms to the multiplicity of the composition relation) is not less than the number of assigned IP addresses (the number cannot be negative). Furthermore, all IP addresses are unique. The server is part of a network with the same subnet.

context DHCP_Server
inv: (self.ipBlock_lower > 0) and (self.ipBlock_upper < 255)
inv: self.ipBlock_lower <= self.ipBlock_upper
inv: self.numberOfIPAddresses >= self.numberOfClients
inv: self.numberOfClients >= 0
inv: (self.allIPAddresses->forAll (ip1, ip2 | ip1 <> ip2 ⇒ ip1.ipAddress <> ip2.ipAddress))
inv: (self.network.subnet = self.subnet)

The preconditions for operation initiate are that the lower and upper bound of the range of IP addresses are valid for IP addresses in general and that the lower bound is not higher than the upper one. The postcondition of this operation indicates that these values and the computed number of elements in this range have been assigned to the corresponding private attributes and that the initial value for the number of assigned IP addresses is zero. Furthermore, the postcondition implies that the necessary IP addresses have been created and are unbound.

context DHCP_Server::initiate (lower : Integer, upper : Integer, subnet : String) : DHCP_Server
pre : (lower > 0) and (upper < 255) and (lower <= upper)
post : (self.ipBlock_lower = lower) and (self.ipBlock_upper = upper) and (self.numberOfIPAddresses = upper - lower + 1) and (self.numberOfClients = 0) and (self.allIPAddresses->size = upper - lower + 1) and (self.allIPAddresses->forAll (ip | ip.getBinding() = false))

The precondition of the operation request shows that the object itself is identified by the operation’s corresponding parameter and that the requested IP address had been offered by this server object.

context DHCP_Server::request (requestedIP : IPAddress, client : DHCP_Client, server : DHCP_Server)
pre : (server = self) and (self.allIPAddresses->includes (requestedIP) = true)

The precondition for operation release checks if the IP address has been assigned by this server and the postcondition unbinds the released IP address and decrements the number of already assigned IP addresses.

context DHCP_Server::release (clientIP : IPAddress)
pre : (self.allIPAddresses->includes (clientIP) = true) and (clientIP.getBinding () = true) and (self.numberOfClients > 0)
post : (clientIP.getBinding () = false) and (self.numberOfClients = self.numberOfClients@pre - 1)
3.2. MODELLING THE EXAMPLE NETWORK WITH UML DIAGRAMS

There is no precondition for operation `hasFreeIP`, but its postcondition indicates the existence of unbound IP addresses.

```java
class DHCP_Server {
    hasFreeIP(): Boolean {
        post: result = (self.numberOfIPAddresses > self.numberOfClients)
    }
}
```

The precondition of operation `getFreeIP` checks if there is any unbound IP address and the operation returns one of them (postcondition). The operation can only be called if a previous call of `hasFreeIP` has returned true.

```java
class DHCP_Server {
    getFreeIP(): IPAddress {
        pre: self.numberOfIPAddresses > self.numberOfClients
        post: result.getBinding() = false
    }
}
```

**Class IPAddress**

There are no data or multiplicity invariants for the objects of class `IPAddress`.

The operation `create` results in an unbound IP address.

```java
class IPAddress {
    create(ip: String): IPAddress {
        post: result.bound = false
    }
}
```

The precondition of the operation `bindIPAddress` checks if the IP address is unbound and the postconditions indicates the change to a bound IP address.

```java
class IPAddress {
    bindIPAddress() {
        pre: self.bound = false
        post: self.bound = true
    }
}
```

The precondition of the operation `unbindIPAddress` checks if the object itself is bound first and results in an unbound one (as denoted by the postcondition).

```java
class IPAddress {
    unbindIPAddress() {
        pre: self.bound = true
        post: self.bound = false
    }
}
```

The operation `getBinding` is a query for the value of the private attribute bound.

```java
class IPAddress {
    getBinding(): Boolean {
        post: result = self.bound
    }
}
```

**Class Network**

There are no invariants for this class.
CHAPTER 3. MODELLING AN EXAMPLE

UML Sequence Diagrams

Sequence Diagrams for the Use Case Connect to network

There are three scenarios of use case Connect to network (see Figure 3.5, p. 33) that are depicted in Figure 3.7, Figure 3.8, and Figure 3.9.

Figure 3.7: UML sequence diagram for one scenario of use case Connect to network (see Figure 3.5) showing the successful connection.

Figure 3.7 shows the successful connection of a client to the network including the successful configuration of the IP address. The user initiates the connection by invoking the operation connect and thus starts the DHCP protocol. The object named client of class DHCP_Client that plays the role aClient in this collaboration sends a broadcast message to a multiobject of class DHCP_Server — a set of objects playing the role allServers. At least one of these server objects responds by sending an offer message including an offered IP address ip and a reference to the server itself. The client then requests these parameters by broadcasting a request message and receives an acknowledgement (ack) from the same server. The user is informed by the result string about the successful connection to the network. This return message is represented by a dashed arrow.

Figure 3.8 shows the unsuccessful try to connect to the network. As in Figure 3.7 the user initiates the connection using the operation connect(). The client receives an offer for a free IP address as response to its discover message, but as response to its request message it does receive a negative acknowledgement (a nak() message).

Figure 3.9 shows an unsuccessful try to connect to the network. The user initiates the connection and thus the client sends a message to discover servers that can offer an unbound IP address. But the client receives no responses so it times out and informs the user. The timing out is not modelled because timing constraints are not referred to in this report.
3.2. MODELLING THE EXAMPLE NETWORK WITH UML DIAGRAMS

Figure 3.8: UML sequence diagram for an scenario of use case Connect to network (see Figure 3.5) showing the unsuccessful connection (indicated by the nak message).

Figure 3.9: UML sequence diagram for an scenario of use case Connect to network (see Figure 3.5) showing the unsuccessful connection (indicated by no responses).
Sequence Diagram for the Use Case Disconnect from network

![UML sequence diagram for Disconnect from network](image)

Figure 3.10: UML sequence diagram for a scenario of use case Disconnect from network (see Figure 3.5) closing the connection.

Figure 3.10 shows a scenario for use case Disconnect from network (in Figure 3.5). A user initiates the disconnection by sending a disconnect () message and the client sends a release (myIPAddress) message (with its IP address as a parameter of the message). The user is informed by means of the result string.

Sequence Diagrams for the Use Cases Initiate new DHCP server and Initiate new DHCP client

![UML sequence diagram for Initiate new DHCP server and Initiate new DHCP client](image)

Figure 3.11: UML sequence diagram for a scenario of use case Initiate new DHCP server (a) and use case Initiate new DHCP client (b).

Figure 3.11 shows scenarios for the use cases Initiate new DHCP server (a) and Initiate new DHCP client (b) modelling the creation of a DHCP server and the creation of a DHCP client. These initiations are invoked by messages from a System Administrator (an actor SysAdmin).
3.2. MODELLING THE EXAMPLE NETWORK WITH UML DIAGRAMS

UML Statechart Diagrams

The internal behaviour of the objects of class DHCP_Client and DHCP_Server are shown in Figure 3.12 and Figure 3.13, respectively. The statechart diagrams for the other classes in the class diagram (Figure 3.6) are not given because their state changes are not relevant for the given sequence diagrams (Figure 3.7, Figure 3.8, Figure 3.9, Figure 3.10, and Figure 3.11) and contain only a few states. For example, an object of class IPAddress can only be in the two simple states “bound” and “unbound”.

Statechart Diagram for the Class DHCP_Client

![Statechart Diagram for the Class DHCP_Client](image)

Figure 3.12: UML State Chart for the class DHCP_Client in the DHCP model. The private attributes are set in the operation ack (requestedIP, server).

Figure 3.12 shows a statechart diagram for the class DHCP_Client that contains the four states Not configured, Selecting, Requesting, and Configured. After creation of a new client object of this class (by an initiate () event), the client is in the state Not configured. Receiving a connect () event causes sending a message to allServers to discover an available server. Following the DHCP protocol, the client now waits for an event offer (offeredIP, server) to request this offered IP address from the server referenced by the parameter server. The client then expects to receive either a positive acknowledgement (an ack () event) indicating the successful configuration or a negative acknowledgement (a nak () event). If the configuration process was completed successfully (i.e., the client is in the state Configured), the reception of a disconnect () event causes the IP address to be released (by sending a message to the server).
If the client does not receive any offer (see for example the sequence diagram in Figure 3.9) while in state Selecting, there is a deadlock. A timing event that would put the client back to the Not configured state is missing, but these are not referred to in this report. The statechart diagram does not contain a final state because there is no destructor operation modelled in the class diagram.

**Statechart Diagram for the Class DHCP_Server**

![Statechart Diagram](image)

Figure 3.13: UML State Chart for the class DHCP_Server in the DHCP model.

Figure 3.13 shows a statechart diagram for class DHCP_Server. Server objects can be in three states: Init, Has free IP addresses, and Has no free IP addresses. After creation of a new server object by an initiate event, the guard of the outgoing (completion) transition is evaluated automatically (the completion transition has neither any event nor any action). While the server is in state Has free IP addresses it can react to discover, request and release events. Receiving a discover event from a client and having an unbound IP address, the server offers this IP address to this client. Receiving a request event (while being in state Has free IP addresses), the server evaluates the guards and either effects the sending of an ack or a nak message to the requesting client (in case of a positive acknowledgement it increments its private attribute numberOfClients). While being in state Has no free IP addresses, the server sends a nak message as reply for any received request event.

Receiving a release event (while being in state Has free IP addresses or Has no free IP addresses) results in unbinding the IP address. Additionally, if the server was in state Has no free IP addresses the server changes its state to Has free IP addresses.

There are no deadlock situations in this statechart diagram. The statechart diagram does not contain any final state since there is no event to terminate or destroy a server object.
Chapter 4

Analysis and Synthesis of Properties

In the previous chapters, the UML diagrams that are considered in this report have been described (Chapter 2) and an example system and the UML diagrams used for its specification have been introduced (Chapter 3). In this model, the different views of the system are represented by different diagram types. Although the system modellers may benefit from considering the different views independently, there exists a high risk that the overall specification is inconsistent or incomplete. Incompleteness may occur if necessary information (e.g., about the behaviour of an object) is either absent or specified in another diagram than the one under concern. The idea for semantic analysis and consistency checking of sequence diagrams as described in this report is to integrate the information specified in statechart and class diagrams into sequence diagram. This integration is based on the relationships of the different diagram types (see Chapter 2). The additional properties contain structural as well as behavioural aspects and are represented as constraints attached to certain locations on the lifelines of the objects. This representation ensures that the resulting enriched or extended sequence diagrams can be checked for consistency and completeness.

The following sections describe an approach how to integrate this information from statechart diagrams and class diagrams in sequence diagrams. As well, it is described how to check the enriched sequence diagrams for consistency and completeness. Additionally, context conditions and pre- and postconditions based on the additional information are generated for each sequence diagram. These will be used at the end to deduce dependencies of the scenarios modelled by the sequence diagrams.

The integration algorithm can be summarised as follows:

**Phase 1** Analysis of different diagrams to integrate the information in a sequence diagram

- **Phase 1.a** Retrieving information from statechart diagrams with respect to a particular sequence diagram Section 4.1
- **Phase 1.b** Retrieving information from class diagrams with respect to a particular sequence diagram Section 4.2
Phase 2 Synthesis of context conditions by checking for consistency and completeness

Phase 2.a Investigation of the context conditions at one location (horizontal synthesis) Section 4.4.1

Phase 2.b Investigation of the context conditions at two succeeding locations on the same lifeline (vertical synthesis) Section 4.4.2

Phase 2.c Generation of pre- and postconditions for a sequence diagram Section 4.4.3

In [Tsi00] and [TE00], an algorithm for consistency checking of sequence diagrams with respect to class diagrams has been described concerning existence, visibility, and multiplicity checking. For this reason, the approach concerned in this report is based implicitly on this checking algorithm. Thus, it is not examined in this report whether all classes, attributes, operations, and associations that are “used” in other diagrams are modelled in the class diagram (existence checking) or whether they are visible for the objects which use them (visibility checking). Although the checking algorithm in [Tsi00] also comprises the multiplicities of association ends, this is regarded as well in Section 4.2.3 because these multiplicities have an equivalent representation as multiplicity invariants (see Section 2.5).

