COMPASS:
Introduction of Formal Methods in Code Development and Inspection

Tim Heyer
Real-Time Systems Laboratory
Department of Computer and Information Science
Linköpings Universitet, S-581 83 Linköping, Sweden

April 1998

Abstract

Providing means for the development of correct software and the verification of software are central challenges of computer science. This is also the aim of the COMPASS method. It assists in the development of correct code and enables the automatic generation of relevant, focused questions to be asked during code inspection sessions.

COMPASS has been developed within the ISIS project "Verification Automation in Software Development". The project was carried out in cooperation between ABB Industrial Systems and the Real-Time Systems Laboratory at IDA, Linköping University. The COMPASS method is based on Hoare's method for proving programs correct. The novel key idea is the introduction of informal predicates, which, though not having a formal definition, may have a perfectly legal and unique informal interpretation. Such predicates make it easier to express requirements in terms of assertions, while still allowing the automatic generation of verification conditions. Moreover, informal predicates enable reasoning about assertions and verifying verification conditions at a level which is suitable for man rather than machine.

This thesis describes the COMPASS method in detail and reports on preliminary results.

Acknowledgements

First of all, I would like to thank my supervisors Anders Törne, for the opportunity to work and perform research in the stimulating environment of the Real-Time Systems Laboratory (RTSLAB), Staffan Bonnier, for his inspiring ideas and many fruitful discussions, and last but not least Ulf Nilsson for his support and his comments on the manuscript. Furthermore, I thank the rest of the members of the RTSLAB for the good working environment.

Many thanks also to Anne Eskilsson and Lillemor Wallgren.

I am grateful to ABB Industrial Systems, especially Ulf Hammar and Stefan Frennemo, for remarks that have led to improvements of the COMPASS approach.

Moreover, I would like to express my gratitude to the NUTEK competence center ISIS, for both the financial support and for the stimulating discussions with its members.

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

Contents

1. Introduction

Providing means for the development of correct software and the verification of software are central challenges of computer science.

In November 1995 the project "Verification Automation in Software Development" was commenced. The project ended in November 1997. It was one of several projects within the competence center ISIS (Information Systems for Industrial Control and Supervision), and was carried out in cooperation between ABB Industrial Systems and the Real-Time Systems Laboratory at IDA, Linköping University.

The general goal of the project was to develop practical means for increasing confidence in software correctness. The strategy was to provide semi-formal support both for the development of code and for its inspection. To achieve this goal, COMPASS, a comprehensible assertion method was suggested [BH97]. The method supports the automatic compilation 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 COMPASS approach has its foundation in the well-known theories for 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 [Hoa69] introduced a logic for reasoning about 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. Dijkstra [Dij76] 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's logic.

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. However, the methods are hardly known in industry and even less used. In [BH97] it was argued that 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 the requirements on formal rigor are relaxed in a controlled manner to achieve a method which is more easily used in practice but which still remains partially mechanizable. The novel key idea in COMPASS concerns the predicates which are used for expressing assertions: COMPASS allows such predicates to be used without an associated formal definition. Instead it is expected that they are defined in normal prose in a special kind of comment. The experience so far is that informal predicates enable expressing assertions at a high level which makes the algorithmic content of a program explicit, and enable reasoning about assertions and verifying programs at a level which is suitable for man rather than machine.

The remainder of this thesis is organized as follows: Chapter 2 describes the general objective of this thesis. In Chapter 3 the traditional formal approach to software verification, that is Hoare's logic, is presented. Then the principles of the COMPASS method are elaborated in Chapter 4. This includes both the principles for code development and for code inspection. A prototype tool for the automatic derivation of verification conditions, and for presenting these conditions in a way suitable to form a basis for code inspection has been developed and is presented in Chapter 5. In Chapter 6, the method is illustrated by the development and inspection of a program that computes a part of a controller for a railway station. Chapter 7 presents and compares with related work. Finally, Chapter 8 contains conclusions and future work.

2. General Objective

The general objective of this research has been to suggest improvements to the current software development practice, enabling more rigid but still practically acceptable methods for the development and verification of software. The focus has been on the coding phase. Two methods commonly used today for verification are code inspection and testing.

2.1. Testing and Debugging

Testing is the process of executing software to identify the potential presence of errors. Debugging is the process of locating and correcting these errors. Testing is today certainly the most important method used during the verification process. However, it has two major drawbacks:

- It can in general not be used to show the absence of errors in the software. 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 relied upon.
- Testing can only be used very late in the development process and does not give much support for how to structure and implement the software.

Due to the above drawbacks the testing costs compared to the overall development costs are rapidly increasing. Testing must therefore be complemented with other means to help increasing confidence in the correctness of the software.

2.2. Code Inspection

Code inspection is a visual examination of code to detect errors. A reader is paraphrasing the code while the other members of the inspection team, equipped with special *check lists*, are trying to find errors.

Check lists used for the inspection of code are usually based on so called *style guides*. Style guides are used during the development of the software. The purpose of a style guide is to describe how to write readable and uniform programs and how to avoid common pitfalls. However, these guides do not give much support for developing correct code (that is, software which behaves as intended).

The check lists used for the inspection typically require mainly that the code is written according to the style guide. Hence, the inspection team checks that all naming conventions have been followed, that the indentation is as required, and so on. This is certainly important in order for the code to be readable and uniform (for instance for maintainability). But it is not sufficient for ensuring correctness. In most cases there is a very systematic way to carry out the checks. Indeed, the checks could in principle often be automatized.

Beside the checks based on style guides, check lists sometimes require that the code behaves as intended. But they provide no means for checking this in a systematic manner. Thus, the steps to be taken during inspection in order to verify these rather important checks are very loosely defined – if defined at all.

Extensions to the present inspection process are therefore needed, describing how to systematically check the correctness of the code with respect to its intended behavior. COMPASS has been developed to satisfy this need. The COMPASS approach provides a guide for the development of correct code together with systematic, and partly automatized support for its inspection. Furthermore, COMPASS supports a structured documentation of the code.

3. A Formal Approach to Software Verification

Software verification may be defined as the activity of checking that a program satisfies its specified requirements. An approach that has received a great deal of attention is Hoare's logic. This approach has had a significant impact on later formal methods for designing and verifying imperative, sequential computer programs.

3.1. Hoare's Logic

In 1969 Hoare [Hoa69] introduced an axiomatic method for proving programs correct. This approach was partially based on Floyd's intermediate assertion method [Flo67]. The method developed by Hoare is also known as the *partial correctness assertion method.*

An assertion expresses a condition on the program variables, which is supposed to hold each time a certain point in the execution of the program is reached. By associating with a method (that is a procedure of an object) two special assertions called the *pre-* and the *postcondition* of the method, assertions may be used to specify the effect executing the method is intended to have on the execution state.

Hoare introduced a logic for reasoning about assertions. The formulae of the 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 holds, and if the execution of `S` terminates, then Q holds 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 Figure 3.1 on the following page):

1. **Development of code with assertions inserted at appropriate places**

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

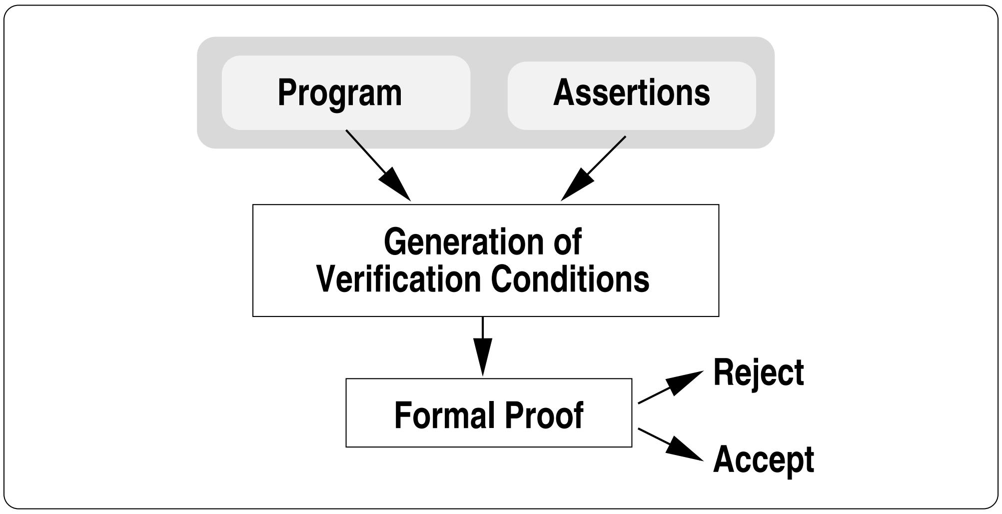

Figure 3.1.: Hoare's partial correctness assertion method

Example 3.1: *The following code with assertions swap the values of two integer variables without using an additional temporary variable:*

$$\{\, x = x\hat{} \wedge y = y\hat{} \,\}\ \texttt{x=x+y; y=x-y; x=x-y;}\ \{\, x = y\hat{} \wedge y = x\hat{} \,\}$$

The postcondition *states that, after the execution of the three assignments, x equals the initial value of y (denoted as $y\hat{}$) and y equals the initial value of x (denoted as $x\hat{}$). The* precondition *simply states that before the execution 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 [Dij76] 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's 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 one of them can be proven then all assertions except the precondition are valid, that is 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 3.2: *The following code from the previous example swaps the values of two integer variables:*

$$\{\, x = x\hat{} \wedge y = y\hat{} \,\}\ \texttt{x=x+y;}\ \texttt{y=x-y;}\ \texttt{x=x-y;}\ \{\, x = y\hat{} \wedge y = x\hat{} \,\}$$

Informally, the verification condition for the above code is generated in the following way:

1. *In order for the postcondition to hold after the execution of the third assignment, everything that is supposed to hold for x in the postcondition must hold for $x-y$ immediately before the execution of the third assignment. That is, the following condition must hold before the final assignment: $x - y = y\hat{} \wedge y = x\hat{}$.*
2. *Accordingly, before the execution of the second assignment the following condition must hold: $x - (x - y) = y\hat{} \wedge x - y = x\hat{}$.*
3. *Finally, before the execution of the first assignment the following condition has to hold: $(x + y) - ((x + y) - y) = y\hat{} \wedge ((x + y) - y) = x\hat{}$.*

The verification condition that results from the above process requires that the computed condition before the first assignment follows from the precondition:

$$x = x\hat{} \wedge y = y\hat{} \implies (x + y) - ((x + y) - y) = y\hat{} \wedge ((x + y) - y) = x\hat{}$$

Similarly, verification conditions for more complex code can be generated. □

It should be noted that the meaning of "+", "−", and the predicate "=" is not involved in the above generation of verification condition. 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.

