Semantic Inspection of Software Artifacts
From Theory to Practice

by

Tim Heyer

Department of Computer and Information Science
Linköpings universitet
SE–581 83 Linköping, Sweden

Linköping 2001
Abstract

Providing means for the development of correct software still remains a central challenge of computer science. In this thesis we present a novel approach to tool-based inspection focusing on the functional correctness of software artifacts. The approach is based on conventional inspection in the style of Fagan, but extended with elements of formal verification in the style of Hoare. In Hoare’s approach a program is annotated with assertions. Assertions express conditions on program variables and are used to specify the intended behavior of the program. Hoare introduced a logic for formally proving the correctness of a program with respect to the assertions.

Our main contribution concerns the predicates used to express assertions. In contrast to Hoare, we allow an incomplete axiomatization of those predicates beyond the point where a formal proof of the correctness of the program may no longer be possible. In our approach predicates may be defined in a completely informal manner (e.g. using natural language). Our hypothesis is, that relaxing the requirements on formal rigor makes it easier for the average developer to express and reason about software artifacts while still allowing the automatic generation of relevant, focused questions that help in finding defects. The questions are addressed in the inspection, thus filling the somewhat loosely defined steps of conventional inspection with a very concrete content. As a side-effect our approach facilitates a novel systematic, asynchronous inspection process based on collecting and assessing the answers to the questions.

We have adapted the method to the inspection of code as well as the inspection of early designs. More precisely, we developed prototype tools for the inspection of programs written in a subset of Java and early designs expressed in a subset of UML. We claim that the method can be adapted to other notations and (intermediate) steps of the software process. Technically, our approach is working and has successfully been applied to small but non-trivial code (up to 1000 lines) and designs (up to five objects and ten messages). An in-depth industrial evaluation requires an investment of substantial resources over many years and has not been conducted. Despite this lack of extensive assessment, our experience shows that our approach indeed makes it easier to express and reason about assertions at a high level of abstraction.
Acknowledgements

First of all, I would like to thank my supervisors Ulf Nilsson, Anders Törne, and Staffan Bonnier, for their inspiring ideas and many fruitful discussions. Furthermore, I thank the rest of the members of the Theoretical Computer Science Laboratory (TCSLAB) and the Real-Time Systems Laboratory (RTSLAB) for the good and stimulating working environment.

I am grateful to Ulf Hammar and Stefan Frennemo at ABB Industrial Systems AB for remarks that have led to improvements of the semantic code inspection approach. Many thanks also to Pär Emanuelson, Tony Olsson, and Jan Lindgren at Ericsson SoftLab AB. Their experience in the development of large-scale, industrial software helped to improve the semantic design inspection approach. Furthermore, I would like to express my gratitude to NUTEK and VINNOVA for their financial support.

Last, but certainly not least, I thank my wife Katrin Wand and my family for their patience and for enduring our long separation.
Contents

1. Introduction ................................................. 1

2. Background .................................................. 5
  2.1. Testing ................................................. 5
  2.2. Inspection ............................................. 6
  2.3. Formal Verification .................................... 7

3. Our Approach .............................................. 11

4. Code Inspection ........................................... 19
  4.1. Specification of Java Programs ......................... 21
    4.1.1. Our Java Subset ..................................... 21
    4.1.2. Class Invariants ..................................... 22
    4.1.3. Interface Specifications ............................ 23
    4.1.4. Other Assertions .................................... 26
  4.2. Program Development ................................... 28
  4.3. The Generation of Questions ......................... 31
    4.3.1. Prerequisites and Limitations ....................... 31
    4.3.2. An Axiomatic Semantics ............................. 34
    4.3.3. Applying the Semantics to Generate Verification Conditions .... 39
  4.4. A Prototype Tool ..................................... 41
    4.4.1. Architecture ........................................ 42
## Contents

4.4.2. Using the Prototype ...................................... 42
4.4.3. Possible Extensions to the Prototype .................. 45
4.5. Example .................................................... 45
4.5.1. Railway Interlocking Systems ........................... 46
4.5.2. Implementation of an Interlocking System .............. 48
4.5.3. Inspection of the Table Generator ...................... 58

5. Design Inspection .............................................. 67
5.1. Development Process ........................................ 68
5.2. Specification Notation ........................................ 70
5.3. Design Notation .............................................. 74
5.3.1. Sequence Diagrams ........................................ 74
5.3.2. Class Diagrams .......................................... 83
5.4. The Generation of Questions ................................. 89
5.4.1. Prerequisites and Limitations ............................ 90
5.4.2. Semantics ................................................. 93
5.4.3. Simplifications .......................................... 107
5.4.4. Alternatives and Extensions ............................ 110
5.5. A Prototype Tool ............................................ 113
5.6. An Example ................................................. 116
5.6.1. The Use-Case Diagram ................................... 118
5.6.2. The Sequence Diagrams .................................. 119
5.6.3. The Class Diagrams ...................................... 126
5.6.4. Questions ................................................ 128
5.6.5. Omitting a Message ...................................... 132
5.6.6. Intermediate Assertion to Weak ......................... 133

6. The Inspection Process ........................................... 135
Contents

7. Related Work 141
  7.1. Software Development ........................................... 141
  7.2. Software Inspections ............................................ 142
    7.2.1. Criticism to Conventional Inspection .................... 143
    7.2.2. Inspection Based on Informal Annotations ............... 145
    7.2.3. Tool Support for Software Inspections .................. 146
  7.3. Formalizations of UML ......................................... 146
  7.4. Sequence Charts .............................................. 148
    7.4.1. Message Sequence Charts .................................. 149
    7.4.2. Live Sequence Charts .................................... 149
  7.5. Assertions .................................................. 150
    7.5.1. Formal Verification ....................................... 150
    7.5.2. Dynamic Checking ......................................... 151
    7.5.3. Documentation ............................................ 152

8. Conclusion and Future Work 153

A. Topology.java .................................................. 167
B. LockTab.java .................................................... 173
C. ExclTab.java ..................................................... 181
D. TableGen.java .................................................... 183
E. Assertions ..................................................... 189
   E.1. Use-Case Diagram ........................................... 189
   E.2. Sequence Diagram ........................................... 190
   E.3. Class Diagram ............................................... 194
1. Introduction

Providing means for the development of high-quality software artifacts (e.g. a design or an implementation) remains a central challenge of computer science. The overall aim of the work described in this thesis is to develop a new method for increasing the confidence in the correctness of software. Our focus is on functional properties (and not on real-time properties, performance, reliability etc). To achieve our goal we integrate formal and informal verification methods. The result is a novel approach to the inspection of software artifacts.

Today’s most important method to ensure the correct behavior of software is testing (see Sect. 2.1). That is, the software is executed with the intention to detect defects. However, executable artifacts are available rather late during the software development. Therefore, testing (and especially removing the defects discovered) is usually considered an expensive and time-consuming activity. Other approaches to increase the confidence in the correctness of the software rely on artifacts available earlier than executable artifacts. These approaches are:

Formal verification. Formal verification (see Sect. 2.3) is based on mathematical principles to demonstrate the correctness of a formal artifact with respect to a formal specification. The required knowledge of the formal notation and its proof system is often experienced as a significant barrier against its industrial use. Hence formal verification is typically used only for especially critical components.

Inspection. Inspection (see Sect. 2.2) is the process of finding defects in the artifact by human examination. Several techniques for the inspection phase are in use. Inspection is reported to be very effective both with respects to defect detection and cost (see Sect. 7.2). However, generally the focus is on style rather than on functionality. Since functionality changes from artifact to artifact it is difficult to give general guidelines for how to systematically check an artifact for defects with respect to its intended functionality. Thus the effectiveness of inspection varies very much with the experience and discipline of the personnel involved.

In this thesis we present novel principles for tool-supported inspections. The inspections focus on the functional correctness of software artifacts. The approach combines today’s
inspection with elements of formal verification to yield a practical and systematic in-
spection process. We believe that there are several advantages of integrating formal and
informal approaches (see e.g. [17]):

- Formal and informal approaches complement each other. Formal approaches con-
  tribute with their precise semantic basis (enabling advanced tool support) whereas
  informal approaches provide a simpler and more intuitive view.
- Integrated approaches are easier to introduce in an industrial organization because
  changes to the development process in place are smaller.
- Specification and verification are possible at varying levels of formality depending
  on the desired confidence level.

Our idea is to annotate software artifacts with assertions describing presumed and de-
sired behavior of the software. These assertions may partly be expressed in an informal
manner, e.g. in natural language. Nevertheless, if the semantics of the notation (apart
from the assertions) used to describe each artifact is sufficiently formal then it is pos-
sible to automatically generate the questions that need to be addressed to ensure the
(functional) correctness of the artifact. These questions form the basis for our novel type
of inspection. We fill the somewhat loosely defined steps of the conventional inspection
with a very concrete content, specifying a highly structured and tool supported protocol
for inspecting software artifacts. Our approach was developed over the last six years in
the scope of two projects:

**Verification Automation in Software Development.** The project started in November
1995 and ended in November 1997. It was one of several projects within the
competence center for Information Systems for Industrial Control and Supervision
(ISIS), and was carried out in cooperation between ABB Industrial Systems and
the Real-Time Systems Laboratory (RTSLAB) at the Department of Computer
and Information Science (IDA), Linköping University. The activities in ISIS are
funded by the University, industry, and the Swedish National Board for Industrial
and Technical Development (NUTEK).

The general goal of the project was to develop practical means for the early ver-
ification of code. Our strategy was to provide semi-formal support both for the
development of code and for its inspection. To achieve our goal, we suggested
COMPASS, a comprehensible assertion method (see [13]). The method supports
the automatic generation of those questions which are relevant for the correctness
of the code, and whose answers hence provide a systematic explanation of why the
code works as intended. Such an explanation helps either in pinpointing errors, or
in convincing the inspection team of the correctness of the code.

Our approach has its foundation in the well-known theories of so called *assertional*
programming. An assertion specifies a condition that program variables must satisfy each time a certain point in the program execution is reached. Thus, assertions may be used to specify the intended result of executing a piece of code. In 1969 Hoare (see [53]) introduced a logic for reasoning with assertions. His theory provides means for formally proving the correctness of a program with respect to given assertions. Dijkstra (see [29]) noted that verifying code after it has been developed is not entirely realistic. He proposed that code should be developed along with the arguments for its correctness and suggested a discipline of programming based on Hoare Logic.

Both Hoare’s method and Dijkstra’s program development discipline are, within academia, well established since a very long time. However, the methods are hardly known in industry and even less used. We believe this lack of understanding of Hoare’s and Dijkstra’s ideas is due to the fact that most expositions approach the subject from a quite formalistic point of view.

In Compass we relax the requirements on formal rigor in a controlled manner to achieve a method which is more easily used in practice but which still remains partially mechanizable. The key idea in Compass concerns the predicates which are used for expressing assertions: in Compass we allow such predicates to be used without an associated formal definition even when the possibility of a formal proof is disabled. Instead we expect that the predicates are defined in natural language.

We developed a prototype tool for the inspection of programs written in a subset of Java and successfully applied it to small but non-trivial programs. Even though we have not conducted an industrial field study, our experience indicates that our approach enables expressing and reasoning about assertions at a high level which makes the algorithmic content of a program explicit.

**Tool Support for Design Inspection.** The project started in April 1999 and ended in August 2001. It is funded by the Swedish Agency for Innovation Systems (VINNOVA) and is carried out in cooperation between Ericsson SoftLab AB and the Theoretical Computer Science Laboratory (TCSLAB) at IDA, Linköping University.

The goal of the project was to develop methods that facilitate tool support for the inspection of early designs expressed in a subset of UML. To achieve our goal we applied the same principles used for the inspection of Java programs to the inspection of early UML designs. However, since early UML designs are typically incomplete, scenario-based, graphical, and on a high level of abstraction we face new challenges. Nevertheless, the basic idea is to automatically generate those questions that need to be addressed during the inspection from a specification and a design. Both the specification and the design are expressed in a subset of UML.
extended with assertions.

To evaluate our approach we implemented a prototype and applied it to small but non-trivial designs. Thus, we demonstrate that it is possible to automatically generate those question that have to be addressed in the inspection to check early, incomplete designs for functional correctness. We have not conducted an in-depth industrial evaluation but our experience shows that our approach makes it easier to express and reason about assertions at a high level of abstraction which emphasizes the algorithmic content of the design.

The remainder of the thesis is organized as follows: in Chap. 2 we describe existing approaches to verify software artifacts (namely testing, conventional inspection, and formal verification). The strong and weak points of the different approaches are discussed. How our approach combines conventional inspection with elements of formal verification to enable the automatic generation of questions to be asked during the inspection is presented in Chap. 3. The application of the general principles to code artifacts is demonstrated in Chap. 4 and the application to design artifacts in Chap. 5. Both chapters include detailed descriptions of the involved notations, methods, prototype tool implementations, and examples. In the following Chap. 6 we briefly introduce an inspection process that exploits our automatic generation of questions. Chapter 7 presents related work. Finally, Chap. 8 contains conclusions and opportunities for future work.
2. Background

As mentioned, the general objective of this work was to suggest improvements to the current software development practice, enabling more rigid but still practically acceptable methods for the verification of software. In this chapter we briefly describe and compare three existing approaches to verify software (focusing on functional properties). In Sect. 2.1 we discuss testing, in Sect. 2.2 conventional inspection, and in Sect. 2.3 formal verification.

2.1. Testing

The single most commonly used verification approach of today is certainly testing. Testing is the process of executing software to detect the potential presence of defects. Testing is closely related to debugging, i.e. the process of locating and correcting these defects. Even though testing is today’s most important verification method it has several important drawbacks compared to conventional inspection and formal verification:

- Testing relies on an executable artifact. Therefore, it is used later than inspection and formal verification. To correct defects which are detected late is often expensive.

- Testing is in general incomplete. Except for very small programs it is not possible to test the software for all possible inputs. Thus, testing as the only means to guarantee the correctness of the software cannot be completely relied upon.

- Testing usually is on a very low level of abstraction making it e.g. cumbersome to specify test cases.

- Testing does not provide much guidance for the development of correct software, in particular when compared to formal verification in the style of Hoare. Test cases specify what the software has to accomplish but they give no hints on how.

Because of these drawbacks the costs of testing software systems are rapidly increasing compared to the overall development costs. Testing must therefore be complemented
2. Background

with other means to help increasing the confidence in the correctness of the software. These alternative means should preferably be applicable much earlier in the software development process and they should provide some support for how to develop correct software artifacts right from the beginning.

2.2. Inspection

In contrast to testing inspection allows the early detection of defects. Inspection, first introduced by Fagan in 1976 (see [34]), is the process of finding defects in the artifact by human examination. Typically, the inspection process consists of several phases (a detailed description of software inspection can be found e.g. in [43] and in [57]):

1. Planning and overview phase
   The phase starts when the author of a software artifact requests its inspection. A selected inspection leader then checks whether the artifact is ready for inspection. If the artifact is ready for inspection the leader determines who should participate and schedules a kick-off meeting. The meeting is intended to ensure that all participants understand the artifact to inspect as well as their roles in the inspection.

2. Inspection phase
   The participants (equipped with rules, procedures, and checklists intended to help to discover defects) then individually check the artifact. Afterwards the members of the inspection team meet to discuss their findings with each other. The (presumably) found defects are logged and it is determined who is responsible for resolving each defect.

3. Rework and follow-up phase
   The inspection leader controls that the defects are being repaired or in some other way dealt with.

The most significant phase is the inspection (defect-detection) phase. Several techniques for the inspection phase are in use (see Sect. 7.2). Most approaches rely on checklists of some form to facilitate the inspection phase. However, generally the focus of checklists is on style rather than on functionality. For example, in code inspections the inspection team usually checks that all naming conventions have been followed, that the indentation is as required etc. This is certainly important in order for the code to be readable and uniform (e.g. for maintainability), but it is not sufficient for ensuring correctness. In most cases there is a very systematic way to carry out these checks. Indeed, the checks could in principle often be automated. In addition, code inspection checklists sometimes require the inspector to check that the code behaves as intended. But the checklist provides no means for checking functional correctness in a systematic manner. Since
2.3. Formal Verification

functionality changes from artifact to artifact it is difficult to give general guidelines for how to systematically check the artifact for defects with respect to its intended functionality. Despite its problems, inspection is reported to be very effective both with respect to defect-detection and cost (see Sect. 7.2).

Nevertheless, the effectiveness of inspection is very much depending on the experience and discipline of the personnel involved. Several tools to support the inspection process are available. For example, Macdonald and Miller compared 16 tools in 1999 (see [79]). However, the focus of these tools is almost entirely on administrative tasks like scheduling meetings and collecting defect reports. What is missing are guidelines that focus on the functional correctness of the particular artifact, i.e. guidelines that precisely state which questions to address to find defects. Ideally, such questions should be generated automatically from a given artifact. The semantic inspection approach introduced in this thesis has been designed to fill this need.

2.3. Formal Verification

Another approach that allows the early detection of defects is formal verification. Formal verification is based on mathematical principles to demonstrate the correctness of a formal software artifact with respect to a formal specification.

A code verification approach that has received a great deal of attention is Hoare Logic introduced by Hoare (see Sect. 7.5.1). This approach has had a significant impact on later formal methods for designing and verifying imperative, sequential computer programs. Hoare Logic is an axiomatic method for proving programs correct also known as the partial correctness assertion method. An assertion specifies a condition that program variables must satisfy each time a certain point in the program execution is reached. By associating with an operation (i.e., a method of an object) two special assertions called the pre- and the postcondition of the operation, assertions may be used to specify the effect executing the operation is intended to have on the computation state. In 1969 Hoare (see [53]) introduced a logic for reasoning with assertions. His theory provides means for formally proving the correctness of a program with respect to given assertions which have been added to the program after its development. The formulae of Hoare Logic are so called Hoare Triples \( \{P\} S \{Q\} \), where \( P \) and \( Q \) are assertions, and \( S \) is a piece of code. It is to be read “if \( S \) starts executing in a state where \( P \) is satisfied, and if the execution of \( S \) terminates, then \( Q \) is satisfied upon termination”. The method proposed by Hoare for proving such formulae, presupposes the existence of proof rules for the programming language under consideration. The method may be considered to consist of the following three phases (see Fig. 2.1 on the following page):
2. Background

![Diagram: The three steps of Hoare's partial correctness assertion method]

Fig. 2.1. The three steps of Hoare’s partial correctness assertion method

1. Development of code with assertions inserted at appropriate places

   In the first phase the code is developed and corresponding assertions are added. The assertions specify the intended behavior of the code.

   Example 2.1

   The following code with assertions swaps the values of two integer variables without using an additional temporary variable:

   \[
   \{ x = x@\text{pre} \land y = y@\text{pre} \} \\
   x = x + y; y = x - y; x = x - y; \\
   \{ x = y@\text{pre} \land y = x@\text{pre} \}
   \]

   The postcondition expresses that, after the execution of the three assignments, \( x \) equals the initial value of \( y \) (denoted as \( y@\text{pre} \)) and \( y \) equals the initial value of \( x \) (denoted as \( x@\text{pre} \)). The precondition simply expresses that before the execution both \( x \) and \( y \) equal their initial values.

   Although the program consists only of three assignments it is not obvious that it actually is correct.

   Dijkstra (see [29]) noted that verifying code after it has been developed is not entirely realistic. Dijkstra instead proposed that code should be developed along with arguments for its correctness. For this purpose he suggested a discipline of programming, based on Hoare Logic, where one first states the assertions the code is to establish, and then uses the assertions to guide the development of the code.

2. Generation of a set of logical formulae (called verification conditions)

   In the second phase so called verification conditions are generated from the code and the assertions. Verification conditions are logical formulae. If each of them can be proven then all assertions except the precondition are valid, i.e. the code is
correct with respect to the assertions. Verification conditions are generated using the axioms and inference rules (axiomatic semantics) of the programming language under consideration.

**Example 2.2**
Consider again the program from Example 2.1 on the preceding page:

\[
\begin{align*}
\{ x &= x@pre \land y = y@pre \\
& \quad \quad \quad x = x + y; y = x - y; x = x - y; \\
& \quad \quad \quad \{ x = y@pre \land y = x@pre \}
\end{align*}
\]

Informally, the verification condition is generated like this:

1. For the postcondition to be satisfied after the execution of the third assignment, everything that is supposed to be valid for \(x\) in the postcondition must be valid for \(x - y\) immediately before the execution of the third assignment. That is, the following condition must be satisfied before the final assignment:

\[
x - y = y@pre \land y = x@pre.
\]

2. Accordingly, before the second assignment the following condition must be satisfied:

\[
x - (x - y) = y@pre \land x - y = x@pre.
\]

3. Finally, before the first assignment the following condition has to be satisfied:

\[
(x + y) - ((x + y) - y) = y@pre \land ((x + y) - y) = x@pre.
\]

The verification condition is that the condition just derived must be implied by the precondition:

\[
x = x@pre \land y = y@pre \implies \quad (x + y) - ((x + y) - y) = y@pre \land ((x + y) - y) = x@pre
\]

Verification conditions for more complex programs are generated in a similar way.

It should be noted that the meaning of “+”, “−”, and the predicate “=” is not involved in the above generation of verification conditions. In fact, the generation of verification conditions comprises mainly the textual substitution of variables by expressions and can be performed automatically without taking the meaning of involved predicates and operations into account.

3. **A formal proof of the verification conditions**

The third phase is a formal proof of the verification conditions, using axioms and proof rules for the domain over which the program variables range.
2. Background

Example 2.3
The verification condition generated in the previous Example 2.2 on the preceding page is:

\[ x = x^{\text{pre}} \land y = y^{\text{pre}} \implies (x + y) - ((x + y) - y) = y^{\text{pre}} \land ((x + y) - y) = x^{\text{pre}} \]

Using conventional arithmetic laws the above formula can easily be simplified to:

\[ x = x^{\text{pre}} \land y = y^{\text{pre}} \implies x = x^{\text{pre}} \land y = y^{\text{pre}} \]

The proof of the above formula is trivial. Hence the code is correct with respect to the assertions, i.e. the code in Example 2.1 on page 8 indeed swaps the values of two integer variables.

Both Hoare’s method and Dijkstra’s program development discipline are, within academia, well established since a very long time, and are also recognized to be the predominant methods for formal development and verification of sequential programs in imperative languages (see Chap. 7). In this perspective, it is quite remarkable that the methods are hardly known in industry, and even less used. Indeed, to the extent asserted programs are developed at all, the assertions are mostly used as run-time checks during debugging and testing (see Sect. 7.5.2). We believe that the lack of understanding of Hoare’s and Dijkstra’s ideas is due to the fact that most expositions approach the subject from a quite formalistic point of view, and thus give the feeling that full formality is a requirement for its applicability. The demand for full formality arises from the third phase of Hoare’s method: to be able to perform a formal proof of the verification conditions, a formal axiomatization is required for each predicate and function symbol which occurs in the assertions and which hence is used to express the intended behavior of the program. However, such axiomatizations do often have a non-obvious connection to the intuitive understanding of the property the predicate is to represent (see Example 3.2 on page 13). As a consequence, they are both difficult to state and to reason with.

Even for other formal approaches the required knowledge of the formal notation and its proof system is often a significant barrier against the industrial use. Hence formal verification is typically used only for especially critical components (see e.g. [106] for a discussion on why formal methods have not been adopted by industry to the extent one would expect).
3. Our Approach

The general aim of our work is to develop practical means for increasing confidence in software correctness. The approach presented in this thesis combines conventional inspection with elements of formal verification. Traditional inspection lacks guidelines that precisely state which questions need to be addressed to find defects. Formal verification is often perceived as very difficult due to the required knowledge of the formal notation and its proof system. Our basic hypothesis is that a method which is easier to use than formal verification and which remains partially mechanizable, may be achieved by relaxing the requirements on formal rigor in a controlled manner. The idea is to automatically generate the questions to be answered during the inspection. The questions are generated from a given annotated software artifact. We are not aware of other approaches that extend conventional inspection with elements of formal verification to yield a systematic software inspection. We call our inspection *semantic inspection* for two reasons:

- We are interested in functional properties of the software artifact, i.e., we are interested in what the artifact means and what it does.

- Our approach requires that the meaning of the underlying notation (which is used to describe the inspected artifact) is formally defined. For example, we need to know the semantics of the programming language Java and of the UML diagrams for early designs since we apply our semantic inspection later on artifacts expressed in these notations (see Chap. 4 and Chap. 5).

The cornerstones on which our semantic inspection is based are Hoare’s method for proving programs correct (see Sect. 2.3) and Fagan’s work on code inspection (see Sect. 2.2). Our semantic inspection method is similar to Hoare’s partial correctness assertion method. However, our approach differs in two important ways from Hoare’s approach:

- The key idea in the semantic inspection approach concerns the predicates and functions ranging over the domain of artifact variables, and which hence are used for expressing assertions. In our approach such predicates and functions are allowed to appear without an associated formal definition (axiomatization) beyond
3. Our Approach

the point where a formal proof of the correctness of the artifact may no longer be possible. Instead the predicate or function is defined in any way suitable for humans to understand its meaning (e.g. in natural language). The point is that such predicates and functions may still have a perfectly legal informal interpretation, and by expressing it informally rather than formally its meaning is more directly accessible to the human reader. For convenience we refer to predicates and functions without formal axiomatization as informal predicates and informal functions.

- We apply the idea of generating verification conditions from code and assertions to other artifacts as well. For example, in Chap. 5 we describe how verification conditions can be generated from annotated early designs expressed in a subset of UML.

With the introduction of informal predicates and functions we relax the requirements on formal rigor in a controlled manner obtaining a method which is more easily used in practice but which still allows automatic tool support. As mentioned, we allow informal functions and predicates to such an extent that a formal proof of the correctness of the artifact may no longer be possible. Therefore, we replace the formal proof with a human inspection and get the following steps (see Fig. 3.1 on page 14):

1. Development of software artifact and assertions

   The software artifact is developed as usual (i.e., with an existing development process). However, the artifact is annotated with assertions that may contain informal functions and informal predicates. Informal functions and predicates make it easy to express conditions in terms of assertions. Of course, using natural language increases the risk of ambiguous function and predicate definitions. Therefore, our inspection may be considered “weaker” than formal verification. However, even formal verification faces that challenge since natural language is the most widely used notation for stating requirements which may later be translated into formal specifications (see e.g. [61] on how ambiguities in requirements can be detected).

**Example 3.1**

An assertion that states that an array of integers is sorted in the interval from left to right could be expressed with help of the informal predicate symbol sorted. The definition of the symbol is then supplied e.g. in a special comment in the code:

```plaintext
define sorted(array, left, right) informally
    The array “array” is sorted in increasing order between “left” and “right”.
end define ;
```

The predicate symbol sorted may now be used in assertions either by itself or combined with other predicate symbols (both formally and informally defined ones).
It should be noted that assertions provide a structured documentation of the artifact and thus improve the communication between different developers (and possibly customers). The importance and role of accurate, structured documentation has been described e.g. by Parnas in [94]. Apart from being used for documentation purposes, the assertions may be exploited to drive the development of the artifact in the manner of Dijkstra’s discipline of programming (see [29]).

2. Automatic generation of verification conditions

The generation of verification conditions is not dependent on the meaning of the predicates appearing in the assertions. Thus it is possible to automatically generate the verification conditions necessary to verify the artifact.

3. Human inspection of the verification conditions

The verification conditions typically contain informal functions and predicates, generally disabling the possibility of a formal proof. Therefore the formal proof of Hoare’s method is replaced by human inspection. During the inspection the verification conditions are informally justified by the inspectors. However, informal functions and predicates facilitate human reasoning with assertions and verifying verification conditions on a high level of abstraction making the algorithmic content of the artifact visible.

Example 3.2

In the formal verification of a program that sorts an array of integers the following verification condition may arise:

\[
\forall i, 1 \leq i \leq n - 1 : a[i] \leq a[i + 1] \wedge \forall i, 1 \leq i \leq n : a[n + 1] \geq a[i] \implies \\
\forall i, 1 \leq i \leq n : a[i] \leq a[i + 1]
\]

Using a predicate symbol \textit{sorted} and a function symbol \textit{max} the same verification condition may be presented in the following form instead:

Assume:
1. \textit{sorted}(a, 1, n)
2. \textit{a}[n + 1] \geq \textit{max}(a, 1, n)

Then:
1. \textit{sorted}(a, 1, n + 1)

Is the conclusion satisfied?

The predicate symbol \textit{sorted} (and the function symbol \textit{max}) describes an informal predicate (and function) as defined in Example 3.1 on the facing page. Thus, the first condition with the formal axiomatization can formally be proven while
3. Our Approach

Our Approach

Fig. 3.1. The three steps of the semantic inspection approach

the second condition requires a human to decide whether it is satisfied or not. However, the informal version is much easier to read and to understand; a human reader is able to answer the question easily without having to struggle with the formal axiomatization. The second verification condition is also presented in a simpler (Horn clause) form avoiding nested formulae and quantifiers.

Each verification condition is presented as a set of questions of the form “Assume that $P$ is satisfied, is then $Q$ satisfied as well?” where $P$ and $Q$ are formulae as described later in this chapter. If all questions can be answered with “yes” then the artifact is assumed to be correct with respect to the assertions (see also Chap. 6).

In later chapters we describe how the semantic inspection approach can be applied to Java programs (see Chap. 4) and also to early UML designs (see Chap. 5). Certain types of assertions may be attached both to code and to designs (e.g. operation pre- and postconditions), whereas other types may not. However, common to all assertions is that they are expressed in a notation similar to first-order predicate logic. In a first order theory some set of objects is selected, and all the statements of the theory are statements about these objects. The objects of primary concern here are the abstract data structures of the software artifact (i.e., assertions specify conditions on the state of the modeled system). The initial focus of our work has been on the application areas of our industrial partners (see Chap. 1), i.e., process control and telecommunications.

The software artifacts we have studied are implementations expressed in a subset of the programming language Java (see Chap. 4) and designs expressed in a subset of UML (see Chap. 5). Both the design and code notations are based on object-orientation. Our assertion notation which is presented in this section has been developed to support the application areas and software artifacts mentioned.

An assertion is a well-formed formula. Well-formed formulae may contain predicates which have arguments. The arguments of predicates are well-formed expressions which we define first. To define expressions we use the following syntactic categories and meta-
variables which range over constructs of each category:

- \( n \) will range over numerals, \textbf{Num}
- \( x \) will range over variables, \textbf{Var}
- \( E \) will range over expressions, \textbf{Exp}
- \( f \) will range over (user-defined) function symbols, \textbf{Fun}

All variables that occur in an artifact can be used in assertions (e.g. for Java programs this includes qualified variables). These variables are called \textit{artifact variables} (or \textit{program variables} if the artifact is a program and \textit{design variables} if the artifact is a design). In principle, assertions in the style of Hoare specify a condition on the state of the software as captured in the program variables. The program variables occur as free variables in the assertions. However, often it is necessary to refer to initial values of artifact variables. For example, if we want to specify what an operation accomplishes we may need to refer to the initial values of its parameters, e.g. to express that a car has half its initial speed after breaking shortly. Therefore, in Hoare Logic e.g. initial values of variables are captured in logical variables which do not appear in the artifact. Unlike artifact variables, logical variables do not change their values during the execution of the software. The logical variables are (implicitly) universally quantified. To distinguish logical variables from artifact variables they have a special appendage “@pre” or “#n” (where \( n \) is a natural number). For convenience we implicitly define for each artifact variable a corresponding logical variable with the name and appendage “@pre”. These logical variables refer to the initial value of the artifact variable with respect to the context of the assertion. For example, an assertion that expresses that a car has half its initial speed after the “brake-shortly” operation could be written as \( 2 \ast \text{speed} = \text{speed}@pre \). With logical variables, assertions (in particular postconditions) specify a relation between two states rather than one state. More details on the role of logical variables and the relationship between Hoare Logic and VDM (the Vienna Development Method, see e.g. [60]) can be found in [63].

3.1 Definition (Well-formed expression)

Numerals and variables are expressions. Complex expressions can be obtained by combining expressions with predefined operators or user-defined functions. The abstract syntax of expressions is:

\[
E ::= n \mid x \mid \text{this} \mid \text{result} \mid -E \mid E - E \mid E + E \mid E/E \mid E \ast E \mid f(E, \ldots, E)
\]

The meaning and precedence of negation, addition, subtraction, multiplication, and
3. Our Approach

truncating division is the standard one. We actually require our language to be typed with a type system that depends on the artifact. Since we are dealing with several kinds of artifacts here we defer from developing such type systems and assume that all expressions are type correct.

Some additional comments on expressions: the variables this and result are special artifact variables. The variable this generically refers to each single instance of a class, i.e., it is a place-holder for the name of that instance (often when assertions are specified the name of the instance is not known). By using this it is possible to express a condition on the state of each single instance of a class (see also Sect. 4.1 and Sect. 5.3). The variable result refers to the return value of an operation in case it has one (see also Sect. 4.1).

Moreover, all expressions are classified according to their type, i.e., the kind of values they can assume. Possible types are integer (i.e., negative and positive natural numbers) and classes. The notion of a class is the cornerstone of object-orientation. A class can be seen as an abstract data type with attributes and operations to manipulate these attributes. As mentioned, we assume for the remainder of this thesis that all expressions etc. are type correct, in fact when it comes to predicate and function definitions we do not even list the types of parameters like it is done e.g. in Java.

To define formulae we use the following additional syntactic categories and meta-variables which range over constructs of each category:

- $p$ will range over (user-defined) predicate symbols, \textbf{Pre}
- $F$ will range over formulae, \textbf{For}

3.2 Definition (Well-formed formula)

The boolean constants true and false are formulae. Predefined and user-defined predicates are atomic formulae. Complex formulae can be obtained by combining formulae with predefined operators. The abstract syntax of formulae is:

$$F ::= \text{true} \mid \text{false} \mid \neg F \mid F \land F \mid F \lor F \mid F \Rightarrow F \mid F \Leftarrow F \mid E < E \mid E \leq E \mid E > E \mid E \geq E \mid E = E \mid E \neq E \mid \text{if } F \text{ then } F \text{ else } F \mid p(E, \ldots, E)$$

The meaning of the predefined predicates and of negation, disjunction, conjunction, and implication is like in conventional logic. The connectives $\land$ and $\lor$ are conditional variants of $\land$ and $\lor$ to handle partial functions. The result of a conditional conjunction is false if the first argument is false and the result of a conditional disjunction is true if the first argument is true. The meaning of if $F$ then $F_1$ else $F_2$ is $(F \Rightarrow F_1) \land (\neg F \Rightarrow F_2)$. Moreover, we assume that all well-formed formulae are type-correct.
Partial functions are common in software engineering. For example, the value obtained by addressing an array \( a \) with an index \( i \) outside the index set of \( a \) is typically undefined. Several approaches to treat partial functions have been presented. We use Dijkstra’s (see [29]) approach of conditional (asymmetric) conjunction and disjunction. Dijkstra’s approach impose an evaluation procedure to allow the use of partial functions. A symmetric approach to treat partial functions without using a three-valued logic is described by Parnas in [91]. Programming languages typically describe in which order expressions are evaluated. Since questions generated from code usually contain programming language expressions, we chose Dijkstra’s approach. However, other approaches could be used as well.

Some comments on equality: if both arguments of the equality relationship have the same value then they are equal. It is not necessary that both arguments have the same name (or address when it comes to implementations). For example, if we have two variables \( \text{joesCar} \) and \( \text{johnsCar} \) then \( \text{joesCar} = \text{johnsCar} \) if and only if both cars are equal with respect to all their attributes (i.e., if they have the same color, the same age, the same number of horse powers, …). If the car owner is an attribute of a car then \( \text{joesCar} \) and \( \text{johnsCar} \) are not equal.

**Example 3.3**

A formula expressing that a phone \( \text{phoneA} \) is off-hook and connected to another phone \( \text{phoneB} \) over a network \( \text{net} \) could look like this:

\[
\text{offHook(phoneA)} \land \text{connected(phoneA, net, phoneB)}
\]

The formula uses two user-defined predicate symbols, namely \( \text{offHook} \) and \( \text{connected} \). The symbols are defined formally or informally e.g. as shown in Example 3.4 on the next page.

A function or predicate symbol may be defined formally (using an expression respectively a formula) or informally (using e.g. natural language). The point is that any means is possible as long as it provides a unique interpretation of the function or predicate to the human reader. In this thesis function and predicate symbols are usually defined in natural language. An informal definition is given in the following form:

\[
\text{define } p(\text{parameter-list}) \text{ informally} \quad \text{Informal function or predicate definition} \\
\text{end define ;}
\]

A function or predicate symbol may be defined formally using an expression as defined in Def. 3.1 respectively a formula as defined in Def. 3.2. Since formulae may not contain quantifiers, it is in general not possible to completely axiomatize a predicate. That is,
3. Our Approach

the formula used to define a predicate usually contains informally defined functions or predicates. A formal definition is given in the following form:

\[
\text{define } p(\text{parameter-list}) \text{ formally} \\
\text{Formal function or predicate definition} \\
\text{end define} ;
\]

In both cases (informal and formal definition), \( p \) is the name of the function or predicate and \( \text{parameter-list} \) is the comma-separated list of the formal parameters of the function or predicate.