4.1 Analysing UML Statechart Diagrams

The intra-object behaviour of each object (or set of objects in case of multiobjects) in the sequence diagram is described by a statechart diagram. Regarding these two behavioural diagram types — the sequence diagrams and the statechart diagrams — properties defined in statechart diagrams can specify the behaviour in the sequence diagrams in more detail. By analysing the statechart diagrams to enrich sequence diagrams with additional constraints, the possible states of an object and the conditions guarding the transitions as well as their actions are considered.

As described in Section 2.4, a state is entered by a transition (an incoming transition) and exited by (usually) another transition (an outgoing transition). Thus, if an object is in a particular state the firing of a transition may change its state. A transition is usually triggered by an event (except completion transitions) and fired if the optional guard of the transition holds. There are several types of events (to trigger a transition), but only those events that invoke an operation by receiving a message are regarded in this report. Transitions can effect specified actions. These actions themselves are usually messages sent to another object or even to the same object (see Section 2.4 for these relationships between events, actions, and messages). Thus, if a sender object sends a message to a receiver object, both objects may change their states. If the transition in the sender’s or receiver’s statechart connects the same state, the corresponding object does actually not change its state. If the receiver object is not in a state where any transition is triggered and fired by the message, only the sender object changes its state.\(^1\) The sender and receiver can even be objects from the same class (i.e., they have the same statechart diagram but may be in different states). Because a transition can effect a sequence of actions, it is possible that a sender object receiving one message sends more than one message (e.g., one message for each action).

\(^1\)The problem of events that are ignored generally or only in some states, i.e., where no transition is triggered, is further discussed in Section 4.1.1 on page 46.
4.1. ANALYSING UML STATECHART DIAGRAMS

Figure 4.1 shows the relationship between the firing of a transition in a statechart and a message sent between a sender and a receiver object. The statecharts statechart 1 and statechart 2 describe the intraobject behaviour of the sender and receiver object, respectively. In statechart 1, transition 1 is triggered by an event and fired because the condition (guard 1) holds. Because the source state of transition 2 in statechart 2 is active and action 1 conforms to event 2, transition 2 is triggered. Assuming that the guard condition holds, transition 2 is fired. Both objects change their respective state. The result of action 1 can be the sending of a message. By receiving this message, another object’s transition (transition r in Fig. 4.1) can be triggered and fired if the guard holds.

![Diagram of statechart and message relationship](image)

Having a message in a sequence diagram invoking an operation, the statechart diagrams of the sender and the receiver object have to be investigated. This concerns the transitions in the sender’s statechart that can effect the message’s sending and the transitions in the receiver’s statechart that can be triggered by the message. Two areas have to be considered: The source and target states of those transitions that may have been fired because of the sequence of messages in the sequence diagram; the guards that allowed the transitions to be fired because of the sequence of messages. The following Section 4.1.1 (p. 46) deals with the enrichment of the sequence diagram by constraints that are OCL expressions for the possible states. Guards are considered in Section 4.1.2 that describes the extension of the sequence diagrams with corresponding constraints.

Investigating a message in a sequence diagram includes analysing the statechart diagram regarding state changes caused by particular transitions. This results in two sets of transitions: First of all, the set of transitions that can effect sending the specific message and, thereafter,
the set of transitions that generally can be triggered by the message. The former set is referenced in this report as $T_{\text{action}}(\text{message})$ and is generated using the statechart of the sender object. The latter referenced as $T_{\text{event}}(\text{message})$ is generated using the receiver’s statechart. These sets are required for the following analysis.

Considering the sequence of messages sent to an object and sent by this object, some of the messages are the resulting actions of a transition. If the sending of a message $msg_j$ is the direct result of the reception of a previous message $msg_i$ (i.e., the action of the transition triggered by the received message), the object is between the receiving and the sending neither in the source nor in the target state of the transition. Such a direct result is indicated by the overlapping of the two sets $T_{\text{event}}(msg_i)$ and $T_{\text{action}}(msg_j)$. As a transition can effect a sequence of actions, it is possible to have more than one direct result.

If the sending of a message is the result of a completion transition (i.e., the two sets of transitions do not overlap), it is referred to in this report as an indirect result. Thus, after having received the event and having sent the caused messages, the object may change its state. As well, it is indicated by firing the completion transition that its optional guard holds.

4.1.1 States

Considering a message $msg$ in the sequence diagram, the sets of transitions of the concerned statecharts are generated as described before. Using the set $T_{\text{action}}(msg)$, it is possible to produce the set of possible target states of the sender that are attached as a constraint to its lifeline after the sending of all directly resulting messages. Similarly, using the set $T_{\text{event}}(msg)$, the set of possible source states is produced and then attached as a constraint to the receiver’s lifeline before receiving the message.

The algorithm guarantees that in case of directly resulting action messages, no state constraints are added between the event message and the directly resulting action messages, because the object is neither in the source nor in the target state.

Since, it is possible that more than one transition in the sender’s statechart diagram can cause the message or more than one transition in the receiver’s statechart diagram can be fired by the message, the sets of transitions can contain more than one element. Consequently, the corresponding source and target states have to be distinguished by the name of their transition. The transition names are created as temporary but unique names in the corresponding statechart diagram and may consist of the source and target state and the transition properties (event, guard and actions). For example, possible transition names are $s_1^{e[i]}a_{s_2}$ or $\text{transition 1}$. Furthermore, this allows that exactly one element in the source state constraint is linked to one in the target state constraint.

Each element state in the set of possible source or target states is represented as the corresponding OCL expression: $\text{object-name}.\text{oclInState(state)}$. The elements in the sets are separated by the boolean operator or. It may happen that the sets of transitions are empty, i.e., that no transition exists in the statechart diagrams that can be triggered by the message or that effects the message. Since this indicates a possible incompleteness or inconsistency (see critical points in Section 4.4), an empty state constraint is added. The empty state constraint is indicated by $\text{object-name}.\text{oclInState()}$. This constraint should not be mixed with constraints for states without name — a nameless state is given a generated unique name.
4.1. ANALYSING UML STATECHART DIAGRAMS

Figure 4.2 shows the state constraints attached to the locations before and after the messages using the statechart diagrams of Figure 3.12 (p. 41) and Figure 3.13 (p. 42). The message release (myIPAddress) sent by the client is the direct result of the received message disconnect (). Therefore, there is no state constraint attached after the received message and before the sending of the resulting message. As well, the return message is sent to the actor User before the object client has terminated the execution of its operation.

4.1.2 Guards of Transitions

The sets of transitions produced for each message msg in a sequence diagram can be used to generate the sets of possible source and target states (see previous Section 4.1.1) and to identify the sets of guards that have to hold. Based on the set $T_{event}(msg)$ a set of guards is produced that is attached as constraints to the receiver’s lifeline before receiving the message msg. The correct location is identified by the location of the corresponding state constraint (see Section 4.1.1).

In case of messages that are either the direct result of a completion transition or that are caused by events (i.e., other messages) that are not shown in the particular sequence diagram, another set of guards is identified based on the set $T_{action}(msg)$. Constraints containing these guards are attached to the sender’s lifeline before sending the message.

Each element in the set of guards is an OCL expression and thus can contain parameters from the event triggering the transition as well as references to the object itself and navigation expressions starting at the object or the parameters. These expressions have to be changed as described in Table 4.1 in order to provide a consistent name space for the sequence diagram. For example, the guard server=self in the server’s statechart diagram (Figure 3.13) denotes the event’s parameter (i.e., the parameter of the corresponding message). This parameter is a reference to an object named server. The message is sent to an object with name server and the keyword self in its statechart diagram references this object name. The resulting guard
constraint \texttt{server=server} shows an expression that can be evaluated to true and thus the guard holds for this object.

If the sets of transitions contain more than one element, the guards in the guard constraints are separated by the boolean operator \texttt{or} and each is additionally labelled by the name of its transition (see Section 4.1.1 for transition names).

If a message in a sequence diagram is received by an object, it is assumed that one of the triggered transitions is fired. This also means that a sequence diagram describes a possible scenario that may happen under certain circumstances. These circumstances are further investigated in Section 4.4.3 that deals with the generation of pre and post conditions of the sequence diagram itself.

<table>
<thead>
<tr>
<th>\textit{guard of a transition} in a statechart diagram</th>
<th>\textit{resulting OCL expression} in added constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{self} reference to the context object</td>
<td>\texttt{object-name}</td>
</tr>
<tr>
<td>\texttt{self.attribute-name}</td>
<td>\texttt{self} is substituted as described above</td>
</tr>
<tr>
<td>\texttt{self.query-operation}</td>
<td></td>
</tr>
<tr>
<td>\texttt{self.navigation-expression}</td>
<td></td>
</tr>
<tr>
<td>navigation expressions without side effects starting at the object itself</td>
<td></td>
</tr>
<tr>
<td>\texttt{event-parameter} reference to a parameter of the event</td>
<td>\texttt{object-name} object name of the sender object</td>
</tr>
<tr>
<td></td>
<td>\texttt{variable-name} variable name of a return value of a previous message in this diagram</td>
</tr>
<tr>
<td>\texttt{event-parameter.attribute}</td>
<td>\texttt{event-parameter} is substituted as described above</td>
</tr>
<tr>
<td>\texttt{event-parameter.query-operation}</td>
<td></td>
</tr>
<tr>
<td>navigation expressions without side effects starting at a parameter of the event</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Changing OCL expressions used in guards of statecharts into their corresponding guard constraints attached to a particular sequence diagram.

Figure 4.3 shows the guard constraints attached to the \texttt{server}'s lifeline. There are no transitions with guards in the statechart of the \texttt{client} object (Figure 3.12). See Figure 3.13 for the statechart of the server object.
Figure 4.3: Sequence diagram of Figure 3.10 with additional constraints containing the guards.
CHAPTER 4. ANALYSIS AND SYNTHESIS OF PROPERTIES

4.2 Analysing UML Class Diagrams

Objects in sequence diagrams are instances of classes in the class diagram. The class diagram does not only specify the attributes and operations of the classes, but may separately also contain object constraints. The object constraints are method specifications consisting of pre- and postconditions as well as data and multiplicity invariants. Analysing class diagrams, additional constraints for these properties are attached to the lifelines of the objects. The algorithm guarantees that the constraints are attached only to those locations where the object is in a (stable) state. These locations are indicated by attached state constraints (see Section 4.1.1). This ensures as well that the properties like data invariants are only tested (in Section 4.4) while the object is in a (stable) state. The following sections ((Section 4.2.1, Section 4.2.2 and Section 4.2.3) give more detailed descriptions of the corresponding constraints and their locations.

Messages in a sequence diagram either invoke one of the specified methods that are the implementations of operations defined in the receiver class or are messages indicating the return value of a previous method. The latter messages are of special interest for constraints containing the postconditions, since they indicate the end of the execution of an operation.

4.2.1 Method Specifications

Considering a message in a sequence diagram, it has to be checked if it invokes an operation of the receiver object\(^2\) and if a method specification exists. If a pre- and postconditions exists, they have to hold before the object receives the message and after the object has executed the method, respectively.