Example 3.3: *The verification condition that has been generated in the previous example for the code that is swapping the values of two integer variables is:*

$$x = x\hat{} \wedge y = y\hat{} \implies (x + y) - ((x + y) - y) = y\hat{} \wedge ((x + y) - y) = x\hat{}$$

The above formula can easily be simplified to the following formula using basic arithmetic laws:

$$x = \hat{x} \wedge y = \hat{y} \implies x = \hat{x} \wedge y = \hat{y}$$

The proof of the implication is then trivial and hence the code is correct with respect to the assertions, that is the code swaps the values of two integer variables. □

Both Hoare's method (see for instance [Apt81]) 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. 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 Section 7.4).

3.2. Problems with Hoare's Logic

The lack of understanding of Hoare's and Dijkstra's ideas may be due to the fact that, though vast in quantity, 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.

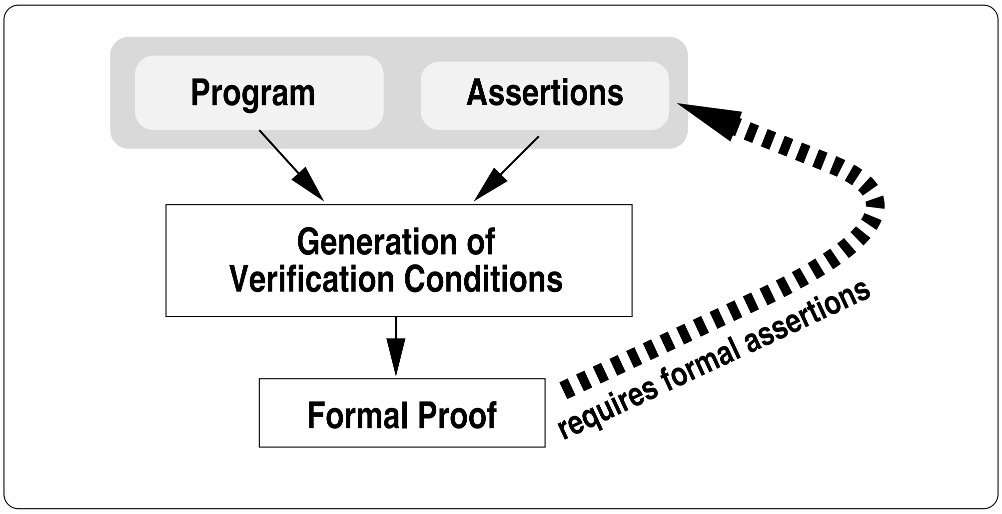

Figure 3.2.: The need for formality

The demand for full formality arises from the third phase of Hoare's method (see Figure 3.2): to be able to perform a formal proof of the verification conditions, a formal axiomatization is required for each predicate which occurs in the assertions and which hence is used to express the intended behavior of executing 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 4.2 on page 12). As a consequence, they are both difficult to state and to reason about.

The basic hypothesis presented in this thesis is that a method which is more easily used in practice, and which remains partially mechanizable, may be achieved by relaxing the requirements on formal rigor in a controlled manner.

4. The COMPASS Approach

The general goal of this research has been to develop practical means for increasing confidence in software correctness. To achieve this goal the COMPASS method has been developed. This approach provides a guide for the development of correct code together with automatic support for its inspection. Furthermore, COMPASS supports a structured documentation of the code.

The COMPASS approach is based on Hoare's method for verifying code as briefly presented in Section 3.1. However, formal approaches require a formal axiomatization of each predicate used to express the intended behavior of the code. As discussed in Section 3.2, such axiomatizations often have a non-obvious connection to the intuitive understanding of the property the predicate is to represent. Therefore, they are both difficult to state and to reason about. In COMPASS the requirements on formal rigor are relaxed by the introduction of *informal predicates* resulting in a method which is more easily used in practice but which still remains partially mechanizable.

Even though the COMPASS method is easier to use than the formal approach it will probably mainly be used for software for which correctness is very important, for instance for software embedded in control systems.

4.1. Informal Predicates

As mentioned, the novel key idea in COMPASS concerns the predicates ranging over the domain of program variables, and which hence are used for expressing assertions. In COMPASS such predicates are allowed to occur without an associated formal definition (axiomatization). They are instead expected to be defined in normal prose in special comments. Thus assertions have a formal syntax, but an informal semantics. The point is that such informal predicates may still have a perfectly legal and unique informal interpretation, and by expressing it informally rather than formally its meaning is more directly accessible to a human reader.

Example 4.1: *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* `sorted`*. The* *definition of this predicate is then supplied in a special code comment, which may look as follows:*

```
define sorted( arr, left, right ) informally

  The array "arr" is sorted in increasing order inbetween "left" and
  "right".

end define;
```

The predicate `sorted` *may now be used in assertions either by itself or combined with other predicates (both formally and informally defined ones).* □

Informal predicates make it easy to express requirements in terms of assertions. Moreover, they enable reasoning about assertions and verifying verification conditions at a level which is suitable for man rather than machine.

Example 4.2: *In a formal proof of a program that sorts an array of integers the following verification condition may occur:*

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

Using a predicate `sorted` *the same verification condition may be presented in the following form instead:*

```
Suppose
   1. sorted( a, 1, n )
   2. a[ n + 1 ] >= max( a, 1, n )

Then
   1. sorted( a, 1, n + 1 )
```

In COMPASS, the predicate `sorted` *may be an informal predicate and defined as in Example 4.1 on the page before. Thus, the first condition with the formal axiomatization can formally be proven while the second condition requires a person to decide whether it holds 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.* □

4.2. Overview over the COMPASS Method

The cornerstones on which the COMPASS method is based are Hoare's method for proving programs correct [Hoa69], Dijkstra's discipline for program development [Dij76], and

Fagan's work on code inspection [Fag76]. The method is similar to Hoare's partial correctness assertion method as outlined in Section 3.1. However, in COMPASS predicates are allowed to occur without a formal definition. Therefore, the Hoare's partial correctness assertion method has to be adapted. The resulting method consists of the following three phases (see also Figure 4.1 on the following page):

1. **Development of code with assertions inserted at appropriate places**

   The code is developed according to Dijkstra's discipline of programming. However, the formulation of assertions is simplified since it is no longer required to formally define all the predicates. It should be noted that the assertions with informal predicates even if they are not used for the generation of verification conditions provide a structured, semi-formal documentation of the code.

   Section 4.4 contains a detailed description of how to specify the intended behavior of a subset of Java programs. The method for developing code is outlined in Section 4.5.

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

   Since the generation of verification conditions is not dependent on the meaning of the predicates involved, the verification conditions sufficient for the correctness of the program may be automatically derived.

   The generation of verification conditions and its prerequisites are presented in Section 4.6.

3. **Human inspection of the verification conditions**

   The generated verification conditions will in general themselves contain informal predicates, thus disabling the possibility of a formal proof.

   Therefore the formal proof of Hoare's method is substituted by an inspection session. During the inspection session the verification conditions are informally justified. In addition, both the informal definitions of predicates and the adequacy of the assertions should be examined during the inspection.

   A detailed description of the COMPASS inspection process can be found in Section 4.7.

A prototype tool for the automatic generation of verification conditions and for presenting these conditions in a way suitable to form a basis for code inspection has been developed. It is presented in Chapter 5.