**Example 3.4**
The informal definition of a predicate symbol \( \text{connected} \) that expresses that two phones are connected over a network could look like this:

\[
\text{define } \text{connected}(a, n, b) \text{ informally} \\
\text{The phone "a" and phone "b" are connected over a dedicated, full-duplex line through the telephone network "n".} \\
\text{end define} ;
\]

As shown the predicate symbol \( \text{connected} \) has three parameters. The parameters are substituted with arguments in a formula where the predicate symbol is used (see e.g. Example 3.3 on the preceding page).

**Example 3.5**
That a program computes an integer approximation of the square root of a number may be expressed with the predicate symbol \( \text{maxApproxSquareRoot} \). An informal definition may look as follows:

\[
\text{define } \text{maxApproxSquareRoot}(n, x) \text{ informally} \\
\text{"x" is the largest natural number that is smaller than or equal to the positive square root of "n".} \\
\text{end define} ;
\]

An alternative, formal definition of \( \text{maxApproxSquareRoot} \) could look like this:

\[
\text{define } \text{maxApproxSquareRoot}(n, x) \text{ formally} \\
0 \leq x \land x \times x \leq n \land n < (x + 1) \times (x + 1) \\
\text{end define} ;
\]
4. Code Inspection

In this chapter we will demonstrate how the principles of semantic inspection introduced in Chap. 3 can be applied to the inspection of code. For the inspection of code written in an annotated subset of the Java programming language we developed Compass, a comprehensible assertion method (see [13]). Our method supports the automatic generation of those questions which are relevant for the correctness of the code, and whose answers hence provide a systematic explanation of why the code works as intended. Such an explanation constitutes the heart of code inspection, and helps either in pinpointing errors, or in convincing the inspection team of the correctness of the code. The method consists of the following three steps (see Fig. 4.1 on the next page):

1. **Development of code with assertions inserted at specific places**
   Assertions express conditions that are assumed to, respectively supposed to, be valid at specific points during the computation. In particular, assertions specify both what a piece of code is supposed to establish and what it may expect to be valid. Our assertions may contain predicates without associated formal definition. This enables the formulation of and reasoning with assertions on a high level of abstraction. Moreover, the assertions provide a structured documentation of the code even if they are not used for the generation of the questions.

2. **Generation of a set of questions (i.e., verification conditions)**
   The generation of the questions is not dependent on the meaning of the predicates appearing in the assertions. However, the meaning (i.e., the semantics) of the programming language has to be defined. Then the questions may be automatically generated from the assertions and the code. The questions are the only questions needed to be addressed in the code inspection (for functional correctness).

3. **Human inspection of the questions**
   The questions generated in the second step in general contain informal predicates, thus disabling the possibility of a formal proof. Therefore the questions are presented to a human inspector who will answer them (and informally justify the answer). If all questions can be answered positively then the code is assumed to be correct with respect to the assertions.
It should be clear that not any program nor any property is amenable to verification along the principles of the COMPASS method. We mentioned earlier that we are interested in functional properties alone. To be more precise, we focus on partial correctness as opposed to total correctness. That is, we verify that the program is correct if it terminates but we do not verify that it terminates. To prove the termination a so-called bound function (or variant function) is provided by the programmer for each loop. Then it is verified for each loop that the bound function is bounded from below as long as the loop has not terminated, and that each loop iteration decreases the bound function. If both conditions are satisfied it can be concluded that the loop terminates. Both conditions can be expressed as Hoare Triples. Thus, it is in principle possible to verify termination with our approach. Recursive procedures are treated in a similar manner (see e.g. [51], [1], and [82]). However, our attention is not on termination but on high level functional properties representable as relations between program variables. Furthermore, the code has to be well-structured in order for the method to be applicable (i.e., essentially it has to be developed according to Dijkstra’s discipline of programming).

We continue in Sect. 4.1 with a description of the various types of assertions that are used to specify the intended behavior of code (in the Java programming language). This is related to the first step of our semantic code inspection method introduced earlier. In Sect. 4.2 we give an overview of Dijkstra’s discipline of programming (i.e., how to develop code by first specifying in assertions what the code should accomplish, and then by exploiting the assertions to write the code). The second step, how the questions are generated from an annotated program, is presented in Sect. 4.3. We have implemented a prototype tool to demonstrate the feasibility of our approach. This tool is described in Sect. 4.4. Finally, Sect. 4.5 contains a small but non-trivial example to illustrate our approach. The third step of the semantic code inspection method is the same as for the semantic design inspection method described in Chap. 5. What the human inspection of the questions generated in the second step may look like is therefore described in Chap. 6.
4.1. Specification of Java Programs

The notion of informal predicates has been introduced to enable reasoning about the high level algorithmic content of a program. Therefore, the use of data abstraction is central to our COMPASS method. By formulating assertions in terms of objects of an abstract data type, verification of a high level algorithmic nature can be separated from the verification of low level invariants of the representation of the data type (such as e.g. non-corruption of the data representation).

To be able to evaluate the COMPASS method in an industrial setting, it was first adapted to the programming language C. However, data abstraction is not supported in C. Although it is possible to apply data abstraction without support in the language, it is not realistic to expect such a discipline to be followed. Another property of C that counteracts the intention with informal predicates is C's primitive memory management which makes it difficult to reason about program correctness without taking low level properties into consideration. Even though it theoretically might be possible to include such properties into the verification, the complexity does in practice become unmanageable. Thus, the programming language to which the COMPASS method is applied should have the following properties for a smooth and practical application of the method:

1. Support for data abstraction and
2. Simple memory management (e.g. some form of automatic garbage collection).

With respect to the above considerations all examples in this thesis concern code written in a subset of the Java programming language. Java supports data abstraction (object-oriented), has automatic garbage collection and is hence an appropriate candidate for our COMPASS approach.

As mentioned, assertions are used to specify the intended behavior of a program. To be able to easily compile the Java code with its specification included, all additional constructs that are introduced for our COMPASS method are encapsulated in special comments. These comments start with /*+ and end with */. In the following Sect. 4.1.1 we describe our subset of the Java programming language in general terms. The notion of class invariants is introduced in Sect. 4.1.2. Finally, in Sect. 4.1.3 we explain how the Java function or method interfaces are specified.

4.1.1. Our Java Subset

We choose a subset of Java to simplify the implementation of a prototype tool to support code inspection. Some of the restrictions are of a syntactic nature and are not limiting the expressibility of the language. Other restrictions are real semantic restrictions and
4. Code Inspection

have partly been introduced to keep the prototype tool simple and partly to avoid some of the difficulties described later in Sect. 4.3.1. Some of the major limitations are:

- Single thread assumption. Why our focus has been on sequential programs is explained in Sect. 5.4.1.

- Class variables have to be declared as private and methods as public. We explain the reasons for this restriction in Sect. 5.4.1.

- Java interfaces and class inheritance are not supported. Some principal difficulties related to inheritance are presented in Sect. 4.3.1. However, the main reason for this restriction is to simplify the development and implementation of our approach.

- Functions, i.e. methods that return a value, are neither allowed to modify any of their arguments nor class variables. It is not difficult to handle functions which have side-effects, but such functions make the questions more complex. The restriction allows us to substitute every occurrence of a function invocation with the result specified in the postcondition of the function.

- Local variable declarations are allowed only directly after the beginning bracket of the body of a method. This restriction is introduced for practical reasons alone and not because of principal difficulties.

- No static variables are allowed. A static variable could be seen as a class variable that is visible only in a single method (i.e., there are no principal difficulties to handle static variables). We do not support static variables for practical reasons.

- \textit{While} loops are permitted, but not \textit{for} and \textit{repeat} loops. There are no principal difficulties to handle for and repeat loops; the reasons for not supporting them are purely practical.

- Assignment in expressions is not supported. Assignments in expressions are like function with side-effects (see above).

- Java’s exception handling is not considered because it would make the development of our approach more complex. However, there are no principal difficulties.

It may seem that many Java programming language constructs are not supported. However, in many cases it is rather straightforward to express an unsupported construct with the help of the remaining ones.

4.1.2. Class Invariants

We support the specification of a \textit{class invariant}, i.e., a property shared by all instances of a class. This property is supposed to be preserved by the methods of the class. This
4.1. Specification of Java Programs

means, that if the class invariant is satisfied for an object before the invocation of a method then it is satisfied after its completion (for the constructor method the invariant only needs to be satisfied after the execution). The notation for a class invariant is:

```
   maintains
   Specification of class invariant;
```

The class invariant is a formula as described in Chap. 3. It is supposed to be satisfied whenever the object is accessed. Only the program variables `this` (or functions on `this`) are allowed to occur in the class invariant (since class attributes are not visible outside the class but the invariant is). A class invariant is written in a COMPASS comment directly after the class declaration. If the invariant (or any other assertion) contains user-defined predicates or functions, then their definitions are provided in the same file as the invariant in the textual form presented earlier in Chap. 3.

**Example 4.1**

A class for simulating traffic lights may have an invariant which states that an instance of the class always shows a valid combination of lights:

```java
   public class TrafficLight
   /*++ maintains
     validSignal(this);
   */
```

To access particular properties of a class functions on `this` have to be used (the properties may be attributes or values derived from attributes). For example, we could use user-defined functions to access the individual lights of a traffic light. An invariant using a 3-ary predicate symbol `validSignal` could look like this: `validSignal(redOn(this), yellowOn(this), greenOn(this))`.

### 4.1.3. Interface Specifications

An interface specification describes the name and the formal parameters of a function or method. In addition, it specifies under which conditions the function or method may be invoked and what result it is intended to deliver. An interface specification consists of the function or method head followed by a special comment containing a precondition, a postcondition, and a so called `modifies-clause` (for methods only). The function or method head is written in standard Java notation:

```
   public type name (parameter-list)
```
4. Code Inspection

For functions the type describes the type of the returned value. For methods the type is void. The name is the name of the function or method, and parameter-list is a comma-separated list of formal parameters (type and name).

Formal parameters may be defined to be constant by writing the key-word const in a Compass comment in front of the parameter's type. A constant parameter is assumed to remain completely unchanged during the execution of the function or method. However, we do not verify that constant parameters actually remain unchanged. If the constant parameter is a compound object then it is assumed that no subcomponent is changed either. Constant parameters simplify the specification of assertions. Without constant parameters it may be necessary to explicitly state that some parameters remain unchanged, i.e. that their current value always equal their initial value. The problem that it in general is not sufficient to specify what a program does change but also what it does not change is known as the frame problem. For example, that a car car has reached a certain speed s after accelerating could be expressed like this: speed(car) = s. However, methods which ensure that this condition is satisfied after their execution may also turn on the car's head lights. Intuitively, we would want that only those variables necessary to establish the condition are changed. Therefore, it is not enough to specify only what changes but also what does not. Several approaches to deal with this problem have been suggested (see e.g. [7] and [14]). We tackle the frame problem with constant declarations, the modifies-clause introduced later, and by explicitly stating in predicates which variables do not change.

Example 4.2

The following could be the head of a function that searches for the first position of an element of array a with the same value as x:

```java
public int search(/*+const*/ int x, /*+const*/ int[] a)
```

Since both x and a are constants it is assumed that they do not change their value during the execution of the body, i.e. it is assumed that x = x@pre and a = a@pre always are valid. This means that neither a itself nor any of its elements are modified during the execution of the method’s body.

In a COMPASS comment following directly after the head of the function or the method, the precondition (requires-clause), the postcondition (ensures-clause) and a list of variables which are modified by the method (modifies-clause) are given. The precondition and postcondition have the following forms:

```java
requires
    Specification of method precondition ;

ensures
```

24
Both the method precondition and postcondition are formulae as described earlier in Chap. 3.

The precondition expresses a condition that is supposed to be satisfied before the execution of the body of the method. Thus, the formula is only allowed to contain logical variables, namely the ones referring to the initial values of the parameters (and the object).

The postcondition, on the other hand, expresses a condition on the program variables that is supposed to be satisfied when the method invocation returns. That is, the assertion expresses what the method is intended to accomplish. If the method actually is a function, the postcondition always has the form result = f, where result denotes the return value of the function and f represents the (formally or informally defined) function that is performed by the method.

Example 4.3
In a class Computer we may have a function uptime that returns the time the computer has been continuously on. A corresponding interface specification may look like this:

```java
public Time uptime()
/**
   * ensures
   * result = uptime.this@pre;
   */
```

The function uptime is defined elsewhere in a COMPASS code comment. For example, an informal definition could look like this:

```java
define uptime(computer) informally
   The function returns the time since the last boot of the computer “computer” in seconds.
end define;
```

The requires-clause and the ensures-clause are followed by the modifies-clause which has the following form:

```java
modifies
   variable-list;
```

The variable-list is a comma separated list of the (program) variables which are modified by the method.
4. Code Inspection

The modifies-clause describes the variables that may be altered due to execution of the method’s body. The variable list may only contain formal parameters and/or a reference to the object itself using this. Only elements of a structured type may be modified by a method, since only structured types are handled by reference in Java. Objects and arrays are structured types. Primitive types are handled by value, i.e. the actual values are passed to methods, and changes to these values are not reflected in the actual arguments. The variable this has to appear in the variable list when class variables may be modified by the method (see Sect. 4.3.1 and Sect. 4.1.1). Finally, functions do not have a modifies-clause, since they are not allowed to modify the arguments or the state of the object (see Sect. 4.1.1).

Any of the three clauses may be omitted. If the precondition is omitted then it is assumed to be true. If the postcondition is omitted it defaults to true as well. The modifies-clause defaults to the empty list, i.e., it is assumed that the method is neither altering any parameters nor the object itself.

Example 4.4
An interface specification for a method that sorts an array between a left and a right index may look as follows:

```java
public void sort(int[] v, /*const*/ int left, /*const*/ int right)
    /*ensures*/
    sorted(v@pre, v, left@pre, right@pre);
    modifies v;
    */
```

The requires-clause is omitted since the method has no restrictions on the state prior to the execution of its body (except of course type correctness which is checked by the compiler).

4.1.4. Other Assertions

Apart from the assertions needed to specify the class invariant and the method interface, two other assertions, namely loop invariants and intermediate assertions, may appear within the code in a method’s body. Loop invariants are specified like this:

```java
maintains Specification of loop invariant;
```

A loop invariant is a formula as described in Chap. 3. It expresses a condition on the state of the computation which is satisfied after each execution of the body of the loop,
assuming that it is satisfied before the execution. More about loop invariants may be found in Sect. 4.2 and Sect. 4.3.2. Loop invariants have to be provided for the generation of the verification conditions.

Example 4.5
A program that given x and array a determines the first occurrence of x in a (it is assumed that such an element exists) may be implemented like this:

```java
while(a[i] ≠ x)
    /*+ maintains
    inSecondPart(x@pre, a@pre, i); */
    i = i + 1;
```

The invariant expresses that the element x@pre is not among the first i – 1 elements of a@pre but among the rest of the elements (i.e., it is in the second part of the array).

The other additional assertions that may be specified along with the code in a method’s body are intermediate assertions. Intermediate assertions look like this:

```java
ensures
    Specification of intermediate assertion ;
```

An intermediate assertion is a formula as explained in Chap. 3. It is used to express a condition which is expected to be satisfied each time execution reaches the point of the assertion.

Example 4.6
The following piece of code may be part of an implementation of the quicksort sorting algorithm:

```java
split(v, left, right, i);
    /*+ ensures
    partition(v@pre, v, left@pre, right@pre, i); */
    */
    quicksort(v, left, i.val() – 1);
    quicksort(v, i.val() + 1, right);
```

The intermediate assertion states that, after the invocation of the method split, the array v is partitioned in a particular way (namely, all elements to the left of i are less than or equal to v[i] and all elements to the right of i are greater than or equal to v[i]).
4. Code Inspection

4.2. Program Development

The first phase of the COMPASS method consists of the development of code with embedded assertions. In 1976 Dijkstra suggested a discipline for developing code with assertions along with arguments for its correctness (see [29] and [28]). The development method that is part of our COMPASS method is the same as proposed by Dijkstra except for the informal predicates that are allowed to occur in COMPASS assertions. Since the program development method is described at length in the literature (see e.g. [46] and [5]) only a brief overview will be given here.

Programming is considered a goal-oriented activity, i.e., the desired result (postcondition) plays a more important role than the precondition. Therefore, before trying to solve a problem, one should make oneself confident with the problem and develop the corresponding pre- and postconditions. Then, given the postcondition (and precondition), the aim is to develop a program that terminates in a state satisfying the postcondition.

The correctness of such refinement steps from a specification to a program have been studied by Back (see e.g. [2] and [3]).

Two common building blocks for a program are the conditional statement and the iterative statement:

- To invent a conditional statement (e.g. an if or switch statement in Java), a statement $S$ has to be found, that establishes the desired postcondition $Q$ in at least some cases. A boolean expression that is a precondition for the command $S$ and postcondition $Q$ may be used as a guard (i.e., as the condition when to execute the corresponding statement) for the conditional statement. This process has to be continued until the precondition implies that at least one guard (or the default case) is true.

- Given pre-, postcondition, and a loop invariant, an iterative statement may be developed as follows:
  1. The loop invariant has to be established before the first execution of the loop by appropriately initializing the involved variables.
  2. The guard must be developed. A boolean expression whose negation in conjunction with the loop invariant implies the postcondition may be used as the guard.
  3. Finally, the loop body is developed in such a way that it improves towards termination while reestablishing the loop invariant.

The problem, how to discover a loop invariant remains. However, in general one already has a certain algorithm in mind when developing a program. Writing down
the loop invariant then should be rather simple and is a good way of documenting
the idea underlying the algorithm.

The development method is presented here as described in Gries (see [46]). However,
since it is not required that all predicates are formally defined assertions can be expressed
at a higher level of abstraction.

Example 4.7
In this example it will be explained how a program may be developed with the develop-
ment method described in this section. The program to be developed is an implementa-
tion of the sorting algorithm quicksort. Although well-known by now, it still serves as a
good candidate for illustrating some of the key issues of the COMPASS method.

Quicksort takes as input an array to be sorted, picks out a splitting element, splits
the array in two subarrays, containing those elements less than or equal to respectively
greater than or equal to the splitting element. It then recursively sorts the two subarrays.

The starting point is to provide an interface specification as described in Sect. 4.1.3.
There are no requirements on the state prior to the execution of quicksort (besides that
all arguments are of the correct type). Thus, the requires-clause may be omitted. After
the termination of quicksort the array parameter is sorted. The array v is modified by
the method whereas left and right are constant:

```java
public void quicksort(int[] v, /*const*/ int left, /*const*/ int right)
    /** ensures
        sorted(v@pre, v, left@pre, right@pre);
     * modifies
       v;
    */
```

The predicate symbol sorted is a typical example of an informal predicate, having only
an informal definition. The relation expressed by sorted(a, b, left, right) is, that b is
sorted in increasing order in the interval [left, right], and that a and b are permutations
in this interval while identical outside. The following definition is provided in a special
code comment:

```plaintext
define sorted(a, b, left, right) informally
    1. “b” is sorted in increasing order in between “left” and “right”.
    2. “a” and “b” are permutations in the interval [“left”, “right”], i.e. contain
       the same elements in this interval.
    3. “a” and “b” are identical outside [“left”, “right”]
end define;
```

29
If the array segment contains only one or no element at all the postcondition is established right away, i.e. the array segment is already sorted. Only if the array segment contains more than one element it has to be sorted:

```java
public void quicksort(int[] v, /*+const*/ int left, /*+const*/ int right)
/** ensures */
    sorted(v@pre, v, left@pre, right@pre);
/modifies v;
*/
{
    if(left < right)
        /* sort v inbetween left and right */
}
```

As mentioned, quicksort sorts by first splitting the array around a splitting element, and then recursively sorting the two array segments. Therefore an informal predicate symbol partition is introduced. This predicate states that the array is split around a certain index i:

```java
define partition(a, b, left, right, i) informally
    1. "b"["left"], . . . , "b"[value("i")], . . . , "b"["right"]
       \leq "b"[value("i")]
       \geq "b"[value("i")]
    2. "a" and "b" are permutations in the range ["left", "right"].
    3. "a" and "b" are identical outside ["left", "right"]
end define ;
```

Assuming a method split which does the actual job of splitting the array is given, the complete method quicksort may look as follows:

```java
public void quicksort(int[] v, /*+const*/ int left, /*+const*/ int right)
/** ensures */
    sorted(v@pre, v, left@pre, right@pre);
/modifies v;
*/
{
    WrapInt i = new WrapInt(0);
    if(left < right)
        { split(v, left, right, i);
          quicksort(v, left, i.val() - 1);
          quicksort(v, i.val() + 1, right);
        }
```
4.3. The Generation of Questions

The idea is that split splits the array $v$ and returns the splitting position in $i$. However, our prototype tool does not allow a method that both modifies its parameters and returns a value with the return statement (see Sect. 4.1.1). Hence the splitting position has to be returned through a parameter. However, basic types are in Java always passed by value. Thus split is called with a wrapper around an integer to enable returning a value.

The interface specification of split may look as follows:

```java
public void split(int[] v, /*const*/ int left, /*const*/ int right, WrapInt i)
/**
 * requires
 * left@pre \leq \text{right}@pre ;
 * ensures
 * partition(v@pre, v, left@pre, right@pre, i) ;
 * modifies
 * v, i ;
 */
```

Now the method split would have to be developed in a manner similar to that used to develop quicksort.

### 4.3. The Generation of Questions

The second phase of the COMPASS method consists of the generation of a set of verification conditions. As mentioned earlier, this generation presupposes the existence of proof rules (semantics) of the programming language, in this case for Java. We start in Sect. 4.3.1 with a discussion of the prerequisites and limitations of the question generation. In Sect. 4.3.2 we present an axiomatic semantics for our subset of Java. Finally, in Sect. 4.3.10 we explain how the questions actually are generated from an annotated piece of code.

#### 4.3.1. Prerequisites and Limitations

The question generation depends on the type of properties that we want to verify and also on the programming language. We mentioned earlier (see the introduction to Chap. 4 and to Sect. 4.1), that our focus is on functional properties of the code written in a subset (see Sect. 4.1.1) of the Java programming language.
The generation of questions is independent from the meaning of the (user-defined) predicates and functions in assertions. It relies solely on the existence of formal proof rules (i.e., an axiomatic semantics) for the programming language. Thus, our approach inherits some of the problems of the completely formal approach, namely sharing, dynamic binding, overriding, and implementation independence. However, the aim of this work has not been to overcome all the challenges related to formal specification and verification of object-oriented programs; COMPASS should be considered an experiment to assess whether relaxing the requirements on formal rigor yields a method which aids verification in a similar way as a formal method, but which is easier to use.

The difficulties related to the formal specification and verification of object-oriented programs are briefly presented in the following sections. First we discuss open issues related to overriding. Then we discuss dynamic binding and implementation-independence. Finally we address challenges due to sharing. Further information on these topics can be found e.g. in [97]. In the PhD thesis by Poetzsch-Heffter a detailed description of different approaches to the specification and verification of object-oriented programs can be found. The thesis contains a formal framework for interface specifications and programming language semantics.

### Overriding

Java, like most object-oriented programming languages, allows the overriding of methods. That is, if a class declares a method then the declaration of that method overrides all methods with the same signature (i.e., the same name, return type, and number and type of formal parameters) in the more general classes.

Conceptually, a subclass should be a subtype of the superclass. This principle is called the Liskov substitution principle (Ivar Jacobson calls it subtyping – see [59]). It was first introduced by Barbara Liskov in 1988 (see [75]):

> If for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$ then $S$ is a subtype of $T$.

In other words, when an instance of a more general class is replaced with an instance of a more specialized class, then the program still has to work correctly. This principle is strongly related to the concept of Design by Contract by Bertrand Meyer (see [84]) and gives rise to additional proof obligations:

- The invariant (see Sect. 4.1.2) of the more specialized class has to imply the invariant of the more general class.
4.3. The Generation of Questions

- The precondition of each method in the more general class has to imply the precondition of the same method in the more specialized class and the postcondition of each method in the more specialized class is to imply the postcondition of the same method in the more general class.

The above requirements are informally fairly clear. There are however some difficulties. Even for equivalent specifications it may be impossible to prove the above constraints. For example, let us assume that the precondition of a method \( a \) is \( P \) and the postcondition of \( a \) is \( Q \). Let us further assume that the precondition of a method \( b \) is \( \text{true} \) and the postcondition of \( a \) is \( P \implies Q \). When it comes to partial correctness then the two methods \( a \) and \( b \) are clearly equivalent. However, if the method \( a \) in the more general class is overridden by the method \( b \) in the more specialized class then \( P \) has to imply \( \text{true} \) and \( P \implies Q \) has to imply \( Q \). This is obviously not the case.

Dynamic Binding

Another aspect of object-oriented programming languages that is related to overriding is dynamic (or late) binding. That is, which method is executed when a method is invoked is not statically but dynamically determined. As a result, a method that invokes another method would have to be verified with respect to all methods that possibly could be executed at the point of the invocation. In principle, this leads to more verification conditions. However, we may exploit the fact that more specialized classes are subtypes of their more general classes (i.e., that the Liskov substitution principle is honored). In that case we only have to show that the strongest precondition (i.e., the precondition of the method in the topmost class) is satisfied when the method is invoked and we may only assume that the weakest postcondition (i.e., the postcondition of the method in the topmost class) is valid after the method’s execution. The topmost class (which can be determined statically) is the most general class whose instances may be accessed with the method invocation.

Implementation-Independence

Another difficulty is related to the scope of variables. Private class variables (class variables are also called attributes) are visible to methods of that class and could thus be used in interface specifications. However, private class variables are not visible at the point of the method invocation. Thus using class variables in interface specifications makes the specification implementation dependent. The problems can be avoided by using a reference to the whole object (usually this) and additional informal predicates (functions) or an abstract model of the class in interface specifications.
4. Code Inspection

Sharing (or Aliasing)

Sharing is present if two or more variables refer to the same object. This phenomenon is also referred to as aliasing. The problem is that an object or a variable may change its value without being explicitly referred to. This change might be missed when the verification conditions are generated in which case the conditions probably are erroneous (i.e., the program could be incorrect even though this is not reflected in the verification conditions). For example, if two objects \( a \) and \( b \) share a common object \( x \) then a modification of \( x \) initiated from \( a \) may leave the class invariant of \( b \) violated. This problem even occurs when it is known that the two objects share a common object.

To overcome the problems with aliasing one may ask the developer to specify which variables may be aliased. For some applications it also may be reasonable to forbid aliasing completely since it is a common source of errors. It is also possible (but cumbersome) to extend Hoare Logic to correctly handle aliasing (see [20]).

4.3.2. An Axiomatic Semantics

The questions to be addressed during our semantic inspection are generated with an inference system. The inference system specifies an axiomatic semantics. It consists of a set of axioms and rules. The rules are typically written in the form:

\[
E_1 \quad E_2 \quad \cdots \quad E_n \quad \frac{}{E}
\]

The above rule states that, if the formulae \( E_1, E_2, \ldots, E_n \) are satisfied, then so is the formula \( E \) (an axiom is a rule where \( n = 0 \)). The \( E_i \) and \( E \) are formulæ of the inference system. In our case they are either well-formed formulæ as described in Chap. 3 or they are triples of the form \( \{P\} S \{Q\} \) where \( S \) is a statement in the Java programming language and where \( P \) and \( Q \) are well-formed formulæ. The triples (which are usually called Hoare Triples) are to be read:

If executing \( S \) in a state where \( P \) is satisfied terminates then \( Q \) is satisfied upon termination.

The statement \( S \) then is said to be partially correct (because we do not know whether the execution terminates due to the iterative statement and recursion).

In the following, a simplified subset of the axiomatic semantics used by the prototype tool (see Sect. 4.4) is introduced. The axiomatic semantics as presented here is inspired by the one found in Hoare’s and Wirth’s “An Axiomatic Definition of the Programming Language PASCAL” (see [56]) and also of the one in Gries’ “The Science of Programming” (see [46]). It should be noted, that our main goal is to demonstrate a new method
4.3. The Generation of Questions

for the inspection of Java code. It is not our primary goal to provide a precise semantic
description of Java.

The Assignment Statement

A very important construct of any imperative programming language is the assignment
statement. The effect of an assignment is that variables in a formula are substituted
with other variables. Therefore we introduce the notation $P_{\bar{v}}$ where $P$ is a formula, \(\bar{v}\) is a
vector of variables $v_1, v_2, \ldots, v_n$, and \(\bar{e}\) is a vector of matching expressions $e_1, e_2, \ldots, e_n$.
$P_{\bar{v}}$ denotes the formula obtained by simultaneously substituting in $P$ all free occurrences
of the variables $\bar{v}$ (since we do not support quantifiers in well-formed formulae there are
only free variables) with the corresponding expressions $\bar{e}$.

4.1 Axiom (Assignment statement)

Let $x$ be a simple variable (i.e., no array element variable or class attribute variable)
and $e$ an expression of the same type. Executing $x = e$ in a state where $P_{\bar{v}}$ is satisfied
results in a state where $P$ is satisfied:

$$\{ P_{\bar{v}} \} x = e \{ P \}$$

Arrays are treated as functions (see e.g. [46]). An assignment to an array element,
a[i] = e, is thus interpreted as the assignment of a new function to a. That new function
is the same as the original one, except for the argument $i$, where it now has the value $e$.
The new function is written as $\langle a; i : e \rangle$. Thus, executing a[i] = e in a state where
$P_{(a, i, e)}$ is satisfied results in a state where $P$ is satisfied:

$$\{ P_{(a, i, e)} \} a[i] = e \{ P \}$$

Similar to arrays, class variables are treated as arguments for the function this.
An assignment to a class variable $c = e$ is thus interpreted as an assignment of a new
function to this. That new function is the same as the original one, except for the
argument $c$, where it has the value $e$. That is, executing $c = e$ in a state where $P_{\text{this}}$
is satisfied results in a state where $P$ is satisfied:

$$\{ P_{\text{this}} \} c = e \{ P \}$$

The assignment axioms are not actually axioms but axiom schemes, generating separate
axioms for each choice of the predicate $P$. 

35
4. Code Inspection

The Sequential Composition

Complex statements are constructed by sequential composition of simpler statements. Therefore a corresponding inference rule is needed.

4.2 Rule (Sequential composition)
If executing $S_1$ in a state where $P$ is satisfied results in a state where $R$ is satisfied, and if executing $S_2$ in a state where $R$ is satisfied results in a state where $Q$ is satisfied, then executing $S_1; S_2$ in a state where $P$ is satisfied must result in a state where $Q$ is satisfied:

$$\{P\} S_1 \{R\} \{R\} S_2 \{Q\} \quad \{P\} S_1; S_2 \{Q\}$$

The Conditional Statements

Java provides a set of conditional statements. Among these are the if-then-else statement (and the special form hereof, the if-then statement) which is included in our Java subset. Other conditional statements (like the switch statement) are not supported. However, there are no principal difficulties to handle them.

4.3 Rule (Conditional statement 1)
If executing $S_1$ in a state where $P$ is satisfied, and executing $S_2$ in a state where $R$ is satisfied, both result in a state where $Q$ is satisfied, then executing if($B$) $S_1$ else $S_2$ in a state where $(B \implies P) \land (\neg B \implies R)$ is satisfied must result in a state where $Q$ is satisfied:

$$\{P\} S_1 \{Q\}, \{R\} S_2 \{Q\} \quad \{(B \implies P) \land (\neg B \implies R)\} \text{if}(B) S_1 \text{else} S_2 \{Q\}$$

4.4 Rule (Conditional statement 2)
If executing $S$ in a state where $P$ is satisfied results in a state where $Q$ is satisfied, then executing if($B$) $S$ in a state where $(B \implies P) \land (\neg B \implies Q)$ is satisfied must result in a state where $Q$ is satisfied:

$$\{P\} S \{Q\} \quad \{(B \implies P) \land (\neg B \implies Q)\} \text{if}(B) S \{Q\}$$

The Iterative Statements

Other important and rather complicated constructs of imperative programming languages are the iterative statements like the while statement. The while statement is included in our Java subset whereas other iterative statements (like the for loop and the repeat loop) are not supported. Again, there are no principal difficulties to handle them.
4.3 The Generation of Questions

4.5 Rule (Iterative statement)
Let \( P \) be the invariant of the loop \( \text{while}(B) \) \( S \). If executing \( S \) in a state where \( B \land P \) is satisfied results in a state where \( P \) is satisfied, then executing \( \text{while}(B) \) \( S \) in a state where \( P \) is satisfied must result in a state where \( \neg B \land P \) is satisfied:

\[
\begin{align*}
\{ B \land P \} & \quad S \quad \{ P \} \\
\{ P \} & \quad \text{while}(B) \quad S \quad \{ \neg B \land P \}
\end{align*}
\]

The Function Call

Methods that return a value, (i.e., that are not declared as \texttt{void}) are functions. As described in Sect. 4.1.1, we do not allow functions to modify variables apart from local ones.

Several kinds of statements cause Java expressions to become part of verification conditions (e.g., because of the substitution in the assignment statement). These Java expressions may contain function calls which are resolved first (i.e., before any other axiom or inference rule is applied). The function calls are substituted by the corresponding functions used in the postconditions. At the same time, the implicit parameters of the function call are turned into explicit parameters. Moreover, if a statement contains a function call it has to be proven that the precondition of the statement implies the precondition of that function.

Example 4.8
In Example 4.3 on page 25 we declared a function \texttt{uptime}. An assignment statement in a program may contain a call to that function:

\[ x = \texttt{compi.uptime()} ; \]

Now, instead of substituting all occurrences of \( x \) in the postcondition of the assignment statement with \texttt{compi.uptime()} they are substituted with \texttt{upTime(compi)} (from the postcondition of the interface specification of \texttt{uptime}).

Method Invocation

An important means of controlling abstraction is the method (difficulties and limitations related to methods have been described in Sect. 4.3.1 and Sect. 4.1.1). Methods may be invoked in the same class as they are declared. In that case an invocation looks as follows:

\[ \texttt{method-name (argument-list)} ; \]
A qualified name has to be used for the invocation of a method that is declared in a different class:

\[ \text{object-name} \cdot \text{method-name} (\text{argument-list}) ; \]

The object-name is the (possibly qualified) name of an object which provides the latter method, method-name is the name of that method, and argument-list is a comma-separated list of arguments to the method.

It is assumed that the interface specification of the method has the form that is given earlier in Sect. 4.1.3. Moreover, when generating the verification condition for a method \( m \) then it is assumed that each method \( n \) that is invoked by \( m \) is partially correct (i.e., \( \{ P \} b_n \{ Q \} \) is valid, where \( P \) is the precondition of \( n \), \( Q \) is the postcondition of \( n \), and \( b_n \) is the body of \( n \)).

### 4.6 Axiom (Method invocation)

Let \( P \) be the precondition and \( Q \) the postcondition of the method \( m \) of the object \( o \) of the class \( c \). Let \( C \) be the class invariant of \( c \). Let \( \vec{x}@pre \) be the vector of logical variables representing the initial values of all formal parameters followed by this@pre. Let \( \vec{y} \) be the vector of all formal parameters (and possibly this) which are modified (according to the modifies-clause). Let \( \vec{a} \) be the vector of all arguments \( \vec{v} \) and \( o \). Let \( \vec{b} \) be the vector of all arguments (and possibly \( o \)) which are modified (according to the modifies-clause). Executing \( o.m(\vec{v}) \) in a state where \( P_{\vec{x}@pre} \land (\forall \vec{u}: (Q \land C)_{\vec{y},\vec{a}@pre} \implies R_{\vec{b}}) \) is satisfied must result in a state where \( R \) is satisfied:

\[
\{ P_{\vec{x}@pre} \land (\forall \vec{u}: (Q \land C)_{\vec{y},\vec{a}@pre} \implies R_{\vec{b}}) \} o.m(\vec{v}) \{ R \}
\]

Axiom 4.6 essentially states that the precondition of a method must be satisfied when it is invoked (\( P \)), and that all valid results of the method (\( Q \land C \)) must imply whatever is supposed to be satisfied after the method invocation (\( R \)).

**Example 4.9**

Let us assume we have a method \text{flyHigher(height)} of a class \text{Plane}. The idea is that invoking the method makes a plane climb \text{height} units. The precondition \( P \) of the method is \text{height}>0 and the postcondition \( Q \) is \text{flightHeight(this)} = \text{flightHeight(this}@pre) + \text{height}. The function symbol \text{flightHeight} represents a user-defined function returning the flight height of an instance of the class \text{Plane}. The class invariant \( C \) is “true” and not considered.

We invoke the method with \text{plane.flyHigher(10)} and want to show that the condition \( R \), which is \text{flightHeight(plane)} = 100, is satisfied after the invocation.
4.3. The Generation of Questions

Axiom 4.6 tells us to replace height with 10 in $P$, this@pre with plane, plane with plane#1, and height with 10 in $Q$, and plane with plane#1 in $R$. We get that the following formula has to be satisfied before the invocation of the method:

$$10 > 0 \land (\forall \text{plane}#1 : \text{flightHeight}(\text{plane}#1) = \text{flightHeight}(\text{plane}) + 10 \implies \text{flightHeight}(\text{plane}#1) = 100)$$

In other words, the flight height of the plane before the method invocation has to be 90.