The pre- and postconditions consist of logical expressions that can be evaluated depending on the actual state of the object and further expressions. Thus, the following has to be regarded: parameters of the operation, attributes of the context object, and navigation expressions starting at parameters, attributes or role names. For the navigation expressions, the links and link paths of the context object are used. The parameters themselves are in fact attributes of the sender object, variables (used for return values in previous messages), and navigation expressions using links or link paths of the sender object. Therefore, a pre- or postcondition contains three sets of expressions: A set of logical expressions that only contains parameters (e.g. parameter1>parameter2), a set of logical expressions that only contains attribute values of the context object and navigation expressions using links of the context object (e.g., \(\text{self.privateAttribute > self.role1.publicAttribute}\)), and a set of logical expressions that contains both (e.g., \(\text{parameter1 + self.protectedAttribute = parameter2}\)). If an operation has no parameters, the first and the last set of logical expressions are empty. If the precondition specifies the requirements of a constructor (i.e., the operation that creates a new object), only the first set can be nonempty (because the object to be created has no attributes or links before being created).

The pre- and postcondition have to hold before receiving the message and after executing the method, respectively. Thus, the respective constraints are attached to the receiver's lifeline directly before and directly after it, respectively. Since both constraints depend on the system's

\(^2\)This existence checking is not especially referred to in this report. See the introductory notes of this chapter on page 43.
4.2. ANALYSING UML CLASS DIAGRAMS

state, *directly before* means that the receiver object does not enter a (stable) state between checking the precondition and receiving the message. Thus, the additional constraints are attached to those locations where a state constraint is attached (see Section 4.1.1). Similarly, *directly after* the execution means that the receiver object usually does not enter a (stable) state in between the execution ending and checking the postcondition. In case that an explicit return message is sent, the postcondition is attached to the location directly after its sending.

Messages that invoke the creation of a new object (stereotype &lt;&lt;create&gt;&gt;) can also have preconditions but do not have a state before their creation, i.e., these preconditions cannot be attached to such a location. By using the characteristics of these preconditions (i.e., that the set of logical expression contain only parameters such as attributes from the sender object and navigation expressions starting at the sender object), these constraints are attached *directly before* sending the message at the sender’s lifeline. If an actor is the sender of this message, the constraint containing the precondition may be attached at the actor’s lifeline. Since the actor represents the environment and thus is not part of the system, constraints attached to the lifelines of actors are not considered in the synthesis (see Section 4.4).

The pre- and postconditions contain OCL expressions. These logical expressions have to be changed in a way that yields a consistent name space for the sequence diagram. Thus, the parameter names as well as the attributes and navigation expressions of the receiver object in the logical expressions of the method specifications are changed to correspond with the argument names (i.e., the actual parameter names) in the sequence diagram. The receiver object is also called context object. Table 4.2 summarises these changes. If any of the objects does not have an object name a temporary object name is created.