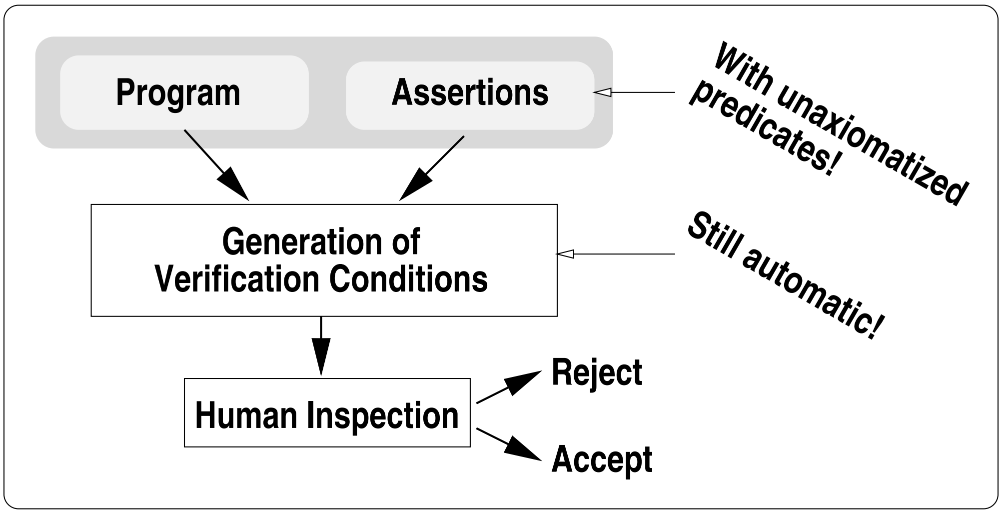

Figure 4.1.: The COMPASS method

4.3. Prerequisites and Limitations

Not all programming languages are amenable for writing verifiable programs. Furthermore, the ability to reason about assertions and verification conditions at a high level imposes restrictions on the properties of programs that are appropriate for inspection.

4.3.1. Properties to be Inspected

It should be clear that not any program nor any property is amenable to verification along the principles of the COMPASS method. The COMPASS method is based on Hoare's partial correctness assertion method and – as his method – it focuses on partial correctness as opposed to total correctness. That is, COMPASS gives no support for verifying termination. The considerations for termination are independent of partial correctness and not taken into account here.

The attention is restricted to 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 (that is, essentially it has to developed according to Dijkstra's discipline of programming).

4.3.2. The Choice of the Programming Language

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 the 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 for instance non-corruption of the representing data structure).

To be able to evaluate the COMPASS method in an industrial setting, it was initially adapted to the C programming language. However, data abstraction is not supported in C. Though 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 basic memory management which makes it difficult to argue for 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 (implying some form of automatic garbage collection).

With respect to the above considerations all examples in this report concern code written in the Java programming language. The implemented prototype tool to support code inspection (see Chapter 5) also concerns Java code only. Java supports data abstraction (object-oriented) and has garbage collection and is hence an appropriate candidate for COMPASS.

4.3.3. Open Questions

The COMPASS method allows predicates without a formal definition to occur in assertions. However, the automatic generation of verification conditions presupposes the existence of formal proof rules (that is, an axiomatic semantics) for the programming language. Thus, the semi-formal approach inherits some of the problems of the 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. Further information can be found in the literature (see for instance [PH97]).

Overriding

Java, as most object-oriented programming languages, allows *overriding* of methods. That is, if a class declares a method then the declaration of that method overrides all methods with the same signature (that is the same name, return type, and number and type of formal parameters) in the superclasses that would otherwise be accessible to code in the class.

Conceptually, the method in the subclass is a specialization of the method of the superclass. This gives rise to additional proof obligations:

- The invariant (see Section 4.4.3) of the subclass has to imply the invariant of the superclass.
- The method of a subclass that overrides a method in the superclass has to show the same behavior as the method in the superclass, that is the precondition of the method in the superclass has to imply the precondition of the method in the subclass and the postcondition of the method in the subclass has to imply the postcondition of the corresponding method in the superclass.

The above requirements are informally fairly clear. There are however some difficulties. The requirement that the precondition of a method in the superclass has to imply the precondition of the corresponding method in the subclass does in general not have to hold. Another problem is connected with the format of the specification, that is the pre- and postcondition. Even for equivalent specifications it can be impossible to prove above constraints if the specifications are not given in a normal form.

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 at the point of a method invocation 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. This leads to more complex verification conditions. However, the relation between a method in a subclass to a corresponding method in its superclass may be used to reduce the complexity of verification conditions.

Implementation-Independence

Another difficulty is related to the scope of variables. Private class variables are visible for methods of that class and could thus be used in interface specifications. However, private class variables are in general not visible at the point of the method invocation. Moreover, the use of class variables would make the specification implementation dependent. These problems can be avoided by using a reference to the whole object (usually *this* or *self*) and additional informal predicates or an abstract model of the class in interface specifications.

Sharing

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 would probably be erroneous (that is, the program could be incorrect even though this is not captured by the verification conditions).

Another difficulty related to sharing concerns class invariants. If two objects share a common object a modification of the common object initiated from one of the objects may violate the class invariant of the other object. As a result, it would be necessary to prove for *all* methods that they satisfy *all* class invariants.

A possible approach to the mentioned problems may be to ask the user to specify which variables actually might be aliased. For some applications it also might be possible to forbid aliasing completely since it is a common source of errors. Other approaches can be found in the literature (see Chapter 7).

4.3.4. A Java Subset

To be able to evaluate the COMPASS method, it has been applied to the Java programming language. An appropriate subset has been chosen to simplify the implementation of a prototype tool to support code inspection.

Some of the restrictions are of a rather syntactic nature and are not limiting the expressablity of the language. Other restrictions are real semantic restrictions and have partly been introduced to keep the prototype tool simple and partly to avoid some of the difficulties mentioned in Section 4.3.3. Some of the major syntactical restrictions are:

- Java interfaces and class inheritance are not supported.
- Class variables have to be declared as private and methods as public.
- Functions, that is methods that return a value, are not allowed to modify any actual parameters or class variables.
- Local variable declarations are allowed only directly after the beginning bracket of the body of a method.
- No static variables are allowed.
- Only while loops are permitted.
- Assignments in expressions are not supported.
- Java's exception handling has been removed.

It may seem that many common Java programming language constructs have been forbidden. However, in most cases it is rather straight forward to express an unsupported construct with the help of the remaining ones.

4.4. Specification of Java Programs

As mentioned, assertions are used to specify what a program is intended to accomplish. An assertion expresses a condition on the program variables, which is supposed to hold each time a certain point in the execution of the program is reached. Thus assertions may be used to specify what the code is to accomplish.

To be able to easily compile the Java code with its specification included, all additional constructs that are introduced for the COMPASS method are encapsulated in special comments. These comments start with `/**` and end with `*/`.

4.4.1. The Assertion Language

The language in which the assertions may be expressed is a so called *first order theory*. 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 manipulated by a program. In the following paragraphs a definition of the language of assertions is provided.

Expressions

Expressions form the basis of the language of assertions:

- *Constants* are expressions, for instance `0`, `-123`, and `42`.
- *Variables* are expressions (see next paragraph).
- If a_1, a_2 to a_n are expressions, then so is $f(a_1, a_2, \ldots, a_n)$ where f is the name of a function.

Certain predefined functions are written in infix or prefix notation. These functions cover the most commonly used arithmetic operators and are collected in Table 4.1.

Binary operators		Unary operators
`a / b` (division)	`a + b` (addition)	`+a`
`a * b` (multiplication)	`a - b` (subtraction)	`-b` (negation)
`a % b` (modulo)		

Table 4.1.: Basic arithmetic operators

Variables may be *program variables* or so called *logical variables*. Program variables are variables that occur in the program, whereas logical variables are special variables that must not appear in any statement of the program under investigation. Unlike program variables, logical variables do not change their values. They are typically used to refer to the initial values of program variables.

Program variables are used in the same form as in the Java code itself. Logical variables are marked with a succeeding circumflex "^". This notation is the same as used in Larch [CL94]. For each variable occurring as a formal parameter of a function, a corresponding logical variable is implicitly defined. These logical variables refer to the initial values of the actual parameters.

There are two special, predefined variables. These variables are:

`result` refers to the return value of a method and is used in the postcondition of functions. For further details see Section 4.4.4.

`this` refers to the actual object itself. It is used to express a condition on the state of the object. More detailed descriptions of the usage of `this` can be found in Section 4.4.3 and Section 4.4.4.

Example 4.3: *All legal Java variable names (including qualified names) may be used in assertions as for instance* `ret_val` *or* `name[i]`*. That a certain method of a class* `TrafficLight` *is to switch to the red signal may be expressed with the postcondition*

redSignal(this)*. A postcondition for a method which computes the power x^y may look as follows:* **`result == power( x^, y^ )`**. □

Logical Formulae

With the help of expressions so called *logical formulae* can be introduced: The constants ***true*** and ***false*** are logical formulae and a *predicate* is a logical formula. A predicate has the form $P(a_1, a_2, \ldots, a_n)$ where P is the name of the predicate and a_1, a_2 to a_n are expressions. Table 4.2 shows some common, predefined predicates that are written in infix notation.

s == *t* (equal)	*s* != *t* (not equal)	*s* < *t* (less than)
s > *t* (greater than)	*s* <= *t* (less or equal)	*s* >= *t* (greater or equal)

Table 4.2.: Comparison operators

Complex logical formulae may be constructed using the logical connectives shown in Table 4.3, where P and Q are formulae. The precedence is ***not***, ***and***, and ***or*** with ***not*** binding strongest.

Binary connectives		Unary connectives
P ***and*** *Q* (conjunction)	*P* ***or*** *Q* (disjunction)	***not*** *P* (negation)

Table 4.3.: Logical connectives

Finally, the legal sentences of the assertion language are the above described logical formulae.