Consequence

The axioms and rules presented so far are still not sufficient for the generation of questions. For a program $S$ with the precondition $P$ and the postcondition $Q$ we are able to automatically derive a condition $R$ such that $\{R\} S \{Q\}$ holds. Our initial goal however was to show that $\{P\} S \{Q\}$ holds. Therefore we need an additional inference rule, namely the rule of consequence.

4.7 Rule (Consequence)

If executing $S$ in a state where $R$ is satisfied results in a state where $Q$ is satisfied and if $P \implies R$ then executing $S$ in a state where $P$ is satisfied also results in a state where $Q$ is satisfied:

$$\begin{align*}
\{R\} S \{Q\} & \quad P \implies R \\
\{P\} S \{Q\} & 
\end{align*}$$

In general, the formula $P \implies R$ contains insufficiently axiomatized predicates and can in general not be formally verified. Instead it constitutes the questions which are presented to and answered by the inspectors.

4.3.3. Applying the Semantics to Generate Verification Conditions

With the semantics presented in the previous section, it is possible to generate a set of questions from a Java program with embedded assertions. We allow predicates without associated formal definition to occur in those assertions. Since the generation of questions is independent of the meaning of the predicates it can still be carried out automatically. The questions form the basis of our semantic inspection approach. All questions stem from the formula $P \implies R$ in Rule 4.7. How the questions are presented to the inspectors and what answers are possible is explained in Chap. 6.
Example 4.10

This example is a continuation of Example 4.7. It shows how a number of questions is generated from the definition of quicksort and the interface specification of split. The interface specification of both methods can be found in Example 4.7 on page 29. The body of quicksort looks like this:

```java
WrapInt i = new WrapInt(0);
if(left < right)
{
    split(v, left, right, i);
    quicksort(v, left, i.val() - 1);
    quicksort(v, i.val() + 1, right);
}
```

The questions are generated by applying the axioms and inference rules presented in Sect. 4.3.2. We use the following abbreviations: left is abbreviated with l, right with r, i.val() with i, quicksort with q, and split with s. The inference tree looks as follows:

The formula that remains to be verified is $P \implies V$. Since the formula typically contains insufficiently axiomatized predicates it can in general not be formally proven. Instead it is presented to the inspectors as a set of questions. Here the remaining formula can easily and automatically (see Chap. 6) be rewritten as the following three questions (exploiting the fact that left and right are not modified by the code):

\[
\begin{align*}
\text{P} &\implies V \quad \{ V \} \quad \text{i=0} \quad \{ U \} \quad \{ S \} \quad q(v,i,i-1) \quad \{ R \} \quad q(v,i+1,r) \quad \{ Q \} \\
\{ T \} &\implies s(v,l,r,i) \quad \{ S \} \\
\{ T \} &\implies q(v,i,i-1) \quad \{ Q \} \\
\{ U \} &\implies \text{if}(l<r)(s(v,l,r,i); q(v,i,i-1); q(v,i+1,r)) \quad \{ Q \}
\end{align*}
\]

where:

\[
\begin{align*}
\text{P} &\text{ : true} \\
\text{Q} &\text{ : sorted(v@pre, v, l@pre, r@pre)} \\
\text{R} &\text{ : sorted(v, v#1, i + 1, r@pre) }\implies\text{sorted(v@pre, v#1, l@pre, r@pre)} \\
\text{S} &\text{ : sorted(v, v#2, l@pre, i - 1) }\implies\text{Rv#2} \\
\text{T} &\text{ : l@pre }\leq\text{ r@pre }\land\text{ (partition(v, v#3, l@pre, r@pre, i#1) }\implies\text{Sv#3,i#1)} \\
\text{U} &\text{ : (l@pre }<\text{ r@pre }\implies\text{T) }\land\text{ (l@pre }\geq\text{ r@pre }\implies\text{Q)} } \\
\text{V} &\text{ : U0i} \\
\end{align*}
\]

The formula that remains to be verified is $P \implies V$. Since the formula typically contains insufficiently axiomatized predicates it can in general not be formally proven. Instead it is presented to the inspectors as a set of questions. Here the remaining formula can easily and automatically (see Chap. 6) be rewritten as the following three questions (exploiting the fact that left and right are not modified by the code):
4.4. A Prototype Tool

Question 1

Assume:
1. \text{left@pre} < \text{right@pre}

Then:
1. \text{left@pre} \leq \text{right@pre}

Is the conclusion satisfied?

Question 2

Assume:
1. \text{left@pre} < \text{right@pre}
2. \text{partition(v@pre, v#3, left@pre, right@pre, i#1)}
3. \text{sorted(v#3, v#2, left@pre, value(i#1) - 1)}
4. \text{sorted(v#2, v#1, value(i#1) + 1, right@pre)}

Then:
1. \text{sorted(v@pre, v#1, left@pre, right@pre)}

Is the conclusion satisfied?

Question 3

Assume:
1. \text{left@pre} \geq \text{right@pre}

Then:
1. \text{sorted(v@pre, v@pre, left@pre, right@pre)}

Is the conclusion satisfied?

Question 1 may be answered automatically. Question 2 and 3 however require an understanding of the informal predicates. Thus, the question may not be answered automatically. How the questions are presented and answered during a code inspection session is described in detail later in Chap. 6.

4.4. A Prototype Tool

A prototype tool that supports our semantic code inspection has been implemented in SICStus Prolog version 3 #5 (see [112]). The tool generates the questions to be asked during code inspection. The Prolog programming language has been chosen especially for its support of definite clause grammars. Such grammars are automatically transformed into parsers for the language specified. Moreover, it is rather easy to implement the
4. Code Inspection

symbolic manipulations required to generate the verification conditions in Prolog (see e.g. [23] and [110]).

It should be noted that the tool is just a prototype. The tool is rather limited in its capabilities and it supports only a small subset of the Java programming language (as presented in Sect. 4.1.1). Suggestions for extensions of the prototype tool may be found in Sect. 4.4.3.

4.4.1. Architecture

The architecture of the prototype tool is similar to that of a multi-pass compiler and is shown in Fig. 4.2. The prototype tool consists of a scanner, a parser, and a question generator. The scanner reads a file containing a Java class with assertions as a stream of characters and produces a list of tokens (including information on their location in the source file). The parser constructs a parse tree from the list of tokens. Both the scanner and parser are implemented with the help of definite clause grammars. The parse tree is used to generate the verification conditions as outlined in Sect. 4.3. Finally, the conditions are simplified if possible. The resulting list of verification conditions is then, one by one, presented to and validated by the inspector.

4.4.2. Using the Prototype

The user interface of the prototype is given by the Prolog top loop itself. Commands to load Java classes and to inspect all or a single method of them are available after the prototype has been loaded.

Loading Classes

Before the verification conditions can be generated all involved Java classes have to be loaded into the Prolog database. This includes the classes to be inspected as well as
4.4. A Prototype Tool

the classes used indirectly. It is however sufficient for the generation of the conditions if
the latter classes contain “empty” methods with their interface specifications only. The
command used to load classes is:

\texttt{loadClass(file-name)}.

Where \textit{file-name} is the name of the file with extension containing the corresponding Java
class.

If the Java class is syntactically correct (with respect to the restrictions imposed by the
prototype tool) the system indicates success, e.g.:

\texttt{Loading TableGenerator.java . . . done.}

If the code is not syntactically correct the system responds with an error message and
an excerpt of the code which contains the error, e.g.:

\texttt{Loading TableGenerator.java . . .

Couldn’t parse. . .

157 if (!overlap) excl_table.allowRoutes(route1, route2);
158 route2 = route2 + 1}

After all required classes have been loaded the verification conditions, i.e. the questions
relevant to establish the correctness of the code (with respect to the assertions), can be
generated.

**Inspecting Classes and Methods**

It is possible to inspect all or individual methods of a class that has been loaded. The
command to inspect all methods of a class is:

\texttt{inspect(class-name)}.

Where \textit{class-name} is the name of the class whose methods are to be inspected. The
corresponding command to inspect only a single method is:

\texttt{inspect(class-name, method-name)}.

Where \textit{method-name} is the name of the method of the class \textit{class-name} that is to be
inspected.

In both cases the system will respond with a number of questions relevant to establish
the correctness of the corresponding method or methods. The questions are introduced
4. Code Inspection

with a consecutive number and the name of the method and class. Each question itself consists of the premise (preceded by Assume) and the conclusion (preceded by Then). Finally, the system asks if it is correct that the premise implies the conclusion. For example:

VC-1 generateExclusions :: TableGenerator

Assume:
1. emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))

Then:
1. partMatchExclTab(ltab@pre, etab@pre, etab@pre, 1, 1)

Is the conclusion satisfied?

Now the system waits for the inspector to answer the question with “yes” or “no”. To support the inspector the system is able to provide function and predicate definitions. The inspector can get a definition by typing the name of the corresponding function or predicate symbol whenever the system waits for an input. For example:

Is the conclusion satisfied? partMatchExclTab

partMatchLockTab(topo, ltab1, ltab2, track)

Locking table “ltab2” is the same as locking table “ltab1” except that all train routes starting from those of the first “track” – 1 track segments in “topo” that are connected to only one point have been added (as long as they fit).

Is the conclusion satisfied?

If the inspector answers “yes” to all questions generated for a method, then the system will respond accordingly, e.g.:

Method “generateExclusions::TableGenerator” is correct.

If the inspector answers no to a question, then either the assertions or the code is incorrect. The system does not ask further questions regarding the method currently under investigation. Instead information about the assertions and the code used to generate the last question is given, e.g.:

Method “generateExclusions::TableGenerator” is incorrect.

Class inv. and precond. of method “generateExclusions” (line 112)
Invocation of method “initExclTable” (line 126)
Postconditions of method invocations in assignment (line 127)
Invariant of while loop (line 128)
4.5. Example

Given this information the code and assertions can easily be checked to locate and correct the defect, e.g. above output indicates that the invariant of a while loop (found in line 128) is initially not satisfied.

4.4.3. Possible Extensions to the Prototype

The prototype tool may be extended in several ways. The following list contains some suggestions for possible improvements which could rather easily be realized:

- The assertions and definitions should be checked for consistency. For example, it could be checked that only logical and visible program variables appear in assertions and that all function and predicate symbols used in the assertions are defined.

- Parameters to predicates could be specified with their type. This would result in further possibilities to check the consistency of the program and the assertions. In addition types would enable for overloading of definitions.

- Several classes may provide similar methods with equal names. However, the predicates in assertions must be different for all classes. Methods with the same names but different classes are distinguished by qualified names. Similar techniques could be used for the predicates.

- The user interface could be improved in several ways. For example, each question could automatically be presented together with the corresponding code, and function and predicate definitions.

- If some additional notation is used to represent the formal parameters within informal definitions then the formal parameters could be replaced with the actual parameters (for a more thorough discussion see Sect. 5.5).

- The prototype may be extended to cover a larger portion of the Java programming language, especially those constructs which do not cause principal difficulties, e.g. for and repeat loops (see also Sect. 4.3.1).

Suggestions for more general improvements of the COMPASS method itself rather than of the prototype are discussed in Chap. 8.

4.5. Example

In this section we demonstrate how the COMPASS method can be used to develop and inspect a small, but non-trivial program. The program to be developed and inspected is
4. Code Inspection

supposed to generate so called locking and exclusion tables. These tables may be used in a system that controls the movement of trains in a railway station (a so called railway interlocking system).

4.5.1. Railway Interlocking Systems

Systems to control train traffic have been built since a long time. With the increase of tracks it became necessary to control trains within the railway stations (for a simple model of a railway station with 17 track segments and 9 points see Fig. 4.3). Therefore interlocking systems have been constructed. The interlocking systems control the train movements within the railway station. In the beginning these systems were entirely mechanical. They developed then into electro-mechanical and later into relay based systems. Nowadays, interlocking systems are often computer controlled.

The main task of an interlocking system is to prevent trains from colliding and derailing while at the same time allowing them to move. To fulfill this task the system observes the states of track segments (e.g. empty or occupied), points (e.g. in bent or straight position), and signals (e.g. drive or halt). Based on the states the system controls the points and signals.

Typically, interlocking systems are based on the notion of train routes. A train route is a set of track elements (i.e., track segments and points) which is reserved for a particular train and enables the train to move within the railway station. For example, the track segments 1, 2, and 3 together with the points 1 and 9 form a train route. It is assumed that each train has at most one train route. Given the notion of train routes a collision of two trains is possible only if at least one of the trains is not on its train route or if both trains are on their train route but these train routes lead to a common track segment. A more detailed description of interlocking systems and their safety requirements may be found e.g. in [49].
4.5. Example

Table 4.1
Part of a locking table

<table>
<thead>
<tr>
<th>Route</th>
<th>Track segments</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&gt; &lt;&lt;</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>&gt; &lt;</td>
<td>− − +  +</td>
</tr>
<tr>
<td>3</td>
<td>&gt; &lt; &lt; &lt; &lt;</td>
<td>− − − + +</td>
</tr>
<tr>
<td>4</td>
<td>&gt; &lt; &lt; &lt; &lt; &lt;</td>
<td>− − +  − −</td>
</tr>
<tr>
<td>5</td>
<td>&gt; &lt; &lt; &lt; &lt; &lt; &lt;</td>
<td>− − − − + − − +</td>
</tr>
<tr>
<td>6</td>
<td>&gt; &lt; &lt; &lt; &lt; &lt; &lt;</td>
<td>− − − − + − − −</td>
</tr>
<tr>
<td>7</td>
<td>&gt; &gt;</td>
<td>+ +</td>
</tr>
<tr>
<td>8</td>
<td>&lt; &gt; &gt;</td>
<td>+ +</td>
</tr>
</tbody>
</table>

Table 4.2
Part of an exclusion table

<table>
<thead>
<tr>
<th>Route</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34</td>
</tr>
<tr>
<td>2</td>
<td>x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>3</td>
<td>x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>4</td>
<td>x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>5</td>
<td>x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x</td>
</tr>
</tbody>
</table>

A particular interlocking system for a railway station may be realized with the help of two special tables. One table, the so-called locking table, describes each train route by listing the track elements that are part of it (and which thus have to be locked if the train route is to be used). Table 4.1 shows a part of the locking table for the railway station of Fig. 4.3 on the facing page. A “+” indicates that the point is used in the straight position and a “−” that the point is used in the bent position. The symbols “<” and “>” indicate in which direction the corresponding track segment is used. An “>” indicates that the corresponding track segment is used as given by the arrow in Fig. 4.3 on the preceding page and an “<” indicates that it is used in the opposite direction.

The other table is called the exclusion table and it describes which routes may be used concurrently, i.e. which pairs of routes do not lead to a common track segment. Table 4.2 shows a part of the exclusion table for the railway station of Fig. 4.3 on the facing page. An “x” in the table indicates that the corresponding train routes may not be used concurrently.

Given these two tables an interlocking system could easily be implemented. Regarding train routes, the interlocking system has to perform the following subtasks:
4. Code Inspection

1. Allocation and locking of a train route
   Check whether the requested train route may be used concurrently with the other already allocated train routes (according to the exclusion table). Allocate the track elements that are part of the train route (as listed in the locking table). Change the points and signals accordingly. Lock the track elements (i.e., prevent by some means that the elements are deallocated or manipulated by accident).

2. Monitoring of a train route
   Monitor the train route while in use.

3. Unlocking of a train route
   Unlock the track segments, points, and signals and deallocate them.

Of course the above procedure works safely only if the tables are correct. That is, if the track elements of each train route (according to the locking table) actually form a route and if no two train routes that are concurrently allowed (according to the exclusion table) actually lead to a common track segment.

4.5.2. Implementation of an Interlocking System

In this section we present an implementation of a program that given a model of a railway station topology (built only from those primitive elements in Fig. 4.3) computes corresponding locking and exclusion tables. However, since the aim of this example is to demonstrate the COMPASS method and not to produce a full scale implementation of the table generator, we introduce some simplifications:

- The station topology is modeled with track segments and points only (i.e., signals are not considered).
- Each train route is starting from and ending at a track segment that is connected to only one point, e.g. track segments 1, 3, 6, 10, 13, 16, and 17 in Fig. 4.3 on page 46.
- Each train route is an alternating sequence of connected track segments and points that are directly used by a train moving along the train route (without changing its direction).
- The locking table will be based on the notion of a train route as described above. It contains information about the track elements that are part of a train route and also in which state (i.e., position or direction) each track element has to be locked.
- The exclusion table will be based on the locking table as described above. Hence it will only disallow two train routes from being used concurrently if they directly share a track segment or point.
As a result of the above simplifications the resulting controller may be considered less safe. In this example the train route is the sequence of track elements a train actually uses, i.e., drives on. However, usually even other track elements are part of a train route e.g. point 2 (in plus position) could be part of train route 1-1-2-9-3 to protect the train route's flank (see Fig. 4.3 on page 46). Point 2 then could still be part of another train route that is used concurrently as long as it is used in the position necessary to protect the train route 1-1-2-9-3.

The implementation of the interlocking system is later inspected with the help of the prototype tool described in Sect. 4.4. Therefore, when developing the code the language restrictions described in Sect. 4.1.1 have to be taken into account.

The Overall Design

The design of a program as outlined previously is rather straightforward. The topology of the railway station, i.e. the basis for the computation of the tables, is modeled by the class `Topology`. The locking table and the exclusion table are each modeled by a class of their own, namely `LockTab` and `ExclTab`. Finally, the generator that is to compute the tables is implemented in the class `TableGen`.

Implementation of the Topology

The topology has to provide services to add track segments and points to the topology of a railway station. It must be possible to connect track segments and points with each other. Moreover the class has to provide functions to investigate how many track segments and points are part of the railway station and how they are interconnected.

The topology of the railway station is modeled in the following way: the track segments are numbered from 1 to \( k \) and the points from 1 to \( n \). Each track segment is connected to zero, one, or two different points and each point is connected to three different track segments. An example of a model of a railway station can be found in Fig. 4.3 on page 46. The above requirements on the interconnection of track segments and points can be expressed as an invariant of the class:

\[
\text{maintains validTopology(this)};
\]

The informal definition of the predicate symbol `validTopology` is supplied somewhere with the code:

\[
\text{define validTopology(topo) informally}
\]
4. Code Inspection

Each point in “topo” is connected to exactly three different track segments.
Each track segment is connected to zero, one, or two different points. The
number of points is zero or greater. The number of track segments is zero or
greater.

end define ;

The interface specifications and implementations of the methods are very simple and
can be found in Appendix A. The topology is implemented with two arrays: one to keep
the information for the track segments and the other one to keep the information for the
points.

Implementation of the Locking Table

The class LockTab models the locking table. It has to provide services to manipulate
a locking table by adding new train routes, by copying train routes, and by assigning
track segments and points to train routes. Moreover, methods to observe the state of a
locking table (e.g. which track segments and points are used for a certain train route)
are needed.

The locking table is modeled similarly to the topology of the railway station: the routes
are sequentially numbered starting with 1 and the track segments and points are identi-
fied by their number. Track segments can be assigned to train routes with a direction,
i.e. forward or backward, whereas points can be assigned to train routes with a position,
i.e. plus (for the straight branch) or minus (for the bent branch).

The interface specifications and implementations of the methods are simple and can
be found in Appendix B. The locking table is implemented with the help of two 2-
dimensional arrays: one for the information about the track segments and one for the
information about the points. Because we use arrays, which have a fixed size, it might
become impossible to add new train routes to the locking table because it is full.

Implementation of the Exclusion Table

The class that models the exclusion table has to provide services to allow train routes to
be used concurrently and to observe the state of instances, i.e. which train routes may
be used concurrently by two trains and how many train routes there are.

The interface specifications of the methods are simple. They can be found in Appendix C.
The exclusion table is implemented with help of a 2-dimensional array.
### Implementation of the Table Generator

The class `TableGen` contains the main functionality of this example, namely the computation of the locking and exclusion tables for a given topology of a railway station.

In the following paragraphs we describe the development of the method `generateLockings`. This method computes all possible train routes through a railway station given its topology. The first step in the development of the method, which has two formal parameters (namely, a topology and a locking table), is to write its interface specification:

- The topology is not changed during the execution of the method and is thus declared as constant. The locking table is modified by the execution of the method.
- The method is supposed to establish a certain relation between the topology, the initial, and the final locking table. The predicate symbol `matchLockTab` is used to represent this relation.
- Initially, it is required that the topology is valid in the sense described in this section and that the locking table is empty and has the same number of track segments and points as the topology.

The corresponding interface specification of the method `generateLockings` looks like this:

```java
public void generateLockings(/*+const*/ Topology topo, LockTab ltab)

/** requires
   validTopology(topo@pre) ∧
   emptyLockTab(ltab@pre, numTracksTopo(topo@pre),
   numPointsTopo(topo@pre), routes@pre);

   ensures
   matchLockTab(topo@pre, ltab@pre, ltab);

   modifies
   ltab;

*/
```

The predicate symbol `validTopology` was defined earlier. The definition of `emptyLockTab` is provided in the class `LockTab`:

```java
define emptyLockTab(itab, ntracks, npoints, nroutes) informally
  Locking table “ltab” is large enough to store at least “nroutes” train routes
  with at least “ntracks” track segments and at least “npoints” points for each
  train route. However, the table does not contain any train routes, track
  segments, or points.
end define;
```
4. Code Inspection

The definition of the predicate symbol `matchLockTab` is provided in class `TableGen` and looks as follows:

```plaintext
define matchLockTab(topo, ltab1, ltab2) informally
    Locking table “ltab2” is the same as locking table “ltab1” except that all (or as many as fit into the table) possible train routes through the station have been added. The track segments and points are given by the topology “topo”.
end define ;
```

The postcondition states that all possible train routes have been added to the locking table. All train routes start from track segments that are connected to only one point. Thus, we can use a loop over the track segments to construct the locking table. In each iteration the train routes starting from the current track segment are added to the locking table (if the track segment is connected to only one point and if the table is not full). If all track segments have been processed the postcondition will be satisfied. Thus, the program so far looks like this:

```plaintext
class TableGen {
    public void generateLockings(/**+const*/ Topology topo, LockTab ltab) {
        /**+ requires */
        validTopology(topo@pre) ∧
        emptyLockTab(ltab@pre, numTracksTopo(topo@pre), numPointsTopo(topo@pre), routes@pre) ;
        /**+ ensures */
        matchLockTab(topo@pre, ltab@pre, ltab) ;
        /**+ modifies */
        ltab ;
        */
        { /* + track = 1; */
            while (track < topo.numTracksTopo()+1) {
                /**+ maintains */
                partMatchLockTab(topo@pre, ltab@pre, ltab, track) ∧
                0 < track ∧ track ≤ numTracksTopo(topo@pre)+1 ;
                */
                { if ((topo.trackLeft(track) == 0 || topo.trackRight(track) == 0) &&
                    topo.trackLeft(track) != topo.trackRight(track) &&
                    ltab.routeEmpty()) /* add all routes starting from track segment “track” */
                    track = track + 1;
                }
            }
        }
    }
}
```

The definition of the predicate symbol partMatchLockTab is given in class TableGen and reads as follows:

```plaintext
define partMatchLockTab(topo, ltab1, ltab2, track) informally
    Locking table “ltab2” is the same as locking table “ltab1” except that all train
    routes (if any) starting from the first “track” − 1 track segments have been
    added (as long as the table is not full). The track segments and points are
    given by the topology “topo”.
end define;
```

Next, code has to be developed to add all train routes starting from the current track segment to the locking table (as long as the table is not full). Here we have chosen a recursive solution. It is assumed that a train route with no track segments or points assigned to it is given. Then this train route has to be completed, i.e. the current track segment and recursively other connected track segments and points have to be assigned to the train route until another track segment that is connected to only one point is reached. Each time a point is entered such that the train route can continue in two different ways, the train route so far has to be duplicated and both train routes then have to be completed for different positions of the point. If this is implemented in a method findFromTrack the complete method generateLockings looks like this:

```java
public void generateLockings(/*++ const */ Topology topo, LockTab ltab)
{ /*++ requires*/
    validTopology(topo@pre) ∧
    emptyLockTab(ltab@pre, numTracksTopo(topo@pre),
    numPointsTopo(topo@pre), routes@pre);
    /*++ ensures*/
    matchLockTab(topo@pre, ltab@pre, ltab);  
    /*++ modifies*/
    ltab;
    { int track = 1;
        while(track < topo.numTracksTopo() + 1)  
        { /*++ maintains*/
            partMatchLockTab(topo@pre, ltab@pre, ltab, track) ∧
            0 < track ∧ track ≤ numTracksTopo(topo@pre) + 1;  
            /*++ */
            if((topo.trackLeft(track) == 0 || topo.trackRight(track) == 0) &&
            topo.trackLeft(track) != topo.trackRight(track) &&
            ltab.routeEmpty())
        }
    }

53
```
4. Code Inspection

```java
ltab.incRoutesByOne();
findFromTrack(topo, ltab, ltab.numRoutesLockTab(), 0, track);
```

It is only meaningful to invoke the method `findFromTrack` if the topology is valid. Moreover, the locking table has to have the same number of track segments and points as the topology. This is expressed by the predicate symbol `matchTrackPoint`:

```java
define matchTrackPoint(topo, ltab) informally
    The topology "topo" and the locking table "ltab" have the same number of
    track segments and points.
end define;
```

Finally, the point, track, and route have to be within certain intervals. Thus, the interface specification of `findFromTrack` looks as follows:

```java
public void findFromTrack(/*+*/ const Topology topo, LockTab ltab, /*+*/ int route, /*+*/ int point, /*+*/ int track)
    requires
        validTopology(topo@pre) ∧ matchTrackPoint(topo@pre, ltab@pre) ∧
        0<route@pre ∧ route@pre ≤ numRoutesLockTab(ltab@pre) ∧
        0≤point@pre ∧ point@pre ≤ numPointsTopo(topo@pre) ∧
        0<track@pre ∧ track@pre ≤ numTracksTopo(topo@pre);
    ensures
        routesAddedTrack(topo@pre, ltab@pre, ltab, route@pre, point@pre, track@pre);
    modifies
        ltab;
    */
```

The definition of the predicate symbol `routesAddedTrack` is provided as a COMPASS comment and looks as follows:

```java
define routesAddedTrack(topo, ltab1, ltab2, route, point, track) informally
    – The train route "route" enables a train to move from a track segment that
      is connected with only one point to the point "point".
    – Locking table "ltab2" is the same as locking table "ltab1" except that the
      train route has been completed such that a train could continue from point
```

54
4.5. Example

"point" via track segment "track" to another segment that is connected with only one point.

Furthermore, all other train routes with the same beginning as train route "route" but different continuations have been appended to the locking table (as long as the number of train routes in the locking table is less than or equal to the maximal number possible).

Everything according to the topology "topo" of the railway station.

end define ;

The implementation of method findFromTrack and other methods used can be found in Appendix D.

Now, given a locking table describing all train routes through a railway station, a method that computes the corresponding exclusion table is developed. The exclusion table will describe which pairs of train routes may be used concurrently. The method is named generateExclusions. We start again by writing the interface specification of the method which has two formal parameters (namely, a locking table and an exclusion table):

- The locking table is not changed during the execution of the method and is thus declared to be constant. The exclusion table is of course modified.
- The method is to establish a certain relation between the locking table and the final exclusion table. The predicate symbol matchExclTab represents this relation.
- Before the execution of the method, the exclusion table has to be empty in the sense that no train routes are allowed to be used concurrently. Moreover, it has to have the same number of train routes as the locking table.

Thus, the corresponding interface specification for the method generateExclusions looks as follows:

public void generateExclusions(/*+const*/ LockTab ltab, ExclTab etab) 
/** requires 
   emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre)) ; 
   ensures 
   matchExclTab(ltab@pre, etab) ; 
   modifies 
   etab ; 
*/

The predicate symbol emptyExclTab is informally defined in the class ExclTab. The definition reads as follows:

define emptyExclTab(etab, nroutes) informally
4. Code Inspection

Exclusion table “etab” contains “nroutes” train routes. The number of train routes is zero or greater. No train routes are allowed to be used concurrently.

end define;

The definition of the predicate symbol matchExclTab is provided in the class TableGen and looks as follows:

define matchExclTab(ltab, etab1, etab2) informally
Exclusion table “etab2” is identical to “etab1” except that any two train routes that do not share a track segment or point according to locking table “ltab” may be used concurrently.
end define;

One way to establish the postcondition is to compare all train routes pairwise and to allow the concurrent use if the train routes do not share a track segment or point. A pairwise comparison can be implemented with two nested loops each over all train routes starting with the first. The invariants of the loops can be derived from the postcondition. Both invariants can be expressed with the same predicate symbol partMatchExclTab. The resulting code so far reads as follows:

public void generateExclusions(/**const*/ LockTab ltab, ExclTab etab)
/** requires */
emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre));
ensures
matchExclTab(ltab@pre, etab);
modifies
etab;
*/
{
int route_a = 1, route_b;
while(route_a < ltab.numRoutesLockTab() + 1)
/** maintains */
partMatchExclTab(ltab@pre, etab@pre, etab, route_a, 1) ∧
0 < route_a ∧ route_a ≤ numRoutesLockTab(ltab@pre) + 1;
*/
{
route_b = 1;
while(route_b < ltab.numRoutesLockTab() + 1)
/** maintains */
partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b) ∧
0 < route_a ∧ route_a < numRoutesLockTab(ltab@pre) + 1 ∧
0 < route_b ∧ route_b ≤ numRoutesLockTab(ltab@pre) + 1;
4.5. Example

The definition of the predicate symbol partMatchExclTab is provided in a Compass comment. It looks as follows:

```
define partMatchExclTab(ltab, etab1, etab2, route1, route2) informally
Exclusion table "etab2" is the same as "etab1" except that in "etab2" any train route X has been allowed to be used concurrently with any train route Y if and only if they do not share a track segment or point according to locking table "ltab". Moreover, X is either less than or equal to "route1" − 1 or Y is "route1" and Y is less than or equal to "route2" − 1.
end define;
```

The actual comparison to check whether the two train routes share a track segment or point is implemented in the method shareElems. The method does not modify any of its parameters, it just checks if two train routes share a track segment or point according to a locking table. The interface specification of the method looks as follows:

```
public boolean shareElems(/*+ const*/ LockTab ltab, /*+ const*/ int route_a, /*+ const*/ int route_b)
/**+ requires
  0 < route_a@pre ∧ route_a@pre < numRoutesLockTab(ltab@pre) + 1 ∧
  0 < route_b@pre ∧ route_b@pre < numRoutesLockTab(ltab@pre) + 1 ;
*/
ensures
  result = shareElems(ltab@pre, route_a@pre, route_b@pre) ;
```

The definition of the predicate symbol shareElems is provided as a comment within class TableGen and looks as follows:

```
define shareElems(ltab, route1, route2) informally
The train routes "route1" and "route2" share a track segment or point according to locking table "ltab".
end define;
```
4. Code Inspection

Given the method shareElems, the final version of the method generateExclusions looks as follows:

```java
public void generateExclusions(/*+const*/ LockTab ltab, ExclTab etab)
   /*+ requires */
   emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre));
   ensures
   matchExclTab(ltab@pre, etab);
   modifies
   etab;
/*
int route_a = 1, route_b;
while(route_a < ltab.numRoutesLockTab() + 1)
   /*+ maintains */
   partMatchExclTab(ltab@pre, etab@pre, etab, route_a, 1) ∧
   0 < route_a ∧ route_a ≤ numRoutesLockTab(ltab@pre) + 1;
/*
route_b = 1;
while(route_b < ltab.numRoutesLockTab() + 1)
   /*+ maintains */
   partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b) ∧
   0 < route_a ∧ route_a ≤ numRoutesLockTab(ltab@pre) + 1 ∧
   0 < route_b ∧ route_b ≤ numRoutesLockTab(ltab@pre) + 1;
*/
{ if(!shareElems(ltab, route_a, route_b))
   etab.allowRoutes(route_a, route_b);
   route_b = route_b + 1;
}
route_a = route_a + 1;
}
```

The implementation of method shareElems and other methods can be found in the class TableGen in Appendix D.

4.5.3. Inspection of the Table Generator

The implementations of the classes Topology, LockTab, and ExclTab are simple and so are the questions generated to be addressed during the code inspection. Therefore, we
will discuss here only the inspection of parts of the more complex, and hence more interesting, implementation of the class `TableGen`.

**Inspection of the Locking Table Generator**

The generation of the locking table is implemented with help of the three methods `generateLockings`, `findFromTrack`, and `findFromPoint`. The prototype tool generates the following 7 questions regarding the method `generateLockings`:

```
VC-1 generateLockings :: TableGen ________________________________
Assume:
  1. validTopology(topo@pre)
  2. emptyLockTab(ltab@pre, numTracksTopo(topo@pre),
                  numPointsTopo(topo@pre), routes@pre)
Then:
  1. partMatchLockTab(topo@pre, ltab@pre, ltab@pre, 1)
  2. 1 ≤ numTracksTopo(topo@pre) + 1
Is the conclusion satisfied?
```

The first question concerns the establishment of the loop invariant before the first iteration of the loop. The question is, whether the loop invariant is satisfied before the loop body is executed for the first time. Obviously, the initial locking table is the same as the initial locking table with all train routes starting from no track segment added. Part 2 of the conclusion is satisfied since the number of track segments in the topology is always greater or equal to zero according to `validTopology`.

```
VC-2 generateLockings :: TableGen ________________________________
Assume:
  1. validTopology(topo@pre)
  2. emptyLockTab(ltab@pre, numTracksTopo(topo@pre),
                  numPointsTopo(topo@pre), routes@pre)
  3. partMatchLockTab(topo@pre, ltab@pre, ltab@pre, track)
  4. 0 < track
  5. track < numTracksTopo(topo@pre) + 1
Then:
  1. track ≤ numTracksTopo(topo@pre)
Is the conclusion satisfied?
```

In the method `generateLockings` the functions `trackLeft` and `trackRight` are invoked. The above question is to check if their preconditions are satisfied before they are executed.
4. Code Inspection

Obviously, the conclusion is satisfied. The question could be automatically discarded by a more advanced inspection tool.

**VC-3 generateLockings :: TableGen**

**Assume:**
1. validTopology(topo@pre)
2. emptyLockTab(ltab@pre, numTracksTopo(topo@pre), numPointsTopo(topo@pre), routes@pre)
3. partMatchLockTab(topo@pre, ltab@pre, ltab, track)
4. 0 < track
5. track < numTracksTopo(topo@pre) + 1
6. (trackLeft(topo@pre, track) = 0 ⊻ trackRight(topo@pre, track) = 0) ⊼ trackLeft(topo@pre, track) ≠ trackRight(topo@pre, track) ⊼ routeEmpty(ltab)

**Then:**
1. routeEmpty(ltab)

Is the conclusion satisfied?

The method generateLockings invokes the method incRoutesByOne. The above question is to check if the precondition of incRoutesByOne is satisfied before it is executed. The conclusion is satisfied since it appears in the premise. In Java expressions are evaluated from left to right and the evaluation is aborted as soon as the result can be determined. Therefore, premise 6 cannot be split like classical conjunction.

**VC-4 generateLockings :: TableGen**

**Assume:**
1. validTopology(topo@pre)
2. emptyLockTab(ltab@pre, numTracksTopo(topo@pre), numPointsTopo(topo@pre), routes@pre)
3. partMatchLockTab(topo@pre, ltab@pre, ltab, track)
4. 0 < track
5. track < numTracksTopo(topo@pre) + 1
6. (trackLeft(topo@pre, track) = 0 ⊻ trackRight(topo@pre, track) = 0) ⊼ trackLeft(topo@pre, track) ≠ trackRight(topo@pre, track) ⊼ routeEmpty(ltab)
7. incRoutesByOneMod(ltab, ltab#2)

**Then:**
1. matchTrackPoint(topo@pre, ltab#2)
2. 0 < numRoutesLockTab(ltab#2)
3. 0 ≤ numPointsTopo(topo@pre)
4. track ≤ numTracksTopo(topo@pre)

Is the conclusion satisfied?
In the method `generateLockings` the method `findFromTrack` is invoked. The above question is to check if the precondition of `findFromTrack` is satisfied before the invocation. Conclusion 4 clearly is satisfied due to premise 5. Conclusion 3 is satisfied directly because of premise 1. That the locking table `ltab#2` contains at least one train route (conclusion 2) is satisfied due to premise 7 (`ltab#2` has one more train route than `ltab`), premise 3 (`ltab` is the same as `ltab@pre` except that zero or more train routes have been added), and premise 2 (`ltab@pre` has zero train routes). Finally, `ltab#2` has the same number of track segments and points as `topo@pre` (conclusion 1). That `ltab#2` is the same as `ltab@pre` except that zero or more train routes have been added, can be concluded from premise 8 and 7 (all routes from track segment `track` added) and premise 3 (all routes starting from the first `track` – 1 track segments added).