In the sequence diagram of Figure 3.10 (p. 40), a message sent by the DHCP client to the DHCP server invokes the operation *release (myIPAddress)*. As specified in Section 3.2, there exists a precondition for the operation *release (clientIP)*. Figure 4.4 shows the extension of this sequence diagram with a constraint for the precondition and a constraint for the postcondition. The object references *self* and the parameter names in the conditions are changed as defined in Table 4.2.

```
DHCP_Server
server /aServer :

DHCP_Client
client /aClient :

User

disconnect ()

"Connection closed."

pre
  and ( server.numberOfClients > 0 )
  and ( client.myIPAddress.getBinding ( ) = true )
{ ( server.allIPAddresses->includes (client.myIPAddress) = true )
  and ( client.myIPAddress.getBinding ( ) = true )
  and ( server.numberOfClients > 0 ) }

release (myIPAddress)

post
{ ( ( client.myIPAddress.getBinding ( ) = false ) and
  ( server.numberOfClients = server.numberOfClients@pre -1 ) )


Figure 4.4: Sequence diagram of Figure 3.10 with additional constraints containing the pre- and postcondition.
### CHAPTER 4. ANALYSIS AND SYNTHESIS OF PROPERTIES

<table>
<thead>
<tr>
<th><strong>logical expressions</strong> in pre- or postconditions</th>
<th><strong>resulting logical expressions</strong> in added constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter-name</td>
<td>object-name</td>
</tr>
<tr>
<td></td>
<td>object name of the sender object</td>
</tr>
<tr>
<td></td>
<td>variable-name</td>
</tr>
<tr>
<td></td>
<td>variable name of a return value of a previous message</td>
</tr>
<tr>
<td></td>
<td>in this diagram</td>
</tr>
<tr>
<td></td>
<td>object-name.attribute-name</td>
</tr>
<tr>
<td></td>
<td>reference to an attribute of the sender object</td>
</tr>
<tr>
<td></td>
<td>object-name.navigation-expression</td>
</tr>
<tr>
<td></td>
<td>navigation expression starting at the sender object</td>
</tr>
<tr>
<td></td>
<td>resulting in a reference to another object</td>
</tr>
<tr>
<td></td>
<td>CONSTANT</td>
</tr>
<tr>
<td></td>
<td>constant values represent parameters assigned by the</td>
</tr>
<tr>
<td></td>
<td>environment (i.e., actors) during a run of the</td>
</tr>
<tr>
<td></td>
<td>sequence diagram</td>
</tr>
<tr>
<td>parameter.attribute-name</td>
<td>parameter is substituted as described above</td>
</tr>
<tr>
<td>parameter.query-operation</td>
<td>object-name</td>
</tr>
<tr>
<td>parameter.navigation-expression</td>
<td>object-name</td>
</tr>
<tr>
<td></td>
<td>object name of the context object</td>
</tr>
<tr>
<td></td>
<td>object-name.attribute-name</td>
</tr>
<tr>
<td>self</td>
<td>object-name</td>
</tr>
<tr>
<td>reference to the receiver object itself</td>
<td>object-name.query-operation</td>
</tr>
<tr>
<td>self.attribute-name</td>
<td>object-name.attribute-name</td>
</tr>
<tr>
<td>reference to an attribute of the receiver object</td>
<td></td>
</tr>
<tr>
<td>self.query-operation</td>
<td>object-name.query-operation</td>
</tr>
<tr>
<td>reference to the result of an operation of the</td>
<td></td>
</tr>
<tr>
<td>receiver object that has no side effects</td>
<td></td>
</tr>
<tr>
<td>self.navigation-expression</td>
<td>object-name.navigation-expression</td>
</tr>
<tr>
<td>result</td>
<td>result-name</td>
</tr>
<tr>
<td>keyword used only in postconditions for the</td>
<td></td>
</tr>
<tr>
<td>result of the operation</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Changing parameter names used in method specifications to consistent parameter names in constraints with respect to a particular sequence diagram.
4.2. ANALYSING UML CLASS DIAGRAMS

Figure 4.5 shows the results of adding the pre- and postcondition to the sequence diagram of Figure 3.11.(a) (p. 40). Since the message invokes the creation of a new object, the precondition of the constructor is attached to the lifeline of the sender object, i.e., in the example to the lifeline of the actor *SysAdmin*. The postcondition is attached after the creation of the object at the server’s lifeline. The parameters *lower*, *upper* and *subnet* are assigned by constant values during a run of the sequence diagram.

![Diagram](image)

**Figure 4.5**: Extension of the sequence diagram of Figure 3.11.(a) with the pre- and postcondition for the constructor *DHCP_Server.initiate*.

4.2.2 Data Invariants

Data invariants specified for the objects of a class have to hold at any time during the life of an object, i.e., directly after creation of the object until the moment of its termination. Therefore, constraints containing data invariants are added at several points in a sequence diagram: Directly after the creation of a new object (or, if the object already exists, at the beginning of the sequence diagram directly below the object symbol) and before and after each message (unless this location is between two states of the object). Directly after the creation or a message and directly before each message means that the object is not in a (stable) state in between the location of the constraint and receiving or sending the message.

Data invariants represented as OCL expressions can use references to the object itself and navigation expressions to other objects as well as attributes and operations of the object itself. Table 4.3 defines how to change these expressions in the added constraints in order to preserve a consistent name space for the sequence diagram. If, for any object, there is a

---

3The attachment to the actor’s lifeline is optional since the actor as a representative of the environment provides no statement about the system’s state. Constraints attached to the actor’s lifeline are not considered during the synthesis phase.

4This implicit encapsulation of certain intermediate “states” is discussed in Section 4.3 (p. 57).
data invariant but not an object name in the sequence diagram, a temporary object name is created.

<table>
<thead>
<tr>
<th>data invariant of a class in a class diagram</th>
<th>resulting OCL expression in added constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>object-name</td>
</tr>
<tr>
<td>self.attribute-name</td>
<td>object-name.attribute-name</td>
</tr>
<tr>
<td>self.query-operation</td>
<td>object-name.query-operation</td>
</tr>
<tr>
<td>self.navigation-expression</td>
<td>object-name.navigation expression</td>
</tr>
</tbody>
</table>

| reference to the context object            | object-name                                 |
| reference to an attribute of the object itself | object-name.attribute-name                  |
| reference to the result of an operation without side effects | object-name.query-operation                 |
| reference to a linked object and its values (e.g., by using the role name at the other end of the association for navigation) | object-name.navigation expression           |

Table 4.3: Change of OCL expressions used in data invariants of the class with respect to a particular sequence diagram.

Figure 4.6 shows the enrichment of the sequence diagram with the data invariants for the objects defined in Section 3.2. The constraints for the data invariants of the object server of class DHCP_Server are added at the specified locations. There are no data invariants for the object client of class DHCP_Client.

Figure 4.6: Sequence diagram of Figure 3.10 (p. 40) with additional constraint containing the data invariants.
4.2.3 Multiplicities

Multiplicity adornments at association ends can be represented as equivalent multiplicity invariants (see Section 2.5, p. 25). Each multiplicity invariant represents a property of the object configuration and must be maintained as true at any time in the object configuration of a running system. The multiplicity properties are attached to each object of the corresponding class, because each object has to have its necessary links. For example, a multiplicity constraint \( \{ \text{self.allIPAddresses->size} = 1 \} \) (with an object of class \DHCP\_Server as context) claims the existence of at least two objects and a link from the context object to the other object: one object of class \DHCP\_Server\ and one object of the class at the other association end that can play the role \allIPAddresses\, i.e., an object of class \ IPAddress\.

Since a sequence diagram cannot show all objects of an object configuration at runtime, some of the objects are taking part in the communication shown in the sequence diagram and others are not. Nevertheless, if a multiplicity specifies that an object has a link to another object (see example above), this has to be maintained from the creation of this object until its termination.\(^5\)

Similar to data invariants, the multiplicity invariants are added in the sequence diagram below the object symbol (i.e., also below the object symbol of a newly created object). Since every message can invoke the creation or destruction of an object, each message can effect the change of the object configuration. A constraint containing the multiplicity invariants is attached at least to these locations but, for convenience, also at all locations where a state is attached.

Multiplicity invariants contain navigation expressions that result in a set of objects of the class at the other end of a navigable association. There are at least as many objects of this class in the object configuration as specified by the number of elements in the set of objects. Therefore, a multiplicity invariant below the object symbol of an existing object denotes the (minimum and maximum) number of links an object must have. A message between a sender and a receiver object is assumed to be link from the sender to the receiver object. The navigation expression of the invariants is changed as defined in the following table (Table 4.4).

<table>
<thead>
<tr>
<th>multiplicity invariant of a class in the class diagram</th>
<th>resulting OCL expression in added constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.role-name-&gt;size</td>
<td>object-name.role-name-&gt;size</td>
</tr>
</tbody>
</table>

Table 4.4: Change of OCL expressions used in multiplicity invariants of the class with respect to a particular sequence diagram.

Figure 4.7 shows the added multiplicity invariants for the client of class \DHCP\_Client\ and for the server of class \DHCP\_Server\ in the sequence diagram of Figure 3.10 (p. 40). The additional\(^6\)

\(^5\)The links of a newly created object are usually generated as a direct result of the invoked constructor (i.e., the generation of the links are the actions of the transition triggered by the constructor).
constraints are added below the object symbol of the existing objects and after each message interaction. The last dashed arrow is not a message but the result of another message and thus cannot change the object configuration.

Figure 4.7: Extension of the sequence diagram of Figure 3.10 with the multiplicity invariants of all corresponding classes.

Figure 4.8 represents the enrichment of the sequence diagram modelling the creation of a new DHCP client (see Figure 3.11.(b), p. 40). The multiplicity invariant is added directly below the newly created object symbol. Additionally, the figure shows the object configuration as specified in the multiplicity invariant: an object of class Network, the newly created object of class DHCP_Client, and an object of class DHCP_Server. The arrows represent the links that have to exist for the multiplicity invariant to hold (the dashed arrow denotes that the link may not exist).

Figure 4.8: Extension of the sequence diagram of Figure 3.10 with the multiplicity invariants of all corresponding classes. The object configuration shows the objects that have to exist for the multiplicity invariant to hold.
4.3 Encapsulating Temporary Inconsistencies

The main idea of analysing class and statechart diagrams with respect to a particular sequence diagram is to attach constraints to the object lifelines at certain positions in between the message sending and reception points. These constraints represent properties specified in the analysed diagrams (data invariants, multiplicities, states, ...). Some locations in between such points may describe system states that do not conform with the constraints, e.g., because not all links for a newly created object already exist as specified by the multiplicity constraint, or a temporary value for an attribute does not conform to the data invariant. However, such points may be acceptable temporary inconsistencies.

Acceptable and non-acceptable inconsistencies are automatically distinguished by the algorithm\(^6\), in that only the states of the statemachine yield locations for the sequence diagram where constraints can be attached. This means, the statemachines states (i.e., the states of the statechart diagrams) designate object states where the constraints are checked. There may be intermediate states (action sequences in a single transition, pseudo states, ...) in the statechart diagram but these do not generate locations for constraints that have to be checked. The information specified in the statechart diagram is carried over to the sequence diagram by the algorithm and thereby demarcates zones of temporarily acceptable inconsistencies. The step character of the sequence diagram specification allows these temporary inconsistencies between two (or more) interactions.

Figure 4.9.(a) shows a sequence diagram with attached constraints for data invariants, states, guards, and pre- and postconditions. The value of the attribute \(p\) is temporarily not consistent with the data invariant. Since the object is then not in a (stable) state, the data invariant has not to be tested. Figure 4.9.(b) shows the corresponding statechart diagram that indicates the encapsulation of the temporary inconsistencies by the specification of the states and the transition. Figure 4.9.(c) shows the method specification of the operation that is invoked by the message.

---

\(^6\)The algorithm of analysing UML statechart diagrams and UML class diagrams is described in the previous sections, i.e., Section 4.1 (p. 44) and Section 4.2 (p. 50).

---

Figure 4.9: Encapsulation of temporarily acceptable inconsistencies in a sequence diagram.
4.4 Synthesising Context Conditions in Sequence Diagrams

During the analysis phase, all sequence diagrams of a system have been enriched with additional constraints containing properties of the class diagram and the statechart diagrams. The properties included from the statechart diagram are constraints for the possible states and for conditions guarding the possible transitions. The additional constraints from the class diagram are method specifications, and data and multiplicity invariants. These constraints form the context conditions of a sequence diagram.

Figure 4.10 shows a sequence diagram that contains as many additional constraints as possible. These context conditions are the data and multiplicity invariants (directly below the object symbols and at every occurrence of another additional constraint), state constraints (before and after each sending of a message and before the receiving of a message and after the completion of its transition), guards (at the beginning of a transition, i.e., usually before the reception of each message), and pre- or postconditions (before each message reception and after each operation execution, respectively). The transition triggered by message1() causes message2() to be sent. But this message either does not effect any action or these actions are not shown in the sequence diagram. Similarly, the event effecting the sending of message1() is not shown in this sequence diagram.

Figure 4.11 summarises the context conditions that have been added separately in the previous sections to the sequence diagram of Figure 3.10 (p. 40). The content of each attached constraint is described in detail below the sequence diagram.