4.4.2. Predicate and Function Definitions

The predicates and functions used in assertions to express what the code is to accomplish have to be defined either formally or informally (that is in natural language). The notation used for these definitions is:

define *name* ***informally*** *definition* ***end define;***

name — is the name of a predicate or function followed by a comma separated list of its formal parameters in parentheses.

definition — describes the relation between the parameters the predicate is to represent. The description can be in any form, for instance it might be a simple picture with an explanation in natural language.

define *name* **formally** *definition* **end define;**

name is the name of a predicate or function followed by a comma separated list of its formal parameters in parenthesis.

definition describes the relation between the parameters the predicate is to represent. The description has to be given in the formal notation presented in Section 4.4.1.

Several definitions may be written sequentially in one special comment block (that is between a pair of `/*+` and `*/`).

Example 4.4: *That a program is to compute an integer approximation of the square root of a number may be expressed with the predicate* `approx_square_root`. *An informal respectively formal definition of this predicate may look as follows:*

```
/*+

  define approx_square_root1( n, x ) informally

    "x" is the largest integer that is smaller than the square
    root of "n".

  end define;

  define approx_square_root2( n, x ) formally

    0 <= x * x and x * x <= n and n < ( x + 1 ) * ( x + 1 )

  end define;

*/
```

The example shows also two definitions in only one special comment. □

4.4.3. Class Invariants

Typically, classes are intended to use only a subset of their possible set of values. Therefore COMPASS allows the specification of a *class invariant*, that is, a property of the class that is unaffected by the operations on instances of the class. This means that, given that the property (class invariant) holds for an object before the invocation of a method, it does hold after its completion (for the constructor method the invariant only needs to hold *after* the execution). The notation for a class invariant is:

invariant *condition;*

condition is a logical formula as described in Section 4.4.1. The condition on the state of an instance of the class has to be expressed in terms of `this` (see Section 4.3.3 for further explanations). Only program variables are allowed to occur in the formula.

The class invariant has to be written in a special comment directly after the class declaration.

Example 4.5: *A class that is to represent traffic lights may have an invariant which states that an instance of the class always shows a valid combination of lights.*

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

The above construct expresses that all methods of each instance of the class `TrafficLight` *require an initial state where the invariant holds (except for the constructor method) and that they ensure that the invariant holds upon termination.* □

4.4.4. Interface Specifications

In which states a method may safely be invoked (that is for which initial states it produces the desired result) and what it is intended to do is expressed in a so called *interface specification*. An interface specification consists of the method head followed by a special comment.

Method Head

The method head is written in standard Java notation. In addition formal parameters of the method may be defined to be *constants* by writing the keyword `const` in a special COMPASS comment in front of the type of the parameter.

A parameter defined to be a constant is assumed to remain completely unchanged during the execution of the body of the function. If the formal parameter refers to a composed object it is moreover assumed that no part of it is changed. The introduction of constant parameters simplifies the specification of assertions. Without the declaration as constant it would be necessary for some parameters to explicitly state that their current values always equal their initial values during the execution of the method's body. 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* (see for instance [BMR95]).

Example 4.6: *The following could be the head of a method that searches the position of an element of array* `a` *with the same value as* `x`*.*

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

Since both `x` *and* `a` *are defined to be constant it is assumed that they do not change their value during the execution of the body, that is it is assumed that* $x = x\hat{}$ *and* $a = a\hat{}$*. As mentioned this includes that neither* `a` *itself nor any of its elements are modified during the execution of the body.* □

`requires`-, `ensures`-, and `modifies`-clause

In a special comment following directly after the head of 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 notation is similar to that used in Larch [CL94]. The notation is as follows:

`requires` *condition;*

condition is a logical formula as described in Section 4.4.1. The `requires`-clause expresses a condition that is supposed to hold *before* the execution of the body of the method.

Thus, the formula is only allowed to contain logical variables, in particular the ones referring to the initial values of the parameters.

`ensures` *condition;*

condition is a logical formula as described in Section 4.4.1. The `ensures`-clause expresses a condition on the program variables that is supposed to hold when the method invocation returns. That is, this assertion expresses what the method is intended to accomplish.

If the method actually is a function, the postcondition always has the form `result ==` *function*, where `result` denotes the return value of the method and *function* represents the (formally or informally defined) function that is performed by the method.

`modifies` *variables;*

variables is a comma separated list of variable names. 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 (`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*, that is the actual values are passed to methods.

The reference **`this`** has to appear in the variable list when class variables may be modified by the method (see Section 4.3.3 and Section 4.3.4).

Functions do not have a **`modifies`**-clause, since they are not allowed to modify the arguments or the state of the object (see Section 4.3.4).

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, that is, it is assumed that the method is neither altering any parameters nor the object itself.

Example 4.7: *An interface specification for a method that is to sort an array between a left and a right index may look as follows:*

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

The **`requires`***-clause is omitted since the method has no restrictions on the state prior the execution of its body.* □

4.4.5. Other Assertions

Besides the assertions to specify the class invariant and the method interface, two other assertions may be given along with the code in the body of a method. Loop invariants are specified with the following notation:

`invariant` *condition;*

condition is a logical formula as described in Section 4.4.1. It expresses a condition on the state of the computation which holds after each execution of the body of the loop, assuming that it holds before the execution. More about loop invariants may be found in Section 4.5 and Section 4.6.1.

The specification of loop invariants is necessary to carry out the generation of the verification conditions.

Example 4.8: *A program that given* `x` *and array* `a` *determines the position of* `x` *in* `a` *(it is assumed that such an element exists) may be implemented with the following loop:*

```
while( a [ i ] != x )
/*+
  invariant
    in_second_part( x^, a^, i );
*/
  i = i + 1;
```

The invariant expresses that the element is not among the first `i-1` *elements but among the rest of the elements (that is, it is in the second part of the array).* □

The other additional assertions that may be specified along with the code in the body of a method are so called *intermediate assertions.* The notation is:

`assert` *condition;*

condition	is a logical formula as described in Section 4.4.1. It is used to express a condition which is expected to hold each time execution reaches the point of the assertion.

Example 4.9: *The following piece of code may be part of an implementation of the* quicksort *algorithm.*

```
split( v, left, right, i );
/*+
  assert
    partition( v^, v, left^, right^, i );
*/
quicksort( v, left, i.val() - 1 );
quicksort( v, i.val() + 1, right );
```

The intermediate assertion states that, after the call of the method `split`*, the array* `v` *is partitioned in a certain way.* □

4.5. 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 [Dij76]. The development method that is part of the 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, for instance [Gri81] and [Bac86] only a brief overview will be given here.

Programming is considered a *goal-oriented* activity, that is, 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.

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

- To invent a conditional statement (for instance 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 (that is 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 [Gri81]. However, since it is not required that all predicates are formally defined, developing a program along with its assertions can be performed on a level more suitable for humans.

Example 4.10: *In this example it will be explained how a program may be developed with the development method described in this section.*

The program to be developed is an implementation 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 respectively less than and greater than the splitting element. It then recursively sorts the two subarrays.

The starting point is to provide an interface specification as described in Section 4.4.4. There are no requirements on the state prior to the execution of `quicksort`*. 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:*

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

The predicate `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. This definition is provided as a special comment as described in Section 4.4.2:*

```
define sorted( a, b, left, right ) informally

  1. "b" is sorted in increasing order inbetween "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;
```

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

```
public void quicksort( int[] v, /*+const*/ int left,
                       /*+const*/ int right )
```

```
/*+
  ensures
    sorted( v^, v, left^, right^ );
  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 sub-arrays. Therefore an informal predicate `partition` *is introduced. This predicate states that the array is split around a certain index* `i`*:*

```
define partition( a, b, left, right, i ) informally

  1. "b"["left"],..,"b"[value("i")],..,"b"["right"]
     \____________/               \______________/
      <= "b"[value("i")]            >= "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:*

```
public void quicksort( int[] v, /*+const*/ int left,
                       /*+const*/ int right )
/*+
  ensures
    sorted( v^, v, left^, right^ );
  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 );
```

```
    }
  }
```

The idea is that `split` *splits the array* `v` *and returns the splitting position in* `i`*. The prototype tool does not allow a method that modifies its parameters to return a value with the* `return` *statement (see Section 4.3.4). Hence the splitting position has to be returned through a parameter. However, basic types are in Java always handled* 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:*

```
public void split( int[] v, /*+const*/ int left,
                   /*+const*/ int right, WrapInt i )
/*+
  requires
    left^ <= right^;
  ensures
    partition( v^, v, left^, right^, i );
  modifies
    v, i;
*/
```

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

4.6. The Generation of Verification Conditions

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 of the programming language, in this case for Java.

4.6.1. An Axiomatic Semantics

The verification conditions are generated with the help of an *inference system*. The formulae of the inference system have the form $\{\, P \,\}\ \ S\ \ \{\, Q \,\}$, where S is a statement in the Java programming language and P and Q are logical formulae as described in Section 4.4.1. These triples are called *Hoare triples* and are to be read "if S starts executing in a state where P holds, and if the execution of S terminates, then Q holds upon termination".

The inference system specifies an *axiomatic semantics*. It consists of a set of *axioms* and *rules*. The rules are typically written in the form

$$\frac{E_1, E_2, \ldots, E_n}{E}$$

where the E_i and E stand for arbitrary propositions. The rule indicates that if formulae $E_1, E_2, \ldots, E_n$ hold, then so does formula E.

In the following, a simplified subset of the axiomatic semantics used by the prototype tool (see Chapter 5) 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" [HW73] and also of the one in Gries's "The Science of Programming" [Gri81].