The execution of the body of the loop has to reestablish the loop invariant. The above and the following questions check if this is the case. The above question is generated for the case when the current track segment is connected to only one point and when the locking table is not full (i.e., train routes have been added to the locking table). Clearly, conclusion 2 is satisfied because of premise 4 and conclusion 3 because of premise 5. That `ltab#1` is the same as `ltab@pre`, but with all train routes starting from the first track track segments added, can be concluded from premise 8 and 7 (all routes from track segment `track` added) and premise 3 (all routes starting from the first track – 1 track segments added).
4. Code Inspection

VC-6 generateLockings :: TableGen

Assume:
1. validTopology(topo@pre)
2. emptyLockTab(ltab@pre, numTracksTopo(topo@pre),
   numPointsTopo(topo@pre), routes@pre)
3. partMatchLockTab(topo@pre, ltab@pre, ltab, track)
4. 0 < track
5. track < numTracksTopo(topo@pre) + 1
6. trackLeft(topo@pre, track) ≠ 0 ∧ trackRight(topo@pre, track) ≠ 0 ∀
   trackLeft(topo@pre, track) = trackRight(topo@pre, track) ∀
   ¬ routeEmpty(ltab)

Then:
1. partMatchLockTab(topo@pre, ltab@pre, ltab, track + 1)
2. 0 < track + 1
3. track + 1 ≤ numTracksTopo(topo@pre) + 1

Is the conclusion satisfied?

The sixth question is generated for the case when either the current track segment cannot
be the starting segment of a train route or when the locking table is full. Conclusions 2
and 3 are obviously satisfied. Conclusion 1 is satisfied because of premise 3 and 6.

VC-7 generateLockings :: TableGen

Assume:
1. track = numTracksTopo(topo@pre) + 1
2. validTopology(topo@pre)
3. emptyLockTab(ltab@pre, numTracksTopo(topo@pre),
   numPointsTopo(topo@pre), routes@pre)
4. partMatchLockTab(topo@pre, ltab@pre, ltab, track)
5. 0 < track

Then:
1. matchLockTab(topo@pre, ltab@pre, ltab)

Is the conclusion satisfied?

The last question is to check that the loop invariant together with the negation of the
loop guard implies the postcondition of the method generateLockings. The conclusion
that ltab is the same as ltab@pre except that all train routes through the station have
been added (according to matchLockTab) is satisfied because of premise 3 and premise 1
(track = numTracksTopo(topo@pre) + 1).
Since all questions can be answered with yes, it can be concluded that the method `generateLockings` is correct with respect to the assertions. Exact how the questions may be answered is described in detail in Chap. 6.

**Inspection of the Exclusion Table Generator**

The generation of the exclusion table is implemented in the methods `generateExclusions` and `shareElems`. The prototype tool generates the following questions regarding the method `generateExclusions`:

**VC-1 generateExclusions :: TableGen _________________**

Assume:
1. `emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))`

Then:
1. `partMatchExclTab(ltab@pre, etab@pre, etab@pre, 1, 1)`
2. `1 ≤ numRoutesLockTab(ltab@pre) + 1`

Is the conclusion satisfied?

The first question concerns the initial establishment of the invariant for the outer loop. The question is, if this loop invariant is satisfied before the loop body is executed for the first time. Conclusion 1 is satisfied due to the premise (see the definition of `partMatchExclTab`). Conclusion 2 clearly is satisfied, since the number of train routes in the locking table is at least zero.

**VC-2 generateExclusions :: TableGen _________________**

Assume:
1. `emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))`
2. `partMatchExclTab(ltab@pre, etab@pre, etab@pre, route, 1)`
3. `0 < route`
4. `route < numRoutesLockTab(ltab@pre) + 1`

Then:
1. `1 ≤ numRoutesLockTab(ltab@pre) + 1`

Is the conclusion satisfied?

As for the outer loop, the invariant of the inner loop has to be initially satisfied. The above question reflects that. The conclusion clearly is satisfied, since the number of train routes in the locking table is at least zero (premise 1).

**VC-3 generateExclusions :: TableGen _________________**

Assume:
4. Code Inspection

1. route_a = numRoutesLockTab(ltab@pre) + 1
2. emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))
3. partMatchExclTab(ltab@pre, etab@pre, etab, route_a, 1)
4. 0 < route_a

Then:
1. matchExclTab(ltab@pre, etab@pre, etab)

Is the conclusion satisfied?

The above question checks if the postcondition of the method generateExclusions is satisfied because of the invariant of the outer loop together with the negation of its guard. According to premise 1 and 3 all pairs of train routes have been checked for shared track segments or points and the exclusion table etab has been updated accordingly. Thus, the conclusion is satisfied.

VC-4 generateExclusions :: TableGen

Assume:
1. emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))
2. partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b)
3. 0 < route_a
4. route_a < numRoutesLockTab(ltab@pre) + 1
5. 0 < route_b
6. route_b < numRoutesLockTab(ltab@pre) + 1
7. shareElems(ltab@pre, route_a, route_b)

Then:
1. route_a ≤ numRoutesExclTab(etab)
2. route_b ≤ numRoutesExclTab(etab)

Is the conclusion satisfied?

In the inner loop of method generateExclusions the method allowRoutes is invoked. The above question is to check if the precondition of allowRoutes is satisfied before it is invoked. The conclusion clearly is satisfied due to premise 4 and 6 since exclusion table etab@pre (premise 1), and thus exclusion table etab (premise 2), have the same number of train routes as the locking table ltab@pre.

VC-5 generateExclusions :: TableGen

Assume:
1. emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))
2. partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b)
3. 0 < route_a
4. route_a < numRoutesLockTab(ltab@pre) + 1
4.5. Example

Then:
1. partMatchExclTab(ltab@pre, etab@pre, etab#1, route_a, route_b + 1)
2. 0 < route_b + 1
3. route_b + 1 ≤ numRoutesLockTab(ltab@pre) + 1

Is the conclusion satisfied?

The execution of the body of the inner loop has to reestablish the corresponding loop invariant. The above and the following questions check if this is the case. The above question is generated for the case when the two currently examined train routes do not share a track segment or point. Conclusion 1 above is satisfied according to the definitions of the involved predicates. Conclusion 2 and 3 clearly are satisfied because of premise 4 and 6.

VC-6 generateExclusions :: TableGen

Assume:
1. emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))
2. partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b)
3. 0 < route_a
4. route_a < numRoutesLockTab(ltab@pre) + 1
5. 0 < route_b
6. route_b < numRoutesLockTab(ltab@pre) + 1
7. shareElems(ltab@pre, route_a, route_b)

Then:
1. partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b + 1)
2. 0 < route_b + 1
3. route_b + 1 ≤ numRoutesLockTab(ltab@pre) + 1

Is the conclusion satisfied?

The sixth question is generated for the case when the two currently examined train routes do share a track segment or point. Conclusion 1 is satisfied because of premise 1, 2, and 7. Conclusion 2 and 3 clearly are satisfied due to premise 4 and 6.

VC-7 generateExclusions :: TableGen

Assume:
1. route_b = numRoutesLockTab(ltab@pre) + 1
2. emptyExclTab(etab@pre, numRoutesLockTab(ltab@pre))
4. Code Inspection

3. partMatchExclTab(ltab@pre, etab@pre, etab, route_a, route_b)
4. 0 < route_a
5. route_a < numRoutesLockTab(ltab@pre) + 1
6. 0 < route_b

Then:
1. partMatchExclTab(ltab@pre, etab@pre, etab, route_a + 1, 1)
2. 0 < route_a + 1
3. route_a + 1 ≤ numRoutesLockTab(ltab@pre) + 1

Is the conclusion satisfied?

Finally, the seventh question is generated to check that each iteration of the outer loop reestablishes the corresponding loop invariant. The first conclusion is satisfied since the premise states that it has been examined which train routes may be used concurrently with train route route_a (premise 1 and 3). Conclusion 2 and 3 clearly are satisfied due to premise 4 and 5.

Since all questions generated for the method generateExclusions can be answered with yes, it can be concluded that the method is correct with respect to the assertions. Exact how the questions may be answered is described in detail in Chap. 6.
5. Design Inspection

In the previous chapter we described the COMPASS method which is concerned with the inspection of code. However, a large portion of the mistakes which eventually lead to erroneous system behavior have their origin in earlier phases of the development process (see e.g. [77]). Therefore, we demonstrate in this chapter how the principles of semantic inspection introduced in Chap. 3 can be applied to the inspection of early designs (see also [52]). The specification and the design are expressed in a subset of the Unified Modeling Language (UML) annotated with assertions similar to the ones used for the inspection of Java code (see Sect. 4). In contrast to code, early designs are typically incomplete, scenario-based, and on a high level of abstraction. However, for the automatic generation of questions we need detailed information on the functionality of operations which are not implemented yet. Thus, the inspection of early designs creates new challenges. We tackle these challenges with the introduction of additional types of assertions (e.g. sequence pre- and postconditions, and sequence intermediate assertions).

For the inspection of early designs our approach consists of the following three steps (see Fig. 5.1 on the following page):

1. Development of an early design annotated with assertions
   As mentioned, assertions express conditions that are assumed to, respectively supposed to, be valid at specific points during the computation. In particular, assertions specify both what certain design elements are supposed to establish and what they may expect to be valid. Our assertions may contain insufficiently axiomatized predicates. This enables the formulation of and reasoning with assertions on a high level of abstraction. It should be noted that the assertions provide a structured documentation of the design even if they are not used for the generation of the questions.

2. Generation of a set of questions (i.e., verification conditions)
   The generation of the questions is not dependent on the meaning of the predicates appearing in the assertions. However, the meaning (i.e., the semantics) of the design notation has to be defined. Then the questions may be automatically generated from the annotated design. The questions are the only questions needed to be addressed in the design inspection (for functional correctness).
5. Design Inspection

![Diagram](image)

Fig. 5.1. The three steps of the semantic design inspection method

3. Human inspection of the questions

The questions generated in the second step contain in general informal predicates, thus disabling the possibility of a formal proof. Therefore the questions are presented to a human inspector who will answer them (and informally justify the answer). If all questions can be answered positively then the design is assumed to be correct with respect to the assertions (but in contrast to code inspection this statement is weaker since the design typically is incomplete).

In Sect. 5.1 we start out by briefly introducing an object-oriented software development process which uses UML. The process helps us to pinpoint the notations used in the first step of our semantic design inspection method. The specification notation is then described in Sect. 5.2 and the design notation in Sect. 5.3. The second step, namely the generation of the questions, is presented in Sect. 5.4. We have implemented a prototype tool to demonstrate the feasibility of our approach. This tool is described in Sect. 5.5. Finally, in Sect. 5.6 we give an example to further elaborate on our semantic design inspection. The third step of the semantic design inspection method is the same as for the semantic code inspection method. What the human inspection of the questions generated in the second step may look like is therefore discussed in Chap. 6.

5.1. Development Process

Our specification and design notation is based on UML 1.3 (see [86]) with special annotations (assertions). The development of UML was initiated by the Object Management Group (OMG) to define a standard notation for modeling object-oriented systems. However, UML is nothing more (and nothing less) than a modeling language, i.e. it does provide means to express different views of a system but it does not provide a particular process to develop such systems. Therefore, e.g. the Unified Software Development Process (USDP) has been presented by Jacobson, Booch, and Rumbaugh (see [58]). The USDP is a use-case driven, architecture-centered, iterative, and incremental software de-
5.1. Development Process

Development process. It describes how to develop software. The process results in a set of models which describe the software from different angles and at different stages of the development. Each model uses different kinds of diagrams (the diagrams are expressed using UML). The primary model set of USDP comprises the following models:

**Use-Case Model.** The use-case model describes the requirements as seen from the outside of the system. It is used by customers and developers.

**Analysis Model.** The analysis model describes the requirements more precisely as seen from the inside of the system. It is used by developers to increase their understanding of the requirements. The analysis model is usually incomplete and may sometimes be omitted entirely.

**Design Model.** The design model describes the static structure of the system. Moreover, the design model describes how the elements of the static structure interact to realize the use-cases. The design model is used by programmers and forms the basis of the implementation.

**Implementation Model.** The implementation model describes components (which represent source code) and how the elements of the static structure are mapped to the components.

**Deployment Model.** The deployment model describes physical nodes of computers and how the components (from the implementation model) are mapped to the nodes.

**Test Model.** The test model describes the test cases that verify the use-cases.

We adopt the notation of the use-case model for the specification notation. Of course, it is backed up by annotations expressed in the assertion language introduced earlier in Chap. 3. Assertions are attached to certain parts of the use-case model (and later to the design model). They express conditions on the state of the system or subcomponents which are supposed to be satisfied at certain points. Actually, UML already provides for some annotations (as constraint stereotypes, e.g. operation pre- and postconditions). Moreover, UML includes the Object Constraint Language (OCL) which is used to express various conditions attached to UML diagrams (OCL is defined together with the UML in [86]). OCL is very similar to the formal parts of the assertion notation mentioned. Since UML already provides for some annotations and for a formal language it should be easy to introduce our semantic inspection approach into an existing development process that uses UML. We could have adopted the notation of the analysis model as specification notation, but as the analysis model is often omitted, it was not considered an appropriate choice.

The natural choice for our design notation is to adopt the notation of the design model. As mentioned, the analysis model is usually very incomplete or even omitted and thus not a good candidate. Moreover, the analysis model (if it exists) is quite similar to the
5. Design Inspection

Fig. 5.2. Graphical representation of an actor

design model and may in fact be seen as a special form of the design model. That is, we are able to verify an existing analysis model in the same manner as the design model.

The remaining models (i.e., the implementation, deployment, and test model) are not considered since they are created after the design.

5.2. Specification Notation

In USDP the specification (i.e., the use-case model) consists of use-case diagrams with attached documentation. A use-case diagram describes the relationships between various uses of the system and its environment. We have selected a subset of UML both for the specification and the design notation. The subset simplifies the development and implementation of methods and tools. At the same time the subset is sufficiently large to demonstrate the principles of semantic inspection. The subset had to be suitable for the telecommunication domain because the design inspection has been developed in a project carried out in cooperation with Ericsson SoftLab AB. The details and implications are discussed in Sect. 5.4.1 on the prerequisites and limitations of the generation of questions.

When it comes to the specification notation, then our UML subset includes use-cases, actors, and communicate relationships between actors and use-cases:

**Actor.** An actor represents a role of an entity that is outside the system and that interacts with the system. Actors can be persons, things, or other systems. An actor has a name and a type. In USDP the details of the role that an actor represents are described in the attached documentation. In the semantic inspection approach the details (or at least some of them) are expressed in assertions. An actor is rendered as a stick figure with the object and class name below it (see Fig. 5.2).

**Use-Case.** A use-case represents one specific use of the system. Each use-case has a name. In USDP the details are described in the attached documentation. In the semantic inspection approach the details are expressed in assertions. A use-case is drawn as an oval bubble with its name inside (see Fig. 5.3 on the facing page).

**Relationship.** The only relationship allowed in our subset of use-case diagrams is a communicates relationship between two entities. Such a relationship indicates that
5.2. Specification Notation

Fig. 5.3. Graphical representation of a use-case

the two connected elements are communicating with each other. A relationship is indicated by a straight line connecting two entities (see e.g. Example 5.1).

Example 5.1
The following figure shows a use-case diagram used to describe on a high level what services are expected from an internet provider:

![Diagram](attachment:use-case-diagram.png)

The diagram contains two actors and two use-cases. The actors are called client and server and are both of class (type) Computer. The two use-cases are dialIn and retrieveFile. They represent dialing in to an internet provider (the system) respectively retrieving a file.

In the USDP the documentation consists of a set of natural language descriptions of when it is safe to apply a certain use-case, what the outcome of the use-case is, and roughly how the system establishes that outcome. Instead of this completely informal description, in the semantic inspection approach specific diagrams and diagram elements are annotated in a semi-formal way. These annotations are directly attached to the diagram or graphical component to which they apply.1 Thus, the context to which the annotations apply follows immediately from the graphical component to which they are attached. However, to provide examples in this thesis a textual form for annotations is used:

```
context element-type element-name
  Annotation descriptions etc.
end context element-type element-name
```

The element-type specifies the type of diagram or diagram element for which the body applies. Possible element types are sequence, use-case, loop, object, intermediate, class,

---

1That requires that the tool used to draw the different diagrams supports these annotations. A prototype for such a tool has been developed and is described later in Sect. 5.5.

71
and operation. It is assumed that classes and use-cases are globally unique and hence no context environment for class- and use-case diagrams are necessary. The element-name is the name of the element as it appears in a diagram (including arguments/parameters if applicable). Context environments are usually nested and the nesting follows the structure of the graphical representation which is illustrated in Fig. 5.4. Annotations both in the specification and in the design belong to such a context environment. Furthermore, for convenience we assume that predicate and function symbols (see Chap. 3) are globally unique and hence their definitions are not given within a context environment. The textual form of an informal respectively formal definition is either of the following two:

```
define p(parameter-list) informally
    Informal predicate or function definition
end define ;

define p(parameter-list) formally
    Formal predicate or function definition
end define ;
```

The $p$ is simply the name of the predicate or function. The parameter-list is the comma-separated list of parameters to the predicate or function. Whether the definition is formal or informal is indicated by the corresponding key-word formally respectively informally (see Example 5.2 on the next page).
The most important annotations of the specification are use-case pre- and postconditions. Use-cases represent the various uses of the system and the details of each use-case are described with a pair of use-case pre- and postconditions.

5.1 Definition (Use-case pre- and postconditions)

A use-case precondition is a formula that is assumed to be satisfied prior to the execution of the use-case. Thus, the precondition specifies the states in which the use-case can be invoked. A use-case postcondition is a formula that has to be satisfied after the execution of the use-case. Hence, the postcondition specifies the service that is provided by the use-case (given that the precondition was satisfied when it was invoked).

It follows from the nature of use-case diagrams that use-case pre- and postcondition may contain references to related actors, the system in general, and possible input and output to the use-case. A use-case precondition is identified by the key-word requires (since the execution of the use-case requires that the subsequent condition is satisfied before the use-case is invoked):

\[
\text{requires} \\
\quad \text{Specification of the use-case precondition ;}
\]

Similarly, a use-case postcondition is identified by the key-word ensures (since the execution of the use-case ensures that the subsequent condition is satisfied after the use-case terminated):

\[
\text{ensures} \\
\quad \text{Specification of the use-case postcondition ;}
\]

Example 5.2

The text below shows the textual representation of the pre- and postconditions of a use-case connect. The use-case is part of a use-case diagram telephoneNetwork and represents the establishment of a connection between two phones.

\[
\text{context Use-Case connect} \\
\quad \text{requires} \\
\quad \quad \text{idle(callerPhone@pre) and idle(calleePhone@pre) ;} \\
\quad \text{ensures} \\
\quad \quad \text{connectionEstablished(callerPhone, calleePhone) ;} \\
\quad \text{define idle(phone) informally} \\
\quad \quad \text{The phone “phone” is not in use.} \\
\quad \text{end define ;} \\
\quad \text{define connectionEstablished(phoneA, phoneB) informally}
\]
5. Design Inspection

An open, full-duplex connection exists between the two telephones
"phoneA" and "phoneB".

end define;
end context Use-Case connect

The use-case precondition states that both phones have to be idle before a connection may be established. The postcondition states that a connection exists between the two phones and that they are reserved for that connection. The use-case pre- and postconditions in the example make use of two predicate symbols, namely idle and connectionEstablished. The symbols are defined in natural language as shown above.

Annotated use-case diagrams constitute the highest level of specification with respect to USDP. In the remainder of this thesis it is called a system specification.

5.3. Design Notation

In USDP the design (i.e., the design model) is described in class diagrams and sequence diagrams. Class diagrams describe the static structure of the whole system whereas sequence diagrams describe dynamic behavior of the system. Class diagrams actually describe what to implement. Sequence diagrams are traces of specific executions of the system. These traces are therefore also called scenarios. They describe the interaction of concrete instances (i.e., objects) of classes. Sequence diagrams (design) are closely related to use-cases (specification) – one use-case is described in more detail by a set of sequence diagrams. However, sequence diagrams usually describe just a few scenarios and not the complete behavior of the system. The relationships of the diagrams are illustrated in Fig. 5.5 on the facing page.

We mentioned that we selected a subset of UML. The following sections describe the subsets of sequence diagrams and class diagrams we support.

5.3.1. Sequence Diagrams

A sequence diagram is concerned with the realization of use-cases. It focuses on the temporal ordering of messages that are exchanged between objects for particular scenarios. In our subset active objects, asynchronous (signals) and synchronous messages (operation invocations), lifelines, activations, and loops are supported:

Object. An object is a (named) instance of a (named) class. All objects in our subset of UML are active objects (as opposed to passive objects). Active objects own a
5.3. Design Notation

![Diagram of relationships between specification, design, and involved diagrams]

Fig. 5.5. Relationships between specification, design, and involved diagrams

![Diagram of object, lifeline, and activation]

Fig. 5.6. The graphical representation of an object, a lifeline, and an activation

thread of control and can consequently initiate activity (passive objects have no own thread of control and can thus only initiate activity after they have received control from an active object). How objects are presented is shown in Fig. 5.6.

**Lifeline.** Every object has a lifeline. The lifeline indicates the time during which the object exists and is represented by a vertical line (see Fig. 5.6). In our subset of UML objects cannot be destroyed and must persist during the whole scenario.

**Activation.** Lifelines specify the time during which the object exists. Activations (see Fig. 5.6) specify the time during which the object is active. Even though all objects in our subset of UML are active objects there may be intervals during which an object is not active, namely when the object is waiting for an operation to return. We divide activations into so called *activation segments*. An activation segment starts respectively ends when the object receives or dispatches a message, when a loop starts or ends, or when the lifeline itself starts or ends. Activation segments are closely related to the notion of *reference points*. A reference point is a point on the time axis of a sequence diagram where an activation segment starts or ends.

**Message.** Objects communicate with each other through (named and parameterized)
5. Design Inspection

Fig. 5.7. The graphical representation of operation invocations, operation returns, and signals

messages. In our subset of UML all messages arrive at the receiver without delay (delayed messages may be simulated with additional objects). We support two types of message, namely operation invocations (with matching operation returns) and signals. When an object invokes an operation it is blocked (i.e., the activation ends) until the invoked object dispatches a return message (i.e., the activation starts again). Hence, operation invocations are called synchronous messages. When an object sends a signal it is not blocked. Therefore signals are called asynchronous messages. It is possible for an object to dispatch messages to itself. How the different messages are presented graphically is illustrated in Fig. 5.7.

Loop. UML supports loops in sequence diagrams. A connected set of messages may be enclosed and marked as a loop. However, the meaning of loops in UML is not precisely defined. Nevertheless, loops are an essential element in the generation of questions described later in Sect. 5.4. To clarify the meaning of loops we require that every loop is accompanied by an iterator variable (e.g. an integer variable \(i\), an initialization expression specifying the value of the iterator variable before the first iteration (e.g. 0), a guard specifying if a next iteration should occur (e.g. \(i < N\)), and a progress expression specifying how the iterator variable progress from one iteration to the next (e.g. \(i = i + 1\)). The iterator variable may be used as argument to expressions and formulae appearing within the loop. What loops look like is illustrated in Fig. 5.8.
Example 5.3

In a use-case diagram describing a telephone network we may have a use-case `connect`. The use-case represents the establishment of a connection between two telephones. A sequence diagram that realizes the use-case in the “normal” operation may look as shown in Fig. 5.9.

The sequence diagram describes the temporal ordering of messages for establishing a connection between the two telephones `caller` and `callee` over the network `net`. Firstly, the caller lifts the receiver and hears a signal that indicates that the telephone number may be dialed. Then the caller dials the number of the recipient – one digit at a time (therefore the loop). If the number is complete the callee’s telephone rings and the callee lifts the receiver which establishes the connection. It should be noted that the figure also shows the reference points occurring in the sequence diagram on the left side.

So far the elements and presentation of sequence diagrams have been described in an informal manner. More formally a well-formed sequence diagram is constructed as described in the next three definitions.

5.2 Definition (Partial sequence)

Below we will use the following symbol to represent a partial sequence containing an arbitrary (but fixed) number \( n \) of objects:

1. An empty sequence is a partial sequence. An empty sequence consisting of \( n \) objects is denoted \( \epsilon_n \).
5. Design Inspection

2. If $S$ is a partial sequence then so is $S'$ obtained by prepending a signal and two activation segments to $S$:

3. If $S$ is a partial sequence then so is $S'$ obtained by prepending an operation invocation and two activation segments, and by appending an operation return and two activation segments to $S$:

The above figure indicates that both objects get an additional activation segment attached to their top segment in $S$. However, since operation invocations are synchronous, the dispatching object is blocked after the invocation. This is indicated by drawing the corresponding activation segment as a line instead of a block.

4. If $S$ is a partial sequence then so is $S'$ obtained by using $S$ as the body of a loop (i.e., the loop stretches over all objects of $S$):

5. If $S_1$ and $S_2$ are partial sequences over the same set of objects then so is $S'$ obtained by prepending $S_1$ to $S_2$ (written $S_1 \circ S_2$):

5.3 Definition (Complete sequence)

If $S$ is a partial sequence then $S'$, obtained by prepending an activation segment to each object, is a complete sequence:
5.3 Design Notation

5.4 Definition (Well-formed sequence)

A complete sequence is well-formed if and only if every object is active when it dispatches a message or enters a loop.

Example 5.4

In Example 5.3 on page 76 we introduced a sequence shown in Fig. 5.9 on page 77. In this example we will demonstrate that the sequence indeed is a complete sequence according to Def. 5.3. We do so by constructing the sequence of Example 5.3 as described in Def. 5.3 and Def. 5.2.

We start with an empty partial sequence $S_0$ of the three objects caller, net, callee (Rule 1 in Def. 5.2). Next we prepend two activation segments and a signal connected to $S_0$ (Rule 2 in Def. 5.2) to build the partial sequence $S_1$:

$$S_1 : \text{connected}$$

In the same manner we prepend further activation segments and the signals connected, offHook, and ring to build the partial sequences $S_2$, $S_3$, and $S_4$:

$$S_4 : \text{connected} \quad \text{offHook} \quad \text{ring} \quad \text{connected}$$

Then we start again with the empty partial sequence $S_0$ and prepend two activation segments and a signal dial to $S_0$ (Rule 2 in Def. 5.2) to build the partial sequence $S_5$:

$$S_5 : \text{dial(digit)}$$

Now we use the partial sequence $S_5$ as the body of a loop. With Rule 4 in Def. 5.2 we get the partial sequence $S_6$:

$$S_6 : \text{dial(digit)}$$

We prepend further activation segments and the signals idleSound and offHook to $S_6$ to build the partial sequences $S_7$ and $S_8$ (using Rule 2 in Def. 5.2):
5. Design Inspection

To combine the partial sequences $S_4$ and $S_8$ we use Rule 5 in Def. 5.2. We build the partial sequence $S_9$:

Finally, we prepend (according to Def. 5.3) an activation segment to each object to make the partial sequence $S_9$ complete and to obtain the complete sequence shown in Fig. 5.9.

Since a sequence diagram is a trace of an execution of the system it is usually only applicable under certain circumstances. For example, it is common to have sequence diagrams describing execution under normal conditions and under exceptional conditions. Thus, a sequence diagram alone is not sufficient to fully understand a scenario. Additional information about prerequisites, limitations, and intended outcome of the specific trace is needed. In USDP this information is provided in an informal manner in attached documents. In our semantic design inspection approach, specific diagrams and diagram elements are annotated directly in a semi-formal manner. There are four types of annotations, namely sequence preconditions, sequence postconditions, sequence intermediate assertions, and loop invariants.

5.5 Definition (Sequence pre- and postcondition)
A sequence precondition is a formula that is supposed to be satisfied prior to the execution of the sequence. A postcondition is a formula that has to be satisfied after the execution of the sequence (given that the precondition was satisfied when the trace started).

Both of the formulae may refer to the state of all objects that appear in the sequence diagram. However, a precondition may only refer to the initial states of objects (using the appendage “@pre”) whereas a postcondition may refer both to the initial and final state of objects. The textual form of sequence pre- and postconditions looks exactly like

---

2For convenience we use “sequence” when we mean “well-formed sequence”.
the textual form of use-case pre- and postconditions (the context determines what kind of condition it is). A sequence precondition is identified by the key-word requires and a sequence postcondition by the key-word ensures:

```
requires
    Specification of the sequence precondition ;
ensures
    Specification of the sequence postcondition ;
```

The sequence pre- and postcondition specify when the trace can occur and what results can be expected. Therefore, a pair of sequence pre- and postconditions is called a scenario specification in the remainder of this thesis.

Our focus is on the functional correctness of software artifacts like early designs. However, the information contained in a sequence diagram alone is not enough for verifying a design. A sequence diagram shows the temporal ordering of information exchange between objects but it does not show what the objects actually do. Thus, additional assertions are needed, namely sequence intermediate assertions and loop invariants.

5.6 Definition (Sequence intermediate assertion)
Each activation segment is accompanied by a sequence intermediate assertion. An intermediate assertion is a relation that specifies how the state of the corresponding object changes in that segment. That is, an intermediate assertion is a formula that is assumed to be satisfied immediately after the activation segment.

If no intermediate assertion is provided for an activation segment then nothing is known about the state. Also, even if an intermediate assertion is supplied, only what is explicitly specified by the assertion is known. Hence, it is in general not enough to specify which variables did change (and how). Instead it has to be specified which variables did not change as well. This problem is known as the frame problem (and has already been discussed in Sect. 4.1.3). When it comes to assertions attached to early designs we assume that assertions explicitly specify which variables did and did not change (as far as it is needed to establish the correctness of the design with respect to the scenario respectively the system specification). When it comes to objects this is usually done by defining predicates in the style of “Object x is the same as object x@pre except that...” (see e.g. Example 5.5 on the next page and Example 5.7 on page 87).

In general, intermediate assertions can refer to the state of the object itself and to other objects it receives through messages. If the intermediate assertions appear inside a loop they can also refer to the iterator variable. Since a sequence intermediate assertion ensures that the subsequent formula is satisfied immediately after the activation segment.
the key-word is ensures. The textual form for presenting intermediate assertion therefore is:

\[
\text{ensures} \\
\text{Specification of the sequence intermediate assertion ;}
\]

In the textual form of annotations it is necessary to specify to which activation segment a sequence intermediate assertion applies. For each object the activation segments are sequentially numbered starting with 1 for the top segment. Moreover, the context allows to distinguish sequence intermediate assertion from use-case postconditions and sequence diagram postconditions which use the same key-word (see Example 5.5).

**Example 5.5**
The activation segments belonging to a specific object in a specific sequence are sequentially numbered from top to bottom (starting with 1). The intermediate assertions attached to the activation segments 3 and 10 of object net in Example 5.3 on page 76 may look as follows:

```plaintext
context Sequence normalConnect
  context Object net
    context Intermediate 3
      ensures
      readyForDigits(this,this@pre) ;
    end context Intermediate 3
    context Intermediate 10
      ensures
      this = this@pre ;
    end context Intermediate 10
  end context Object net
end context Sequence normalConnect
```

The intermediate assertion to activation segment 3 is intended to express that the network net (i.e., this) at the end of the activation segment is the same as it was at the beginning of the activation segment except that it now is ready to accept digits from the caller. The telephone caller does not occur as an argument to the predicate readyForDigits because it is already known to, and somehow stored in, the network (e.g. as an attribute of class Network).

After activation segment 9 (at reference point 9) the connection is already established. The intermediate assertion to activation segment 10 therefore needs to express that the state of the network does not change.
5.3. Design Notation

5.3.2. Class Diagrams

Class diagrams describe the static structure of a software system. They specify the classes involved (i.e., general descriptions of objects) and their relationships. In our subset of UML class diagrams consist of classes (with attributes and operations), and association and generalization relationships between classes:

Class. A class describes the properties common to a certain “type” of objects. Common properties include both attributes and operations. For example, a class Phone could state that any instance of that class may be either on-hook or off-hook (attribute), that any instance allows to dial a number (operation) etc. A class is represented by a rectangular box. The name of the class appears inside the box. Attributes and operations are written inside the box below the name of the class (see Fig. 5.10).

![Class Diagram](image)

Fig. 5.10. Graphical representation of a class with operations

5.7 Definition (Loop invariant)

A loop invariant is a formula that is supposed to be maintained by a loop, i.e., a condition that has to be satisfied prior to the first iteration and that is reestablished after each iteration over a contiguous part of a sequence diagram.

A loop invariant can refer to all objects in the sequence diagram. Moreover, the iterator variable that is defined for the loop can be used. A loop invariant is identified by the key-word maintains (as the subsequent condition is maintained by the loop):

maintains

Specification of loop invariant ;

The same key-word is used for other annotations as well (namely class invariants). The context determines which kind of annotation it actually is.

Intermediate assertions and loop invariants are applicable only to a single scenario. Other conditions that facilitate the understanding of a single scenario have a more general character and are hence provided as annotations to class diagrams.
5. Design Inspection

Attribute. Attributes are used to capture the state of class instances. Each attribute has a name, followed by a type and a default value:

\[ \text{name : type} = \text{default-value} \]

The type restricts the values which the attribute may assume. Possible types are boolean, integer and classes.\(^3\) The default-value is a valid value expression of the previous type. Both the equal sign and the default value may be omitted.

In UML it is possible to specify the visibility of each attribute. The subset for semantic design inspection includes private attributes only, i.e. attributes are visible only to instances of the class. The reasons for this restriction are discussed in Sect. 5.4.1.

Operation. An operation describes what an object of the corresponding class actually can do. Every operation consists of a name followed by a (possibly empty) list of parameters, and the return type:

\[ \text{name (parameter-list)} : \text{return-type} \]

The return-type restricts the values that the operation may return (see description of attributes). If the operation does not return values then the colon and the return type are omitted. The parameter-list is a comma-separated list of formal parameters. Each list element is of the form:

\[ \text{kind name : type} = \text{default-value} \]

The kind is in, out, or inout. If it is omitted then it is assumed to be in. The kind of a parameter expresses whether the parameter is used by the operation (in), calculated by the operation (out), or both (inout). Operations that return values (i.e., functions) are not permitted to change the state of the system; hence they may only have in parameters. The type restricts the values the parameter may assume (see description of attributes). Finally, the default-value is a valid value expression of the previous type. Both the equal sign and the default value may be omitted.

As for attributes it is possible to specify the visibility of operations in UML. The subset for semantic design inspection includes public operations only (i.e., all operations are visible to all users of the class). The reasons for this restriction are discussed in Sect. 5.4.1.

Association. An association represents a general structural relationship between two classes. An association may have a name and an indicator for clarifying in which

---

\(^3\)In UML the types are dependent of the implementation language.
5.3. Design Notation

Direction the name should be read. Moreover, associations may be tagged with multiplicity values that indicate how many objects of the given class may be linked to how many objects of the other class. Possible multiplicity values are 1 (one and only one), 0..1 (zero or one), M..N (from M to N, both natural numbers), * (any natural number), and 1..* (any positive integer). Associations are represented by a solid line with the name and the direction indicator on top and the multiplicity below it (see Fig. 5.11).

**Generalization.** A generalization refers to the classification relationship between classes, i.e., to the relationship between a general and a more specific object. A more specific object is an object that contains more information than the general object while remaining completely consistent with the description of the general object. In other words the more general class, called *super class*, is an abstraction of the more specific class, called *subclass* (the implications of inheritance were already discussed in Sect. 4.3.1). The generalization relationship between classes is represented by an hollow arrow pointing from the subclass to the super class (see Fig. 5.12).

**Example 5.6**
The class diagram in Fig. 5.13 on the following page describes the classes and their relationships as they may occur in a (non-existent) wireless communication network. The diagram shows the three (incomplete) classes MobileDevice, MobilePhone, and BaseStation.
5. Design Inspection

The diagram specifies that the class BaseStation has two attributes and four operations, the class MobileDevice has one attribute and two operations, and the class MobilePhone has no additional attributes or operations. Moreover, the diagram shows that instances of the class BaseStation control instances of the class MobileDevice, and that the class MobilePhone is a subclass of the class MobileDevice.

Annotations (assertions) are attached to certain class diagram elements. These annotations describe in detail what functionality is provided by each class and what other properties it has. General properties of a class (i.e., of each instance of a class) are described by a class invariant:

5.8 Definition (Class invariant)

A class invariant is a condition (formula) that is maintained by each instance of the class. That is, it is assumed that the condition is satisfied before any invocation of an operation of the object and that it is reestablished afterwards. During the execution of an operation the condition may be temporarily violated.

A class invariant may refer to the state of the class instance by using this or functions (see e.g. Example 5.11 on page 101) on this alone. Class attributes cannot be used directly in a class invariant because they are not visible outside the class but the invariant is. However, this is not limiting the expressiveness of class invariants since the state that is captured in class attributes can be addressed indirectly using this (see Example 5.7 on the facing page). Moreover, an invariant may only refer to the current state of any class instance and not to its initial state (which is denoted by the appendage “@pre”). In the textual form a class invariant is identified by the key-word maintains (because the subsequent conditions is maintained by each instance of the class):
5.3. Design Notation

maintains
   Specification of the class invariant ;

The same key-word is used for other annotations as well (namely loop invariants). The context determines which kind of annotation it actually is.