Figure 4.11: Sequence diagram of Figure 3.10 containing all additional context conditions. These are described in detail separately below the sequence diagram.
The synthesis as described in the following deals with two complementary approaches: A horizontal synthesis and a vertical synthesis. The horizontal synthesis investigates the context conditions at one location, e.g., all constraints directly before the reception of a message. Section 4.4.1 describes this approach. To combine the constraints attached to the locations of an object’s lifeline, the vertical synthesis checks these constraints upwards and downwards for consistency (Section 4.4.2). Both approaches result in marking critical points, i.e., points where the added constraints do not match. Furthermore, pre and postconditions of the sequence diagram itself are generated that contain the necessary object configurations, states of these objects, and other attribute values as relevant for the sequence diagram (Section 4.4.3).

4.4.1 Horizontal Synthesis of the Context Conditions

The horizontal synthesis investigates the context conditions at one location in the sequence diagram, i.e., there are six locations in Figure 4.11 to be investigated. The investigation comprises two parts: On one hand, each attached constraint is checked for consistency (verifying that it is not contradictory in itself), and, on the other hand, the combination of the constraints is examined. Both — the constraints themselves and their combinations — are logical expressions. These evaluate to true (i.e., no inconsistency) or false (i.e., the model is inconsistent or incomplete). The resulting value may depend on the values of variables, e.g. the values of attributes. If, for each possible assignment of the variables, the logical expression is evaluated to false, it is contradictory and definitely points out an inconsistency.

An example for such a logical expression is \((x > y) \land (x < y)\). The result of other logical expressions, e.g., \((x < y) \lor (x > y)\), may depend entirely of the current assignment of the variables (in the example, it is false if \(x = y\) and otherwise true).

Checking each constraint itself for consistency can be summarised as follows:

1. Constraints for data invariants, multiplicity invariants, pre- and postconditions can be inconsistent. If so, the contradictions are inside the corresponding parts of the class diagram or the separate object constraints.

2. Constraints for guards can be inconsistent and then show a contradiction inside the statechart diagram.

3. Constraints for states cannot be inconsistent since logical expressions combined by an or operator cannot be contradictory.

The constraints at one location are combined by the and operator. To identify possible critical points at one location more precisely, constraints are combined and tested pairwise. The result of the complete investigation at one location is the combination of these single examinations. In the following, the possible combinations and their inconsistencies are summarised. For example constraints, consider Figure 4.11 (p. 59) or Figure 4.13 (p. 63).
4.4. SYNTHESISING CONTEXT CONDITIONS IN SEQUENCE DIAGRAMS

1. It is possible that the combination of constraints for data invariants and constraints for multiplicity invariants is contradictory. Then the specification of the class diagram (for the multiplicity invariants) and the separate object constraints (for the data invariants) are inconsistent. Usually, they are not contradictory because they describe distinct properties.

2. If constraints for data invariants or constraints for multiplicity invariants are combined with constraints for guards, their logical expressions can be contradictory. Since it is assumed in this report that the scenario described by the sequence diagram is a commitment and not just an example that does not need to hold, an inconsistency is detected. The inconsistency is between the specification of the class diagram and its separate object constraints and the specification of the statechart diagram.

3. The combination of constraints for data or multiplicity invariants and constraints containing a precondition for an operation can either show redundant information, no contradictions, or a contradiction of the constraints. Redundant information exists if the precondition is more general than the data or multiplicity invariant. If the data or multiplicity invariant is more general than the precondition or if the set of attributes used to declare the constraints do not overlap, no contradictions are possible. A contradictory specification can occur if the possible attribute values of the object specified by the data or multiplicity invariant and the attribute values needed to satisfy the precondition of the message are contradictory. A contradiction demonstrates an inconsistency inside the class diagram and its object constraints and the impossibility to invoke the operation.

4. The combination of constraints for data or multiplicity invariants and constraints containing the postcondition of an operation can be contradictory, too. A possible contradiction demonstrates an inconsistent class diagram.

5. If a constraint containing guards and a constraint containing a precondition lead to a contradiction, this demonstrates inconsistencies between the class diagram and the statechart diagram.
The combinations of constraints containing the possible states and other possible constraints as introduced in Section 4.1 and 4.2 are usually not contradictory (assuming that the other constraints do not contain expressions with states). They describe different properties of the object.

4.4.2 Vertical Synthesis of the Context Conditions

The vertical synthesis checks constraints at two succeeding locations on the same lifeline that have to be consistent because they should describe the same system or object state. Constraints at these locations have been attached by analysing two succeeding messages.\(^7\) To simplify the identification and location of inconsistencies, combinations of exactly two constraints, i.e., one constraint at location \(i\) and one at its immediate successor \(i'\), are used.

The constraints are combined by the \textit{and} operator and the logical expressions are then examined. The number of constraint combinations can be decreased because some of the combinations have already been investigated by horizontal synthesis (e.g., combinations with data or multiplicity invariants). As well, there are other combinations of state constraints with constraints containing pre- or postconditions or guards that can usually not lead to contradictions (assuming that these constraints do not contain expressions with states). Thus, only

\[\text{state} \quad \text{data inv}\]
\[\text{state} \quad \text{multiplicity inv}\]
\[\text{state} \quad \text{guard}\]
\[\text{state} \quad \text{pre}\]
\[\text{state} \quad \text{post}\]

the combinations shown in Figure 4.12 can expose contradictory specifications. These are: (a) state constraints between the reception of two messages, (b) state constraints between the reception of a message and the sending of another message, (c) constraints containing

\(^7\)In this case, two succeeding messages are not the event and the action of one transition but the events of two succeeding transitions. More precisely, a constraint directly before the reception or the sending of a message and another constraint directly afterwards are not succeeding constraints. Regarding the definitions of Section 5.2 in the following Chapter (p. 72), location \(l_1 = 0.7\) and location \(l_2 = 1.3\) are not succeeding locations, but location \(l_2\) and location \(l_3 = 1.7\).
post and preconditions between the reception of two messages, (d) constraints containing postcondition and guards between the reception of two messages.

If a combination leads to a contradiction either

- some additional behaviour has happened in between that is not shown in the sequence diagram (incompleteness of the diagram) or
- the sequence of messages is not consistent to the behaviour described in the statechart diagram of the object (inconsistency of the specification).

These critical points can be marked in the sequence diagram.

Figure 4.13 shows the sequence diagram of Figure 3.8 with the additional context conditions. The vertical synthesis has detected and marked a critical point. Table 4.5 describes the context conditions in more detail. The horizontal synthesis has not led to any contradictions.

Figure 4.13: Sequence diagram of Figure 3.8 with context conditions and marking of the detected critical point.
List of Context Conditions

\begin{itemize}
  \item \text{mi (c)} \quad \{ (\text{client.aServer->size} \leq 1) \text{ and } (\text{client.network->size}=1) \} \\
  \item \text{s (c1), s (c6)} \quad \{ \text{client.oclInState (Not configured)} \} \\
  \item \text{s (c2), s (c3)} \quad \{ \text{client.oclInState (Selecting)} \} \\
  \item \text{s (c4), s (c5)} \quad \{ \text{client.oclInState (Requesting)} \} \\
  \item \text{di (s)} \quad \{ (\text{server.ipBlock\_lower}>0) \text{ and } (\text{server.ipBlock\_upper}<255) \\
  \quad \text{and } (\text{server.ipBlock\_lower} \leq \text{server.ipBlock\_upper}) \\
  \quad \text{and } (\text{server.numberOfClients} \geq 0) \\
  \quad \text{and } (\text{server.allIPAddresses->forAll } (\text{ip1,ip2} | \\
  \quad \text{ip1.ipAddress} \neq \text{ip2.ipAddress} )) \} \\
  \item \text{mi (s)} \quad \{ (\text{server.allIPAddresses->size} = \text{ipBlock\_upper-ipBlock\_lower+1}) \\
  \quad \text{and } (\text{server.network->size} = 1) \} \\
  \item \text{s (s1), s (s2)} \quad \{ \text{server.oclInState (Has free IP addresses)} \} \\
  \item \text{s (s3), s (s4)} \quad \{ (\text{server.oclInState (Has free IP addresses)})t1 \\
  \quad \text{or } (\text{server.oclInState (Has no free IP address)})t2 \} \\
  \item \text{g (s1)} \quad \{ \text{server.hasFreeIP()} = \text{true} \} \\
  \item \text{g (s3)} \quad \{ (\text{server=server}) \text{ and } (\text{ip.getBinding = true})t1 \\
  \quad \text{or } ((\text{server=server}) \text{ and } (\text{ip.getBinding()} = \text{true}))t2 \} \\
  \item \text{pr (s1)} \quad \{ \text{server.numberOfIPAddresses > server.numberOfClients} \} \\
  \item \text{pr (s3)} \quad \{ (\text{server=server}) \\
  \quad \text{and } (\text{server.allIPAddresses->includes(ip)=true}) \} \\
  \item \text{po (s2)} \quad \{ \text{ip.getBinding()} = \text{false} \}
\end{itemize}

Table 4.5: Description of the context conditions in Figure 4.13.
4.4.3 Synthesis of Pre- and Postconditions for Sequence Diagrams

In the previous sections, the enriched sequence diagrams have been checked for consistency and completeness. The aim of this section is to describe how to generate a pre- and post-condition for each sequence diagram. These conditions of a sequence diagram contain the object configuration and the possible states and attribute values for these objects. If the precondition is maintained before the execution of the sequence diagram, it is assured that the postconditions holds afterwards.

**Precondition for a Sequence Diagram**

A precondition for a sequence diagram is generated using the context conditions attached to the lifelines of the objects before the reception or sending of the first message. The context conditions used for the precondition are constraints for data and multiplicity invariants, states, guards, and preconditions of operations to be invoked.

Figure 4.14 shows the sequence diagram of Figure 4.10 (p. 58) with the generated precondition. The precondition contains the context conditions of object1 (constraints containing the data and multiplicity invariants, state and a guard) and object2 (constraints containing the data and multiplicity invariant, state, guard and precondition).

![Sequence diagram with generated precondition](image)

Figure 4.14: Sequence diagram of Fig. 4.10 with the generated precondition for the sequence diagram.

In Section 4.1 (p. 44) it has been necessary that all transitions that can be triggered by a message or that cause the message are considered for the integration of the possible states and guards. Thus, the resulting constraints contains a set of states or guards combined by the
boolean operator or where each state or guard is labelled by its transition’s name. Depending on the sequence diagram, it might be possible to specify the precise state of the object or guard by deriving it from other state constraints attached to the same lifeline. Since it is possible that also preconditions contain such state and guard constraints, this also helps to define the precondition of the sequence diagram more precisely. In the following, an algorithm is described that works “upwards”, i.e., that starts at the bottom of the sequence diagram and helps to specify the precondition more precisely.

The derivation upwards is possible if the state constraints at two succeeding locations overlap in a way that the state constraint at location $i''$ is part of the state constraint at its preceding location $i'$. Since state constraints at two succeeding locations are supposed to be equivalent, this means that the objects must have been in the same state. Accordingly, the state constraint at location $i'$ can be reduced to the state that has been specified at location $i''$.

Thus, the state constraint at location $i'$ has been specified more precisely. The state contained in the constraint is the target state of the transition indicated by the label. The state constraint at the “preceding” location $i$ contains a state labelled with the same transition name. This state is the source state of the transition and the state constraint can be reduced to it. The optional guard constraint at the same location $i$ can also be reduced to the guard labelled by the same transition name.

![Figure 4.15: Derivation of state and guard constraints (upwards).](image)

Figure 4.15.(a) shows a part of a sequence diagram that contains combined states and guards (each labelled by its transition name). The context conditions are attached to three locations: before the reception of message1 () (location $i$), after the sending of message2 () (location $i'$), and the succeeding location (location $i''$). The state and guard constraints at location $i'$ and

---

*If state constraints at two succeeding locations are equivalent, they are not contradictory. Otherwise, they are contradictory and this usually indicates a critical point (see Section 4.4.2, p. 62).

*This is the previous location $i$ at the same lifeline that is separated from the correct location $i'$ by the reception or sending of messages. Using the definition in the following Chapter 5, the following locations are addresses (where $x \in \{1, 2, \ldots, |max_{m,i}|\}$): $i = x.3$, $i' = x.7$, and $i'' = (x + 1).3$. 
location $i$ can be decreased based on the state constraint at location $i''$. This derivation is shown in Figure 4.15(b). The resulting sequence diagram should be checked again for contradictions as described in Section 4.4.1 (horizontal synthesis) and Section 4.4.2 (vertical synthesis).

**Postcondition for a Sequence Diagram**

A postcondition for a sequence diagram is generated using the context conditions attached to the end of the lifelines, i.e., after the reception or the sending of the last message. Therefore, the following context conditions are relevant for a postcondition: constraints for data and multiplicity invariants, states, and postconditions of invoked operations.

Figure 4.16 shows a sequence diagram (see also Figure 4.10, p. 58) with the generated postcondition for the sequence diagram.