The Function Call

Methods that return a value, that is, that are not declared as ***void***, may be seen as functions. As described in Section 4.3.4, in the Java subset functions are not allowed to modify any variable except of local ones. Thus the ***modifies***-clause of function declarations is always omitted. The interface definition of a function typically looks as follows:

```
public <type> <function_name>( <parameter_list> )
/*+
  requires
    <condition>;
  ensures
    result == <function>;
*/
```

The meaning of the key words enclosed in "<" and ">" is given by the Java language itself and by the definition of interface specifications (see Section 4.4.4).

Several kinds of statements cause Java expressions to become part of verification conditions (for instance because of the substitution in the assignment statement). These Java expressions may contain function calls which have to be handled first. The function calls are substituted by the corresponding functions used in the postconditions of the interface specification of the function. 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 4.11: *The* `ensures` *clause of an interface specification of a function* `f` *may look as follows:*

```
ensures result == fct( this^, a^ );
```

An assignment statement in the program to be inspected may contain a call to that function:

```
x = m.f( 0 );
```

Now, instead of substituting all occurrences of `x` *in the postcondition of the assignment statement with* `m.f( a )` *they are substituted with* `fct( m, 0 )`*.* □

The Assignment Statement

A very important construct of any imperative programming language is the assignment statement. The axiom for the assignment statement of the Java subset is:

$$\{ P^x_e \} \texttt{ x = e; } \{ P \} \tag{4.1a}$$

P^x_e denotes the expression obtained by simultaneously substituting expression e for all free occurrences of x in expression P. The variable x may be a simple variable, an array variable, or a field variable:

Indexed variables: Arrays are treated as functions as for instance in [Gri81]. 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. That is:

$$\{ P^a_{(a;\ i:e)} \} \texttt{ a[i] = e; } \{ P \} \tag{4.1b}$$

Class variables: Similar to arrays, field 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:

$$\{ P^{this}_{(this;\ c:e)} \} \texttt{ c = e; } \{ P \} \tag{4.1c}$$

The axioms 4.1a, 4.1b, and 4.1c are not actually axioms but axiom schemes, generating separate axioms for each choice of a predicate P.

The Sequential Composition

More complex statements may be composed from smaller statements using sequential composition. The rule of inference for sequential composition is:

$$\frac{\{ P_{i-1} \}\ S_i\ \{ P_i \} \text{ for } i = 1, \ldots n}{\{ P_0 \}\ \texttt{\{}\ S_1\ S_2\ \ldots\ S_n\ \texttt{\}}\ \{ P_n \}} \tag{4.2}$$

This rule states that if P_0 holds before the execution of a block of statements `{` S_1 S_2 $\ldots$ S_n `}` and if the execution terminates, then P_n holds in the final state, provided that there is a set of logical formulae P_i such that $\{ P_{i-1} \}\ S_i\ \{ P_i \}$ holds for $i = 1$ to n.

The Conditional Statements

Java provides a set of conditional statements. Among these are the *if-then* statement and the *if-then-else* statement. The rule of inference for the *if-then* statement is:

$$\frac{\{\,P\,\}\ S\ \{\,Q\,\}}{\{\,(B \implies P) \wedge (\neg B \implies Q)\,\}\ \texttt{if(}\ B\ \texttt{)}\ S\ \{\,Q\,\}} \tag{4.3}$$

The rule says that if the *if-then* statement is executed from a state where $(B \implies P) \wedge (\neg B \implies Q)$ holds and if the execution terminates then Q holds for the final state provided that, if statement S is executed from a state where P holds and if the execution terminates then Q holds in the final state.

The second conditional statement included in the Java subset supported by the prototype tool is the *if-then-else* statement. The corresponding rule of inference is:

$$\frac{\{\,P\,\}\ S_1\ \{\,Q\,\}, \{\,R\,\}\ S_2\ \{\,Q\,\}}{\{\,(B \implies P) \wedge (\neg B \implies R)\,\}\ \texttt{if(}\ B\ \texttt{)}\ S\ \texttt{else}\ S_2\ \{\,Q\,\}} \tag{4.4}$$

This rule expresses that if the *if-then-else* statement is executed from a state where $(B \implies P) \wedge (\neg B \implies R)$ holds and if the execution terminates then Q holds for the final state provided that:

1. if statement S_1 is executed from a state where P holds and if the execution terminates then Q holds in the final state and
2. if statement S_2 is executed from a state where R holds and if the execution terminates then Q holds in the final state.

Other conditional statements as the *switch* statement that are part of the Java programming language are not included in the Java subset supported by the prototype.

The Iterative Statements

A very important and rather complicated construct of imperative programming languages are the iterative statements, for instance the *while* statement. The inference rule for the *while* statement is:

$$\frac{\{\,B \wedge P\,\}\ S\ \{\,P\,\}}{\{\,P\,\}\ \texttt{while(}\ B\ \texttt{)}\ S\ \{\,\neg B \wedge P\,\}} \tag{4.5}$$

The logical formula P is called an *invariant* for the loop and the idea is that it will hold before and after each execution of the body S of the loop. The rule states that if additionally B holds before each execution of the body of the loop then $\neg B$ will hold when the execution of the loop has terminated.

Other iterative statements of the Java programming language as for instance the *for* loop and the *repeat* loop are not included in the subset supported by the prototype tool.

Method Invocation

An important means of abstraction is the method. The interface specification of a method typically looks as follows:

```
public void <meth_name>( <parameter_list> )
/*+
  requires
    <P>;
  ensures
    <Q>;
  modifies
    <variable_list>;
*/
```

The meaning of the key words enclosed in "<" and ">" is defined by the Java language itself and by the definition of interface specifications (see Section 4.4.4). The difficulties and limitations that apply for the method invocation are described in Section 4.3.3 and Section 4.3.4.

The Java subset allows the invocation of methods declared in the same class and of methods declared in a different class. In the former case, an invocation looks as follows:

```
<meth_name>( <argument_list> );
```

A qualified name has to be used for the invocation of a method that is declared in a different class:

```
<obj_name>.<meth_name>( <argument_list> );
```

It is assumed that the interface specification of the method has the form that is given earlier in this section. Moreover, it is also assumed that the implementation of the method is partially correct, that is, that $\{ P \}$ *method body* $\{ Q \}$ holds. The axiom for the method invocation then is:

$$\{ P^{\bar{x},\bar{y}}_{\bar{a},\bar{b}} \land (\forall \bar{u} : Q^{\bar{y}}_{\bar{u}} \Rightarrow R^{\bar{b}}_{\bar{u}}) \} \ \mathit{obj\_name.meth\_name}(\ \bar{a}, \bar{b}\ );\ \{ R \} \qquad (4.6)$$

The $\bar{x}$ represents the variables that are not modified by the method whereas the $\bar{y}$ represents the variables that may be modified and thus are listed in the ***modifies***-clause of the interface specification. The $\bar{a}$ and $\bar{b}$ represent the corresponding arguments of the method invocation. The *obj_name* is treated as an implicit argument to the method and the corresponding formal parameter is `this`.

If a class invariant has been specified then it has to be treated as an additional pre- and postcondition of the method (except for the constructors, in which case it is only treated as a postcondition).

4.6.2. Applying the Semantics to Generate Verification Conditions

The axiomatic semantics that is outlined in the previous section, enables the generation of a set of verification conditions given a program with assertions. The generation of verification conditions is, as mentioned, not dependent on the meaning of the predicates involved. Thus, the generation can still be performed automatically, even though assertions in general contain informal predicates.

The verification conditions are usually split into several independent conditions. Thus, the complexity of the questions to be answered during code inspection sessions is reduced. Moreover, it is often possible to further simplify the verification conditions using basic axioms and inference rules for the domain over which the program variables range. Some verification conditions may be completely solved due to this simplifications.

Example 4.12: *This examples is a continuation of Example 4.10 on page 27. It is intended to show how, from the definition of* `quicksort` *(see page 28) and the interface specification of* `split` *(see page 29), a number of verification conditions may be derived (using the axiomatic semantics presented in Section 4.6.1). The program together with an assertion between each pair of statements, a so called* proof outline *(that is an outline of a formal proof), may look as follows:*

$$\begin{aligned}
&\{P\!: \ \mathit{true}\} \\
&\{P1\!: \ \mathit{left}\hat{\ } < \mathit{right}\hat{\ } \implies (\ \mathit{left}\hat{\ } \leq \mathit{right}\hat{\ } \wedge (\ \mathit{partition}(\ v\hat{\ }, v_3, \mathit{left}\hat{\ }, \mathit{right}\hat{\ }, i_1\ ) \implies \\
&\qquad \mathit{sorted}(\ v_3, v_2, \mathit{left}\hat{\ }, \mathit{value}(i_1){-}1\ ) \implies \\
&\qquad \mathit{sorted}(\ v_2, v_1, \mathit{value}(i_1){+}1, \mathit{right}\hat{\ }\ ) \implies \mathit{sorted}(\ v\hat{\ }, v_1, \mathit{left}\hat{\ }, \mathit{right}\hat{\ }\ )\ )\ )\ )\ ) \wedge \\
&\qquad \mathit{left}\hat{\ } \geq \mathit{right}\hat{\ } \implies \mathit{sorted}(\ v\hat{\ }, v\hat{\ }, \mathit{left}\hat{\ }, \mathit{right}\hat{\ }\ )\}
\end{aligned}$$

```
      {
        WrapInt i = new WrapInt( 0 );
```