Example 5.7
Example 5.6 on page 85 shows a class BaseStation. The textual representation of a class invariant expressing that never more than 100 mobile phones may be simultaneously connected to the base station may look as follows:

context Class BaseStation
   maintains
      noOverload(this) ;
   define noOverload(station) informally
      There are never more than 100 mobile phones simultaneously connected to the base station “station”.
   end define ;
end context Class BaseStation

The class invariant is expressed with a single predicate symbol noOverload. The predicate has one argument denoting the class instance itself, namely this. The number of phones currently connected to the base station may be kept in the attribute numConnected. However, that attribute is not visible outside the class instance itself and may thus not be used in the class invariant.

Class invariants are not the only type of annotations to class diagram elements. In addition to invariants, each operation of a class is annotated with an operation pre- and postcondition. Operation pre- and postconditions and class invariants form a contract between the object that invokes an operation and the object that provides the operation. It is up to the invoking object to ensure that the precondition is satisfied and it is then up to the providing object to ensure that the postcondition and class invariant are satisfied. That the providing object actually fulfills its part of the contract (i.e., the postcondition and the class invariant are satisfied upon termination) is usually not checked at the early design stage. Instead it is postponed until more details on how the operation is realized are known. How this could be done is explained in Chap. 4.

5.9 Definition (Operation pre- and postcondition)
An operation precondition is a condition (formula) that is supposed to be satisfied prior to the execution of the operation. A postcondition is a formula that has to be satisfied after the execution of the operation.
5. Design Inspection

Both assertions can refer to the state (respectively the initial state) of the object (by using this) and to arguments to the operation (if present). Like use-case and sequence preconditions, an operation precondition is also identified by the key-word requires:

\[
\text{requires} \\
\text{Specification of the operation precondition ;}
\]

In the textual representation an operation postcondition is identified by the key-word ensures:

\[
\text{ensures} \\
\text{Specification of the operation postcondition ;}
\]

The key-words requires and ensures are used for use-case and sequence pre- and postconditions as well. The context determines which kind of annotation it actually is.

Example 5.8
Example 5.6 on page 85 shows a class BaseStation with an operation addPhone. The textual representation of the pre- and postconditions of operation addPhone may look like this:

\[
\text{context Class BaseStation} \\
\text{context Operation addPhone(phone)} \\
\text{requires} \\
\text{emptyChannel(this@pre) } \land \text{ notControlled(phone@pre) ;} \\
\text{ensures} \\
\text{phoneAdded(this, this@pre, phone) ;} \\
\text{define emptyChannel(station) informally} \\
\text{It is possible to connect another phone to the base station "station".} \\
\text{end define ;} \\
\text{define notControlled(phone) informally} \\
\text{The phone "phone" is not under control of any base station.} \\
\text{end define ;} \\
\text{define phoneAdded(oldStation, newStation, phone) informally} \\
\text{Base station "newStation" is the same as base station "oldStation" except that an additional mobile phone "phone" is added to the phones controlled by "newStation".} \\
\text{end define ;} \\
\text{end context Operation addPhone(phone)} \\
\text{end context Class BaseStation}
\]
The operation \texttt{addPhone} simply adds a new phone to the list of phones controlled by a base station (an instance of class \texttt{BaseStation}). It is only possible to add a phone if the station is capable of controlling one more phone (expressed with the predicate symbol \texttt{emptyChannel}) and if the phone is not already under the control of a base station (expressed with the predicate symbol \texttt{notControlled}).

After the operation has been performed the phone is under the control of the base station. It is important to express not only what has been changed but also what did not (the frame problem as discussed earlier). This is conveniently done by comparing the new state of the base station (\texttt{station}) to its old (initial) state (\texttt{station@pre}).

### 5.4. The Generation of Questions

In this section we explain in detail how the questions to be addressed in our semantic design inspection are generated. The questions are generated from a specification and a design in the notations introduced earlier in Sect. 5.2 and Sect. 5.3.

For the sequences and the use-case of an early design several questions are generated. The questions intend to verify that every:

- operation precondition is satisfied when the operation is invoked,
- loop invariant is satisfied when the loop is reached,
- loop invariant is maintained by the loop,
- sequence postcondition is satisfied when the sequence ends, and
- use-case postcondition is satisfied when the use-case ends.

A note on loops (and loop invariants): loops are verified in an inductive way, i.e. first we check that the invariant holds initially, and then we check that the invariant holds after an iteration assuming it held prior to the iteration. Thus we can conclude that the invariant holds after the termination of the loop.

One question is generated for each of the conditions listed above. These questions are the only questions needed to be answered during the design inspection (for functional correctness). If the questions can be answered positively, then the design is assumed to be correct with respect to the assertions. When generating the questions we assume that every:

- loop invariant and guard is satisfied at the beginning of every iteration,
- loop invariant and the negation of the guards is satisfied when the loop terminates,
• intermediate assertion is satisfied when the activation segment ends,

• class invariant is satisfied when operations are invoked, and

• class invariant and operation postcondition is satisfied when operations return.

In principle, each question is of the form “assuming that \( x \) is satisfied, is also \( y \) satisfied?” (in practise the questions may be answered in several steps to allow a better defect classification and localization, see Chap. 6). The premise of a question expresses everything that is known up to the point when the conclusion is reached. This knowledge may concern the state of variables at different reference points in the sequence (as mentioned a reference point is a point on the time axis of a sequence where an activation segment starts or ends). Therefore, variables in the questions may be followed by a hash sign and a number describing a reference point in the sequence diagram. The initial state of a variable (with respect to the context) is denoted by the appendage “\( @pre \)”. Thus the expression \( x\#n \) refers to the value of the variable \( x \) at reference point \( n \). Two occurrences of \( x\#n \) always refer to the same value, while \( x\#m \) may differ from \( x\#n \) when \( m \neq n \).

Questions are generated from bottom to top (i.e., from the point where the assertion to be verified is located, to the beginning of the sequence). On the way up all intermediate assertions, loop invariants etc. are transformed and added to the premise. Since the assertions describe the state-change locally from the beginning to the end of the diagram element they are attached to, some variable substitutions are necessary. That is, all formal variables in the assertions are substituted by the corresponding actual variables with respect to value and time.

In the following Sect. 5.4.1 we discuss the prerequisites and limitations of the generation of questions. A formal semantics is necessary for the generation of questions. Therefore, we define in Sect. 5.4.2 the semantics of our subset of UML. It is not our main intention to give a (new) semantics of UML, but to show the feasibility of our semantic inspection of early UML designs. Alternative semantics (with a focus on functional properties) are possible. In Sect. 5.4.3 we explain how the questions can automatically be simplified to some extent. Finally, in Sect. 5.4.4 we suggest some alternatives and extensions.

5.4.1. Prerequisites and Limitations

The questions, which are generated from an annotated UML design, are intended to serve as the basis for establishing the functional correctness of the design. Before we explain in detail how the questions are generated based on the UML semantics described in Sect. 5.4.2 we discuss in the following paragraphs the prerequisites and limitations of our approach.
5.4. The Generation of Questions

Scenario Verification

Common to approaches to formal verification is the aim to prove that the software artifact is correct with respect to the specification. Usually, the intention is to prove that the final state of the system satisfies a certain postcondition provided that the initial state satisfies a certain precondition. In our approach the overall behavior of the system is specified by a set of use-cases and corresponding pre- and postconditions. The system is correct if the execution of each use-case in a state that satisfies the precondition ends (if it ends) in a state that satisfies the postcondition. That is, what we would like to verify corresponds to Arrow 1 in Fig. 5.14 on the next page. However, to verify a use-case, its effect on the state has to be completely defined. In our approach each use-case is specified in more detail by a set of sequence diagrams. The sequence diagrams together with the class diagrams constitute our early design. Each sequence (described in a sequence diagram) comes with a pre- and a postcondition. A sequence is correct if its execution in a state that satisfies the precondition ends in a state that satisfies the postcondition (Arrow 2 in Fig. 5.14). We can generate question to verify this since a sequence contains enough detail (intermediate assertions, operation pre- and postconditions, etc.). If we finally verify that each use-case precondition implies the disjunction of the corresponding sequence preconditions (Arrow 3 in Fig. 5.14) and that each sequence postcondition implies the corresponding use-case postcondition (Arrow 4 in Fig. 5.14) we have achieved our original goal. To generate questions to verify the last proposition is straightforward. However, the preceding proposition implies that the sequence diagrams belonging to a use-case cover all possible states when the use-case may be executed. We consider it very hard if at all possible, and seldom desirable to describe the behavior of a system completely with sequence diagrams. We are aiming at the development of practical means for the inspection of designs and our belief is that the verification of critical properties of particularly important uses of the system is sufficient with respect to cost and benefit. It is here where our approach differs in an important way from formal verification. We verify that, if the execution of each sequence starts in a state that satisfies the sequence precondition then it ends (if it ends) in a state that satisfies the use-case postcondition. However, for executions that are not captured in a sequence we do not know if the use-case postcondition will be satisfied.

Single Thread Assumption

Another important limitation is our single thread assumption. That is, we assume that each object has only one thread of its own (i.e., all objects are active objects). Thus, every object can initiate activity (i.e., invoke operations). However, an object cannot be executing more than one operation at a time. Only, after the end of one operation another operation may be started.
Multiple threads within a single object are difficult to handle because the different threads share the same variables (i.e., attributes). For example, when an operation is invoked its precondition (which may involve the state of the object’s attributes) is checked to be satisfied. However, immediately after the operation invocation another thread may change the object’s attributes leaving the operation precondition dissatisfied. At the same time, the correct function of the operation relies on the validity of the precondition. If the state specified by the precondition is changed and that change is not noticed by the operation, then the postcondition is not sure to be satisfied when the operation ends.

Multiple threads could be allowed if other mechanisms to prevent unnoticed state changes are applied (e.g., atomic operations, semaphores, more details in operation specification to find compatible operations). Nevertheless, we do not support multiple threads to simplify the development and presentation of our semantic inspection method.

Our Subset of UML

We support a subset of UML use-case, sequence, and class diagrams. Use-case diagrams are typically used to express the high-level specification of systems. Sequence and class diagrams are usually used to express the early design of systems. UML statechart, activity, deployment, component, object, and collaboration diagrams are not supported because they are needed neither for specifications nor for early designs. The elements of UML use-case, sequence, and class diagrams we omit can be partitioned into three groups:

- We omit many elements because they are not concerned with the functional cor-
5.4. The Generation of Questions

rectness of the design. Among these elements are e.g. templates and qualified associations in class diagrams.

- Other elements (e.g. branching in sequence diagrams and private operations in class diagrams) we omit even though they would influence the functional correctness. However, the impact of these elements on the generation of questions is small and they could easily be added.

- Some elements are not supported because of principal difficulties. An important restriction is that all class attributes are private. Public attributes would cause similar problems as multiple threads. An operation precondition involving attributes could suddenly become invalid if another object changes these attributes.

Finally, for convenience we assume a global name space for predicates, functions, classes, and use-cases (i.e., predicate, function, class, and use-case names are globally unique).

Missing Questions

The semantics as presented below result in questions to check whether each sequence complies to the corresponding scenario specification and system specification (see earlier in this section on scenario verification). It is possible to generate additional questions to check the class diagram for inheritance consistency, i.e., that a subclass is indeed a subtype of a superclass as it would conceptually be expected. Section 4.3.1 contains a more detailed discussion in the paragraph on overriding.

5.4.2. Semantics

Like for the inspection of Java programs, the questions to be discussed during the inspection of early UML designs are generated with an inference system. For our inspection of early designs, the formulae of the inference system are either well-formed formulae as described in Chap. 3 or they are triples of the form \( \{P\} D \{Q\} \) where \( D \) is a use-case or sequence and where \( P \) and \( Q \) are well-formed formulae. The triples are to be read:

If executing \( D \) in a state where \( P \) is satisfied terminates then \( Q \) is satisfied upon termination.

The use-case or sequence \( D \) then is said to be partially correct because we do not know whether the execution terminates or not (because we do not know whether the execution terminates due to the loop construct).

In the remainder of this section we introduce a semantics for our subset of UML. However, it should be noted that our main goal is to demonstrate a new method to the
5. Design Inspection

inspection of early designs expressed in UML; the goal is not primarily to provide a precise semantic description of UML. Concerning well-formed formulae we use the following notation in our semantics:

• If $U$ is a use-case then $P_U$ denotes its precondition and $Q_U$ its postcondition. Let $\bar{u}$ be the vector of the variables in $Q_U$ without appendage. Let $u#r$ be the vector of the same variables but with the appendage “#r”. Then:

$$(Q_U)_{\bar{u}#r}$$

is the condition that is valid when the sequence ends at reference point $r$ (according to a related sequence diagram).

• If $S$ is a sequence then $P_S$ denotes its precondition and $Q_S$ its postcondition. Let $\bar{u}$ be the vector of the variables in $Q_S$ without appendage. Let $u#r$ be the vector of the same variables but with the appendage “#r”. Then:

$$(Q_S)_{\bar{u}#r}$$

is the condition that is valid when the sequence ends at reference point $r$.

• Let $A$ be the intermediate assertion of an activation segment belonging to the object $a$. Let $\bar{u}@pre$ be the vector of the variables in $A$ with the appendage “@pre”. Let $u#r_1$ be the vector of the same variables but with the appendage “#r_1” except for the variable $this$ which in addition is replaced with $a$. Let $\bar{u}$ be the vector of the variables in $A$ without appendage. Let $u#r_2$ be the vector of the same variables but with the appendage “#r_2” except for the variable $this$ which in addition is replaced with $a$. Then:

$$A_{\bar{u}@pre, \bar{u}#r_1, \bar{u}#r_2}$$

is the relation that is satisfied when the activation segment $A$ starts at reference point $r_1$ and ends at reference point $r_2$.

Example 5.9
Assume we have an activation segment of an object box. The activation segment starts at reference point 2 and ends at reference point 4. As intermediate assertion $A$ the designer has specified:

ensures
mystery(this, this@pre, joe, carin@pre):

Then $\bar{u}@pre = [this@pre, carin@pre], \bar{u}#1 = [box#2, carin#2], \bar{u} = [this, joe], \bar{u}#4 = [box#4, joe#4]$, and thus:

$$A_{\bar{u}@pre, \bar{u}#2, \bar{u}#4} = \text{mystery(box#4, box#2, Joe#4, carin#2)}$$
5.4. The Generation of Questions

• Let $L$ be the invariant of a loop. Let $\bar{u}@pre$ be the vector of the variables in $L$ with the appendage “@pre”. Let $\bar{u}@r_1$ be the vector of the same variables but with the appendage “@r_1”. Let $\bar{u}$ be the vector of the variables in $L$ without appendage. Let $\bar{u}$ be the vector of the same variables but with the appendage “@r_2”. Then:

$$L, \bar{u}@pre, \bar{u}@r_1, \bar{u}@r_2$$

is the relation that has to be satisfied after each iteration when the iteration starts at reference point $r_1$ and ends at reference point $r_2$.

It may seem unusual that the loop invariant relates two reference points. However, often the loop invariant has to refer to the state before the first iteration. For example, an invariant for a loop (with an iterator $i$) that sorts a list may be expressed like this: the list (at reference point $r_2$) contains the same elements as the list contained immediately before the first loop iteration (i.e., at reference point $r_1$); however, the first $i$ elements in the list (at reference point $r_2$) are sorted in increasing order.

• If $a$ is an instance of a class $A$ then $C_a$ denotes its class invariant. Moreover:

$$(C_a)^{this}$$

is the invariant condition of $a$ at reference point $r$.

• If $O$ is an operation (of an object $o$) then $I_O$ denotes its precondition and $F_O$ its postcondition. Let $\bar{x}@pre$ be the vector of the variables representing the initial values of all formal parameters and this. Let $\bar{x}$ be the vector of the variables representing the final values of all formal parameters and this (if the operation actually is a function then $\bar{x}$ is preceded by the variable result). Let $\bar{a}@r_1$ be the vector of the variables and (user-defined) functions on variables representing the initial values of all arguments and $o$ when the operation is invoked at reference point $r_1$. Let $\bar{a}@r_2$ be the vector of the variables and (user-defined) functions on variables representing the final values of all arguments and $o$ when the operation is returns at reference point $r_2$. Then:

$$(I_O)^{\bar{x}@pre}$$

is the condition that has to be satisfied when the operation is invoked at reference point $r_1$ and:

$$(F_O)^{\bar{x}, \bar{a}@pre}_{\bar{a}@r_2, \bar{a}@r_1}$$

is the condition that is satisfied when the operation, invoked at reference point $r_1$, returns at reference point $r_2$.

Example 5.11 on page 101 explains in detail how the axioms and inference rules presented in the following sections are applied to generate the questions used for the semantic design inspection.
5. Design Inspection

Use-Case Diagrams

As explained in Sect. 5.4.1 it is not feasible to verify that a use-case precondition implies the disjunction of all associated sequence preconditions. However, for each use-case postcondition it needs to be verified that it is implied by every sequence postcondition of associated sequences. That is, it is checked that there is no state that the system can assume according to the sequence postcondition but not according to the use-case postcondition.

5.1 Rule (Use-case)

Let $S_U$ be a complete sequence associated with the use-case $U$ and let $r$ be the reference point at which the sequence (and hence also the use-case) ends. Then the following rule applies:

$$P_{S_U} \land (Q_{S_U})^u_{\bar{u}\#r} \implies (Q_U)^u_{\bar{u}\#r} \quad \{P_{S_U}\} \quad S_U \quad \{(Q_{S_U})^u_{\bar{u}\#r}\}$$

In general, the above formula:

$$P_{S_U} \land (Q_{S_U})^u_{\bar{u}\#r} \implies (Q_U)^u_{\bar{u}\#r}$$

contains informal predicates and it is therefore in general not possible to formally prove that it is valid. Instead, this formula is, after certain automatic simplifications (see Sect. 5.4.3), presented as questions to the human inspector (see Chap. 6).

Sequence Diagrams

Each sequence diagram describes a specific sample execution of a use-case by means of a sequence. The notion of a sequence is defined in Def. 5.2, Def. 5.3, and Def. 5.4. In particular, Def. 5.2 defines partial sequences recursively by listing a basic partial sequence (i.e., the empty sequence) and by describing how larger partial sequences can be constructed from smaller ones. The semantics of a sequence is based on these definitions, i.e. for each basic sequence and construction rule an axiom or inference rule is specified. Thus we can be sure, that our semantics cover all complete sequences (i.e., we can generate questions for any complete sequence).

5.2 Axiom (Empty sequence)

Let $\epsilon_n$ be the empty partial sequence containing $n$ objects. Then the following axiom applies:

$$\{Q\} \quad \epsilon_n \quad \{Q\}$$
Larger partial sequences may be constructed from smaller ones by prepending messages (see Def. 5.2). Objects communicate via messages. Signals are asynchronous messages. The effect of a signal on the state of a system cannot be expressed in a postcondition as for operation invocations. One reason is that it is difficult to specify when the effects of a signal would actually take place (since there is no return message). Another reason is that it is difficult to express the effects of a signal in general terms because it can be dispatched and received under various conditions (states of the system). Therefore, the effect of a signal is expressed in subsequent intermediate assertions. One could say that signals simply transfer information from one object to another.

5.3 Rule (Signal)

Let $S_E$ be the partial sequence obtained by prepending a signal and two activation segments to the sequence $S$ (see below). Then the following rule applies:

\[
\{P\} \quad S \quad \{Q\}
\]

\[
\{A \oplus \text{pre}, \bar{u} \# r_1, \bar{u} \# r_2 \land B \oplus \text{pre}, \bar{v} \# r_3 \implies P\} \quad S \quad \{Q\}
\]

The effect of the signal $E$ is not obvious from above inference rule. However, the effect is that the arguments to the signal become visible to the receiving object and thus can be used e.g. in the intermediate assertion of activation segment $B$.

The above inference rule as well as the following inference rules and axiom may seem confusing because the formula preceding the sequence $S_E$ refers to logical variables defined later in the sequence (i.e., to later states of design variables). The rules and axioms require backward reasoning. The explanation is that designing (as well as programming) is considered a goal-oriented activity. That is, the final state plays a more important role than the initial state. Thus, the axioms and rules describe which initial states ensure that the final states satisfy the condition $Q$ (the axioms and rules here are very similar to the method invocation axiom for the programming language Java, i.e. Axiom 4.6 on page 38). Since all axioms and rules have similar characteristics we can compute even for a large sequence those initial states that ensure that the final states satisfy the sequence postcondition. Thus it only remains to show that the states described by the sequence precondition are contained in the computed initial states (and that is exactly what the questions presented to the inspectors are about).

Operation invocations (and returns) are synchronous messages. Each operation comes with a precondition and a postcondition. It is necessary to verify that the precondition
is indeed satisfied when the operation is invoked (i.e., questions need to be generated). After the return of the operation the caller can assume that the postcondition and the class invariant are satisfied (i.e., no question need to be generated). Operations usually change the state of a system. However, if an invoked operation actually is a function (i.e., it returns a value) then the state is not changed by that operation but by storing the returned value. To make that state change visible in a sequence, functions receive an additional argument (prepended to the other arguments) which is the return value of the function (i.e., the result variable in the postcondition is substituted with the additional argument, see Example 5.10).

5.4 Axiom (Operation invocation and return)

Let $S_O$ be the partial sequence obtained by prepending an operation invocation and two activation segments, and by appending an operation return and two activation segments to the sequence $S$ (see below). Let $b$ be the object that provides the operation $O$. Then the following axiom applies:

\[
\begin{align*}
(I_O) & \quad \bar{a} \preceq r_1 \
(F_O) & \quad \bar{a} \preceq r_1 
\end{align*}
\]

The above axiom indicates that we do not verify that an operation actually satisfies the postcondition and class invariant. We check that the precondition is satisfied when the operation is invoked and we assume that the postcondition and class invariant is satisfied when it returns. We believe that an early design in general is not detailed enough (i.e., the sequence $S$ above is often the empty sequence) for a meaningful verification of the postcondition and class invariant. It is trivial to verify the postcondition and class invariant if they are repeated as intermediate assertion of the activation segments immediately before the return, but it does not help in finding defects. We suggest that the operation postcondition and class invariant is verified later (e.g., when the code is available then it could be verified as described in Sect. 4). However, an alternative rule which verifies the postcondition and class invariant can be found later in Sect. 5.4.4.

Example 5.10

Figure 5.15 on the next page shows a partial sequence $S$ with a single operation invocation. The operation is actually a function noticeable by the return type (in the figure the function has an additional argument for the return value as explained earlier):

\[
\text{calcSpeed}(\text{in Plane } p) : \text{Speed}
\]
5.4. The Generation of Questions

The idea is that the function (which is provided by an object speedometer of the class Radar) determines the current speed of an airplane as determined by the radar. The precondition of the function as well as the class invariants of the classes Radar and Airplane are “true”. The postcondition $F$ of calcSpeed is:

ensures
\[ \text{result} = \text{currentSpeed}(\text{p@pre}, \text{this@pre}) \; ; \]

The intermediate assertions $A$ and $B$ of both the activation segments following the return of the operation are:

ensures
\[ \text{this} = \text{this@pre} \; ; \]

Now, let us assume that we want to compute the condition that ensures that the following condition is satisfied after $S$ (the formula can only contain logical variables due to the rules and axioms that had to be applied earlier, in particular due to Rule 5.1):

\[ \text{speed(}\text{plane}\#3) = 200 \]

From the Axiom 5.4 we get (we omitted the class invariants and precondition because they all are true and hence do not contribute):

\[
\begin{cases}
A \land B \land \text{speed(}\text{plane}\#2,\text{plane}\#3) = 200
\end{cases}
\]

\[
\begin{cases}
A \land B \land \text{calcSpeed(}\text{speed(}\text{plane}\#), \text{plane})
\end{cases}
\]

\[
\{ \text{speed(}\text{plane}\#3) = 200 \}\]
Thus the following condition has to be satisfied before $S$ (some simplifications as described in Sect. 5.4.3 would automatically be performed by a tool; we omitted them for illustration purposes):

\[
\begin{align*}
\text{plane#2} &= \text{plane#3} \\
\text{speed(plane#2)} &= \text{currentSpeed(plane#1, speedometer#1)} \\
\text{speed(plane#3)} &= 200
\end{align*}
\]

That is, if the speed of the plane plane#1 as determined by the radar speedometer#1 is 200 then the speed of the plane plane#3 is 200 as well.

Our subset of UML supports loops. Loops provide another means to construct larger partial sequences from smaller ones (see Def. 5.2). As described in Sect. 5.3.1, a loop is accompanied by an iterator, its initial value, a description of how the iterator changes during each iteration, a guard, and a loop invariant. To verify a loop questions need to be generated to verify that the invariant is initially satisfied and to verify that the invariant is maintained by the loop. After the termination of the loop it can be assumed that the invariant and the negation of the guard are satisfied. Loops always stretch over all objects. The next inference rule expresses the listed statements.

5.5 Rule (Loop)

Let $S_B$ be a complete sequence of $n$ objects. Let $S_L$ be the partial sequence (of $n$ objects) obtained by using $S_B$ as the body of a loop and by appending an activation segment to each of its $n$ objects (see below). Furthermore, let $G_L$ be the guard of the loop. Then the following rule applies:

\[
\begin{align*}
\{L_{\bar{u}, \bar{u} \text{pre}}^{r_1} \land G_L\} &\implies \{L_{\bar{u}, \bar{u} \text{pre}}^{r_2} \land G_L\} \\
\{L_{\bar{u}, \bar{u} \text{pre}}^{r_1} \land (L_{\bar{u}, \bar{u} \text{pre}}^{r_2} \land \neg G_L \land (A_1)_{\bar{u}_1} \land \cdots \land (A_n)_{\bar{u}_n} \implies Q)\} &\implies \{Q\}
\end{align*}
\]

The last means of constructing larger partial sequences described in Def. 5.2 is sequential composition which is treated in the next inference rule.
5.6 Rule (Composition)
Let $S_C$ be the partial sequence obtained by appending the partial sequence $S_2$ to the partial sequence $S_1$ (see below). Then the following rule applies:

\[
\{P\} \begin{array}{c}
S_1 \\
\{R\}
\end{array} \begin{array}{c}
\{R\}
\end{array} \begin{array}{c}
S_2 \\
\{Q\}
\end{array}
\]

\[
\begin{array}{c}
S_1 \\
\{P\}
\end{array} \begin{array}{c}
\{Q\}
\end{array}
\]

Definition 5.3 defines the notion of a complete sequence. A complete sequence is obtained by sequentially composing a sequence with a single activation segment for each object with a partial sequence. The next rule specifies the meaning of the completion.

5.7 Rule (Completion)
Let $S_S$ be the complete sequence obtained by prepending an activation segment to each object $a_i$ in the partial sequence $S$ (see below). Then the following rule applies:

\[
P \land (C_{a_1})_{a_1 \# r_1} \land \cdots \land (C_{a_n})_{a_n \# r_1} \land \\
(A_{i_1})_{u_{i_1} \# r_1}^v_{u_{i_1} \# r_1} \land \cdots \land (A_{i_n})_{u_{i_n} \# r_1}^v_{u_{i_n} \# r_1} \Rightarrow \begin{array}{c}
\{R\} \\
S \\
\{Q\}
\end{array}
\]

\[
\{P\} \begin{array}{c}
A_1 \\
\vdots
\end{array} \begin{array}{c}
A_n
\end{array} \begin{array}{c}
\vdash
\end{array} \begin{array}{c}
\{Q\}
\end{array}
\]

In general, the upper left formula in Rule 5.7 cannot be formally proven because it usually contains insufficiently axiomatized predicates. Instead the formula is simplified and normalized (see Sect. 5.4.3) to be presented as one or more questions to the human inspectors.

Example 5.11
The example is intended to illustrate how the questions to be answered during the semantic design inspection are generated. It does not describe a real-world scenario.

Figure 5.16 on the following page shows a sequence diagram TransferMsg that describes the temporal order of messages when a message is transferred from an object sender to another object receiver. The sequence diagram describes the normal-case trace without any errors and all required resources available (i.e., both the sender and the receiver are
initially idle). After the execution of the sequence the message has been transferred from the sender to the receiver. The message to be transferred is known to the sender (i.e., it is an attribute of a device) and thus no explicit reference to it is needed. Apart from the new message in receiver the sender and receiver should be unchanged. To express the previous statement a reference to the initial state of the sender and the receiver is required. The sequence pre- and postcondition are:\footnote{The actual definitions of the predicate symbols occurring in this example are omitted as they are not needed for the generation of questions. Of course, they are usually needed to answer the questions.}

\begin{verbatim}
context Sequence TransferMsg
   requires
      available(sender@pre) ∧ available(receiver@pre) ;
   ensures
      messageReceived(sender, sender@pre, receiver, receiver@pre) ;
end context Sequence TransferMsg
\end{verbatim}

The sequence diagram shows that the sequence consists of three steps. First a connection is established between the two devices by the operation \textit{connect}. Then the message is transferred by the operation \textit{transfer}. Finally, the connection is closed by the operation \textit{disconnect}.

Each operation is accompanied by a precondition and a postcondition. The operation \textit{connect} may only be invoked if neither the sender nor the receiver (which provides the
operation and hence is within the operation referred to as this) are already connected to another device. After the completion of the operation the objects are unchanged except that they are now connected. Devices may have other properties than their connection state. To verify other sequences we may need to rely on the fact that the operation connect does not change the state of the involved device in any other way than that it establishes a connection. In general, it is not sufficient to specify what an operation changes but also what it does not change (the frame problem discussed earlier). The pre- and postcondition are:

context Operation connect(sender)

requires
   available(sender@pre) \land available(this@pre) ;

ensures
   connectedEstablished(sender, sender@pre, this, this@pre) ;

end context Operation connect(sender)

The operation transfer may only be invoked if the calling device is connected with the called device. After the operation the devices are the same as before except that the receiver now keeps the transferred message in the corresponding attribute:

context Operation transfer(sender, message)

requires
   connected(sender@pre, this@pre) ;

ensures
   transferred(sender, sender@pre, message@pre, this, this@pre) ;

end context Operation transfer(sender, message)

Finally, the operation disconnect may only be invoked if the calling device is connected with the called device. After the operation the devices are the same as before except that they are now no longer connected:

context Operation disconnect(sender)

requires
   connected(sender@pre, this@pre) ;

ensures
   disconnected(sender, sender@pre) \land disconnected(this, this@pre) ;

end context Operation disconnect(sender)

When it comes to the class invariants we assume that they are all true, i.e. there are no restrictions on the possible states of the two devices. Since we, according to Axiom 5.4,
do not verify operations (i.e., we assume that operations are realized in such a way that they satisfy their postcondition) the intermediate assertions of activation segments $B_2$, $B_4$, and $B_6$ can be set to true. The intermediate assertion of activation segment $A_2$ has to introduce the argument $msg$ of the operation $transfer$. This is accomplished with the following intermediate assertion (where $message$ is a (user-defined) function that retrieves the message of a device):

```plaintext
context Object sender
    context Intermediate 2
        ensures
            this = this@pre ∧ msg = message(this@pre);
    end context Intermediate 2
end context Object sender
```

The remaining activation segments are assumed not to change the state of the system which is expressed by attaching the following intermediate assertion to them:

```plaintext
context Object sender
    context Intermediate 1,3,4
        ensures
            this = this@pre;
    end context Intermediate 1,3,4
end context Object sender
context Object receiver
    context Intermediate 1,3,5,7
        ensures
            this = this@pre;
    end context Intermediate 1,3,5,7
end context Object receiver
```

The question generation is based on the axioms and rules presented earlier in this section. Moreover, the notation used is the same as introduced earlier. The questions for the example at hand are generated by applying Rule 5.7 once, and Rule 5.6 and Axiom 5.4.
5.4. The Generation of Questions

three times:

\[
\begin{array}{c}
\{R_2\} \quad \{R_4\} \quad \{R_3\} \\
\hline
\{R_2\} \quad \{R_3\} \\
\hline
\{R_3\} \quad \{Q\}
\end{array}
\]

where (we omitted the class invariants since they are both “true”):

\[
\begin{align*}
R_1 & : P \land (A_1 \land (B_1) \Rightarrow R_2) \\
R_2 & : (I_{\text{connect}} \land (A_2 \land (B_2) \Rightarrow R_4) \\
R_3 & : (I_{\text{disconnect}} \land (A_3 \land (B_3) \Rightarrow Q) \\
R_4 & : (I_{\text{transfer}} \land (A_4 \land (B_4) \Rightarrow R_3)
\end{align*}
\]

Finally, the remaining formula that needs to be verified is \( R_1 \). If this formula is valid then the original formula, i.e. \( \{P\} \text{TransferMsg} \{Q\} \), holds as well. In general, the formula \( R_1 \) cannot be formally proven because it typically contains informal predicates. Therefore it is presented as a question to the inspectors. The first step is to rewrite the formula in the form of one or more questions (see Sect. 5.4.3). Using standard equalities of predicate logic the formula can be rewritten into four questions. The first question checks whether the precondition of the operation \( \text{connect} \) is satisfied when the operation is invoked. After some straightforward simplifications (see again Sect. 5.4.3) the first question looks like this:

**Assume:**
1. available(sender@pre)
2. available(receiver@pre)
5. Design Inspection

Then:
1. available(sender@pre)
2. available(receiver@pre)

Is the conclusion satisfied?

Obviously, the answer to that question is “yes”. In fact, the question could be automatically answered since the meaning of the involved predicate is not required.

The second question checks whether the precondition of the operation transfer is satisfied when the operation is invoked:

Assume:
1. available(sender@pre)
2. available(receiver@pre)
3. connectionEstablished(sender#2, sender@pre, receiver#2, receiver@pre)

Then:
1. connected(sender#2, receiver#2)

Is the conclusion satisfied?

The premise states that at reference point 2 a connection is established between two previously available devices (i.e., a sender and a receiver). The question is whether these two devices are connected at reference point 2. This is clearly the case and the answer to the question is therefore “yes”. If the involved predicates are defined informally, then the question cannot be answered automatically. To answer the question it is necessary to understand the informal meaning of the involved predicates. What the predicates mean and why the look as they do was discussed (though not defined) earlier in this example.

The third question checks whether the precondition of the operation transfer is satisfied when the operation is invoked:

Assume:
1. available(sender@pre)
2. available(receiver@pre)
3. connectionEstablished(sender#2, sender@pre, receiver#2, receiver@pre)
4. transfered(sender#4, sender#2, message(receiver#2), receiver#4, receiver@pre)

Then:
1. connected(sender#4, receiver#4)

Is the conclusion satisfied?

The premise states that two initially available devices (i.e., a sender and a receiver) are connected at reference point 2 and that a message was transfered at reference point 4.
5.4. The Generation of Questions

The questions is whether these two devices are still connected at reference point 4. This is the case because the predicate transfered express that the two devices are not changed apart from the new message in receiver. The answer to the question is therefore “yes”. As for the second question, the third question cannot be answered automatically unless the involved predicates are defined formally.

Finally, the fourth and last question checks whether the sequence satisfies its postcondition:

Assume:
1. available(sender@pre)
2. available(receiver@pre)
3. connectionEstablished(sender#2, sender@pre, receiver#2, receiver@pre)
4. transfered(sender#4, sender#2, message(receiver#2), receiver#4, receiver#2)
5. disconnected(sender#6, sender#4)
6. disconnected(receiver#6, receiver#4)

Then:
1. messageReceived(sender#6, sender@pre, receiver#6, receiver@pre)

Is the conclusion satisfied?

The premise states that two initially available devices (i.e., a sender and a receiver) are connected at reference point 2, that a message was transfered at reference point 4, and that the devices are disconnected at reference point 6. The question is whether the two devices at reference point 6 are the same as they were initially (i.e., at “@pre”) except that the receiver now stores the message it has received from the sender. The answer to this question is “yes” because the operation connect followed by the operation disconnect leaves the devices in a state which is the same as the initial one. The only state change that survives until the end of the sequence is caused by the operation transfer and that operation is establishing the condition demanded in the sequence postcondition. Like the second and third question, the above question cannot be answered automatically if the involved predicates are defined informally.