```
object 1 : Class1

postcondition for sequence diagram
```

Figure 4.16: Sequence diagram of Fig. 4.10 with the generated postcondition for the sequence diagram.

Similarly to the algorithm for “upwards derivation”, it is possible to apply it “downwards”, i.e., to derive more precise state constraints at the end of the sequence diagrams and even in the postcondition. The derivation is possible, if the state constraint attached to location $i$ is part of the state constraint at its succeeding location $i'$. According to the algorithm
for “upwards derivation”, the less precise state constraint (i.e., the one at location \( i' \)) can be reduced to the same state constraint as specified at location \( i \). This state is the source state of a transition marked by the label of the constraint. The guard constraint at the same location \( i' \) can be reduced to the guard of this transition, too. Furthermore, it is possible to reduce the state constraint at the “following” location \( i'' \) to the target state of this transition.

![Figure 4.17: Derivation of state and guard constraints (downwards).](image)

Figure 4.17: Derivation of state and guard constraints (downwards).

Figure 4.17.(a) shows a part of a sequence diagram that contains combined states and guards (at the locations \( i' \) and \( i'' \)) each labelled by its transition name. The state and guard constraint at these locations can be specified more precisely based on the state constraint at location \( i \). This “downwards derivation” is shown in Figure 4.17.(b). Since the attached context conditions have changed, the diagram should be checked again for contradictions as described in the previous sections (Section 4.4.1 and Section 4.4.2).
Chapter 5

Formalisation of Sequence Diagrams and their Extensions

UML sequence diagrams have their origin among others in Message Sequence Charts (MSCs) that have been standardised by the ITU ([Z.120]). In [DH98]¹, the formal semantics of MSCs and their abstract syntax have been extended by W. Damm and D. Harel to Live Sequence Charts (LSCs) in order to specify additional aspects, e.g., liveness², possible and necessary behaviour, branching, and iteration.

In the following sections, the abstract syntax of UML sequence diagrams is defined based on the formalism of LSCs (Section 5.1) and afterwards (Section 5.2) extended to also cover the extended sequence diagrams as discussed in the previous chapter (see Section 4.1 (p. 44) and Section 4.2 (p. 50)) in that additional locations are introduced where conditions can be attached.

5.1 Formalisation of Sequence Diagrams

A sequence diagram shows the interaction of objects arranged in time sequence. Let \( \text{inst}_m \) be a set of identifiers for the object instances that participate in the interaction sequence of a sequence diagram \( m \):

\[
\text{inst}_m = \{i_1, \ldots, i_n\}
\]

The environment of the (sub)system that is specified by the sequence diagram is represented by actors. The actors and their lifelines are not considered in the formalisation of the sequence diagram. Instead, messages without source or target are allowed, representing the communication with the environment. This conforms to the LSC formalisation and is in one-to-one correspondence with the UML definition.

The points on a lifeline of an object instance where messages can be sent or received are called locations. Accordingly, a finite number of abstract and discrete locations \( l \) on the object’s

¹The technical report is enlarged by the authors W. Damm and D. Harel from time to time (often without explicit versioning) and the author generally refers to the current version [DH00] from July 2000.

²This is the reason for calling them Live Sequence Charts.
lifeline is associated with each instance $i$. Let $\text{dom}_{m,i}$ be the set of locations associated with a single instance $i$ and $\text{dom}_m$ be the collection of all locations of a sequence diagram $m$:

$$
\text{dom}_{m,i} = \{0, \ldots, l_{\max_{m,i}}\}
$$

$$
\text{dom}_m = \{\langle i, l \rangle : i \in \text{inst}_m \land l \in \text{dom}_{m,i}\}
$$

At each location a single or set of messages can be sent (e.g., to another object or to a multiobject) or a message can be received (e.g., from the same or another object). Also, it is possible to receive messages from or send messages to the environment, i.e., to actors participating in the sequence diagram. These messages have no source or target location. In this report, only those messages are considered that invoke operations in the receiver object. Therefore, a message is labelled by a message or operation name and the corresponding parameter list. In order to distinguish two messages invoking the same operation between the same sender and receiver object, each message is given a unique (sequence) number. This message label is called $\text{message}\_id$ and is an element of the set $\text{MessageIds}$ that specifies all possible message labels. Since the types of the parameters must conform to the operation specification given in the class diagram $\text{MessageIds}$ is only a subset of all possible combinations.

$$
\text{MessageIds} \subseteq \text{MessageNames} \times \text{ParameterLists} \times \mathbb{N}
$$

If the operation invoked by a message yields a return value and this is sent by an explicit return message, the message label $\text{return-}\text{message}\_id$ is used, which is an element of the set of $\text{ReturnMessageIds}$:

$$
\text{ReturnMessageIds} \subseteq \text{MessageIds} \times \text{ReturnValue}
$$

Similar to the set $\text{MessageIds}$, the set $\text{ReturnMessageIds}$ contains only those combinations of return messages that are syntactically type correct as specified by the class diagram. Each message in a sequence diagram relates two points on the object lifelines: the point where the message is sent and the point where the message is received. The sending or receiving of a message can also be guarded by a condition, but this is not referred to in this report. The following notation is used for the sending or receiving of a message in the abstract syntax: $!$ for sending and $?$ for receiving. The set $\text{Messages}$ comprises all messages of a sequence diagram:

$$
\text{Messages} = \text{MessageIds} \times \{!, ?\} \cup \text{ReturnMessageIds} \times \{!, ?\}
$$

Each location in a sequence diagram can be labelled with an arbitrary number of messages. Usually it is exactly one. For a sequence diagram $m$, the association between locations and messages is given by a labelling function $\text{msg}_m$:

$$
\text{msg}_m : \text{dom}_m \rightarrow \mathcal{P}(\text{Messages})
$$

The functions $\text{source}$ and $\text{target}$ define the explicit source and target of a message. So messages sent to or received from the environment $E$ are identified.

$$
\text{source}, \text{target} : \text{Messages} \rightarrow \text{inst}_m \cup \{E\}
$$

where $\forall l \in \text{dom}_m$

$$
\text{source} : \mu \rightarrow \begin{cases} 
  i & : \text{if } \mu = \langle m\_id, ! \rangle \in \text{msg}_m((i, l)) \\
  E & : \text{else}
\end{cases}
$$

$$
\text{target} : \mu \rightarrow \begin{cases} 
  i & : \text{if } \mu = \langle m\_id, ? \rangle \in \text{msg}_m((i, l)) \\
  E & : \text{else}
\end{cases}
$$
The locations in the sequence diagram are partially ordered. The relation \( \leq \subseteq \text{dom}_m \times \text{dom}_m \) is defined by

1. \( \langle i, l \rangle \leq \langle i, l+1 \rangle \)
2. \( \langle m_{id}, ! \rangle \in \text{msg}_m(\langle i, l \rangle) \land \langle m_{id}, ? \rangle \in \text{msg}_m(\langle i', l' \rangle) \Rightarrow \langle i, l \rangle \leq \langle i', l' \rangle \)
3. Transitive closure induced by 1 and 2.

The labelling function \( \text{msg}_m \) has to maintain the following constraints for consistency:

1. Each message (i.e., each \( m_{id} \in \text{MessageIds} \)) is unique in the sequence diagram.
   \[ \forall \langle i, l \rangle, \langle i', l' \rangle \in \text{dom}_m, x \in \{!, ?\} : \langle m_{id}, x \rangle \in \text{msg}_m(\langle i, l \rangle) \land \langle m_{id}, x \rangle \in \text{msg}_m(\langle i', l' \rangle) \Rightarrow \langle i, l \rangle = \langle i', l' \rangle \]
2. It is not possible to receive more than one message at each location or even to receive a message and to send another message at the same location.
   \[ \forall \langle i, l \rangle \in \text{dom}_m, \langle m_{id}, x \rangle \in \text{msg}_m(\langle i, l \rangle) : \text{card}(\text{msg}_m(\langle i, l \rangle)) > 1 \Rightarrow x = ! \]
3. Each return message by an instance \( i \) sends the return value \( v \) of a previous message to the sender of this message.
   \[ \forall \langle i, l \rangle \in \text{dom}_m, v \in \text{ReturnValue} : \langle \langle m_{id}, v \rangle, ! \rangle \in \text{msg}_m(\langle i, l \rangle) \Rightarrow \exists l' \in \text{dom}_{m,i} : \langle m_{id}, ? \rangle \in \text{msg}_m(\langle i, l' \rangle) \land l' \leq l \land \text{target}(\langle m_{id}, ? \rangle) = \text{source}(\langle \langle m_{id}, v \rangle, ! \rangle) \]
4. No message cannot have more than one return message.
   \[ \forall \langle m_{id}, v \rangle, \langle m_{id}', v' \rangle \in \text{ReturnMessageIds} : \exists \langle i, l \rangle, \langle i', l' \rangle \in \text{dom}_m : \langle m_{id}, v \rangle \in \text{msg}_m(\langle i, l \rangle) \land \langle m_{id}', v' \rangle \in \text{msg}_m(\langle i', l' \rangle) \land m_{id} = m_{id}' \Rightarrow \langle i, l \rangle = \langle i', l' \rangle \land v = v' \]

Summarising, the abstract syntax of a sequence diagram \( m \) is defined by the following triple:

\[ m = (\text{inst}_m, \text{dom}_m, \text{msg}_m) \]

Figure 5.1 shows the sequence diagram of Figure 3.10 (p. 40) with the locations marked by ◦. Messages received from the environment are messages without a source location. Messages sent to the environment are without a target location.

According to the previous definitions, the following results for the sequence diagram \( m \) of Figure 5.1:

\[ m = (\text{inst}_m, \text{dom}_m, \text{msg}_m) \]
\[ \text{inst}_m = \{ \text{client}, \text{server} \} \]
\[ \text{dom}_m = \{ \langle \text{client}, 0 \rangle, \langle \text{client}, 1 \rangle, \langle \text{client}, 2 \rangle, \langle \text{client}, 3 \rangle, \langle \text{server}, 0 \rangle, \langle \text{server}, 1 \rangle \} \]
5.2 Formalisation of Enriched Sequence Diagrams

In the previous chapter — more precisely in Section 4.1 and Section 4.2 — the sequence diagrams have been enriched by context conditions. These are constraints containing specific properties derived from the statecharts and class diagrams: states, guards, pre- and postconditions, and data- and multiplicity invariants. These constraints have been attached to the object’s lifelines directly before or directly after the point where the object receives or sends a messages.

Hence, for example, if an object receives a message at location $l = 1$ and this is the only message that the object sends or receives, new locations $l' = 0.7$ and $l'' = 1.3$ are introduced and the constraints are added directly before (i.e., at location $l'$) and directly afterwards (i.e., at location $l''$).

Let $\text{dom}_{m,i} = \{0, 1, 2, \ldots, l_{max_{m,i}}\}$ be the set of locations associated with an instance $i$ of the sequence diagram $m$ (see Section 5.1, p. 70). For enriched sequence diagrams, this yields two sets:

The set $\text{dom}_{m,i}^{\text{msg}}$ that collects all locations associated with an instance $i$ where messages can be sent or received, and the set $\text{dom}_{m,i}^{\text{cons}}$ that contains the additional locations associated with an instance $i$ to attach the constraints.

\[
\begin{align*}
\text{dom}_{m,i}^{\text{msg}} &= \{1, 2, \ldots, l_{max_{m,i}}\} \\
\text{dom}_{m,i}^{\text{cons}} &= \{0, 0.7, 1.3, \ldots, (l_{max_{m,i}} - 1) \cdot 7, (l_{max_{m,i}}) \cdot 3, (l_{max_{m,i}} + 1)\}
\end{align*}
\]
The sets \( \text{dom}_m, \text{dom}^{\text{msg}}_m, \text{and} \text{dom}^{\text{cons}}_m \) that are collections of locations of a sequence diagram \( m \) are defined as following:

\[
\begin{align*}
\text{dom}_m &= \{ (i, l) : i \in \text{inst}_m \land l \in \text{dom}_{m,i} \} \\
\text{dom}^{\text{msg}}_m &= \{ (i, l) : i \in \text{inst}_m \land l \in \text{dom}^{\text{msg}}_{m,i} \} \\
\text{dom}^{\text{cons}}_m &= \{ (i, l) : i \in \text{inst}_m \land l \in \text{dom}^{\text{cons}}_{m,i} \}
\end{align*}
\]

The constraints attached to the sequence diagram are OCL expressions (i.e., boolean expressions) and collected in the set \( \text{Constraints} \). The unique pre- and postconditions of a sequence diagram \( m \) (see Section 4.4.3, p. 65) are distinguished by the predicates \( \text{pre}_m \) and \( \text{post}_m \).

As described above, the integer-numbered locations (i.e., locations in \( \text{dom}^{\text{msg}}_{m,i} \)) are labelled by messages using the function \( \text{msg}_m \). The fraction-numbered locations (i.e., locations in \( \text{dom}^{\text{cons}}_{m,i} \)) are labelled by the constraints. For a sequence diagram \( m \), the association between the locations and the messages is given by a labelling function \( \text{msg}_m \) and the association between the new locations and the constraints is given by a labelling function \( \text{cons}_m \):

\[
\begin{align*}
\text{msg}_m &: \text{dom}^{\text{msg}}_m \rightarrow \mathcal{P}(\text{Messages}) \\
\text{cons}_m &: \text{dom}^{\text{cons}}_m \rightarrow \mathcal{P}(\text{Constraints})
\end{align*}
\]

There are also additional consistency rules concerning the constraints:

1. The pre- and postconditions of a sequence diagram are associated with the first and the last additional location.

\[
\begin{align*}
\forall (i, l) \in \text{dom}^{\text{cons}}_m, c \in \text{cons}_m((i, l)) : & \quad \text{pre}_m(c) \Rightarrow l = 0 \\
\forall (i, l) \in \text{dom}^{\text{cons}}_m, c \in \text{cons}_m((i, l)) : & \quad \text{post}_m(c) \Rightarrow l = (l_{\text{max}}_{m,i} + 1)
\end{align*}
\]

2. For each sequence diagram, there is exactly one pre- and postcondition marked by the predicates \( \text{pre}_m \) and \( \text{post}_m \) respectively, and these are associated with each instance at location 0 and \( l_{\text{max}}_{m,i} \), respectively:

\[
\forall i \in \text{inst}_m : \quad \text{pre}_m(\text{cons}_m((i, 0))) \land \text{post}_m(\text{cons}_m((i, l_{\text{max}}_{m,i})))
\]

The abstract syntax of an enriched sequence diagram \( m \) is accordingly defined by:

\[ m = (\text{inst}_m, \text{dom}_m, \text{msg}_m, \text{cons}_m) \]

Figure 5.2 shows the sequence diagram of Figure 5.1 with the additional locations marked by \( \bullet \). The additional locations can be labelled with the constraints, i.e., either the pre- or postcondition, or other conditions. Those locations that can have attached constraints have been discussed in Section 4.1 (p. 44) and Section 4.2 (p. 50). The boolean expressions of the conditions are given in Figure 4.11 (p. 59).
The abstract syntax for the sequence diagram \( m \) of Figure 5.2 is given as:

\[
m = (\text{inst}_m, \text{dom}_m, \text{msg}_m, \text{cons}_m)
\]

\[
\text{inst}_m = \{\text{client}, \text{server}\}
\]

\[
\text{dom}_m = \{(\text{client}, 0), (\text{client}, 1), (\text{client}, 2), (\text{client}, 3), (\text{server}, 0), (\text{server}, 1)\}
\]

This yields:

\[
\text{dom}_m^{\text{msg}} = \{(\text{client}, 1), (\text{client}, 2), (\text{client}, 3), (\text{server}, 1)\}
\]

\[