$$\{P2\!: \ \mathit{left}\hat{\ } < \mathit{right}\hat{\ } \implies P1 \wedge \mathit{left}\hat{\ } \geq \mathit{right}\hat{\ } \implies Q\}$$

```
        if ( left < right )
        {
```

$$\begin{aligned}
&\{P3\!: \ \mathit{left}\hat{\ } \leq \mathit{right}\hat{\ } \wedge (\ \mathit{partition}(\ v, v_3, \mathit{left}\hat{\ }, \mathit{right}\hat{\ }, i_1\ ) \implies \\
&\qquad \mathit{sorted}(\ v_3, v_2, \mathit{left}\hat{\ }, \mathit{value}(i_1){-}1\ ) \implies \\
&\qquad \mathit{sorted}(\ v_2, v_1, \mathit{value}(i_1){+}1, \mathit{right}\hat{\ }\ ) \implies \mathit{sorted}(\ v\hat{\ }, v_1, \mathit{left}\hat{\ }, \mathit{right}\hat{\ }\ )\ )\ )\ )\}
\end{aligned}$$

```
05              split( v, left, right, i );
```

$$\{P4\text{:}\ sorted(\ v,\ v_2,\ left\hat{},\ value(i){-}1\ ) \Longrightarrow$$
$$(\ sorted(\ v_2,\ v_1,\ value(i){+}1,\ right\hat{}\ ) \Longrightarrow sorted(\ v\hat{},\ v_1,\ left\hat{},\ right\hat{}\ )\ )\}$$

```
06              quicksort( v, left, i.val() - 1 );
```

$$\{P5\text{:}\ sorted(\ v,\ v_1,\ value(i){+}1,\ right\hat{}\ ) \Longrightarrow sorted(\ v\hat{},\ v_1,\ left\hat{},\ right\hat{}\ )\}$$

```
07              quicksort( v, i.val() + 1, right );
08          }
09      }
```

$$\{Q\text{:}\ sorted(\ v\hat{},\ v,\ left\hat{},\ right\hat{}\ )\}$$

The generation of verification conditions typically starts with the postcondition and is performed from the bottom to the top. The generation takes into account that `left` *and* `right` *are defined as constant, that is that* left = left^ *and* right = right^ *always hold.*

After the last recursive invocation of `quicksort` *in line 7 the array has to be sorted, that is* Q *has to hold. The axiom for method invocation (4.6) states that this is the case if* P5 *holds before the execution of* `quicksort` *in line 7. Similarly,* P4 *has to hold before execution of line 6 and* P3 *before execution of line 5.*

The rule of sequential composition (4.2) says that if execution of the sequence of statements from line 5 to 7 starts in a state where P3 *holds and if the execution terminates then* Q *holds in the final state. This enables for the application of the rule for the* if-then *statement (4.3). The rule states that* P2 *has to hold prior to execution of the* if-then *statement. The additional application of the assignment axiom (4.1a) for line 2 and the rule of sequential composition (4.2) does not change* P2 *any further.*

Finally, the result is that the precondition of the method `quicksort`*, that is* true*, has to imply the derived condition* P1*. This implication is split into three independent conditions that constitute the questions to be answered during code inspection:*

1. $left\hat{} < right\hat{}$
   $\Longrightarrow left\hat{} \leq right\hat{}$
2. $left\hat{} < right\hat{} \wedge partition(\ v\hat{},\ v_3,\ left\hat{},\ right\hat{},\ i_1\ ) \wedge$
   $sorted(\ v_3,\ v_2,\ left\hat{},\ value(i_1){-}1\ ) \wedge sorted(\ v_2,\ v_1,\ value(i_1){+}1,\ right\hat{}\ )$
   $\Longrightarrow sorted(\ v\hat{},\ v_1,\ left\hat{},\ right\hat{}\ )$
3. $left\hat{} \geq right\hat{}$
   $\Longrightarrow sorted(\ v\hat{},\ v\hat{},\ left\hat{},\ right\hat{}\ )$

It should be noted that condition 1 may be answered automatically. Condition 2 and 3 however contain informal predicates and may thus not be automatically answered. How the questions are presented and answered during a code inspection session is described in detail in Section 4.7 □

4.7. Code Inspection

The third phase of the COMPASS method consist in a human inspection of the verification conditions. In addition to justifying the validity of the verification conditions, both the informal definitions of predicates and the adequacy of the assertions should be examined during the inspection.

Code inspection, as it is defined in [Fag76], aims only at isolating faults in the code. The COMPASS approach goes one step further; it is based on well known formal techniques for verifying code in a structured manner. Verifying the whole consists of verifying the parts. Failure in verifying a part means that a fault has been isolated. However, in addition to this, the method is guaranteed to generate all questions relevant for establishing the assertions. Thus, if the questions are thoroughly investigated and no errors are found, the code must be considered verified (and hence correct) with respect to its assertions.

The code inspection can be performed either by the programmer during code development (individual inspection) or later, by a group of reviewers (group inspection).

4.7.1. Individual Inspections

As soon as a method is implemented, the programmer may perform a code inspection to check whether the method is correct with respect to the assertions. It is not required that all other methods called by the method under investigation are fully implemented, as long as the interface specifications of the subfunctions in form of the ***`requires`***-, ***`ensures`***-, and ***`modifies`***-clauses are known. Thus the COMPASS approach enables the programmer to find faults in the software, or an argument for its correctness as soon as possible.

In general, the individual inspection process as part of the COMPASS method may be considered to consist of the following four phases:

1. Evaluation of the formally and informally defined functions and predicates.
2. Check whether the pre- and postconditions correspond to the intended behavior of the program.
3. The automatic derivation of verification conditions as it is described in detail in Section 4.6.
4. An informal justification of why each of the automatically generated verification condition holds.

The verification conditions and the justifications given by the programmer may be stored in a database. They may be reused for instance after modifications of the source code or during later group inspections.

Since the decision whether a certain verification condition holds or not is left to the same programmer who wrote both the assertions and the code, possibly faulty verification conditions may pass as correct. In order to exclude this source of uncertainty, group inspections should be performed.

4.7.2. Group Inspections

A group inspection is performed similar to the process first described by Fagan in [Fag76]. However, since the COMPASS approach has a formal basis and can be supported by several tools, the roles of the participants and the process itself can be defined in more detail.

An inspection group with two to four members is suggested. The different roles that may be adapted by the participants during the code inspection process are:

Moderator The moderator is responsible for organizing and moderating the inspection, that is schedules the meeting, organizes a room, and distributes the code, the generated questions, and the corresponding formal and informal definitions to be inspected. Moreover, the moderator is responsible for keeping a protocol of the meeting and for the assignment of work resulting from the inspection if any.

During the inspection session the moderator is also participating in the discussion about the assertions and the verification conditions.

Designer The designer contributes to the discussion of the verification conditions with knowledge about the domain (in particular the abstract data types) and the intended behavior of the program. She has to check whether the interface specification correspond to the intended function of the software.

As the other members of the inspection team, the designer participates actively in the discussion of the assertions and the generated verification conditions.

Implementor The programmer who is responsible for coding the software according to the development method outlined in Section 4.5 contributes with detailed knowledge about the code.

The implementor participates as the other members of the inspection team in the discussion of the assertions and the verification conditions.

The overall goal of the inspection process as part of the COMPASS method is to give rational and systematic explanations of *why* a program works as intended, and, if not, to isolate the fault.

The code inspection process suggested with the COMPASS method may be distinguished in the following three phases:

1. **Planning phase**

   The moderator schedules the meeting and invites the participants. In good time before the meeting the code, the generated questions, and the corresponding formal and informal definitions to be inspected are distributed by the moderator.

2. **Code inspection phase**

   The goal of an inspection session is to explain why the program works as intended. This is done by justifying the correctness of the automatically derived verification conditions.

   The code inspection session itself can be further divided into the following, distinct steps:

   1. Presentation of the general problem. Explanation and discussion of the informally defined functions and predicates.
   2. Controlling whether the pre- and postconditions correspond to the intended behavior of the program. If the assertions do not properly describe the intended behavior, the inspection should be adjourned.
   3. The group has to decide for each verification condition whether the condition holds with respect to its informal interpretation (that is the interpretation determined by the informal predicate and function definitions). The verification condition together with the argument for its correctness or fault is saved.

3. **Rework phase**

   What has to be done after the code inspection depends on the result of the main phase of the inspection session:

   - If the assertions correspond to the intended behavior of the program, and if all verification conditions are deemed to be satisfied, then the inspected program is considered to be correct.
   - If the assertions are the intended ones, but not all of the verification conditions are correct, then faults have been found. In this case, the moderator assigns programmers to remedy all defects found. The non-valid verification conditions give a very delimited piece of code which contains the fault. The moderator moreover decides whether a new inspection is necessary.
   - Finally, if assertions are not according to the intentions, the assertions have to be corrected before discussing verification conditions is meaningful.

   If faults have been detected, the inspection process should be redone. However, since earlier results of the code inspection session may be saved, some verification conditions may not need to be reevaluated.

Since the COMPASS approach guarantees the generation of questions relevant for the correctness with respect to the assertions, it fills the somewhat loosely defined steps of Fagan's code inspection method with a concrete content. It hence provides a structured and machine supported protocol for inspecting the code.