5.4.3. Simplifications

In the previous section we explained how the questions for the inspection of an early design are generated. The generation is based on the semantics of the notation used to express the design (i.e., UML). All well-formed formulae in the semantics are of the form \(A_1 \land (A_2 \implies A_3)\) where the \(A_1, A_2,\) and \(A_3\) are assertions, loop guards, or “true”. Thus, all the formulae (the implications \(P_S \land Q_{S_U} (r) \implies Q_U (r)\) and \(P \land \cdots \implies R\) in Rule 5.1 respectively Rule 5.7) that constitute the questions which
are eventually presented to the inspectors are of the form
\[ A_1 \Rightarrow (A_2 \land (A_3 \Rightarrow (A_4 \land (A_5 \Rightarrow (A_6 \land \ldots)))))) \]

where \( A_1, A_2, \ldots \) are again assertions, loop guards, or “true”. These formulae are easily (and automatically) rewritten in the form
\[ (A_1 \Rightarrow A_2) \land (A_1 \land A_3 \Rightarrow A_4) \land (A_1 \land A_3 \land A_5 \Rightarrow A_6) \land \cdots \]

Now, in principle each of the conjuncts in the previous formula is one question presented to the inspectors. However, we suggest that a few simplifications are carried out before the questions are actually presented. These simplifications require no further knowledge of the meaning of user-defined predicates that occur in the questions. Thus, they may be performed automatically. Let us assume that we have constructed a formula which will be presented as a single question to the inspectors. The formula is of the form
\[ (P_1 \land P_2 \land \cdots) \Rightarrow (C_1 \land C_2 \land \cdots) \]

where \( P_1, P_2, \ldots, C_1, C_2, \ldots \) are formulae as specified in Def. 3.2. We suggest that at least the following simplifications are carried out before the question is presented:

**Variable substitutions.** Sequence intermediate assertions are often of the form \( x = f(y_1@pre,\ldots,y_n@pre) \) where the right-hand side is a (user-defined) function in \( n \) arguments. For example, we may use \( \text{this} = \text{this}@pre \) to specify that the state of an object does not change or we may use \( \text{msg} = \text{message(this}@pre) \) to address an attribute of an object. As a consequence, some of the formulae in \( \{P_1, P_2, \ldots\} \) are of the form \( x#r_0 = f(y_1#r_1,\ldots,y_n#r_n) \) where \( r_0, r_1, \ldots, r_n \) are reference points of the sequence and \( \forall i : x#r_0 \neq y_i#r_i \). Then all occurrences of \( x#r_0 \) in the premise and conclusion of the question may be replaced with \( f(y_1#r_1,\ldots,y_n#r_n) \). Since functions have no side-effects the substitution obviously leaves the meaning of the question unchanged. After the substitution the identity \( f(y_1#r_1,\ldots,y_n#r_n) = f(y_1#r_1,\ldots,y_n#r_n) \) is removed from the premise.

**Conclusion reduction.** It may happen that some formulae appear both in the premise and in the conclusion of a question. That is, some of the formulae in \( \{P_1, P_2, \ldots\} \) are syntactically equal to some of the formulae in \( \{C_1, C_2, \ldots\} \) (e.g. when the same formula is used in a sequence precondition and an operation precondition). Then each formula \( A \) that appears both in the premise and the conclusion may be removed from the conclusion because we know that the \( A \) in the conclusion is implied by the \( A \) in the premise. If a conclusion is empty (i.e., it is true) the answer to the question has to be “yes” and hence answering the question is not necessary.

**Premise reduction.** The formulae in \( \{P_1, P_2, \ldots\} \) aggregate everything valid before the conclusion. However, usually all this information is not needed to answer a question. For example, to check whether an operation precondition of an object \( o \) is satisfied no information about other objects that previously neither directly nor indirectly dispatched messages to \( o \) is needed. Therefore, those formulae in \( \{P_1, P_2, \ldots\} \) that originate from unrelated objects are removed from the premise.
The formulae which can be removed are determined in a recursive fashion. Firstly, let \( F_1 \) be the set of formulae that share variables with any formulae in the conclusion. Then, let \( F_i \) be the set of formulae that share variables with any formulae in \( F_{i-1} \) or the conclusion. The second step is repeated until \( F_n = F_1 = F_{i-1} \). All the formulae in \( F_n \) may have an impact on the variables appearing in the conclusion and are hence needed to answer the question. All other formulae in \( \{ P_1, P_2, \ldots \} \) (i.e., the ones not in \( F_n \)) may be removed from the premise since they have no influence on the conclusion.

Additional simplifications may be possible by exploiting the meaning of the predefined operations and predicates (e.g. \(+, -, \leq, >\)). However, usually these operations and predicates occur only occasionally in early designs (and hence questions) and are not considered here.

**Example 5.12**

Let us assume that we have generated the following question from an early design with assertions (without any simplifications yet):

**Assume:**
1. \( m\#2 = \text{message}(\text{channel}\#1) \)
2. \( \text{validMessage}(m\#2) \)
3. \( \text{idle}(\text{receiver}\#1) \)
4. \( \text{idle}(\text{sender}\#1) \)

**Then:**
1. \( \text{idle}(\text{receiver}\#1) \)
2. \( \text{messageArrived}(m\#2,\text{receiver}\#1,\text{receiver}\#2) \)

Is the conclusion satisfied?

First we substitute all variables as described above. We have one formula in the premise which is of the form \( m\#2 = \text{message}(\text{channel}\#1) \). Thus we replace all occurrences of \( m\#2 \) with \( \text{message}(\text{channel}\#1) \) and we remove \( m\#2 = \text{message}(\text{channel}\#1) \) from the premise:

**Assume:**
1. \( \text{validMessage}(\text{message}(\text{channel}\#1)) \)
2. \( \text{idle}(\text{receiver}\#1) \)
3. \( \text{idle}(\text{sender}\#1) \)

**Then:**
1. \( \text{idle}(\text{receiver}\#1) \)
2. \( \text{messageArrived}(\text{message}(\text{channel}\#1),\text{receiver}\#1,\text{receiver}\#2) \)

Is the conclusion satisfied?
5. Design Inspection

Next we reduce the conclusion. The formulae $\text{idle(receiver#1)}$ appears both in the premise and in the conclusion. Hence, it is removed from the conclusion:

Assume:
1. $\text{validMessage(message(channel#1))}$
2. $\text{idle(receiver#1)}$
3. $\text{idle(sender#1)}$

Then:
1. $\text{messageArrived(message(channel#1),receiver#1,receiver#2)}$

Is the conclusion satisfied?

Finally, we reduce the premise. The variable $\text{sender#1}$ neither appears in the conclusion nor in any formula in the premise that shares variables with any formula in the conclusion. Thus all formulae that contain $\text{sender#1}$ as only variable are removed from the premise:

Assume:
1. $\text{validMessage(message(channel#1))}$
2. $\text{idle(receiver#1)}$

Then:
1. $\text{messageArrived(message(channel#1),receiver#1,receiver#2)}$

Is the conclusion satisfied?

The simplifications mentioned in this section make no use of the formal definitions supplied by the designer. When formal definitions are available they could be included in the simplification process. However, it is not obvious that the resulting questions would be easier to answer. Using the formal definitions probably allows to discard some questions automatically. On the other hand, some questions might be more difficult to answer because they now contain conditions on a lower level of abstraction.

5.4.4. Alternatives and Extensions

Several alternatives and extensions to the inspection of early UML designs as presented in this chapter are possible. In the following we present an alternative treatment of operations and two additional types of assertions to sequences (namely, global intermediate assertions and sequence invariants).
Alternative Treatment of Operations

Axiom 5.4 defines the meaning of the operation invocation. It specifies that the operation postcondition and class invariant are satisfied when the operation ends. However, the axiom does not express that the operation postcondition and class invariant must be satisfied when the operation ends. That is, according to Axiom 5.4 no questions are generated to verify that the operation postcondition and the class invariant actually are satisfied upon operation return. It is simply assumed that later implementations of operations always fulfill their part of the contract, i.e. that operations satisfy their postcondition and class invariant upon termination. The assumption may be verified at a later stage when the implementation is available (e.g. with semantic code inspection as described in Chap. 4). However, it is possible and may be desirable to verify the operation postcondition and class invariant already during the semantic design inspection. In that case Axiom 5.4 is replaced with the following Rule 5.8.

5.8 Rule (Operation return)

Let $S_O$ be the partial sequence obtained by prepending an operation invocation and one activation segments, and by appending an operation return and two activation segments to the sequence $S$ (see below). Let $b$ be the object that provides the operation $O$. Then the following rule applies:

Verifying already in the design inspection that each operation ends with the operation postcondition and the class invariant satisfied has several drawbacks:

- The design inspection is scenario based. Thus, it is only possible to verify that the operation ends in a state where the postcondition is satisfied for these particular scenarios. It is not known how the operation behaves under other circumstances.

- Even when the postcondition is partly verified in the design inspection it is necessary to verify that the actual implementation does the same (e.g. by using testing or the semantic code inspection).
5. Design Inspection

- In an early design details on how an operation behaves internally (in terms of dispatching and receiving messages) are often omitted. Hence, it does not make much sense to verify the postcondition of each operations. A verification in that context would require the designer essentially to copy the operation postcondition into the only activation segment between operation invocation and return.

However, applying Rule 5.8 instead of Axiom 5.4 also has advantages: for designs on a lower level of abstraction (i.e., design which describe the communication occurring within each operation) the rule provides an early indication of whether an operation implemented as expressed in the design is going to be correct or not.

Thus, whether Axiom 5.4 or Rule 5.8 should be preferred depends mainly on the level of abstraction in the design. For designs on a high level of abstraction Axiom 5.4 is the better choice whereas Rule 5.8 is better for designs on a lower level of abstraction.

Global Sequence Intermediate Assertions

A large sequence with many messages and no loops results in a few but complex questions (loops “partition” the sequence and lead to more but simpler questions). Global sequence intermediate assertions may be used to decrease the complexity of questions while increasing their number. A global sequence intermediate assertion is an assertion that stretches over all objects in a sequence and hence introduces a new reference point common to all objects. The global intermediate assertion then specifies the state of each object for that reference point. The idea basically is that the global intermediate assertion “splits” the sequence into two sequences with the assertion as postcondition respectively precondition of the upper respectively lower part. Thus, the more global intermediate assertions a sequence contains, the more but less complex questions are generated for the inspection. The obvious drawback of global intermediate assertions is that they introduce an artificial point of synchronization (i.e., all objects are synchronized where the global intermediate assertions is given). These synchronization may not be required or present in an actual implementation of the system. Nevertheless, a full-scale implementation of our semantic design inspection approach should include global intermediate assertions.

Sequences Invariants

A sequence describes the execution of a use-case in a state where the sequence precondition is satisfied. Often these preconditions (or parts thereof) are supposed to be satisfied during the whole sequence and not just when the sequence starts. For example, a sequence describing the normal execution of a use-case commonly may have a precondition
expressing that certain resources are available or that certain disturbances are absent. Often such a condition is supposed to be satisfied during the whole sequence. Therefore, other assertions in the sequence (e.g. intermediate assertions) explicitly have to state that the condition specified by the precondition is maintained.

A better solution may be to introduce *sequence invariants*, i.e. conditions that are assumed to be satisfied until the sequence ends. It is impractical to verify such a condition because it would be necessary to answer one question for each reference point at each object (i.e., very many questions). Hence, sequence invariants weaken our verification slightly, because we then verify that the postcondition of each sequence is satisfied if the precondition is initially satisfied and the invariant is always satisfied. Without sequence invariants we verify that the postcondition of each sequence is satisfied if the precondition is initially satisfied. However, sequence invariants simplify expressing and reason about the other assertions (e.g. intermediate assertions).

5.5. A Prototype Tool

We have implemented a prototype tool in the programming language Java (see e.g. [36]) with Java 2 SDK 1.3 and Swing. Swing is a GUI component kit for windowing components (e.g. menus, tool bars, dialogs). The implementation of our prototype comprises about 50 classes with a total of about 5500 lines of code. The prototype allows to enter a specification and an early design, and it generates and presents resulting questions in a single user environment. It does not support voting and re-inspections as described later in Chap. 6. All screenshots in this section are taken from the prototype tool.

The specification (see Fig. 5.17 on the next page) and the early design (see Fig. 5.18 on page 115 and Fig. 5.19 on page 116) cover the notations presented earlier in Sect. 5.2 and in Sect. 5.3 with the exception that neither non-void operations (i.e., operations that return a value) nor typed parameters are supported. Non-void operations have not been considered necessary since operations can modify their arguments. Types are not supported because they are not necessary for the generation of questions. However, types could be used to automatically find several kinds of defects and inconsistencies that would otherwise remain unnoticed. In a full-scale implementation of our semantic inspection types of some kind would certainly be included.

The questions which are presented to the inspector are generated as described earlier in Sect. 5.4.2. The prototype simplifies the question as explained in the same section. Figure 5.20 on page 117 shows how the questions are presented to the inspectors. In the

---

5The screenshots have been slightly altered to reflect recent changes in the graphical representation of method invocations.
5. Design Inspection

Fig. 5.17. A screenshot of the use-case diagram window and the use-case property window

upper half of the figure the sequence diagram window is shown. The lower half shows the question window. The right half of the question window displays the actual question. To guide the inspector to answer the question the definitions of all function and predicate symbols occurring in the question are provided in the left half of the question window. Moreover, in the sequence diagram window the relevant portion is highlighted and the reference points appearing in the question (1, 2, and 3) are listed at the left side of the sequence.

The tool we implemented is only a prototype to demonstrate the feasibility of our approach and to give an impression of what a full-scale tool may look like. For a full-scale tool we suggest the following extensions:

**Replacing formal parameters with actual parameters.** To answer the questions the inspector is supplied with the function and predicate definitions. To improve the readability all parameters in the definitions could be substituted with the corresponding arguments (of course it must be considered that the same variables may occur in parameters and in arguments in which case the affected parameters are renamed before the substitution). For informal function and predicate definitions the substitution is only possible if all formal parameters are recognizable e.g. by being enclosed in quotes.
Fig. 5.18. A screenshot of the sequence diagram window, the loop property window, and the activation segment property window
5. Design Inspection

Expanding and collapsing formally defined predicates. To help the inspector answering the questions, formally defined functions and predicates occurring in the questions could be replaced with their definition when clicked upon (and vice versa).

Showing informal definitions of predicates as tool tip. Instead of providing the definitions of predicate symbols in a compartment of their own they could be given in a tool tip when the mouse hovers over the predicate symbol in the definition.

Named reference points. Instead of sequentially numbering all reference points, full-scale tool implementations may allow to specify meaningful names for reference points.

5.6. An Example

To illustrate our approach we provide a small but non-trivial example. We describe what the specification, the (early) design, and the resulting questions may look like. This section contains all assertions (i.e., use-case pre- and postconditions, class invariants, intermediate assertions etc.) and questions. However, since the assertions and questions are intertwined with text, we provide the complete lists also in App. E on page 189 respectively App. F on page 195.
5.6. An Example

Fig. 5.20. A screenshot of the sequence inspection window and the sequence diagram window with the corresponding sequence elements highlighted
The design outlined could be part of a larger system, namely a base station in a mobile telephone network. Such a station mediates between the mobile telephones in a cell and the telephone network. We are going to look at a small part of the protocol that may be used by the station to communicate with the telephone network.

5.6.1. The Use-Case Diagram

We consider a single use-case sendMsg that describes the sending of a single message from a sender to a receiver over a given connection between the two. The corresponding use-case diagram can be found in Fig. 5.21. It contains one use-case named sendMsg and two actors which are the objects sender and receiver of class LinkUser. The use-case sendMsg is accompanied by a precondition and a postcondition. The precondition expresses what is satisfied before the message is sent – here mainly that the system is operational:

context Use-Case sendMsg
    requires
        readyToSend(sender@pre, receiver@pre);
end context Use-Case sendMsg

We use a single predicate symbol to express the precondition of the use-case. The symbol readyToSend is defined informally:

define readyToSend(sender, receiver) informally
    A full-duplex connection between the link user "sender" and the link user "receiver" is available. The link user "sender" has a message it wants to transfer to the link user "receiver". The link user "receiver" is prepared to receive a message from the link user "sender".
end define;

The postcondition specifies the result we expect after applying the use-case; in the case at hand, that the message is successfully transferred assuming that the connection is reliable:
5.6. An Example

context Use-Case sendMsg

ensures
  reliablyConnected(sender@pre, receiver@pre)
  $\implies$ message(sender@pre) = message(receiver);

end context Use-Case sendMsg

We use a new predicate symbol `reliablyConnected` and a new function symbol `message` in the postcondition. They are defined informally:

define reliablyConnected(sender, receiver) informally
  A full-duplex reliable connection between the link user “sender” and the link user “receiver” is available.
end define;

define message(entity) informally
  Returns the message of the link user or link entity “entity”.
end define;

Naturally, a full-scale tool (as well as our prototype) supporting our semantic design inspection would allow the assertions and predicate definitions to be attached to and stored with the use-case itself.

5.6.2. The Sequence Diagrams

In our example only one sequence diagram for the normal operation of the above use-case is given, i.e. it is assumed that all components are available and fully functional, and that no errors occur. The sequence diagram (see Fig. 5.22 on the following page) illustrates how a message is transferred in the normal case.

Every sequence diagram is accompanied by a scenario specification. The scenario specification consists of the sequence precondition and postcondition. Here the precondition specifies which conditions are satisfied in the case of the normal operation (i.e., a reliable connection exists between the involved objects, the send entity has not received any acknowledgments yet, and the receive entity has not received any segments yet):

context Sequence normalOp

requires
  reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre,
  receiveEntity@pre, receiver@pre) \&
  noAcksReceived(sendEntity@pre) \& noSegsReceived(receiveEntity@pre) ;
5. Design Inspection

Fig. 5.22. Sequence diagram normalOp
Again, we introduced new predicate and function symbols. The predicate symbol reliablyConnectedChannel, and the function symbols noAcksReceived and noSegsReceived are defined in natural language:

```plaintext
define reliablyConnectedChannel(sender, sEntity, ch, rEntity, receiver) informally
   A full-duplex reliable connection between the link user “sender” and the link
   user “receiver” via the link entity “sEntity”, the L2-channel “ch”, and the link
   entity “rEntity” is available.
end define;
define noAcksReceived(entity) informally
   The link entity “entity” has received no acknowledgment since the last
   successful message transfer.
end define;
define noSegsReceived(entity) informally
   The link entity “entity” has received no segments since the last successful
   message transfer.
end define;
```

The sequence postcondition specifies the conditions that are supposed to be satisfied when the sequence ends. Here it expresses that, when the sequence ends, the message in the receiver should be the same as the message initially in the sender:

```plaintext
context Sequence normalOp
   ensures
      message(sender@pre) = message(receiver);
end context Sequence normalOp
```

The sequence diagram contains five objects of which two are the actors sender and receiver that already appeared in the use-case diagram (see Sect. 5.6.1). The other three objects sendEntity, channel, and receiveEntity are part of our solution to transfer a message from the sender to the receiver. The send entity is responsible for breaking the message in small segments and for sending the segments one by one to the receive entity. After each segment the send entity has to wait for an acknowledgment from the receive entity (i.e., a stop-and-wait error control strategy is used). The receive entity is responsible for accepting and acknowledging the small segments and to eventually assemble them into a single message. Finally, the channel simply has to transfer segments and acknowledgments between send and receive entities.
First of all the message is transferred from the sender to the send entity. The sequence diagram shows that the message is transferred as argument \( \text{msg} \) of the signal \( \text{sendMsg} \). Therefore we have to define the argument \( \text{msg} \) before the signal is dispatched:

```plaintext
context Object sender
  context Intermediate 1
    ensures
      \( \text{msg} = \text{message(sender@pre)} \);  
  end context Intermediate 1
end context Object sender
```

The send entity receives the message and then segments it before the loop starts. We need to specify that the send entity is not changed before the message is received (so we know that it has not sent any segments yet), that the message which is segmented is the same as the send entity received from the sender, and that the segments are not modified before the loop is entered:

```plaintext
context Object sendEntity
  context Intermediate 1
    ensures
      \( \text{sendEntity} = \text{sendEntity@pre} \);  
  end context Intermediate 1
  context Intermediate 2
    ensures
      \( \text{replacedMessage(sendEntity,sendEntity@pre,msg)} \);  
  end context Intermediate 2
  context Intermediate 4
    ensures
      \( \text{sendEntity} = \text{sendEntity@pre} \);  
  end context Intermediate 4
end context Object sendEntity
```

In the second intermediate assertion we use a new predicate symbol \( \text{replacedMessage} \). It is informally defined like this:

```plaintext
define \( \text{replacedMessage(entityB, entityA, message)} \) informally  
The link user or link entity "entityB" is the same as the link user or link entity "entityA" except that the message of link user or link entity "entityB" is the message "message".
```
To ensure that the receive entity has not received any segments when the loop starts we may specify that it has not been modified since the sequence started (because the sequence precondition expresses that no segments nor acknowledgments have been received):

\[
\text{receiveEntity}_\text{pre} = \text{receiveEntity} ;
\]

The sending, transferring, and acknowledging of the segments is performed in a loop which is represented by a rectangular box over all objects in Fig. 5.22 on page 120. Associated with each loop is a so called \textit{iterator variable}. This variable changes its value in a user-defined way and captures the progress of the loop. Here the variable is \( i \) ranging over the natural numbers indicating that \( i - 1 \) segments have been sent and acknowledged. The iterator variable is initially 1 and it is increased by 1 after each iteration. The loop is iterated as long as \( i \leq \text{numOfSegs}(\text{sendEntity}) + 1 \). What the loop actually does is expressed with a loop invariant. Here the loop invariant has to express that when each iteration ends then \( \text{sendEntity} \) and \( \text{receiveEntity} \) are the same as they were when the iteration started, except that the first \( i \) segments have been received and acknowledged:

\[
\text{Loop}
\]

\[
\begin{array}{l}
\text{maintains} \\
\quad \quad i \leq \text{numOfSegs}(\text{sendEntity}) + 1 \land \\
\quad \quad \text{receivedSegAndAck}(r\text{EntityB}, r\text{EntityA}, s\text{EntityB}, s\text{EntityA}, i) ;
\end{array}
\]

To express the loop invariant we introduced one new predicate symbol \text{receivedSegAndAck} and one new function symbol \text{numOfSegs}. The new symbols are defined in natural language:

\[
\text{define receivedSegAndAck}(r\text{EntityB}, r\text{EntityA}, s\text{EntityB}, s\text{EntityA}, i) \text{ informally}
\]

The link entity \( "r\text{EntityB}" \) is the same as the link entity \( "r\text{EntityA}" \) except that the first \( "i" \) segments received by it are the same as sent by the link entity
5. Design Inspection

"sEntityA" (hence the number of received segments of "rEntityB" is "i"). The link entity "sEntityB" is the same as "sEntityA" except that the acknowledgments for the first "i" segments have been received from "rEntityB".

end define;

define numOfSegs(entity) informally
    Returns the total number of segments which the link entity "entity" wants to send.
end define;

An important task is to ensure that the loop invariant is maintained (i.e., that it is satisfied after each iteration of the loop. The loop invariant states that after each iteration another segment has arrived at the receive entity and that another acknowledgment has arrived at the send entity. The send and receive entities are unchanged in all other respects. Hence, we specify for the send entity:

context Object sendEntity
    context Intermediate 5
        ensures
            sendEntity = sendEntity@pre ∧ seg = segment(sendEntity@pre, i);
    end context Intermediate 5
    context Intermediate 6
        ensures
            sendEntity = sendEntity@pre;
    end context Intermediate 6
    context Intermediate 7
        ensures
            receivedPacket(sendEntity, sendEntity@pre, ack@pre);
    end context Intermediate 7
end context Object sendEntity

We use a new predicate symbols receivedPacket and a new function symbol segment in the above intermediate assertions. They are defined informally:

define segment(entity, i) informally
    Returns the "i"th segment to be sent by the link entity "entity".
end define;

define receivedPacket(entityB, entityA, packet) informally
    The link entity "entityB" is the same as the link entity "entityA" except that it has received "packet" (which is either a segment or an acknowledgment).
end define ;

The channel simply has to leave the data it is carrying unchanged. It just forwards the data to the other entity:

```plaintext
context Object  channel
  context Intermediate 3
    ensures
      seg = seg@pre ;
    end context Intermediate 3
  context Intermediate 5
    ensures
      ack = ack@pre ;
    end context Intermediate 5
end context Object  channel
```

In each loop iteration the receive entity receives one segment and sends one acknowledgement. Apart from this the receive entity remains unchanged:

```plaintext
context Object  receiveEntity
  context Intermediate 2
    ensures
      receiveEntity = receiveEntity@pre ;
  end context Intermediate 2
  context Intermediate 3
    ensures
      receivedPacket(receiver, receiver@pre, seg) ∧
      ack = acknowledgment(receiveEntity@pre, i) ;
  end context Intermediate 3
end context Object  receiveEntity
```

This time we use a new function symbol `acknowledgment` in the intermediate assertion for the third activation segment of the receive entity. The definition of the new function symbol is:
5. Design Inspection

\[
\text{\textbf{define} acknowledgment(entity, i) \textbf{informally}}
\]

Returns an acknowledgment from the link entity “entity” to the ”i”th segment.
\[
\text{\textbf{end define}};
\]

After the loop the segments in the receive entity are assembled in the same order they were received, and the resulting message is sent to the receiver:

\[
\begin{array}{l}
\text{context Object receiveEntity} \\
\text{context Intermediate 5} \\
\quad \text{ensures} \\
\quad \quad \text{receiveEntity} = \text{receiveEntity}@pre; \\
\text{end context Intermediate 5} \\
\text{context Intermediate 7} \\
\quad \text{ensures} \\
\quad \quad \text{receiveEntity} = \text{receiveEntity}@pre \land \\
\quad \quad \text{msg} = \text{message(receiveEntity}@pre); \\
\text{end context Intermediate 7} \\
\end{array}
\]

The last activation segment of the receiver needs to have an intermediate assertion which expresses that the receiver contains the recently received message:

\[
\begin{array}{l}
\text{context Object receiver} \\
\text{context Intermediate 4} \\
\quad \text{ensures} \\
\quad \quad \text{replacedMessage(receiver, receiver}@pre, \text{msg}@pre); \\
\text{end context Intermediate 4} \\
\end{array}
\]

This concludes the description of the sequence for the normal operation of the use-case sendMsg.

5.6.3. The Class Diagrams

There are three types of objects (i.e., classes) in the sequence diagram in Fig. 5.22 on page 120. These classes are LinkUser, LinkEntity, and L2Channel. The operations provided by the classes and the relationships between the classes are further described in the class diagram shown in Fig. 5.23 on the next page (for convenience we have...
5.6. An Example

omitted the types and visibility information). The class diagram is annotated with class invariants, and operation pre- and postconditions. In the case at hand, we do not specify any class invariants (i.e., they are all “true”). We only need to specify the pre- and postconditions of the two operations of the class LinkEntity.

The precondition of the operation segment is true (i.e., there is no particular condition that has to be satisfied prior to the invocation of the operation). Hence we omit the requires-clause. The postcondition of the operation expresses that when the operation ends then the link entity is the same as before, except that the message stored in the entity is broken up into segments which are in turn stored in the entity:

context Class LinkEntity
  context Operation segment()
    ensures
      segmented(this, this@pre) ;
  end context Operation segment()
end context Class LinkEntity

To express the postcondition we use a new predicate symbol segmented. It is defined in natural language:

define segmented(entityB, entityA) informally
  The link entity “entityB” is the same as the link entity “entityA” except that the message that “entityA” wants to send has been broken up into numOfSegs("entityA") segments. The message is broken up into at least one segment.
end define ;

Also the precondition of the operation assemble is true and thus omitted. The postcondition of the operation specifies that when the operation ends, then the link entity is the same as before except that the segments in the entity are assembled and stored as the entity’s new message:
5. Design Inspection

context Class LinkEntity
  context Operation assemble()
    ensures
      assembled(this, this@pre);
  end context Operation assemble()
end context Class LinkEntity

Again we introduce a new predicate symbol to express the postcondition of the operation. The predicate symbol \texttt{assembled} is defined like this:

\begin{verbatim}
define assembled(entityB, entityA) informally
  The link entity “entityB” is the same as the link entity “entityA” except that the segments received by it are assembled into a message that is contained in “entityB”.
end define ;
\end{verbatim}

Now, we have given a complete description of an annotated design. The next step is to generate the questions which form the basis for the semantic design inspection.

5.6.4. Questions

For the design to be correct a number of conditions must be satisfied. These conditions constitute the questions to be asked (and answered) during the design inspection. The idea is that all questions must be answered with “yes” for the design to be correct with respect to the specification.

The first questions that we would expect for the example at hand asks the inspectors to check whether the precondition and postcondition of the sequence imply the postcondition of the corresponding use-case:

\begin{verbatim}
Question 1 Assume:
  1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
  2. noAcksReceived(sendEntity@pre)
  3. noSegsReceived(receiveEntity@pre)
  4. message(sender@pre) = message(receiver)

Then:
  1. reliablyConnected(sender@pre, receiver@pre)
     \implies message(sender@pre) = message(receiver)
\end{verbatim}
5.6. An Example

Is the conclusion satisfied?

The premise states that the sender and the receiver are connected and that the transmissions on that connection are error-free. Hence we know that the predicate reliablyConnected in the conclusion is satisfied. This means that the formula \( \text{message}(\text{sender}@\text{pre}) = \text{message}(\text{receiver}) \) in the conclusion needs to be satisfied. We can easily see that the formula is satisfied since it also appears in the premise. Therefore the answer to the question should be “yes”.

The second and third questions ask the inspector to check whether the preconditions of the operations segment respectively assemble are satisfied when the corresponding operation is invoked:

**Question 2**

**Assume:**
1. reliablyConnectedChannel(\( \text{sender}@\text{pre}, \text{sendEntity}@\text{pre}, \text{channel}@\text{pre}, \text{receiveEntity}@\text{pre}, \text{receiver}@\text{pre} \))
2. noAcksReceived(\( \text{sendEntity}@\text{pre} \))
3. noSegsReceived(\( \text{receiveEntity}@\text{pre} \))

**Then:**
1. true

**Is the conclusion satisfied?**

**Question 3**

**Assume:**
1. reliablyConnectedChannel(\( \text{sender}@\text{pre}, \text{sendEntity}@\text{pre}, \text{channel}@\text{pre}, \text{receiveEntity}@\text{pre}, \text{receiver}@\text{pre} \))
2. noAcksReceived(\( \text{sendEntity}@\text{pre} \))
3. noSegsReceived(\( \text{receiveEntity}@\text{pre} \))
4. replacedMessage(\( \text{sendEntity}@2, \text{sendEntity}@\text{pre}, \text{message}(\text{sender}@\text{pre}) \))
5. segmented(\( \text{sendEntity}@3, \text{sendEntity}@2 \))
6. receivedSegAndAck(\( \text{receiveEntity}@9, \text{receiveEntity}@\text{pre}, \text{sendEntity}@9, \text{sendEntity}@3, \text{numOfSegs}(\text{sendEntity}@3) \))

**Then:**
1. true

**Is the conclusion satisfied?**

Since the preconditions of both operations are true, the answers to both questions are trivially “yes”. The questions would be discarded automatically and not presented to the inspector. We list them only for completeness.
5. Design Inspection

The fourth question is concerned with the loop invariant. The loop invariant must be satisfied immediately before the loop is executed for the first time. The question therefore is:

**Question 4**

Assume:
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. replacedMessage(sendEntity#2, sendEntity@pre, message(sender@pre))
5. segmented(sendEntity#3, sendEntity#2)

Then:
1. $1 \leq \text{numOfSegs}(sendEntity#3) + 1$
2. receivedSegAndAck(receiveEntity@pre, receiveEntity@pre, sendEntity#3, sendEntity#3, 0)

Is the conclusion satisfied?

The first item in the conclusion is satisfied because we know from item 5 in the premise that the number of segments in sendEntity#3 is at least one. An entity is equal to itself when it has received neither segments nor acknowledgments. Hence item 2 in the conclusion is satisfied as well. The answer to the question is therefore “yes”.

The fifth question also concerns the loop invariant. The following question checks whether the loop invariant is maintained by the body of the loop:

**Question 5**

Assume:
1. $i \leq \text{numOfSegs}(sendEntity#4) + 1$
2. receivedSegAndAck(receiveEntity#4, receiveEntity#4, sendEntity#4, sendEntity#4, i - 1)
3. receivedPacket(receiveEntity#7, receiveEntity#4, segment(sendEntity#4, i))
4. receivedPacket(sendEntity#9, sendEntity#4, ack(receiveEntity#7, i))

Then:
1. $i \leq \text{numOfSegs}(sendEntity#9)$
2. receivedSegAndAck(receiveEntity#7, receiveEntity#4, sendEntity#9, sendEntity#4, i)

Is the conclusion satisfied?

The major issue is if sendEntity#9 respectively receiveEntity#7 are the same as sendEntity#4 respectively receiveEntity#4 except that i segments have been received and ac-
An Example

We know that for sendEntity#4 and receiveEntity#4 already $i - 1$ segments have been received and acknowledged (item 2 in the conclusion). Furthermore we know that receiveEntity#7 is the same as receiveEntity#4 except that it has received segment $i$ from sendEntity#4 (item 3 in the premise) and that sendEntity#9 is the same as sendEntity#4 except that it has received an acknowledgment to segment $i$ from receiveEntity#7 (item 4 in the premise). Therefore, we can conclude that the answer to the first subquestion should be “yes”.

The minor issue is whether $i$ is less than or equal to the total number of segments that sendEntity#9 wants to send (item 1 in the conclusion). The premise tells us that $i$ is less than or equal to the total number of segments that sendEntity#4 wants to send plus 1 (item 1 in the premise). Moreover, we know that sendEntity#9 is the same as sendEntity#4 when it comes to the number of segments kept in the send entity $i$ from receiveEntity#7 (item 4 in the premise). Hence, the answer to the second subquestion is also “yes”.

The final question is to check if the postcondition of the sequence is satisfied when the sequence ends:

**Question 6**

**Assume:**
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. replacedMessage(sendEntity#2, sendEntity@pre, message(sender@pre))
5. segmented(sendEntity#3, sendEntity#2)
6. receivedSegAndAck(receiveEntity#9, receiveEntity@pre, sendEntity#9, sendEntity#3, numOfSegs(sendEntity#3))
7. assembled(receiveEntity#11, receiveEntity#9)
8. replacedMessage(receiver#13, receiver#12, message(receiveEntity#11))

**Then:**
1. message(sender@pre) = message(receiver#13)

Is the conclusion satisfied?

The question is whether the message in receiver#13 is the same as the one in sender@pre. From item 8 in the premise we know that the message in receiver#13 is the same as in receiveEntity#11. The receiveEntity#11 is the same as the receiveEntity#9 except that the segments in the receiveEntity#11 have been assembled (item 7 in the premise). The receiveEntity#9 is the same as the receiveEntity@pre except that it has received numOfSegs(sendEntity#3) segments from the sendEntity#3 (item 6 in the premise). The
5. Design Inspection

sendEntity\#3 is the same as the sendEntity\#2 except that it contains the segments of
the message in the sendEntity\#2 (item 5 in the premise). Finally, the sendEntity\#2 is
the same as the sendEntity@pre except that it contains the message message(sender@pre)
(item 4 in the premise). Therefore the answer to this question is “yes” as well.

Since we answered each question with “yes” the sequence diagram is considered correct
with respect to the scenario specification.

5.6.5. Omitting a Message

The example design presented in the previous sections is correct (with respect to the
assertions). But if an early design is incorrect, how would an actual defect manifest itself?
First let us assume that we accidentally forgot to invoke the assemble operation of the
receive entity after the loop. As a consequence we would have only one activation segment
between the end of the loop and the receiveMsg event instead of the segments 5, 6, and
7. Let us moreover assume that we simply specified receiveEntity = receiveEntity@pre as
intermediate assertion for that activation segment (i.e., the state of the receive entity
remains unchanged). The only question effected by our omission is the sixth question.
It would now look like this:

Question 6.1

Assume:
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre,
   receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. replacedMessage(sendEntity\#2, sendEntity@pre, message(sender@pre))
5. segmented(sendEntity\#3, sendEntity\#2)
6. receivedSegAndAck(receiveEntity\#9, receiveEntity@pre, sendEntity\#9,
   sendEntity\#3, numOfSegs(sendEntity\#3))
7. replacedMessage(receiver\#13, receiver\#12, message(receiveEntity\#9))

Then:
1. message(sender@pre) = message(receiver\#13)

Is the conclusion satisfied?

The question is whether the message in receiver\#13 is the same as the one in sender@pre
(item 1 in the conclusion). From item 7 in the premise we know that the message in
receiver\#13 is the same as in receiveEntity\#9. The receiveEntity\#9 is the same as the
receiveEntity@pre except that it has received numOfSegs(sendEntity\#3) segments from the
sendEntity\#3 (item 6 in the premise). Thus, the answer to the question is “yes” only if the
message in receiverEntity@pre is the same as the one in sender@pre. However, the sequence precondition gives no hints concerning the message contained in the receiverEntity@pre. Therefore, we cannot answer the question with “yes”.

From item 6 of the premise of question 6.1 we know that receiveEntity#9 already contains all segments that make up the message that the sender wants to transfer to the receiver. Thus we can conclude that the defect resides somewhere after reference point 9 and that it concerns the receive entity and/or the receiver. Moreover, the question indicates that the design is erroneous because it is not assembling the segments into a message.

5.6.6. Intermediate Assertion to Weak

Another defect may be that one of our intermediate assertions is not strong enough. In the extreme case we may completely forget to specify an intermediate assertion which then is assumed to be “true”. Let us assume that we did not specify the intermediate assertion for the fifth activation segment of the channel. The intermediate assertion of the fifth activation segment of the channel only appears in the question to check if the loop invariant is maintained (i.e., question 5). When the intermediate assertion is not given then the question would look like this instead:

**Question 5.1**

Assume:
1. \( i \leq \text{numOfSegs}(\text{sendEntity#4}) + 1 \)
2. receivedSegAndAck(receiveEntity#4, receiveEntity#4, sendEntity#4, sendEntity#4, \( i - 1 \))
3. receivedPacket(receiveEntity#7, receiveEntity#4, segment(sendEntity#4, \( i \))
4. receivedPacket(sendEntity#9, sendEntity#4, ack#8, \( i \))

Then:
1. \( i \leq \text{numOfSegs}(\text{sendEntity#9}) \)
2. receivedSegAndAck(receiveEntity#7, receiveEntity#4, sendEntity#9, sendEntity#4, \( i \))

Is the conclusion satisfied?