\text{dom}_m^{\text{cons}} = \{(\text{client}, 0), (\text{client}, 0.7), (\text{client}, 1.3), (\text{client}, 1.7), (\text{client}, 2.3), (\text{client}, 2.7), (\text{client}, 3.3), (\text{client}, 4), (\text{server}, 0), (\text{server}, 0.7), (\text{server}, 1.3), (\text{server}, 2)\}
\]

\[
\text{msg}_m((\text{client}, 1)) = \{((\text{disconnect}, ()), 0, ?)\}
\]

\[
\text{msg}_m((\text{client}, 2)) = \{((\text{release}, (\text{myIPAddress})), 1, !)\}
\]

\[
\text{msg}_m((\text{client}, 3)) = \{((((\text{disconnect}, ()), 0, "\text{Connection closed}"), 1, !)\}
\]

\[
\text{msg}_m((\text{server}, 1)) = \{((\text{release}, (\text{myIPAddress})), 1, ?)\}
\]

\[
\text{cons}_m((\text{client}, 0)) = \{\text{precondition}\}
\]

\[
\text{cons}_m((\text{client}, 0.7)) = \{\text{condition1}\}
\]

\[
\text{cons}_m((\text{client}, 1.3)) = \emptyset
\]

\[
\text{cons}_m((\text{client}, 1.7)) = \emptyset
\]

\[
\text{cons}_m((\text{client}, 2.3)) = \emptyset
\]

\[
\text{cons}_m((\text{client}, 2.7)) = \emptyset
\]

\[
\text{cons}_m((\text{client}, 3.3)) = \{\text{condition2}\}
\]

\[
\text{cons}_m((\text{client}, 4)) = \{\text{postcondition}\}
\]

\[
\text{cons}_m((\text{server}, 0)) = \{\text{precondition}\}
\]

\[
\text{cons}_m((\text{server}, 0.7)) = \{\text{condition3}\}
\]

\[
\text{cons}_m((\text{server}, 1.3)) = \{\text{condition4}\}
\]

\[
\text{cons}_m((\text{server}, 2)) = \{\text{postcondition}\}
\]
5.2. FORMALISATION OF ENRICHED SEQUENCE DIAGRAMS

The formalisation described in this chapter provides the means to formally define the locations where messages are sent or received and the locations of context conditions (i.e., combinations of constraints attached to one location). The resulting sets are $\text{dom}_{\text{msg}}^m$ and $\text{dom}_{\text{cons}}^m$ and are necessary for the following investigation of the dependencies between the enriched sequence diagrams of a model (see especially Section 6.1). These sets are two contributes of the abstract syntax for enriched sequence diagrams that is defined in this chapter. The given definitions are sufficient for the needs of this report and provide the foundation for a formal description of the integration algorithm developed. However, formally describing the semantics of the algorithm is outside the scope of this report.
Chapter 6

Dependencies between Sequence Diagrams and their Representation in UML Use Case Diagrams

In Chapter 4, an approach is described that defines how information specified by statechart and class diagrams can be integrated in one sequence diagram. The resulting sequence diagram has been enriched or extended by constraints attached to the lifelines of the participating objects. These context conditions have been combined in the synthesis phase to identify critical points (i.e., points where the attached constraints are contradictory). The synthesis phase has resulted in the generation of pre- and postconditions for each sequence diagram.

In Figure 6.1, an abstraction of the sequence diagrams as defined in Section 3.2 and their pre- and postconditions as described in Section 4.4.3 (p. 65) is given. Additionally, the critical point is marked that has been detected by synthesising the extended sequence diagrams (see Section 4.4, p. 58, and Figure 4.13, p. 63). Figure 6.1 shows all sequence diagrams of the system’s model — subsequently referred to as the *collection of sequence diagrams*.

In this chapter, the collection of enriched sequence diagrams is investigated to identify the dependencies between the sequence diagrams. In addition, it is determined whether there exists a representation of these dependencies as relationships between use cases. The refinement of the use case diagram to contain the dependencies of its scenarios (i.e., the sequence diagrams realising the use cases) provides the foundation for the next iteration step in the specification phase.1

The investigation of the collection of enriched sequence diagrams is based on the context conditions such as constraints and pre- and postconditions of the diagrams as generated in Chapter 4. Furthermore, it uses the formalisation of the abstract syntax of the (enriched) sequence diagrams as defined in Chapter 5. The investigation comprises the following: Identification of possible “inclusions” of sequence diagrams at critical points, definition of a required chronological order, and refinement of use case diagrams by refining some of the use cases.

---

1Following an iterative and incremental object-oriented development process, modelling use cases is usually one of the first activities in the specification phase. Afterwards, the scenarios realising these use cases are defined by representing them as sequence diagrams. Every iteration step improves and extends the results of the last iteration. The model as described in Section 3.2 shows the first — or one of the first — iterations in the specification phase.
Figure 6.1: Collection of extended sequence diagrams (i.e., sequence diagrams with pre- and postconditions and additional marks for critical points). Abstraction of the sequence diagrams in Section 3.2: (a) Fig. 3.11.a on p. 40, (b) Fig. 3.11.b on p. 40, (c) Fig. 3.7 on p. 38, (d) Fig. 3.8 on p. 39, (e) Fig. 3.9 on p. 39, (f) Fig. 3.10 on p. 40.
6.1 Refinement of Use Cases

A use case diagram defines the use cases that specify the functionality of the system (see for example Figure 3.5, p. 33). Scenarios define the realisation of these use cases and are represented usually as sequence diagrams. If a use case is realised by more than one scenario (i.e., by more than one sequence diagram), it can be further refined in the next iteration of the specification phase as described in the following.

The first step in this refinement process is to specify a new temporary use case for each scenario. Figure 6.2 describes an abstract representation of this refinement. The “base” use case Connect to network is refined by a temporary use case for each of its scenarios.

For each use case a sequence of actions is defined. As mentioned in Section 2.1, there is no detailed description in the UML standard how this sequence of actions is modelled. It is suggested in this report to represent the sequence of actions by the sequence of messages in a sequence diagram. Using the definitions in the previous Chapter 5, it is also possible to formally specify this sequence of messages.

The scenarios of a use case and thus also their temporary use cases can be investigated for their dependencies. Since each use case describes one of the system’s functions, it can be specified which temporary use case satisfies the goal of the base use case and which one does not. Usually, there is only one success scenario that satisfies the goal and, thus, its sequence of steps defines the sequence of actions in the base use case. For the other scenarios it is investigated if they have some common parts with the base use case. The non common parts are represented in extension use cases derived from their temporary use cases. An extend relationship (as described in Section 2.1, p. 13) between the extension use case and the base use case then includes the variants of the sequence of actions. To define the extension, additional specifications are necessary: the extension points in the base use case, and extension

\[\text{Figure 6.2: Refinement of a use case with a new use case for each scenario.}\]
conditions (i.e., guards that must be fulfilled for the extension to be invoked). Since the UML standard does not specify the way this is described, this report uses a representation based on the ideas described in [CD00] (p. 44).

The notation used in [CD00] defines the use case with a textual description for the sequence of steps (i.e., the sequence of actions). This describes the main success scenario defined in the base use case. The use case steps act as UML extension points. They form the anchor points from which extend relationships to an extension use case may be defined. To assign a unique identification in the sequence of the use case, the locations defined in Chapter 5 are used. For the specification of the steps, only those locations \( l \) at the lifelines of an object instance \( i \) are used where a message \( m \) is sent, i.e., \( \langle m_id, l \rangle \in msg_m((i, l)) \). The action at each such step is then the sending of the corresponding message.

The following textual notation can be given for the base use case Connect to network (see the sequence diagram in Figure 3.7, p. 38):

**Main Success Use Case Connect to network**

1. \( \langle \text{client}, 2 \rangle \) \( msg_m((\text{client}, 2)) = \{\langle \text{discover}, (\text{client}), 1, ! \rangle \} \)
2. \( \langle \text{server}, 1 \rangle \) \( msg_m((\text{server}, 1)) = \{\langle \text{getFreeIP}, ((), 2), ! \rangle \} \)
3. \( \langle \text{server}, 3 \rangle \) \( msg_m((\text{server}, 3)) = \{\langle \text{offer}, (\text{ip}, \text{server}), 3), ! \rangle \} \)
4. \( \langle \text{client}, 4 \rangle \) \( msg_m((\text{client}, 4)) = \{\langle \text{request}, (\text{ip}, \text{client}, \text{server}), 4), ! \rangle \} \)
5. \( \langle \text{server}, 4 \rangle \) \( msg_m((\text{server}, 4)) = \{\langle \text{ack}, ((), 5), ! \rangle \} \)
6. \( \langle \text{client}, 6 \rangle \) \( msg_m((\text{client}, 6)) = \{\langle \text{connect}, ((), 0), \text{"Successfully connected."}, ! \rangle \} \)

The scenarios Unsuccessful connection to network (1) can be given using as well the formalism of Chapter 5. The textual notation for an extension use case contains a reference to a location in the base use case (described above), the condition that guards the extension, and the alternative sequence. In addition, it needs to be specified what happens after the extension use case. Based on the definition in [CD00], the following four alternatives exist: Continue (to indicate that the base use case performs its next step without skipping a step), Fail (to indicate the use case is terminated with the goal unsatisfied), Stop (to indicate that the use case is terminated with the goal satisfied), and Resume N (to indicate that N is the next step to be performed in the main success scenario). The condition guarding the extension at location \( l \) is the context condition at the preceeding location \( l' \), i.e., \( cons_m((i, l')) \) yields the constraints attached to location \( l' = (l - 1).7 \).

**Extension Use Case Unsuccessful connection to network (1)**

5. \( \langle \text{server}, 4 \rangle \) if \( cons_m((\text{server}, 3.7)) \)
   a. \( msg_m((\text{server}, 4)) = \{\langle \text{nak}, ((), 5), ! \rangle \} \)
   b. \( msg_m((\text{client}, 6)) = \{\langle \text{connect}, ((), 0), \text{"Not connected."}, ! \rangle \} \)
   c. Fail

These textual notations have to be regarded in addition to the refined use case diagram. Figure 6.3 shows the refinement of the use case diagram (see Figure 3.5, p. 33) based on the refinement of use case Connect to network. The alternative sequences of steps are specified in the extension use cases Unsuccessful connection to network (1) and Unsuccessful connection to network (2).
Figure 6.3: Representation of the use case refinement

Figure 6.4: Iterative refinement of use case diagrams.
CHAPTER 6. DEPENDENCIES BETWEEN SEQUENCE DIAGRAMS

To represent the iterative refinement process in terms of refinement relationships between use case diagrams, each iteration of a use case diagram is represented as a package and related to the respective diagram at the previous iteration step by a refinement dependency relation ([UML1.3], p. 3-82). A refinement dependency indicates a semantic relationship between two model elements with a mapping between them that does not necessarily have to be complete. Figure 6.4 shows the refinement between the use case diagrams as a dashed arrow between the packages that is annotated by the keyword <<refine>>. This refinement is based on the refinement of use case Connect to network.

6.2 “Sequence Diagram Inclusion”

In the synthesis phase (see especially Section 4.4.2, p. 62), the investigation of the constraints may result in the identification of contradictions (critical points). Two conclusions can be drawn from critical points identified: Either some additional behaviour has happened in between that is not shown in the sequence diagram, or the specification of the system is inconsistent. Considering a collection of sequence diagrams (enriched with their constraints and their pre- and postconditions), these critical points can be examined further.

A critical point in a sequence diagram seq1 is identified by a contradiction between two constraints cons1 and cons2 which are attached to two succeeding locations. To investigate the occurrence of additional behaviour, the collection of sequence diagrams can be investigated to identify sequence diagrams that can be included between these locations. To do this, the combination of the constraints cons1 and cons2 and the pre- and postconditions of the other sequence diagrams in the collection are considered. If there exists a sequence diagram such that the respective constraints are compatible, the behaviour described in this sequence diagram could have occurred in between. Thus, it is possible to include this sequence diagram at the identified location.

Figure 6.5: Inclusion of sequence diagram Successful connection to the network at the critical point of sequence diagram Unsuccessful connection to the network.
Figure 6.5 shows the “inclusion” of sequence diagram **Successful connection to the network** at the critical point of sequence diagram **Unsuccessful connection to the network** (1). A situation for this example scenario can be as follows (see also Figure 3.4, p. 32): A server \( s \) offers an available network address (i.e., an unbound IP address) to a requesting client \( c_1 \). But when the client sends the request message to the server the address is already bound and the server replies by sending a negative acknowledgement. This situation can be explained by the successful request of another client \( c_2 \) in the meantime, i.e., the server has offered the same (still available) network address and replied with a positive acknowledgement to the request message. This binding of the IP address to the second client \( c_2 \) explains the sending of a negative acknowledgement to the first client \( c_1 \).

To identify that a sequence diagram \( \text{seq}_2 \) can be included, the contradictory constraints \( \text{cons}_1 \) and \( \text{cons}_2 \) (of sequence diagram \( \text{seq}_1 \)) are combined pairwise with the pre- and postconditions of the other diagrams in the collection, respectively, and then checked for contradiction or conformance. This approach is similar to the vertical synthesis (Section 4.4.2). In order to be compatible to each other, the object configurations of two constraints (e.g., \( \text{cons}_1 \) and the precondition, \( \text{cons}_2 \) and the postcondition) have to describe the same subset of objects and the states of all these objects have to correspond. If so, the behaviour of sequence diagram \( \text{seq}_2 \) could have been included in sequence diagram \( \text{seq}_1 \). Depending on the pre- and postconditions of the sequence diagrams in the collection, it is possible that more than one sequence diagram could be included.

Figure 6.6 shows the inclusion sketched in Figure 6.5 in more detail. The inclusion is possible because the combination of constraint \( \text{cons}_1 \) and constraint \( \text{cons}_2 \) has led to a contradiction, and these contradictory constraints conform to the pre- and postcondition of sequence diagram **Successful connection to the network**, respectively.

**Representation of the “Sequence Diagram Inclusion”**

How this “sequence diagram inclusion” is represented as relations between the corresponding use cases depends on several factors, including (but not limited to):

1. whether the steps or locations are made explicit in the use cases, i.e., use cases with variants have been refined as described in the previous Section 6.1 (p. 79).

2. whether the use case instances that correspond to the sequence diagrams\(^3\) are instances of the same use case or of different use cases.

3. whether the use case instance that corresponds to the “included sequence diagram” can only be initiated by an actor or whether it can also be initiated by an inclusion.

Provided that the use cases have explicit locations, the representation of possible inclusions as **include** relationships (see Section 2.1, p. 13) is possible if the concerned use case instances are instances of different use cases and if the use case corresponding to the “included sequence

\(^3\)The relationship between a use case and a sequence diagram is defined by an invisible hyperlink ([UML1.3], p. 3-91).
Figure 6.6: Sequence diagram inclusion.
diagram” can be initiated by an inclusion. Otherwise, it is not possible to represent the “inclusion” as an include relation between the concerned use cases.\(^4\)

In case that a representation at the use case level is not possible, it is suggested to represent the inclusion of use case instances of the same use case as a relation at the use case instance level, i.e., as a relation without a corresponding relation at the use case level. This kind of include relation is depicted by a relation between the use case instance that needs the included behaviour and the included use case instance.

If a use case instance is “included” that can only be initiated by an actor, the “including” use case instance depends on the “parallel” existence of the other use case instance. It is suggested to represent this relationship by a newly defined <<assume>> relation between the use case instances.

Considering the example, the “sequence diagram inclusion” can only be represented as an assume relation because the use case instances are instances of the same base use case Connect to network and this base use case can only be initiated by an actor User. Figure 6.7 shows the use case instance level of the example sequence diagram inclusion.\(^5\) The different instances of actors and the participating objects are depicted, too.

---

\(^4\)It is not possible to have cyclic dependencies, i.e., the inclusion of the same use case (see Section 2.1, p. 13), and furthermore, it is not possible to initiate a use case that can only be initiated by an actor. Usually, it makes semantically no sense to initiate such a use case.

\(^5\)It is suggested by the author to represent the use case instances as use cases with underlined use case names and an additional use case instance name. This representation is similar to the representation of class and its object instances.
6.3 Temporal Order of Sequence Diagrams

A collection of extended sequence diagrams as shown in Figure 6.1 (p. 78) can also be analysed with respect to their required causal order. This investigation is based on the pre- and postconditions as generated in Section 4.4.3 (p. 65) and results in deriving a temporal order of the diagrams. It is obtained from the causal relationships detected in the analysis and means that some previous behaviour defined in one sequence diagram is required for the application of another one. Also, it is specified if the application of some sequence diagram is independent.

The application of a sequence diagram $s_2$ may depend on the previous application of another one ($s_1$) if the precondition of $s_2$ is compatible with the postcondition of $s_1$ (i.e., these constraints specify compatible object configurations and compatible states of the objects). Otherwise, the chronological sequence of their application is independent. Sequence diagrams with empty preconditions (i.e., all objects are created during its application) show that no previous interactions are required.\(^6\)

\[\text{pre condition (1)} \quad \text{pre condition (2)} \quad \text{pre condition (3)} \quad \text{pre condition (4)} \quad \text{pre condition (5)} \quad \text{pre condition (6)}\]

\[\text{Initiate new DHCP server} \quad \text{Initiate new DHCP client} \quad \text{Successful connection to the network} \quad \text{Unsuccessful connection to the network (1)} \quad \text{Unsuccessful connection to the network (2)} \quad \text{Disconnection from the network}\]

\[\text{post condition (1)} \quad \text{post condition (2)} \quad \text{post condition (3)} \quad \text{post condition (4)} \quad \text{post condition (5)} \quad \text{post condition (6)}\]

Figure 6.8: Temporal order of a collection of sequence diagrams.

Figure 6.8 shows the collection of sequence diagrams specified in Chapter 3 as an abstract representation (see also Figure 6.1, p. 78). The arrows represent the derived temporal order.

\(^6\)Nevertheless, if the multiplicities of certain objects limit the number of objects, it may not be possible to apply the sequence diagram without creating too many objects in the object configuration of the system.
6.3. TEMPORAL ORDER OF SEQUENCE DIAGRAMS

The preconditions of the sequence diagrams Initiate new DHCP server and Initiate new DHCP client are empty.

It is possible to represent this chronological dependency as a relation between use cases. Thus, a new relationship is defined as a specialisation of the metamodel class Relationship (contained in the Core package). The new relationship is further denoted with the keyword <<require>>.

Figure 6.9 shows the refined use case diagram (see Figure 6.3) with the requirement relationships as detected in this investigation.

![Figure 6.9: Representation of the temporal order](image-url)
Chapter 7

Conclusion

This report has developed an algorithm to check a UML-based model for consistency and completeness. These checks are necessary because, in object-oriented modelling with UML, different views of a system are represented by different diagram types and this bears the risk that the overall specification is inconsistent or incomplete. The algorithm describes how to integrate distributed system information in sequence diagrams and how to check these sequence diagrams based upon the enrichments.

The algorithm comprises an analysis phase, a synthesis phase and a refinement phase. The analysis phase of the algorithm determines how to retrieve necessary information from state-chart and class diagrams and how to add the information to sequence diagrams as constraints attached to certain points on the lifelines of the objects — usually before and after the sending or receiving of a message. The synthesis phase examines the context conditions (i.e., the combination of constraints) at each such location and at two succeeding locations on the same lifeline to detect inconsistencies in or between and incomplete specifications of certain diagrams. Based on the context conditions for each (enriched) sequence diagram, pre- and postconditions are generated that represent the object configurations and the states of the objects before and after the scenario described by the sequence diagram. The refinement phase investigates the dependencies of a collection of enriched sequence diagrams. Based upon these results, the dependencies are represented in use case diagrams as relationships between use cases. In addition, the report describes a formalisation of the abstract syntax of sequence diagrams and their enrichments. Thus, the locations of the attached constraints are made precise.

The algorithm has been applied to an example model to illustrate its functioning and usage. The example model describes a simple client-server system based on DHCP. Although the horizontal synthesis (i.e., the investigation of the context conditions at one location) has not detected any inconsistencies and thus it seems that the model was “perfect” from the beginning, there were several small inconsistencies in previous versions of the diagrams that have been detected and fixed by the author while devising this part of the algorithm.

The algorithm as described in this report considers only a part of the UML diagrams (i.e., the ones that are typically used in the analysis phase) as well as only the most relevant part of their model elements. Nevertheless, it should be possible to enhance the algorithm to also cover other diagram types such as UML collaboration diagrams and UML activity diagrams, since these diagrams contain mostly similar information or even rely on the same
information. To also address other model elements for the currently considered diagram types than those used in this report, it is necessary that the respective information can be represented as constraints. More precisely, they have to contain boolean expressions that can be evaluated during the synthesis phase. Furthermore, an investigation has to check the relations of these model elements to the others and also has to yield the possible locations to attach the constraints (e.g., before each sending of a message).

It is advisable to integrate the described algorithm for semantic analysis of UML diagrams into the specification phase of the software development process to reduce the risks and expenses for debugging an implementation that is based on an inconsistent or incomplete system specification. For a consequent and easy support of the development process by means of the approach presented here, appropriate tools need to be available. Therefore, directions for future work include the development of a UML modelling tool that is able to perform the automatic integration of the different views and their consistency checking as described in this report. Since identifying possible contradictions by examining the logical expressions of the constraints is a main part of the checking algorithm, the integration of a theorem prover is envisaged.

Finally, tool-supported software development necessitates precise definitions of the concepts. The definition for a formalisation of enriched sequence diagrams includes the definition of the abstract syntax, but should be expanded to also specify the semantics of the algorithm.

Summarising, the approach described in this report yields a more precise and refined specification of the system’s model, i.e., usually the resulting diagrams are more specific. Moreover, it allows to emphasise the role of the specification phase in the development process by reducing the effort that is necessary in the following development phases, i.e., in the implementation and verification/validation phase.
Appendix A

UML Metamodel

The architecture of UML is based on a four-layer metamodel structure which consists of the following layers: user objects, model, metamodel, and meta-metamodel. The UML Semantics ([UML1.3], p. 2-1 - 2-190) is primarily concerned with the metamodel layer. The UML metamodel is a logical model that is decomposed into three packages: The Foundation package, the Behavioral Elements package, and the Model Management package. These packages consist of several subpackages.

The metamodel is described using a combination of graphical notation, natural language and formal language. The graphical notation is used for the so-called abstract syntax view that is provided as a model. This model is described in a subset of UML that consists of a UML class diagram. Natural language is also used in the abstract syntax, as well as in the so-called semantics view. A formal language – the Object Constraint Language (OCL) – is used to define the well-formedness rules.

The Foundation package contains the Data Types package, the Core package and the Extension Mechanism package. The Foundation package is the language infrastructure that specifies the static structure of models.

- The Data Types package defines the basic data structures for the language. It specifies, for example, primitive data types such as Integer or String as well as enumeration data types like Boolean.
- The Core package specifies the basic concepts required for an elementary metamodel and defines mechanisms to attach additional language constructs. Concrete constructs specified as metaclasses in the Core package include Class, Attribute, Operation, Association and Constraint.
- The Extension Mechanisms package specifies how model elements are customised and extended with new semantics. It defines the semantics for stereotypes, constraints\(^\text{1}\), and tagged values. Stereotypes may introduce additional constraints and additional tagged values, but it is not necessary to stereotype a model element in order to give it individually distinct constraints or tagged values. Constraints can be directly attached to model elements to specify new semantics using for example a constraint language like OCL.

\(^{1}\)The metaclass Constraint is defined in the Core package but also used in this package.
Constraints are defined by boolean expressions and, for the model to be consistent, the expression must always yield the value “true” when evaluated at any time when the system is stable (i.e., not during the execution of an atomic transaction). The concepts of the Extension Mechanisms package apply to model elements of the metamodel, not to instances and, therefore, represent extensions to the modelling language. Within a model, any model element may have a set of constraints and a set of tagged values. Additionally, it may be classified by an attached stereotype. Any constraints or tagged values on the stereotype are implicitly attached to the model element. If there are any conflicts among multiple constraints or tagged values (inherited or directly specified), then the model is ill-formed and inconsistent.

The Behavioral Elements package is the language superstructure that specifies the behavioural model elements. It is decomposed into the Common Behavior package, the Collaborations package, the Use Cases package, the State Machines package and the Activity Graphs package.

- The Common Behavior package specifies the core concepts required for behavioural elements. Among others, this includes different actions, e.g., call actions that result in the invocation of an operation, create actions that result in the creation of an instance, and destroy actions that result in the destruction of an object.

- The Collaborations package specifies the concepts needed to express how different elements of a model interact with each other from a structural point of view.

- The Use Cases package specifies behaviour using actors and use cases. These concepts are used to define the functionality of a system, a subsystem, or a class.

- The State Machines package defines how discrete behaviour can be modelled using finite state-transition systems. The state machine formalism described is an object-based variant of Harel statecharts and provides the semantic foundation for activity graphs.

- The Activity Graphs package defines this special form of state machines that is used to model processes. The primary focus of activity graphs is on the sequence and conditions for the actions that are taken.

The third package of the UML metamodel is the Model Management package which specifies how model elements are organised into models, packages, and subsystems.
Bibliography