The major issue is if sendEntity#9 respectively receiveEntity#7 are the same as sendEntity#4 respectively receiveEntity#4 except that \( i \) segments have been received and acknowledged (item 2 in the conclusion). From item 2 in the premise we know that for sendEntity#4 and receiveEntity#4 already \( i - 1 \) segments have been received and acknowledged. Furthermore we know that receiveEntity#7 is the same as receiveEntity#4 except that it has received segment \( i \) from sendEntity#4 (item 3 in the premise). Item 4 in the premise tells us that sendEntity#9 is the same as sendEntity#4 except that it
5. Design Inspection

has received an unspecified acknowledgment $\text{ack#8}$. We do not know where the $\text{ack#8}$ comes from and which segment it actually acknowledges. Therefore we cannot answer the question with “yes”.

We may suspect that the defect occurs somewhere between reference point 6 (where the receive entity has received the $i$th segment) and reference point 8 (where the send entity has received an insufficiently defined acknowledgment).
6. The Inspection Process

In this chapter we present an inspection process supported by our generation of questions. The primary aim of this thesis is not the development of a process, however, our approach enables a novel process based on voting. The process is somewhat similar to the three stage inspection process suggested by Sauer, Jeffery, Lau, and Yetton (see [107] and Sect. 7.2).

The core of every inspection process is the defect detection phase. Our approach precisely defines the defect detection phase; it comprises evaluating formulae (of the form \( P \implies C \)) originating from the verification conditions generated from the annotated software artifact (see Sect. 4.3 and Sect. 5.4). An inspector reading such formulae does this with an interpretation, or a set of interpretations, in mind; namely, the possible interpretations of the informal and formal definitions. We refer to these interpretations as models of the definitions. Although most likely not thinking in terms of algebraic structures we still formulate the inspector’s models of the definitions as a set of first order interpretations, \( \text{Mods} \). Ideally there is only one model of the definitions, but natural language is by nature ambiguous leading in general to multiple models of the definitions.

Let \( \text{Mods} \) be the set of models of some definitions. We say that a closed formula \( F \) is satisfiable if there is at least one interpretation in \( \text{Mods} \) where \( F \) is true. We say that \( F \) is valid if \( F \) is true in every interpretation in \( \text{Mods} \). If \( F \) is unsatisfiable, i.e. if there is no interpretation in \( \text{Mods} \) where \( F \) is true, then \( F \) is said to be inconsistent.

Verification conditions contain only logical variables. By the notation \( \forall(F) \) we mean the universal closure of a formula \( F \), i.e. the formula \( F \) with all logical variables universally quantified at the top of the formula. Similarly \( \exists(F) \) denotes the existential closure of \( F \).

The presentation of a formula \( P \implies C \) can be done in several ways:

- The simplest approach is to ask the question \( \forall(P \implies C) ? \) This is the approach adopted in our previous examples although quantifiers are not explicitly written out. We propose allowing three answers to such a question:
The Inspection Process

- **Yes** if $\forall (P \implies C)$ is valid,
- **No** if $\forall (P \implies C)$ is inconsistent,
- **Don’t know** otherwise.

The don’t-know-answer signals a critical ambiguity in the definitions (i.e., $\forall (P \implies C)$ is true in some models and false in others). Of course, if the definitions are unambiguous then the last answer is impossible. Actually, yet another form of don’t-know answer may be motivated, in the case that the inspector is unable to determine the truth-value of a formula for complexity reasons.

- An alternative, more advanced way to present the formula $P \implies C$ is to ask three different questions taking inconsistencies in the premise into account: (1) $\exists P$, (2) $\forall (P \implies C)$, and (3) $\exists (P \wedge C)$? Again, we propose allowing the three answers to each of the three questions:
  - **Yes** if the formula is valid,
  - **No** if the formula is inconsistent,
  - **Don’t know** otherwise.

The three questions are presented one after the other. The second question is only presented if the answer to the first formula is valid and the third question is only presented if the second formula is inconsistent.

The alternative with three questions instead of one allows a better classification and localization of the defect:

- If $\exists P$ is inconsistent, then we have a defect in the artifact or the assertions (D1 in Fig. 6.1 on the facing page). The defect resulted in a contradiction in the premise, e.g. we may have specified both $x = 0$ and $x \neq 0$ in (different) assertions. Thus we have to look either for contradicting assertions or unexpected dependencies in that part of the artifact used to generate the formula $P \implies C$.

- The second question is only presented if the first formula is valid. If $\forall (P \implies C)$ is inconsistent, then the artifact or the assertions contain a defect. The third question helps in the classification of the defect.

- The third question is only presented if the second formula is inconsistent (i.e., we know already that the artifact or the assertions contains a defect). If $\exists (P \wedge C)$ is valid, then our assertions may be too weak while the artifact may still be correct (D2 in Fig. 6.1). For example we may have $x > 0$ as premise and $y < 0$ as conclusion. Then the artifact itself may be correct, we just might be insufficiently strong in our assertions. If the formula is inconsistent instead we have a not further specified defect in those assertions or that part of the artifact used to generate the formula $P \implies C$ (D3 in Fig. 6.1).
Figure 6.1 also indicates that the artifact may be assumed to be correct with respect to the assertions if we answer the first two subquestions of each formula $P \implies C$ with “yes”. If we at any time answer with “don’t know” we have a ambiguity in the informal function or predicate definitions.

For either alternative, the inspectors should provide an informal justification of their answers. Both the answers and the justifications are logged for documentation, group inspections, and possible re-inspections. If the inspector answers “yes” to all questions generated for a specific artifact then the artifact is considered correct with respect to the assertions (and the informal definitions).

Our approach defines precisely what to do in the defect detection phase (namely, answering the automatically generated questions). Hence, our approach may be adapted to different existing inspection processes. The resulting process has the advantage of being more precisely defined, e.g. the inspection phase is easy to document (storing the answers to the questions), and it is repeatable and systematic. Moreover, re-inspections after modifications of the design are facilitated since only those questions that are influenced by the modifications have to be answered. Questions that have not changed from the original inspection need not to be answered again (provided no definitions of involved predicate symbols are altered).

However, the automatic generation of questions facilitates a novel type of inspection based on voting. Even though it was not our aim to define a new inspection process, we suggest the following inspection process to fully exploit our approach:
6. The Inspection Process

1. During the planning phase the inspection team is selected, the software artifact to inspect is chosen, and a deadline for the individual inspections is defined.

2. The inspection phase consists of two steps. In the first step each member of the inspection team answers individually the questions generated for the inspected artifact. After the deadline the answers of all members are collected and compared. If a (qualified) majority answered “yes” to each question then the inspection is over. Here the answers “no” and “don’t know” may be treated differently, e.g. we might use a weight of 3 for “yes”, 1 for “don’t know”, and 0 for “no”. If there is no majority for some questions the inspection team meets to discuss these particular questions. If there still is no majority then a potential defect has been found.

3. Finally, during the rework phase the discovered defects have to be removed. A defect may be a fault or an omission in the specification or in the design. Thus the specification or the design have to be corrected or extended and an additional inspection has to be scheduled. However, as mentioned earlier, only the new questions that arise from the modification have to be investigated. Since each question is generated only from a part of the artifact, the question helps to locate the defect. For example, in a design inspection we might have a question that checks whether a precondition of an operation is satisfied prior to its invocation. However, only a part of the sequence diagram influences the state prior to the invocation of the operation. Hence, the defect must be located in that part.

An inspection may be performed as soon as a single method is implemented or as soon as a single sequence diagram is completed. Of course, the specifications of methods respectively operations used in the artifact have to be available (e.g. pre- and postconditions). Thus, our approach enables the inspection team to find defects in an software artifact early. The size of the inspection team and the required majority for the acceptance of an artifact is dependent on the desired degree of confidence.

The voting approach facilitates distributed, asynchronous inspection. In conventional inspection the inspection phase includes a group meeting for group inspection and logging. Group meetings are typically limited to a maximum duration of two hours. Larger artifacts thus require several inspection meetings. Each meeting is associated with a large overhead due to problems finding a mutually agreeable time, a room for the meeting etc. This overhead is to a large extent avoided in our voting approach.

Tool-support is essential for our type of inspection. When it comes to our inspection of Java code, then the tool should allow to enter the assertions and the Java code. The tool should be able to generate the questions and to present them to the user. The presentation of each questions should include both the question and the relevant parts of the code highlighted. For the inspection of early UML designs the tool should allow to enter the specification and the design (i.e., annotated use-case, class, and sequence dia-
grams). Moreover, the tool should be able to generate the questions and to present them
to the inspector. The presentation should include the question itself and the diagrams
involved with the relevant parts highlighted (as in our prototype implementation). In
both types of inspection the tool has to collect the answers and to establish the outcome
of the voting. All the services above have to be provided in a distributed environment
with a central repository. It is actually not necessary to build such a tool from scratch.
For example, the functionality outlined for UML design inspection may be realized as
a “plug-in” to an existing tool as Rational Rose (see [101]) or a Group Support System
(see [48]). Earlier in Sect. 4.4 and in Sect. 5.5 we described the prototype tools we have
developed to evaluate our approach and to show its feasibility.
6. The Inspection Process
7. Related Work

The basis of our approach to systematic functionality-oriented inspection is conventional inspection and formal verification. A great number of books, articles, and conferences are concerned with these topics and we can only mention a fraction of them. However, existing approaches seem to be either completely formal or completely informal. We are not aware of other approaches that combine conventional inspection and formal verification to achieve a systematic inspection process. The remainder of this chapter is organized as follows. In Sect. 7.1 we will briefly present a few software development processes related to our approach. Different types of conventional inspections, annotation-based inspections, and tool support are discussed in Sect. 7.2. In Sect. 7.3 we shortly introduce and discuss several approaches to formalizing UML (or parts thereof). Our automatic generation of questions relies heavily on sequence diagrams. Sequence diagrams are similar to message sequence charts which we discuss in Sect. 7.4. Finally, different applications of various types of assertions are presented in Sect. 7.5.

7.1. Software Development

For code we suggest Dijkstra’s discipline of programming to be used as development method (see Sect. 4.2). For the more general software development we suggest USDP (see Sect. 5.1). However, many different software development method exist. In this section we further elaborate on software development approaches:

Assertion-driven programming. An approach similar to Dijkstra’s Discipline of Programming called assertion-driven programming has been proposed by van Emden (see [113]). This method allows the development of the required assertions and the code during the same process, where the assertions are driving the code development as in Dijkstra’s method. However, van Emden’s method is based on Floyd’s method for the verification of flowcharts (see [38]) and does not produce code according to structured programming.

Fusion. Fusion (see [24]) is a software development method for object-oriented software and covers analysis, design, and implementation. The analysis phase produces
the object model and the interface model. The object model describes the static structure of the system and its environment with entity-relationship diagrams. The interface model describes the operations the system provides (in the operation model) and the system communication (in the life-cycle model). The design phase produces four types of models called object interaction graphs, visibility graphs, class descriptions, and inheritance graphs. Object interaction graphs describe how run-time objects interact to perform each operation captured in the operation model. Visibility graphs describe how objects have access to each other as required in the object interaction graphs. Class descriptions list all attributes and methods of each class. Finally, inheritance graphs highlight the inheritance relationships between classes.

In Fusion’s operation model every operation is equipped with a precondition and a postcondition (in a so called operation schema). These conditions are expressed informally using a mixture of pseudo-code and natural language. An operation schema produced in the analysis phase would be best compared to a use-case in our approach since it describes a high level operation performed by the system. How the operation works is specified in an object interaction graph which is similar to a sequence diagram. However, an object interaction graph is supposed to capture not just a single scenario but all possible interactions that may occur (decomposition may be used). To achieve this goal, an object interaction graph is accompanied by a description again in a mixture of pseudo-code and natural language. When it comes to the verification it is said on page 78 in [24] (text in “[ ]” added by us):

Check that the functional effect of each object interaction graph satisfies the specification of its system operation given in the operation model.
Check that every clause in the schema Results [i.e., the postcondition of the operation] is satisfied.

Thus, Fusion comprises a complete software development process that relies to a large extent on the notion of operation pre- and postconditions. However, when it comes to the verification the potential of assertions is not exploited and the developer is left without support.

7.2. Software Inspections

Inspection of code was first defined by Fagan in 1976 (see [34]). It is a visual examination of a software artifact to find defects. A reader is paraphrasing the artifact and the other members of the inspection team, equipped with lists of errors known to be likely and clues that usually betray their presence, are trying to find these kinds of errors. For this kind of inspection the team usually has 3 to 6 members. Several variants of the
7.2. Software Inspections

Inspection process have been presented in the literature. For example, Two-Person Inspections by Bisant and Lyle (see [10]), N-Fold Inspections by Schneider, Martin, and Tsai (see [108]), Phased Inspections by Knight and Myers (see [65]), Abstraction-Driven Inspection by Macdonald, Miller, Brooks, Roper, and Wood (see e.g. [81] and [30]), Active Design Reviews by Parnas and Weiss (see below), and Program-Function-Table based inspection by Parnas et al. (see below). Inspection techniques have been introduced for various artifacts that are created in the software development process (i.e., requirements, design, code, and tests). For example, Porter, Votta, and Basili (see [100]) experimentally compared different approaches to requirement inspections.

Some results for code inspections have been reported. Fagan (see [34]) argues that design and code inspections increase the productivity and improve the final program quality. Ten years later, in 1986 he (see [35]) suggests slight modifications to the code inspection process and reports further industrial experiences that support his earlier results. Russell (see [105]) and Kelly, Sherif, and Hops (see [62]) describe similar findings. Glass (see [44]) argues that inspection lets programmers take responsibility for their code thus improving the code quality. In general, software inspections are reported to be very effective both with respects to defect-detection and costs. Still, what actually has to be done in an inspection session is usually only loosely defined – it is mostly dependent on the experience and discipline of the involved personnel. It is not clear how a inspection should be documented or repeated. Changes to the software artifact may require new inspections of large parts of the artifact.

A detailed description of the current practice of inspections, including several case studies, may be found in a book by Gilb and Graham (see [43]). A collection of various articles about inspection including introductory articles, experience reports, process measurements, inspections of other products than code and other inspection-related work has been published by the IEEE (see [115]).

7.2.1. Criticism to Conventional Inspection

Criticism to conventional design reviews has been presented by Parnas and Weiss (see [96]). As problems with conventional design inspection they list among other circumstances the following:

The reviewers are swamped with information, much of which is not necessary in order to understand the design. Structural decisions are hidden in a mass of implementation details.

There is no systematic review procedure and no prepared set of questions to be asked about the design.
7. Related Work

Parnas' and Weiss' approach to avoid the problems of conventional reviews is to redesign the design review process. They suggest that several types of reviews, each with a different focus, should be performed. First of all a good design documentation is needed. Such a documentation makes the assumptions explicit and includes redundant information in the design representation. Possible types of reviews suggested by Parnas and Weiss are then e.g.: assumption validity, assumption sufficiency, and consistency between assumptions and functions. For each type of review a questionnaire is provided to the reviewer. We agree to Parnas' and Weiss' discussion on problems with conventional reviews. However, their solution is still based on generic (with respect to the inspected artifact) checklists. By imposing a certain structure on the provided assumptions and redundant information (i.e., that they are assertions) it is possible for us to generate a questionnaire which is specific for the inspected artifact and abstracts from implementation (or design) details.

Later Parnas et al. presented another approach to the inspection of critical software. Their approach is based on tabular mathematical expressions (see [90]) used to describe programs. Firstly, a tabular description of what the program is supposed to do (i.e., the requirements) is (manually) produced. Secondly, a tabular description of what the program actually does is (manually) produced. Finally, it is checked whether the two descriptions are equivalent (see e.g. [95], [66], and [92]). Their approach is somewhat similar to our approach. We too have a description of what the software artifact is supposed to do and we compare it to what it actually does. However, we allow informal (natural language) elements in the description of what the artifact is supposed to do, and we extract what the artifact is doing automatically.

Porter, Siy, Toman, and Votta (see [99]) have been investigating the effects of structural changes of the (code) inspection process (i.e., team size, number of sessions, repair occasions) on the inspection performance (i.e., inspection effectiveness and interval). However, they discovered that the performance varied widely independent of the treatment used. Also their data indicated that only 15% of the issues reported during individual preparation concern true defects. This suggests that it may be much more important to develop better defect detection techniques than new inspection processes. Porter, Siy, Mockus, and Votta further studied the source of the variations found in there earlier experiment (see [98]). In particular they considered the influence of process inputs, (e.g. code units and reviewers) on the defect detection. Porter et al. found that even when the process inputs are accounted for, then structural changes of the inspection process had little effect on the defect detection. Therefore, according to Porter et al., research on better techniques for the actual defect detection steps should not be neglected.

Similar considerations have been presented by Sauer, Jeffery, Lay, and Yetton. In contrast to most approaches that favor a two stage inspection (i.e., individual defect detection followed by group meeting), they propose a three stage inspection based on
behavorial theory (see [107]). The three stage inspection consists of the individual defect detection, the defect collection, and the defect discrimination. According to Sauer et al. behavioral theory states that the level of expertise of the individual inspectors is the most important factor in determining the effectiveness of the inspections. The effectiveness of inspections can be improved by selecting better inspectors, by training the inspectors, and by increasing the number of inspectors (to the limit of performance). The contribution of the group meeting lays not in the detection of new defects but in the discrimination of true defects and false positives. Later experiments by Land, Jeffery, and Sauer (see [68]) seem to support their approach.

Our semantic software inspection approach is in line with the findings by Porter et al. and Sauer et al. We strive to improve the defect detection abilities of the individual inspector by defining precisely the questions that need to be addressed to check whether the software artifact contains defects.

7.2.2. Inspection Based on Informal Annotations

The Cleanroom method (it presents a practical approach to place software development under statistical quality control to deliver software with known and certified mean time to failure, see [31]) comprises verification-based inspections. These inspections define more strictly how the defect detection should be performed. Verification-based inspections presuppose a design using the constructs from structured programming theory, written in a design language, and organized under a top-down incremental development strategy. That is, the design takes the form of annotated pseudo-code where the annotations are informal assertions written in natural language. The questions to be considered in the inspection typically have the form: "Does the behavior specification of line $x$ satisfy the behavior specification on line $y$ when the predicate on line $z$ is true?".

A similar method describing more precisely what actually has to be done in an inspection session was introduced by van Emden in 1992 (see [113]). His code inspection method is based on Floyd’s method for the verification of flowcharts (see [38]). The basic idea is to first exhaustively annotate the code with completely informal assertions (not necessarily with complete coverage of assumptions). Then, during the inspection session it is checked whether the next assertion along the execution path may be concluded from the former assertion and the instruction between the two assertions.

However, both in the Cleanroom and in van Emden’s inspection the assertions are informal and hence it is not possible to automatically generate verification conditions that reflect the high level algorithmic content of the design. Thus the questions are not independent from the actual text. That is, the inspectors have to consider the effect of the design on the assertions themselves which in general is rather cumbersome. In our
7. Related Work

approach assertions are expressed as well-formed formulae (see Chap. 3). These formulae are formal even though they may contain predicates defined informally (e.g. in natural language). These formulae (together with the semantics of the artifact notation) enable various kinds of automatic support which otherwise are not possible, in particular predicate transformations, arithmetic simplifications, and the generation of verification conditions are possible. Thus our questions make the high level algorithmic content of the design explicit.

7.2.3. Tool Support for Software Inspections

Several tools to support software inspections have been developed. For example, in 1996 Macdonald and Miller (see [80]) published a comparison of five tools. Since then many other inspection tools have been developed and three years later Macdonald and Miller presented in [79] a comparison of 16 inspection tools and their own tool ASSIST (which is described in more detail in [78]). However, the focus of these tools is almost entirely on administrative tasks. The tools support in varying degrees the artifact handling and the data collection, e.g. to ensure that all inspection team members actually inspect the same version of the artifact. The actual inspection phase may be supported by checklists, by ensuring that all members are presented the same piece of the artifact, by checking that all parts of the artifact have been inspected etc. The task of actually finding functional defects in the individual artifact is not supported except by providing the environment and general guidelines. In our approach we provide automatically generated guidelines that are specially tailored for the inspected artifact (e.g. the design) at hand. These guidelines state precisely which questions need to be address to find defects.

7.3. Formalizations of UML

The lack of formal semantics of UML has been recognized as a major problem. Ambiguities hinder the exchange of UML models between different parties and also impede the development of computer support for analysis and verification. For some constructs of the notation not even experts in the field have the same interpretation. Thus, several attempts to formalize UML have been published. An overview can be found e.g. in [32]. A (not complete) list of approaches to formalizing UML comprises:

Use-cases in the refinement calculus. Back, Petre, and Paltor (see [4]) presented an approach that allows formalizing UML use-cases in the refinement calculus. That is, use-cases are annotated with a kind of pre- and postconditions in a formal contract language. The contract language allows to verify the use-cases against a kind of annotated class descriptions.
7.3. Formalizations of UML

**FuZed.** In [18] Bruel and France present *FuZed*. Their approach combines the informal object-oriented analysis technique Fusion (see Sect. 7.1) with the formal specification notation Z. In FuZed informal operation models are, with human intervention, translated into formal validation models expressed in Z. A tool called *FuZE* generates Z specification “shells” from class diagrams. These specifications then have to be completed by a human equipped with guidelines. In [40] France and Bruel describe how the integrated Fusion/Z technique (using UML as notation for the Fusion models where possible) is used to model and to analyze the Invoicing case study proposed for the Invoice ’98 Workshop. In the paper the analysis is restricted to type checking and discharging of general Z proof obligations. However, according to the authors the formalization activity itself already provides additional opportunities for uncovering defects in the models.

**PUML.** Bruel’s and France’s work on formalizing UML (see previous paragraph) is related to the Precise UML (PUML) project. The objective of the PUML research is to develop a formal foundation for a core subset of UML. The PUML project was initiated by Evans, France, Lano, and Rumpe who describe several problems with the missing formal semantics of UML (see e.g. [33] or [39]). Without a precise semantics the readers and the creators of models may have different interpretations of the models without even noticing it, the developers may waste considerable time discussing which interpretation and usage of model elements is appropriate, the rigorous semantic analysis of UML models is difficult, and only limited tool support is possible. Evans, France, Lano, and Rumpe argue that the problems can be overcome by formalizing the UML. The authors chose the specification language Z as the formal language to describe the semantics of UML (see e.g. [41] and [109] on formalizing UML class structures in Z).

**vUML.** Lilius and Paltor (see [74]) developed vUML, a tool for automatically verifying UML models. UML models are translated into the PROcess MEta LAnguage (PROMELA) language and model-checked with the SPIN model checker. The behavior of the objects is described using UML statechart diagrams (the formal semantics of UML state machines is described by the same authors in [73]). The user of the vUML tool neither has to know how to use SPIN nor PROMELA. If the verification of the model fails, a counterexample described in UML sequence diagrams is generated. The vUML tool can check that a UML model is free of deadlocks and livelocks as well as that all the invariants are preserved. In [72] Lilius and Paltor show how the Production Cell (a standard example for evaluating methodologies for designing embedded systems) can be modeled in UML and how the UML model can be verified with the vUML tool.

**Syntropy.** Like our approach, the Syntropy approach presented by Cook and Daniels (see [25]) relies on annotating a graphical notation. Cook and Daniels annotate class
7. Related Work

and statechart diagrams with (formal) formulae in predicate logic. The resulting notation allows e.g. automatic checks for consistency and completeness.

**BOOM.** The general goal of the BOOM project is to develop a formal object-oriented language for the specification of modeling languages like UML. The semantics of the specification language (called *Odal*) is formally defined in the \( \pi \)-calculus (a process algebra). Odal is used to define an operational semantics for some UML elements. In [89] Óvergaard and Palmkvist present a formal approach to UML use-cases and their relationships, in [87] Óvergaard presents a formal approach to UML relationships, and in [88] he presents a formal approach to UML collaborations.

**System Model (refinement of SysLab system model).** The SysLab project (see [15]) aims at the development of a mathematically founded modeling technique for distributed, object-oriented systems. In [16] Breu et al. outline their approach to formalizing UML based on their experience from the SysLab project. The approach is based on the “well-studied and established mathematical theory of streams and stream processing functions” augmented by the notion of *system models*. A system model provides an integrated view of a system and describes both its static and dynamic properties.

The above list shows different approaches to the formalization of UML. A precise semantics of UML is a prerequisite for our semantic inspection of early designs. However, we are not using UML alone; we extend it with assertions. To simplify stating assertions and reasoning with assertions, we chose an assertion notation similar to OCL but with relaxed requirements on formal rigor (see Chap. 3). This allows us to verify *functional* properties (e.g. opposed to time properties as in [71]) of *early* and *incomplete* designs. We can argue for the correctness of the software in certain scenarios which is, as we believe, an easier and more practical approach. The focus of the above approaches is either not on functionality or the approaches require a complete behavioral description (e.g. in the form of statechart diagrams). However, such descriptions are typically not available at an early stage.

7.4. Sequence Charts

The key diagram type for the generation of questions used in the design inspection (see Sect. 5.4) is the sequence diagram. UML sequence diagrams are very similar to Message Sequence Charts (MSC) and Live Sequence Charts (LSC) described in the next two sections (see [103]).
7.4. Sequence Charts

7.4.1. Message Sequence Charts

MSCs are concerned with the communication behavior of protocols and have been standardized by the International Telecommunications Union (ITU). A short introduction to MSCs can be found e.g. in [104].

Several formalizations of MSCs have been suggested to avoid different interpretations of specific features of MSCs and to allow computer support. Among them are approaches based on automata theory (see [67]), Petri net theory (see [45]), and process algebra (see [83]). Common to these approaches is their focus on the communication, i.e. on the order and number of messages received and dispatched. The actual calculations (i.e., state changes) that are performed are not considered. However, it is necessary to take these state changes into account to check whether the system described in the sequence chart provides the intended functionality. In our approach the state changes are expressed, using well-formed formulae (see Chap. 3), within sequence intermediate assertions, loop invariants, operation preconditions, operation postconditions, and class invariants. This enables us to generate questions that help the inspector to detect functional defects in the sequence diagram.

7.4.2. Live Sequence Charts

LSCs introduced by Damm and Harel (see [26]) are an extension to standard MSCs. In contrast to MSCs (which only describe possible behavior), LSCs allow to distinguish between possible and necessary behavior. The intention was to provide an expressive MSC language with a precise formal semantics. The semantics is the basis to relate an inter-object specification (e.g. LSCs) to an intra-object specification (e.g. statecharts) of a system. Klose and Wittke (see [64]) describe how a subset of LSCs can be transformed into Timed Büchi Automata from which temporal logic formulae can be generated. In principle this allows to verify (using a model checker) a statechart model of the system against LSCs (temporal logic formulae). A similar approach is pursued by Harel and Kugler (see [50]). They describe how an object model expressed as a state machine with synchronous communication can be constructed from a restricted LSC language.

In our design inspection we verify a system design (expressed in annotated class diagrams) via scenario descriptions (expressed in annotated sequence diagrams) against a system specification (expressed in annotated use-case diagrams) via scenario specifications (expressed in sequence pre- and postconditions). The approach with LSCs is closer related to our code inspection where we verify a system design (expressed in annotated class diagrams against code). Class diagrams comprise operations with associated pre- and postconditions. These operations appear in sequence diagrams as messages. Hence we have a connection between sequence diagrams and code, i.e. we can indirectly ver-
ify that the implementation complies with the sequence diagram. What is missing in our approach is the verification of the implementation against the sequence intermediate assertions. In principle this should be possible since activation segments end only at specific points (e.g. when a message is received). However, this would need further investigation. Another possibility is to combine our approach with the LSC approach since state machines are common as an intermediate step towards the code. The state machines or temporal logical formulae constructed from a set of LSCs as outlined above focus on temporal properties of the system. We, on the other hand, are interested in functional properties. Therefore, the state machines and the temporal logical formulae would have to consider additional state changes as expressed e.g. in assertions.

7.5. Assertions

Assertions are used mainly for formal verification, dynamic checking, and documentation. Some assertions may serve more than one use. For example, assertions used for formal verification provide a precise documentation as well. In the next three sections we will discuss the three types of applications of assertions.

7.5.1. Formal Verification

Assertions as introduced by Hoare are formal logic formulas. These assertions are typically used to annotate code which allows to formally prove certain properties of the code. A great deal of (academic) work has been conducted in this area. Hoare’s approach has been extended in several ways to cover special programming language constructs, e.g. pointers (see [8]), procedure calls (see [54], [82], [47], and [9]), recursive procedures (see [51]), data structures (see [55]), aliasing (see [20]), and gotos (see [27]). In [114] a Hoare-style calculus for a substantial subset of Java can be found.

One of the newer and more successful approaches to formal software verification is the Ada subset called SPARK (see [19]). SPARK is a subset of Ada 83 that is extended with annotations. The restrictions to the Ada language are partly introduced to ensure predictability of a program’s behavior and partly to ensure simplicity of the formal language definition and the proof arguments. Some annotations are required to perform extended static code analysis and comprise e.g. a specification of which variables are imported and exported by a procedure and how they are related. Other annotations, so called proof contexts, introduce elements of formal specifications and proof obligations, e.g. pre-, postconditions, loop invariants, and intermediate assertions for procedures. Since SPARK has a formally defined semantics, formal program verification is possible and supported by the SPARK Examiner. This tool checks the conformance of a program
to the rules of SPARK, carries out a flow and information analysis of the code, and supports formal verification. However, in contrast to our approach, SPARK aims at low level properties, e.g. the absence of run-time errors (see [42]).

### 7.5.2. Dynamic Checking

Several approaches exploit code annotations to improve dynamic testing, e.g. Robust C (see [37]), APP (see [102]), Anna (see [76]), and C-Patrol (see [116]). These approaches either extend the underlying programming language or they introduce special kinds of comments to allow specifying assertions together with the code. Typically, the assertions are written in the same notation as the code itself. If an assertion is not satisfied during execution then an exception is thrown. A slightly different approach is used by ADLT (see [111]), where the (interface) specifications are not mixed together with the code; the specification is stored in a different file instead.

In [22] Cline and Lea present Annotated C++ (called A++). A++ enables the user to specify for each class the *legal* and the *coherent* objects of that class and its *behavior*. The constraints which instances of a class always have to satisfy identify the legal objects of that class. The coherent objects form a subset of the legal objects and are analogous to self consistency: all public methods of a class can expect a coherent object and they have to ensure to restore coherence before terminating. Thus, the constraints on legal and coherent objects are related to the notion of a class invariant. Finally, several conceptual conditions specify the classes implementation independent behavior. Assertions in A++ may use universal and existential quantification, but otherwise they are expressed using a subset of the expressions of C++. A++ statically verifies to the extent possible that the constraints are satisfied. When this is not possible, A++ generates code to check the constraints dynamically.

In [84] Meyer stresses the importance of reliability for object-oriented technology. As reuse is a cornerstone of the object-oriented approach, incorrect behavior is considered even more serious than in application specific software. The pragmatic techniques presented in the article rely on the theory of design by contract. The notion of contract is taken from everyday life where one may contract out certain subtasks instead of doing them by oneself. Each party of a contract expects some benefits and obligations. The contract protects both the client by specifying how much should be done and the contractor by specifying how little is acceptable. For software assertions play the role of contracts. Meyer refers to the Eiffel (see [85]) programming language and the notation used to specify class invariants and pre- and postconditions of methods there. In Eiffel these conditions are to be specified in the notation of the programming language itself. During the software development these assertion may be compiled together with the application and thus used to monitor if the contracts are fulfilled or not.
7. Related Work

In comparison to simple black box testing the above approaches improve error detection and decrease the necessary debugging effort to find the underlying defect. However, since assertions are executed together with the program they may lead to different behavior of the program with monitoring activated or deactivated. Moreover, in practise exhaustive testing is generally not possible. Therefore additional means to ensure the correctness of the software need to be applied. Our semantic inspection approach is one such instrument.

7.5.3. Documentation

Assertions may also be used for specification and documentation. In practise the documentation of software is typically completely informal and rather unstructured. To use assertions is a way to make the documentation more structured and concise. Assertions used for documentation may range from completely formal assertions to completely informal ones.

Larch/C++ (see [21]) is an interface specification language that enables a precise description of both the behavior and the interface of parts of a C++ class library or module. An interface specification in Larch/C++ consists of two parts: some Larch/C++ specific text and some text in the Larch Shared Language (LSL). The idea behind this approach is that the behavior of an operation of an abstract data type is not described in terms of the representation of that type but in terms of “abstract values”. An interface specification consists of a requires, an ensures, and a modifies clause similar to our semantic code inspection approach. The requires clause contains the precondition of the method and the ensures clause its postcondition. The modifies clause contains a list of variables that are modified by the method.

Blom, Nordby, and Brunström have conducted two industrial case studies (see [11]) to develop a software design method with a focus on semantics called Selma. Selma (see [12]) is based on Meyer’s design by contract approach (see [84]) and provides a structured documentation of code. The basic idea is that contracts help maintaining the semantic integrity of software components. Unfortunately, no reports on their academic and industrial experiments have been published yet.
8. Conclusion and Future Work

We have presented a novel approach to the inspection of software artifacts. In contrast to existing approaches we combine conventional inspection with elements of formal verification. Conventional inspection in the style of Fagan (see [34]) lacks guidelines that precisely state which questions to address to find (functional) defects. Formal verification in the style of Hoare (see [53]) is well established in academia but hardly known in industry. We believe this is the case because formal verification is often perceived as being very difficult due to the required knowledge of the formal notation and its proof system. Our hypothesis is that a method which is more easily used in practice but which still allows automatic tool support, may be achieved by relaxing the requirements on formal rigor in a controlled manner.

In our approach, a software artifact (e.g. a design model or a piece of code) is annotated with assertions. The assertions are expressed in a notation similar to predicate logic and specify a condition on the state of the modeled system. By introducing particular types of assertions it is possible to express the assumed and intended functionality of the system. Our key idea concerns the predicates used to express assertions. We allow the use of predicates without a formal axiomatization beyond the point where a formal proof of the correctness of the artifact may no longer be possible. All predicates may be defined informally, e.g. in natural language. Nevertheless, it is still possible to automatically generate (from an artifact with assertions) those questions that need to be discussed in the inspection. Thus, the somewhat loosely defined content of the steps of Fagan’s code inspection method are filled with a concrete content. If all questions generated with our approach can be answered positively then the artifact can be considered correct with respect to the assertions (limitations are described in Sect. 4.3.1 and Sect. 5.4.1).

It should be noted that the assertions, even if they are not used for the generation of verification conditions, still provide a structured, semi-formal documentation of the artifact.

To illustrate, investigate, and evaluate our approach we applied it to code (the code is expressed in a subset of Java and the specification in class descriptions, see Chap. 4) and early designs (the design is expressed in a subset of UML class and sequence diagrams, and the specification in a subset of UML use-case diagrams, see Chap. 5). For both types
of artifacts we implemented a prototype tool and applied it to small but non-trivial programs (up to 1000 lines) respectively designs (up to five objects and ten messages).

The automatic generation of question allows a new inspection process. In Chap. 6 we presented a distributed, asynchronous, repeatable, and systematic inspection process. Moreover, re-inspections after modifications of the software artifact are facilitated.

Our semantic inspection is far from complete. For the future we would like to see the following issues addressed:

- In-depth evaluations involving large real-world applications are still to come. Our experience so far has been promising. Our industrial partner ABB Industrial Systems has stated that it seems apparent that “COMPASS is a simplification compared to traditional methods”. However, an in-depth evaluation is not possible without an investment of substantial industrial resources over many years.

- An issue requiring more attention is the number and complexity of the assertions and the generated question. It is possible to decrease the complexity of assertions and questions by increasing their number and vice versa. For example, the introduction of sequence diagram invariants (i.e., conditions that remain satisfied during the whole execution of a sequence) decrease the complexity of assertions but increase their number. Global intermediate assertions in sequence diagrams (i.e., conditions that specify the state of the whole system at a certain reference point) increase the number of assertions and questions but decrease their complexity. The future challenge is to find the right mix of complexity and amount of assertions and questions.

- A major area of possible extensions of our semantic inspection concerns the interaction with the user. The method presented here requires the user to specify assertions and to answer questions in a notation similar to predicate logic. However, this might not be the easiest way to express the intended functionality. An alternative may be to write the specification of a program (i.e., the assertions) in a tabular form as presented by Parnas (see [90], [93] and Sect. 7.2) instead. Parnas reports that the tabular notation allows “to describe a program in a more readable manner” compared to the conventional notation.

- To provide precise semantic descriptions of Java and UML was not our primary goal. Nevertheless, as discussed in Sect. 4.3.1 some questions regarding the formal specification and verification of object-oriented programs are still open. Though not directly related to our intentions, it might be of interest to investigate if and how our approach is affected and if it may help to overcome some of these challenges (e.g. integrity and proof obligations due to overriding).

- Dijkstra proposed a discipline of programming where what the program is sup-
posed to do (i.e., the assertions) drive the program development (see Sect. 4.2). Similar techniques may be developed for our semantic inspection approach. It is not reasonable to first develop a software artifact and to annotate it with assertions afterwards. Instead some assertions (mainly pre- and postconditions) should be specified first, and then the software artifact should be developed in a systematic way that ensures its correctness. For our semantic code inspection such a discipline could be very similar to Dijkstra’s. However, for our semantic design inspection such a discipline has to be invented.

• We applied our approach to Java code and early UML designs. The semantic code inspection presupposes the availability of a detailed design in the form of annotated class diagrams. Even though annotated class diagrams are part of our early design, we believe there is a gap. A method to cover the whole software development process has to fill this gap (e.g., using statechart diagrams). Moreover, it would be interesting to investigate how the steps in the development process should be formed to maintain the validity of the assertions. This could be similar to the approach by Lano and Evans (see [70]) who proposed a development process which makes use of transformations on UML models (which are formally defined in Real-time Action Logic, see [69]).

To summarize, our main contributions presented in the thesis at hand are:

• We developed a novel method to specify and to reason about the functionality of software artifacts annotated with assertions. The novelty is that we allow insufficiently (for a formal proof) axiomatized predicates in assertions.

• Although not our main goal, we introduced a new way to understand the meaning of sequence diagrams with a focus on functionality rather than timing.

• We precisely defined which questions need to be addressed in an inspection to find functional defects, thus filling the somewhat loosely defined steps in conventional inspections with a very concrete content.

• We suggested a new asynchronous and distributed inspection process based on voting. The process takes advantage of the fact that the questions which need to be addressed in the inspection can be automatically generated.

• We developed prototype tools and applied our ideas to the inspection of code written in a subset of Java and early designs expressed in a subset of UML. From a technical point of view our approach is working and has successfully been applied to example artifacts.

Despite the open questions and the lack of extensive assessment, our experience so far indeed indicates that informal predicates make it easier to express assertions at a high level of abstraction which makes the algorithmic content of a software artifact explicit.
Moreover, the notion of informal predicates enables reasoning with assertions and verifying software artifacts at an appropriate level of abstraction. However, to successfully apply our semantic inspection method in practice it is indispensable to carefully introduce the potential user in the underlying principles. Furthermore, extended tool support is required.
Bibliography


Bibliography


public class Topology
    /**
     * maintains
     *   validTopology(this);
     */
    {
        private int[][] track_table;
        private int[][] point_table;
        private int num_tracks, num_points;

        public Topology(/* const */ int elems)
            /* ensures */
            emptyTopology(this, elems@pre);
            modifies
                this;
            */
            {
                track_table = new int[elems][2];
                point_table = new int[elems][3];
                num_tracks = 0;
                num_points = 0;
            }

        public void addTracks(/* const */ int tracks)
            /* requires */
            tracksEmpty(this@pre, tracks@pre);
            ensures
                addTracksMod(this@pre, this, tracks@pre);
            modifies
                this;
            */
            {
                num_tracks = num_tracks + tracks;
            }
A. Topology.java

```java
public void addPoint(/*+ const*/ int base, /*+ const*/ int plus, /*+ const*/ int minus)
/**+
requires
pointEmpty(this@pre) ∧
openTracks(this@pre, base@pre, plus@pre, minus@pre);
ensures
addPointMod(this@pre, this, base@pre, plus@pre, minus@pre);
modifies
this;
*/
{
    point_table[num_points][0] = base;
    point_table[num_points][1] = plus;
    point_table[num_points][2] = minus;
    num_points = num_points + 1;
    if(track_table[base - 1][0] == 0)
        track_table[base - 1][0] = num_points;
    else track_table[base - 1][1] = num_points;
    if(track_table[minus - 1][0] == 0)
        track_table[minus - 1][0] = num_points;
    else track_table[minus - 1][1] = num_points;
    if(track_table[plus - 1][0] == 0)
        track_table[plus - 1][0] = num_points;
    else track_table[plus - 1][1] = num_points;
}

public int numTracksTopo()
/**+
ensures
result = numTracksTopo(this@pre);
*/
{
    return num_tracks;
}

public int numPointsTopo()
/**+
ensures
result = numPointsTopo(this@pre);
*/
{
    return num_points;
}
```
public int pointBase(/*\textbf{const}*/ int point)
/** requires */
0 < point@pre ∧ point@pre ≤ numPointsTopo(this@pre) ;
ensures
result = pointBase(this@pre, point@pre) ;
*/
{
    return point_table[point - 1][0];
}

public int pointPlus(/*\textbf{const}*/ int point)
/** requires */
0 < point@pre ∧ point@pre ≤ numPointsTopo(this@pre) ;
ensures
result = pointPlus(this@pre, point@pre) ;
*/
{
    return point_table[point - 1][1];
}

public int pointMinus(/*\textbf{const}*/ int point)
/** requires */
0 < point@pre ∧ point@pre ≤ numPointsTopo(this@pre) ;
ensures
result = pointMinus(this@pre, point@pre) ;
*/
{
    return point_table[point - 1][2];
}

public int trackLeft(/*\textbf{const}*/ int track)
/** requires */
0 < track@pre ∧ track@pre ≤ numTracksTopo(this@pre) ;
ensures
result = trackLeft(this@pre, track@pre) ;
*/
{
    return track_table[track - 1][0];
}

public int trackRight(/*\textbf{const}*/ int track)
/** requires */
0 < track@pre ∧ track@pre ≤ numTracksTopo(this@pre) ;
ensures
result = trackRight(this@pre, track@pre);
*/
{
    return track_table[track − 1][1];
}

/** define validTopology(topo) informally 
  Each point in "topo" is connected to exactly three different track
  segments. Each track segment is connected to zero, one, or two different
  points. The number of points is zero or greater. The number of track
  segments is zero or greater.
end define ;

define emptyTopology(topo, elems) informally 
  Topology "topo" contains no track segments or points. It may contain
  maximal "elems" track segments and "elems" points. The topology is
  initialized such that all possible track segments and points are not
  connected.
end define ;

define tracksEmpty(topo, ntracks) informally
  It is possible to add at least "ntracks" track segments to the topology
  "topo".
end define ;

define addTracksMod(topo1, topo2, ntracks) informally
  Topology "topo2" is the same as "topo1" except that "ntracks" track
  segments have been added to the topology. The new track segments are
  not connected to any point yet.
end define ;

define pointEmpty(topo) informally
  It is possible to add at least one point to the topology "topo".
end define ;

define openTracks(topo, track1, track2, track3) informally
  The track segments "track1", "track2", and "track3" are all different and
  are each connected to no or one point only according to topology "topo".
end define ;

define addPointMod(topo1, topo2, track1, track2, track3) informally
  Topology "topo2" is the same as "topo1" except that:
– a new point has been added and been connected with track segment "track2" to its straight branch, "track3" to its bent branch, and "track1" to the remaining end;
– the track segments "track1", "track2", and "track3" are connected to the new point with one end that was unconnected before.

end define;

define numTracksTopo(topo) informally
Returns the number of track segments in topology "topo".
end define;

define numPointsTopo(topo) informally
Returns the number of points in topology "topo".
end define;

define pointBase(topo, point) informally
Returns the track segment that is connected to the base of point "point" according to topology "topo".
end define;

define pointPlus(topo, point) informally
Returns the track segment that is connected to the straight branch of point "point" according to topology "topo".
end define;

define pointMinus(topo, point) informally
Returns the track segment that is connected to the bent branch of point "point" according to topology "topo".
end define;

define trackLeft(topo, track) informally
Returns the point that is connected to the 'left' end of track segment "track" according to topology "topo" (i.e. in the backward direction).
end define;

define trackRight(topo, track) informally
Returns the point that is connected to the 'right' end of track segment "track" according to topology "topo" (i.e. in the forward direction).
end define;
A. Topology.java
public class LockTab
{
    private int[][] track_table;
    private int[][] point_table;
    private int num_tracks, num_routes, num_points;

    public LockTab(/**+const*/ int tracks, /**+const*/ int points, /**+const*/ int routes)
        /**+ ensures 
            emptyLockTab(this, tracks@pre, points@pre, routes@pre) ;
            modifies 
                this ;
        */
    {
        track_table = new int[routes][tracks];
        point_table = new int[routes][points];
        num_routes = 0;
        num_tracks = tracks;
        num_points = points;
    }

    public void incRoutesByOne()
        /**+ requires 
            routeEmpty(this@pre) ;
            ensures 
                incRoutesByOneMod(this@pre, this) ;
            modifies 
                this ;
        */
    {
        num_routes = num_routes + 1;
    }

    public void copyRoute(/**+const*/ int from, /**+const*/ int to)
        /**+ requires 
            copyRoute(from@pre, to@pre) ;
            ensures 
                copyRoute(from@pre, to@pre) ;
            modifies 
                this ;
        */
    {
    }
}
B. LockTab.java

```java
0 < from@pre \land from@pre \leq \text{numRoutesLockTab}(this@pre) \land
0 < to@pre \land to@pre \leq \text{numRoutesLockTab}(this@pre);

\text{ensures}
\text{copyRouteMod}(this@pre, this, from@pre, to@pre);

\text{modifies}
this;

*/
{
  int i = 0;
  while (i < num_points)
    /*+
    \text{maintains}
    pointsPartlySame(this@pre, this, from@pre, to@pre, i) \land
    0 \leq i \land i \leq \text{num_points};
    */
    point_table[to - 1][i] = point_table[from - 1][i];
    i = i + 1;
}
i = 0;
while (i < num_tracks)
  /*+
  \text{maintains}
  tracksPartlySame(this@pre, this, from@pre, to@pre, i) \land
  0 \leq i \land i \leq \text{num_tracks};
  */
  track_table[to - 1][i] = track_table[from - 1][i];
  i = i + 1;
}

public void lockTrackForw(/+const*/ int route, /+const*/ int track)
/** \text{requires}
  0 < route@pre \land route@pre \leq \text{numRoutesLockTab}(this@pre) \land
  0 < track@pre \land track@pre \leq \text{numTractsLockTab}(this@pre);

\text{ensures}
lockTrackForwMod(this@pre, this, route@pre, track@pre);

\text{modifies}
this;

*/
{
  track_table[route - 1][track - 1] = 1;
}

public void lockTrackBack(/+const*/ int route, /+const*/ int track)
```
/** requires */
0 < route@pre \land route@pre \leq \text{num Routes Lock Tab}(\text{this}@pre) \land
0 < track@pre \land track@pre \leq \text{num Tracks Lock Tab}(\text{this}@pre) ;

ensures
lock Track Back Mod(\text{this}@pre, \text{this}, route@pre, track@pre) ;

modifies
this ;

*/
{
    track_table[route - 1][track - 1] = 2;
}

public void lock Point Minus(/* const*/ int route, /* const*/ int point)
/** requires */
0 < route@pre \land route@pre \leq \text{num Routes Lock Tab}(\text{this}@pre) \land
0 < point@pre \land point@pre \leq \text{num Points Lock Tab}(\text{this}@pre) ;

ensures
lock Point Minus Mod(\text{this}@pre, \text{this}, route@pre, point@pre) ;

modifies
this ;

*/
{
    point_table[route - 1][point - 1] = 2;
}

public void lock Point Plus(/* const*/ int route, /* const*/ int point)
/** requires */
0 < route@pre \land route@pre \leq \text{num Routes Lock Tab}(\text{this}@pre) \land
0 < point@pre \land point@pre \leq \text{num Points Lock Tab}(\text{this}@pre) ;

ensures
lock Point Plus Mod(\text{this}@pre, \text{this}, route@pre, point@pre) ;

modifies
this ;

*/
{
    point_table[route - 1][point - 1] = 1;
}

public boolean track Forw Locked(/* const*/ int route, /* const*/ int track)
/** requires */
0 < route@pre \land route@pre \leq \text{num Routes Lock Tab}(\text{this}@pre) \land
0 < track@pre \land track@pre \leq \text{num Tracks Lock Tab}(\text{this}@pre) ;
ensures
    result = trackForwLocked(this@pre, route@pre, track@pre);
*/
{
    return track_table[route - 1][track - 1] == 1;
}

public boolean trackBackLocked(/*const*/ int route, /*const*/ int track)
/**
  requires
    0 < route@pre ∧ route@pre ≤ numRoutesLockTab(this@pre) ∧
    0 < track@pre ∧ track@pre ≤ numTracksLockTab(this@pre);
  ensures
    result = trackBackLocked(this@pre, route@pre, track@pre);
*/
{
    return track_table[route - 1][track - 1] == 2;
}

public boolean pointMinusLocked(/*const*/ int route, /*const*/ int point)
/**
  requires
    0 < route@pre ∧ route@pre ≤ numRoutesLockTab(this@pre) ∧
    0 < point@pre ∧ point@pre ≤ numPointsLockTab(this@pre);
  ensures
    result = pointMinusLocked(this@pre, route@pre, point@pre);
*/
{
    return point_table[route - 1][point - 1] == 2;
}

public boolean pointPlusLocked(/*const*/ int route, /*const*/ int point)
/**
  requires
    0 < route@pre ∧ route@pre ≤ numRoutesLockTab(this@pre) ∧
    0 < point@pre ∧ point@pre ≤ numPointsLockTab(this@pre);
  ensures
    result = pointPlusLocked(this@pre, route@pre, point@pre);
*/
{
    return point_table[route - 1][point - 1] == 1;
}

public int numTracksLockTab()
/**
  ensures
    result = numTracksLockTab(this@pre);
*/
{   return num_tracks;
}

public int numPointsLockTab()
/**  ensures
     result = numPointsLockTab(this@pre) ;
*/
{   return num_points;
}

public int numRoutesLockTab()
/**  ensures
     result = numRoutesLockTab(this@pre) ;
*/
{   return num_routes;
}

public boolean routeEmpty()
/**  ensures
     result = routeEmpty(this@pre) ;
*/
{   return num_routes <= track_table.length;
}

/**   define emptyLockTab(ltab, ntracks, npoints, nroutes) informally
     Locking table “ltab” is large enough to store at least “nroutes” train routes
     with at least “ntracks” track segments and at least “npoints” points for
     each train route. However, the table does not contain any train routes,
     track segments, or points.
   end define ;

define routeEmpty(ltab) informally
   It is possible to add at least one train route to the locking table “ltab”.
   Not yet used train routes in the locking table do not have any track
   segments or points assigned to them.
end define ;

define incRoutesByOneMod(ltab1, ltab2) informally
   Locking table “ltab2” is the same as “ltab1” except that one train route
   has been appended. No track segments or points are assigned to the new
   train route yet.
end define ;

define copyRouteMod(ltab1, ltab2, route1, route2) informally
Locking table "ltab2" is the same as "ltab1" except that train route "route2" has the same set of track segments and points assigned to it as train route "route1" (with train route "route1" unchanged).
end define ;

define pointsPartlySame(ltab1, ltab2, route1, route2, i) informally
Locking table "ltab2" is the same as "ltab1" except that the same points upto point "i" − 1 that are assigned to train route "route1" are assigned to train route "route2" (with "route1" unchanged).
end define ;

define tracksPartlySame(ltab1, ltab2, route1, route2, i) informally
Locking table "ltab2" is the same as "ltab1" except that the same track segments upto segment "i" − 1 that are assigned to train route "route1" are assigned to train route "route2" (with "route1" unchanged).
end define ;

define lockTrackForwMod(ltab1, ltab2, route, track) informally
Locking table "ltab2" is the same as "ltab1" except that track segment "track" is assigned to train route "route" in the forward direction.
end define ;

define lockTrackBackMod(ltab1, ltab2, route, track) informally
Locking table "ltab2" is the same as "ltab1" except that track segment "track" is assigned to train route "route" in the backward direction.
end define ;

define lockPointMinusMod(ltab1, ltab2, route, point) informally
Locking table "ltab2" is the same as "ltab1" except that point "point" is assigned to train route "route" in the bent branch position.
end define ;

define lockPointPlusMod(ltab1, ltab2, route, point) informally
Locking table "ltab2" is the same as "ltab1" except that point "point" is assigned to train route "route" in the straight branch position.
end define ;

define trackForwLocked(ltab, route, track) informally
Track segment “track” is assigned in forward direction to train route “route” according to locking table “ltab”.
end define ;

define trackBackLocked(ltab, route, track) informally
    Track segment “track” is assigned in backward direction to train route “route” according to locking table “ltab”.
end define ;

define pointMinusLocked(ltab, route, point) informally
    Point “track” is assigned in the bent branch position to train route “route” according to locking table “ltab”.
end define ;

define pointPlusLocked(table, route, point) informally
    Point “track” is assigned in the straight branch position to train route “route” according to locking table “ltab”.
end define ;

define numTracksLockTab(ltab) informally
    Returns the number of track segments that can be assigned to each train route in locking table “ltab”.
end define ;

define numPointsLockTab(ltab) informally
    Returns the number of points that can be assigned to each train route in locking table “ltab”.
end define ;

define numRoutesLockTab(ltab) informally
    Returns the number of train routes that are used in locking table “ltab”.
end define ;

*/
}
B.   LockTab.java
public class ExclTab
{
    private boolean[][] excl_table;
    private int num_routes;

    public ExclTab(/*+ const*/ int routes)
    /** ensures */
        emptyExclTab(this, routes@pre) ;
        modifies this ;
    */
    {
        excl_table = new boolean[routes][routes];
        num_routes = routes;
    }

    public void allowRoutes(/*+ const*/ int route_a,
        /*+ const*/ int route_b)
    /** requires */
        0 < route_a@pre ∧ route_a@pre ≤ numRoutesExclTab(this@pre) ∧
        0 < route_b@pre ∧ route_b@pre ≤ numRoutesExclTab(this@pre) ;
        ensures allowRoutesMod(this@pre, this, route_a@pre, route_b@pre) ;
        modifies this ;
    */
    {
        excl_table[route_a - 1][route_b - 1] = true;
    }

    public boolean routesAllowed(/*+ const*/ int route_a,
        /*+ const*/ int route_b)
    /** requires */
        0 < route_a@pre ∧ route_a@pre ≤ numRoutesExclTab(this@pre) ∧
        0 < route_b@pre ∧ route_b@pre ≤ numRoutesExclTab(this@pre) ;
        ensures
C. ExclTab.java

```java
result = routesAllowed(this@pre, route_a@pre, route_b@pre);
*/
{
    return excl_table[route_a - 1][route_b - 1];
}

public int numRoutesExclTab()
/**+ ensures
    result = numRoutesExclTab(this@pre);
*/
{
    return num_routes;
}

/**+ define emptyExclTab(etab, nroutes) informally
    Exclusion table "etab" contains "nroutes" train routes. The number of
    train routes is zero or greater. No train routes are allowed to be used
    concurrently.
end define ;

define allowRoutesMod(etab1, etab2, route1, route2) informally
    Exclusion table "etab2" is the same as "etab1" except that according to
    "etab2" train route "route1" and train route "route2" are allowed to be
    used concurrently.
end define ;

define routesAllowed(etab, route1, route2) informally
    Exclusion table "etab" contains "nroutes" train routes. The number of
    train routes is zero or greater. No train routes are allowed to be used
    concurrently.
end define ;

define numRoutesExclTab(etab) informally
    Returns the number of train routes in exclusion table "etab".
end define ;
*/
```
public class TableGen
{
    public void findFromPoint(/*const*/ Topology topo, LockTab ltab,
        /*const*/ int route, /*const*/ int track, /*const*/ int point)
        /**< requires */
            validTopology(topo@pre) \&\& matchTrackPoint(topo@pre, ltab@pre) \&\&
            0 < route@pre \&\& route@pre \leq numRoutesLockTab(ltab@pre) \&\&
            0 < track@pre \&\& track@pre \leq numTracksTopo(topo@pre) \&\&
            0 \leq point@pre \&\& point@pre \leq numPointsTopo(topo@pre);
        /**< ensures */
            routesAddedPoint(topo@pre, ltab@pre, ltab, route@pre, track@pre,
                point@pre);
        /**< modifies */
            ltab;
    /**< */
    {
        int new_route;
        if(point != 0)
        {
            if(track == topo.pointPlus(point))
            {
                ltab.lockPointPlus(route, point);
                findFromTrack(topo, ltab, route, point, topo.pointBase(point));
            }
            else if(track == topo.pointMinus(point))
            {
                ltab.lockPointMinus(route, point);
                findFromTrack(topo, ltab, route, point, topo.pointBase(point));
            }
            else
            {
                if(ltab.routeEmpty())
                {
                    ltab.incRoutesByOne();
                    new_route = ltab.numRoutesLockTab();
                    ltab.copyRoute(route, new_route);
                    ltab.lockPointPlus(route, point);
                    findFromTrack(topo, ltab, route, point, topo.pointPlus(point));
                }
            }
        }
    }
}

183
public void findFromTrack(/**+ const*/ Topology topo, LockTab ltab,  
/**+ const*/ int route, /**+ const*/ int point, /**+ const*/ int track)

/** requires 
  validTopology(topo@pre) ∧ matchTrackPoint(topo@pre, ltab@pre) ∧ 
  0 < route@pre ∧ route@pre ≤ numRoutesLockTab(ltab@pre) ∧ 
  0 ≤ point@pre ∧ point@pre ≤ numPointsTopo(topo@pre) ∧ 
  0 < track@pre ∧ track@pre ≤ numTracksTopo(topo@pre) ; 

  ensures 
  routesAddedTrack(topo@pre, ltab@pre, ltab, route@pre, point@pre, track@pre) ; 

  modifies 
  ltab ; 
*/
{
  if (point == topo.trackLeft(track))
    { ltab.lockTrackForw(route, track);
      findFromPoint(topo, ltab, route, track, topo.trackRight(track));
    }
  else
    { ltab.lockTrackBack(route, track);
      findFromPoint(topo, ltab, route, track, topo.trackLeft(track));
    }
}

public void generateLockings(/**+ const*/ Topology topo, LockTab ltab)

/** requires 
  validTopology(topo@pre) ∧ 
  emptyLockTab(ltab@pre, numTracksTopo(topo@pre), 
  numPointsTopo(topo@pre), routes@pre) ; 

  ensures 
  matchLockTab(topo@pre, ltab@pre, ltab) ; 

  modifies 
  ltab ; 
*/
public boolean shareElems(/*+ const*/ LockTab ltab, /*+ const*/ int route_a,
/*+ const*/ int route_b)
/** requires */
0 < route_a@pre ∧ route_a@pre < numRoutesLockTab(ltab@pre) + 1 ∧
0 < route_b@pre ∧ route_b@pre < numRoutesLockTab(ltab@pre) + 1;
/** ensures */
result = shareElems(ltab@pre, route_a@pre, route_b@pre); /* */
{ boolean sharet = false, sharep = false;
  int track = 1, point = 1;
  while(track < ltab.numTracksLockTab() + 1 && !sharet)
    /** maintains */
    shareTrack(ltab@pre, route_a@pre, route_b@pre, track, sharet) ∧
    0 < track ∧ track ≤ numTracksLockTab(ltab@pre) + 1;
  */
  { sharet = (ltab.trackForwLocked(route_a, track) ||
            ltab.trackBackLocked(route_a, track)) &&
           (ltab.trackForwLocked(route_b, track) ||
            ltab.trackBackLocked(route_b, track));
    track = track + 1;
  }
  while(point < ltab.numPointsLockTab() + 1 && !sharep)
    /** maintains */
    sharePoint(ltab@pre, route_a@pre, route_b@pre, track, sharep) ∧
\[
0 < \text{point} \land \text{point} \leq \text{numPointsLockTab}(\text{ltab}@\text{pre}) + 1;
\]

```java
0 < point \land point \leq \text{numPointsLockTab}(\text{ltab}@\text{pre}) + 1;
*/
{
    \text{sharep} = (\text{ltab}.\text{trackForwLocked}(\text{route}_a, \text{point}) ||
                   \text{ltab}.\text{trackBackLocked}(\text{route}_a, \text{point})) &&
    (\text{ltab}.\text{trackForwLocked}(\text{route}_b, \text{point}) ||
     \text{ltab}.\text{trackBackLocked}(\text{route}_b, \text{point}));
    \text{point} = \text{point} + 1;
}
    \text{return}\ \text{sharet} \lor \text{sharep};
}

\text{public void generateExclusions}(/\*+const*/ \text{LockTab}\ \text{ltab}, \text{ExclTab}\ \text{etab})
/++ \text{requires}
    \text{emptyExclTab}(\text{etab}@\text{pre}, \text{numRoutesLockTab}(\text{ltab}@\text{pre}))
    \text{ensures}
     \text{matchExclTab}(\text{ltab}@\text{pre}, \text{etab}@\text{pre}, \text{etab});
    \text{modifies}
     \text{etab} ;
*/
{
    \text{int}\ \text{route}_a = 1, \text{route}_b ;
    \text{while}(\text{route}_a < \text{ltab}.\text{numRoutesLockTab}() + 1)
    /++ \text{maintains}
        \text{partMatchExclTab}(\text{ltab}@\text{pre}, \text{etab}@\text{pre}, \text{etab}, \text{route}_a, 1) \land
        0 < \text{route}_a \land \text{route}_a \leq \text{numRoutesLockTab}(\text{ltab}@\text{pre}) + 1 ;
*/
{
    \text{route}_b = 1;
    \text{while}(\text{route}_b < \text{ltab}.\text{numRoutesLockTab}() + 1)
    /++ \text{maintains}
        \text{partMatchExclTab}(\text{ltab}@\text{pre}, \text{etab}@\text{pre}, \text{etab}, \text{route}_a, \text{route}_b) \land
        0 < \text{route}_a \land \text{route}_a < \text{numRoutesLockTab}(\text{ltab}@\text{pre}) + 1 \land
        0 < \text{route}_b \land \text{route}_b \leq \text{numRoutesLockTab}(\text{ltab}@\text{pre}) + 1 ;
*/
{
    \text{if}(\text{shareElems}(\text{ltab}, \text{route}_a, \text{route}_b))
    \{ \text{etab}.\text{allowRoutes}(\text{route}_a, \text{route}_b) ;
    \}
    \text{route}_b = \text{route}_b + 1;
}
\text{route}_a = \text{route}_a + 1;
}
/** define matchTrackPoint(topo, ltab) informally **
  The topology "topo" and the locking table "ltab" have the same number of track segments and points.
end define;

define routesAddedPoint(topo, ltab1, ltab2, route, track, point) informally
  – Train route "route" enables a train to move from a track segment that is connected with only one point to the track segment "track".
  – Locking table "ltab2" is the same as locking table "ltab1" except that the train route has been completed such that a train could continue from track segment "track" via point "point" to another segment that is connected with only one point.
  – Furthermore, all other train routes with the same beginning as train route "route" but different continuations have been appended to the locking table (as long as they fit).
Everything according to the topology "topo" of the railway station.
end define;

define routesAddedTrack(topo, ltab1, ltab2, route, point, track) informally
  – Train route "route" enables a train to move from a track segment that is connected with only one point to the point "point".
  – Locking table "ltab2" is the same as locking table "ltab1" except that the train route has been completed such that a train could continue from point "point" via track segment "track" to another segment that is connected with only one point.
  – Furthermore, all other train routes with the same beginning as train route "route" but different continuations have been appended to the locking table (as long as they fit).
Everything according to the topology "topo" of the railway station.
end define;

define matchLockTab(topo, ltab1, ltab2) informally
  Locking table "ltab2" is the same as locking table "ltab1" except that all (or as many as fit into the table) possible train routes through the station have been added. The track segments and points are given by the topology "topo".
end define;

define partMatchLockTab(topo, ltab1, ltab2, track) informally
  Locking table "ltab2" is the same as locking table "ltab1" except that all train routes (if any) starting from the first "track" — 1 track segments
have been added (as long as the table is not full). The track segments and points are given by the topology “topo”.

end define ;

define shareElems(ltab, route1, route2) informally
Train route “route1” and train route “route2” share a track segment or point according to locking table “ltab”.
end define ;

define shareTrack(ltab, route1, route2, track, overlap) informally
“overlap” is true if and only if train route “route1” and train route “route2” do not share any of the first “track” — 1 track segments according to locking table “ltab”.
end define ;

define sharePoint(ltab, route1, route2, track, overlap) informally
“overlap” is true if and only if train route “route1” and train route “route2” do not share any of the first “track” — 1 points according to locking table “ltab”.
end define ;

define matchExclTab(ltab, etab1, etab2) informally
Exclusion table “etab2” is identical to “etab1” except that any two train routes that do not share a track segment or point according to locking table “ltab” may be used concurrently.
end define ;

define partMatchExclTab(ltab, etab1, etab2, route1, route2) informally
Exclusion table “etab2” is the same as “etab1” except that in “etab2” any train route X has been allowed to be used concurrently with any train route Y if and only if they do not share a track segment or point according to locking table “ltab”. Moreover, X is either \( \leq \text{route1} - 1 \) or X is “route1” and Y is \( \leq \text{route2} - 1 \).
end define ;

*/
E. Assertions

E.1. Use-Case Diagram

context Use-Case sendMsg
    requires
        readyToSend(sender@pre, receiver@pre) ;
    ensures
        reliablyConnected(sender@pre, receiver@pre) 
            ⇒ message(sender@pre) = message(receiver) ;
end context Use-Case sendMsg

define readyToSend(sender, receiver) informally
    A full-duplex connection between the link user “sender” and the link user “receiver” is available. The link user “sender” has a message it wants to transfer to the link user “receiver”. The link user “receiver” is prepared to receive a message from the link user “sender”.
end define ;

define reliablyConnected(sender, receiver) informally
    A full-duplex reliable connection between the link user “sender” and the link user “receiver” is available.
end define ;

define message(entity) informally
    Returns the message of the link user or link entity “entity”.
end define ;
E. Assertions

E.2. Sequence Diagram

context Sequence normalOp
  requires
    reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre) \∧
    noAcksReceived(sendEntity@pre) \∧ noSegsReceived(receiveEntity@pre) ;
  ensures
    message(sender@pre) = message(receiver) ;

define reliablyConnectedChannel(sender, sEntity, ch, rEntity, receiver) informally
  A full-duplex reliable connection between the link user “sender” and the link user “receiver” via the link entity “sEntity”, the L2-channel “ch”, and the link entity “rEntity” is available.
end define ;

define noAcksReceived(entity) informally
  The link entity “entity” has received no acknowledgment since the last successful message transfer.
end define ;

define noSegsReceived(entity) informally
  The link entity “entity” has received no segments since the last successful message transfer.
end define ;

context Object sender
  context Intermediate 1
    ensures
      msg = message(sender@pre) ;
  end context Intermediate 1
end context Object sender

context Object sendEntity
  context Intermediate 1
    ensures
      sendEntity = sendEntity@pre ;
  end context Intermediate 1
context Intermediate 2
ensures
    replacedMessage(sendEntity, sendEntity@pre, msg) ;
end context Intermediate 2
context Intermediate 4
    ensures
        sendEntity = sendEntity@pre ;
end context Intermediate 4
end context Object sendEntity

context Object receiveEntity
    context Intermediate 1
        ensures
            receiveEntity = receiveEntity@pre ;
    end context Intermediate 1
    context Intermediate 5
        ensures
            receiveEntity = receiveEntity@pre ;
    end context Intermediate 5
    context Intermediate 7
        ensures
            receiveEntity = receiveEntity@pre ∧
            msg = message(receiveEntity@pre) ;
    end context Intermediate 7
end context Object receiveEntity

context Object receiver
    context Intermediate 4
        ensures
            replacedMessage(receiver, receiver@pre, msg@pre) ;
    end context Intermediate 4
end context Object receiver

context Loop i
    maintains
        i ≤ numOfSegs(sendEntity) + 1 ∧
        receivedSegAndAck(receiveEntity, receiveEntity@pre, sendEntity, sendEntity@pre, i − 1) ;
context Object sendEntity
context Intermediate 5
ensures
  sendEntity = sendEntity@pre ∧ seg = segment(sendEntity@pre, i) ;
end context Intermediate 5
context Intermediate 6
ensures
  sendEntity = sendEntity@pre ;
end context Intermediate 6
context Intermediate 7
ensures
  receivedPacket(sendEntity, sendEntity@pre, ack@pre) ;
end context Intermediate 7
end context Object sendEntity

class Object channel
context Intermediate 3
ensures
  seg = seg@pre ;
end context Intermediate 3
context Intermediate 5
ensures
  ack = ack@pre ;
end context Intermediate 5
end context Object channel

class Object receiveEntity
context Intermediate 2
ensures
  receiveEntity = receiveEntity@pre ;
end context Intermediate 2
context Intermediate 3
ensures
  receivedPacket(receiver, receiver@pre, seg) ∧
  ack = acknowledgment(receiveEntity@pre, i) ;
E.2. Sequence Diagram

end context Intermediate 3
context Intermediate 4
  ensures
    receiveEntity = receiveEntity@pre ;
end context Intermediate 4
end context Object receiveEntity
end context Loop i

define replacedMessage(entityB, entityA, message) informally
  The link user or link entity “entityB” is the same as the link user or link entity “entityA” except that the message of link user or link entity “entityB” is the message “message”.
end define ;

define receivedSegAndAck(rEntityB, rEntityA, sEntityB, sEntityA, i) informally
  The link entity “rEntityB” is the same as the link entity “rEntityA” except that the first “i” segments received by it are the same as sent by the link entity “sEntityA” (hence the number of received segments of “rEntityB” is “i”). The link entity “sEntityB” is the same as “sEntityA” except that the acknowledgments for the first “i” segments have been received from “rEntityB”.
end define ;

define numOfSegs(entity) informally
  Returns the total number of segments which the link entity “entity” wants to send.
end define ;

define segment(entity, i) informally
  Returns the “i”th segment to be sent by the link entity “entity”.
end define ;

define receivedPacket(entityB, entityA, packet) informally
  The link entity “entityB” is the same as the link entity “entityA” except that it has received “packet” (which is either a segment or an acknowledgment).
end define ;

define acknowledgment(entity, i) informally
  Returns an acknowledgment from the link entity “entity” to the “i”th segment.
end define ;

end context Sequence normalOp
E. Assertions

E.3. Class Diagram

class LinkEntity

context Class

context Operation segment()
    ensures
        segmented(this, this@pre);
end context Operation

context Operation assemble()
    ensures
        assembled(this, this@pre);
end context Operation

end context Class

define segmented(entityB, entityA) informally
    The link entity "entityB" is the same as the link entity "entityA" except that the
    message that "entityA" wants to send has been broken up into
    numOfSegs("entityA") segments. The message is broken up into at least one
    segment.
end define ;

define assembled(entityB, entityA) informally
    The link entity "entityB" is the same as the link entity "entityA" except that the
    segments received by it are assembled into a message that is contained in "entityB".
end define ;
F. Questions

Question 1

Assume:
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. message(sender@pre) = message(receiver)

Then:
1. reliablyConnected(sender@pre, receiver@pre) \implies message(sender@pre) = message(receiver)

Is the conclusion satisfied?

Question 2

Assume:
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. replacedMessage(sendEntity#2, sendEntity@pre, message(sender@pre))

Then:
1. true

Is the conclusion satisfied?
F. Questions

5. segmented(sendEntity#3, sendEntity#2)
6. receivedSegAndAck(receiveEntity#9, receiveEntity@pre, sendEntity#9, sendEntity#3, numOfSegs(sendEntity#3))

Then:
   1. true

Is the conclusion satisfied?

Question 4

Assume:
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. replacedMessage(sendEntity#2, sendEntity@pre, message(sender@pre))
5. segmented(sendEntity#3, sendEntity#2)

Then:
   1. \(1 \leq \text{numOfSegs}(sendEntity#3) + 1\)
   2. receivedSegAndAck(receiveEntity@pre, receiveEntity@pre, sendEntity#3, sendEntity#3, 0)

Is the conclusion satisfied?

Question 5

Assume:
1. \(i \leq \text{numOfSegs}(sendEntity#4) + 1\)
2. receivedSegAndAck(receiveEntity#4, receiveEntity#4, sendEntity#4, sendEntity#4, i - 1)
3. receivedPacket(receiveEntity#7, receiveEntity#4, segment(sendEntity#4, i))
4. receivedPacket(sendEntity#9, sendEntity#4, ack(receiveEntity#7, i))

Then:
   1. \(i \leq \text{numOfSegs}(sendEntity#9)\)
   2. receivedSegAndAck(receiveEntity#7, receiveEntity#4, sendEntity#9, sendEntity#4, i)

Is the conclusion satisfied?

Question 6

Assume:
1. reliablyConnectedChannel(sender@pre, sendEntity@pre, channel@pre, receiveEntity@pre, receiver@pre)
2. noAcksReceived(sendEntity@pre)
3. noSegsReceived(receiveEntity@pre)
4. replacedMessage(sendEntity#2, sendEntity@pre, message(sender@pre))
5. segmented(sendEntity#3, sendEntity#2)
6. receivedSegAndAck(receiveEntity#9, receiveEntity@pre, sendEntity#9, sendEntity#3, numOfSegs(sendEntity#3))
7. assembled(receiveEntity#11, receiveEntity#9)
8. replacedMessage(receiver#13, receiver#12, message(receiveEntity#11))

Then:
1. message(sender@pre) = message(receiver#13)

Is the conclusion satisfied?
<table>
<thead>
<tr>
<th>No</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
</table>