Example 4.13: *This example is a continuation of Example 4.10 on page 27 and Example 4.12 on page 34. In Example 4.12 on page 34 three verification conditions have been generated from an implementation of the sorting algorithm* quicksort. *These verification conditions have to be informally justified during a code inspection session. The questions presented to the inspection team are:*

```
VC#1 quicksort :: QuickSort -----------------------------------------

Suppose
   1. left^ < right^

Then
   1. left^ <= right^

VC#2 quicksort :: QuickSort -----------------------------------------

Suppose
   1. left^ < right^
   2. partition( v^, v~3, left^, right^, i~1 )
   3. sorted( v~3, v~2, left^, value( i~1 ) - 1 )
   4. sorted( v~2, v~1, value( i~1 ) + 1, right^ )

Then
   1. sorted( v^, v~1, left^, right^ )

VC#3 quicksort :: QuickSort -----------------------------------------

Suppose
   1. left^ >= right^

Then
   1. sorted( v^, v^, left^, right^ )
```

Verification condition 1 and 3 are obviously satisfied. Indeed, the first condition could be automatically discharged. Furthermore, using the informal definitions of `partition` *(see page 28) and* `sorted` *(see page 27) it is not difficult to argue for the correctness of the second condition.* □

5. A Prototype Tool

A prototype tool that supports the code inspection according to the COMPASS method as outlined in Section 4.7 has been implemented in SICStus Prolog version 3 #5 [SIC95]. 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. These grammars are very good for scanning and parsing. Moreover, it is rather easy to implement the textual manipulations required to generate the verification conditions in Prolog (see for instance [CM87] and [SS86]).

It should be noted that the tool is just a prototype. The tool is very restricted and supports only a small subset of the Java programming language (see Section 4.3). Suggestions for extensions of the prototype tool may be found in Section 5.3.

5.1. Architecture

The architecture of the prototype tool is similar to that of a multi-pass compiler and is shown in Figure 5.1.

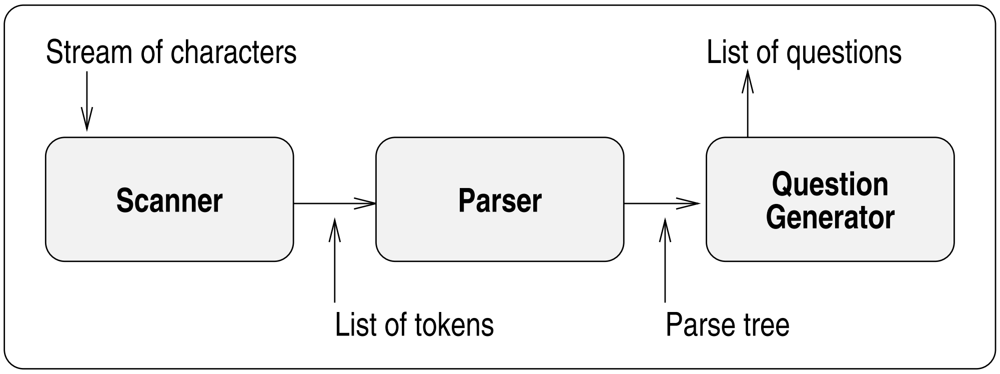

Figure 5.1.: Architecture of the prototype

The prototype tool consists of a scanner, a parser, and a question generator. The scanner (in file "scan.pl") reads a file containing a Java class with assertions as a stream of characters and produces a list of tokens (including information of their location in the source file). The scanner uses additional predicates for the handling of characters found in file “chars.pl”. Out of the list of tokens the parser (in file “parse.pl”) is constructing a parse tree. Both the scanner and parser are implemented with help of definite clause grammars. The parse tree is used to generate the verification conditions (in file “derive.pl”) as outlined in Section 4.6. Finally, the conditions are if possible simplified. The resulting list of verification conditions is then –one by one– presented to and validated by the user.

5.2. Usage of 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 (with `[derive]`).

5.2.1. 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 the classes used by them. 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:

```
loadClass( <FileName> ).
```

Where `<FileName>` 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 will respond with “`Loading <FileName> ... done.`”, for instance:

```
Loading TableGenerator.java ... done.
```

If the code is not syntactically correct the system responds with “`Loading <FileName> ...`” and an excerpt of the code which contains the error, for instance:

```
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, that is the questions relevant to establish the correctness of the code (with respect to the assertions), can be generated.

5.2.2. Inspecting Classes and Methods

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

```
inspect( <ClassName> ).
```

Where *<ClassName>* is the name of the class whose methods are to be inspected. The corresponding command to inspect only a single method of a class is:

```
inspect( <ClassName>, <MethodName> ).
```

Where *<MethodName>* is the name of the method of the class *<ClassName>* 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 with a consecutive number and the name of the method and class. Each question itself consists of the premise (preceded by *Suppose*) and the conclusion (preceded by *Then*). Finally, the system asks if it is correct that the premise implies the conclusion. For instance:

```
VC#1 generateExclusions :: TableGenerator ----------------------------

Suppose
   1. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )

Then
   1. partMatchExclTab( ltab^, etab^, etab^, 1, 1 )

Does verification condition 1 hold?
```

Now the system waits for the user to answer the question with *yes* or *no*. To support the user in her decision the system is able to provide function and predicate definitions. The user can get a requested definition by typing the name of the corresponding function or predicate whenever the system waits for an input. For instance:

```
Does verification condition 1 hold? 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).

Does verification condition 1 hold?
```

If the user answers *yes* all questions generated for a method the code is considered correct with respect to the assertions. The system will accordingly respond with, for instance:

```
+------------------------------------------------------------------+
| Method "generateExclusions::TableGenerator" is correct.          |
+------------------------------------------------------------------+
```

If the user on the other hand answers *no* to a question either the assertions or the code is incorrect. The system will accordingly not ask any further questions regarding the method currently under investigation. Instead information about the assertions and the code that were used to generate the last question will be given, for instance:

```
+------------------------------------------------------------------+
| Method "generateExclusions::TableGenerator" is incorrect.        |
|                                                                  |
| Classinv. 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)                               |
+------------------------------------------------------------------+
```

Given this information the code and assertions can easily be checked to locate and correct the error.

5.3. Possible Extensions

The prototype tool may be extended in several ways. The following list contains some suggestions for possible improvements:

- The prototype may be extended to cover a larger portion of the Java programming language both syntactically and semantically.
- The assertions and definitions should be checked for consistency. For instance, it could be checked that only logical and visible program variables appear in assertions and that all functions and predicates used in assertions are properly defined.

- Moreover, parameters to predicates could be specified with their type. This would result in further possibilities to check the consistency of program and assertions. Furthermore, types would enable for overloading of definitions.
- Several classes may provide similar functions with equal names. However, the predicates in assertions must be different for all classes. Functions with the same names but different classes are distinguished by qualified names. Similar techniques could be used for predicates.
- Experience shows that a large portion of the verification conditions can be simplified to a large extent or even be automatically discharged. Hence it might be worthwhile to extend the simplification capabilities of the tool.
- The user interface could be improved in several ways. For instance, 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 of informal predicate definitions, for instance they could be written in quotation marks, it is possible to replace the formal parameters with their actual argument.

Suggestions for more general improvements of the COMPASS method itself rather than of the prototype tool may be found in Chapter 8.

6. Example

This chapter demonstrates how the COMPASS method may be used to develop and inspect a small program. The program to be developed and inspected is supposed to generate so called locking and exclusion tables. These tables may be used in a system that controls the movement of train in a railway station (a so called *railway interlocking system*).

6.1. 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 Figure 6.1). 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 based.

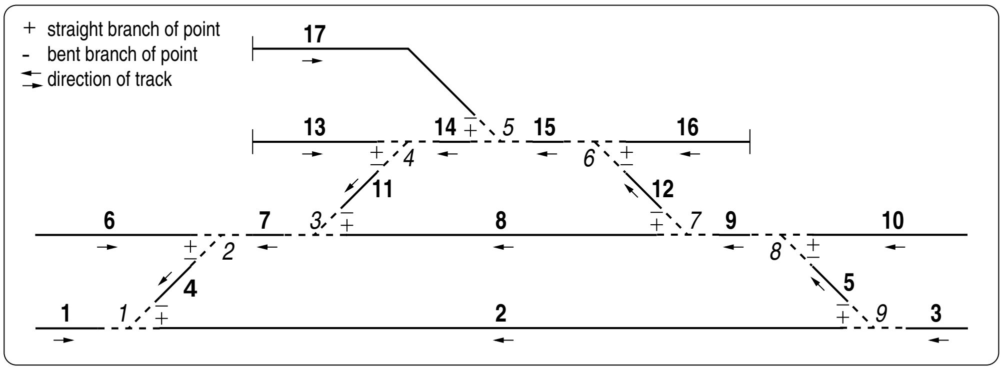

Figure 6.1.: Model of a railway station

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 (for instance empty or occupied), points (for instance in bent or straight position), and signals (for instance 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* (that is track segments and points) which is reserved for a particular train and enables the train to move within the railway station. For instance, 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 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 in [Han96].

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 6.1 shows a part of the locking table for the railway station of Figure 6.1 on the page before. 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 Figure 6.1 on the preceding page and an "<" indicates that it is used in the other direction (see also Section 6.2.3).

Route	Track segments																	Points								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1	2	3	4	5	6	7	8	9
1	>	<	<															+								+
2	>			<			<	<	<	<								−	−	+				+	+	
3	>			<			<				<			<	<	<		−	−	−	−	+	+			
4	>		<	<	<		<	<	<									−	−	+				+	−	−
5	>			<			<		<	<	<	<		<	<			−	−	−	−	+	−	−	+	
6	>		<	<	<		<		<		<	<		<	<			−	−	−	−	+	−	−	−	−
7	<	>	>															+								+
8													<	>	>	>					+	+	+			

Table 6.1.: Part of a locking table

The other table is called the *exclusion table* and describes which routes may be used concurrently, that is which two routes do not lead to a common track segment. Table 6.2 on the next page shows a part of the exclusion table for the railway station of Figure 6.1 on the preceding 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 would have to perform the following subtasks:

Route	Route																																	
	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
1	x	x	x	x	x	x	x		x	x	x	x	x			x		x			x			x			x	x			x			x
2	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
3	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
4	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
5	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

Table 6.2.: Part of an exclusion table

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 (that is prevent by some means that the elements are deallocated or manipulated by accident).

2. **Monitoring of a train route**

   Monitor the train route while it exists.

3. **Unlocking of a train route**

   Unlock the track segments, points, and signals and deallocate them.

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

6.2. Implementation of an Interlocking System

This section presents an implementation of a program that given a model of a railway station topology 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 some simplifications are introduced:

- The station topology is modeled with track segments and points only (that is signals are not considered).
- Each train route is starting from and ending at a track segment that is connected to only one point, that is for instance track segments 1, 3, 6, 10, 13, 16, and 17 in Figure 6.1 on page 47.

- 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 train route as described above. It contains information about which track elements are part of which train routes and also in which state (that is, 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 a train route is a sequence of track elements a train is actually using, that is, driving on. However, usually even other track elements are part of a train route for instance point 2 (in plus position) could be part of train route 1-*1*-2-*9*-3 to protect the train route's flank (see Figure 6.1 on page 47). Point 2 than could still be part of another train route that is used concurrently as long as it is used in the same position.

The implementation of the interlocking system is to be inspected with the help of the prototype tool described in Chapter 5. Therefore, when developing the code the language restrictions described have to be taken into account.

6.2.1. The Overall Design

The design of a program as outlined previously is rather straightforward. The topology of the railway station, that is the basis for the computation of the tables, is modeled by the class `Topology` (see Section 6.2.2). The locking table and the exclusion table are each modeled by a class of their own, namely `LockTab` (see Section 6.2.3) and `ExclTab` (see Section 6.2.4). Finally, the generator that is to compute the tables is implemented in the class `TableGen` (see Section 6.2.5).

6.2.2. 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 Figure 6.1 on page 47. The above requirements on the interconnection of track segments and points are expressed as an invariant of the class:

```
/*
  invariant
    validTopology( this );
*/
```

The definition of the informal predicate `validTopology( topo )` is supplied with the code:

```
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;
```

The interface specifications and implementations of the methods are very simple and can be found in Appendix A on page 81. 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.

6.2.3. 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 instances (that is, for instance which track segments and points are used for a certain train route) are needed.

The locking table is modeled similar to the topology of the railway station: the routes are numbered from 1 to m and the track segments and points are identified by their number. Track segments can be assigned to train routes with a direction, that is forward or backward, whereas points can be assigned to train routes with a position, that is 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 on page 87. The locking table is implemented with the help of two arrays: one for the information about the track segments and one for the information about the points. As a result of this design choice it might become impossible to add new train routes to the locking table because it is full.

6.2.4. 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, that is which train routes may be used concurrently by two trains and how many train routes exist.

The interface specifications of the methods are simple. They can be found in Appendix C on page 95. The exclusion table is implemented with help of a two dimensional array.

6.2.5. Implementation of the Table Generator

The class *TableGen* implements the main functions of this example, namely the computation of the locking and exclusion tables for a given topology of a railway station.

The Generation of the Locking Table.

A method that given the topology of a railway station computes all possible train routes through the station is developed. The method is called *generateLockings*.

The first step in the development of the method is to write its interface specification. The method has two formal parameters: a topology and a locking table.

- 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 to establish a certain relation between the topology, the initial, and the final locking table. The predicate *matchLockTab* is used to represent this relation.
- Initially, it is required that the topology is valid in the sense described in Section 6.2.2 and that the locking table is empty and has the same number of track segments and points as the topology.

Thus, the interface specification looks as follows:

```
public void generateLockings( /*+const*/ Topology topo, LockTab ltab )
/*+
  requires
    validTopology( topo^ ) and
    emptyLockTab( ltab^, numTracksTopo( topo^ ), numPointsTopo( topo^ ),
```

```
      routes^ );
    ensures
      matchLockTab( topo^, ltab^, ltab );
    modifies
      ltab;
  */
```

The fourth argument of the predicate *emptyLockTab*, that is *routes^*, is not actually needed but has to be specified to match the definition of the predicate. The predicate *validTopology* was defined on page 51. The definition of *emptyLockTab* is provided in the class *LockTab* (Appendix B on page 87):

```
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.  The table is initialized such that all
  possible train routes do not contain any track segments or points.

end define;
```

The definition of *matchLockTab* is provided in class *TableGen* and looks as follows:

```
define matchLockTab( topo, ltab1, ltab2 ) informally

  Locking table "ltab2" is the same as locking table "ltab1" except
  that all train routes have been added (as long as the table is not
  full).  The track segments and points are given by the topology
  "topo".

end define;
```

The postcondition states that all train routes have been added to the locking table. All train routes start from track segments that are connected to only one point. Thus, a loop over the track segments can be used. 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 as follows:

```
public void generateLockings( /*+const*/ Topology topo, LockTab ltab )
/*+
  requires
    validTopology( topo^ ) and
    emptyLockTab( ltab^, numTracksTopo( topo^ ), numPointsTopo( topo^ ),
      routes^ );
  ensures
```

```
    matchLockTab( topo^, ltab^, ltab );
  modifies
    ltab;
*/
{
  int track = 1;
  while( track < topo.numTracksTopo() + 1 )
/*+
  invariant
    partMatchLockTab( topo^, ltab^, ltab, track ) and
    0 < track and track <= numTracksTopo( topo^) + 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 ***partMatchLockTab*** is given in class ***TableGen*** and reads as follows:

```
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;
```

Now, 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 an recursive solution has been chosen. 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, that is 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 copied to a new train route and afterwards both train routes have to be completed for different positions of the point. If this is implemented in a method ***findFromTrack*** the complete method ***generateLockings*** reads as follows:

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

```
  requires
    validTopology( topo^ ) and
    emptyLockTab( ltab^, numTracksTopo( topo^ ), numPointsTopo( topo^ ),
      routes^ );
  ensures
    matchLockTab( topo^, ltab^, ltab );
  modifies
    ltab;
*/
{
  int track = 1;
  while( track < topo.numTracksTopo() + 1 )
/*+
  invariant
    partMatchLockTab( topo^, ltab^, ltab, track ) and
    0 < track and track <= numTracksTopo( topo^) + 1;
*/
    {
      if(( topo.trackLeft( track ) == 0 || topo.trackRight( track ) == 0 )
          && topo.trackLeft( track ) != topo.trackRight( track )
          && ltab.routeEmpty() ){
        ltab.incRoutesByOne();
        findFromTrack( topo, ltab, ltab.numRoutesLockTab(), 0, track );
      }
      track = track + 1;
    }
}
```

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 `matchTrackPoint`:

```
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 segment, and train route have to be within certain domains. Thus, the interface specification of `findFromTrack` looks as follows:

```
public void findFromTrack( /*+const*/ Topology topo, LockTab ltab,
                           /*+const*/ int route, /*+const*/ int point,
                           /*+const*/ int track )
/*+
  requires
```

```
      validTopology( topo^ ) and
      matchTrackPoint( topo^, ltab^ ) and
      0 < route^ and route^ <= numRoutesLockTab( ltab^ ) and
      0 <= point^ and point^ <= numPointsTopo( topo^ ) and
      0 < track^ and track^ <= numTracksTopo( topo^ );
    ensures
      routesAddedTrack( topo^, ltab^, ltab, route^, point^, track^ );
    modifies
      ltab;
  */
```

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

```
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 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 on page 99.

The Generation of the Exclusion Table

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 (see Section 6.2.4). The method is named *generateExclusions*.

Again, the first step in the development is to write the interface specification of the method. The method has two formal parameters: 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 `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 looks as follows:

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

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

```
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;
```

The definition of the predicate `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 pair
  of train routes that does not shares a track segment or point
  according to locking table "ltab" may be used concurrently.

end define;
```

One approach 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 ***partMatchExclTab***. The resulting code so far reads as follows:

```
public void generateExclusions( /*+const*/ LockTab ltab, ExclTab etab )
/*+
  requires
    emptyExclTab( etab^, numRoutesLockTab( ltab^ ) );
  ensures
    matchExclTab( ltab^, etab^, etab );
  modifies
    etab;
*/
{
  int route_a = 1, route_b;
  while( route_a < ltab.numRoutesLockTab() + 1 )
/*+
  invariant
    partMatchExclTab( ltab^, etab^, etab, route_a, 1 ) and
    0 < route_a and route_a <= numRoutesLockTab( ltab^ ) + 1;
*/
    {
      route_b = 1;
      while( route_b < ltab.numRoutesLockTab() + 1 )
/*+
  invariant
    partMatchExclTab( ltab^, etab^, etab, route_a, route_b ) and
    0 < route_a and route_a < numRoutesLockTab( ltab^ ) + 1 and
    0 < route_b and route_b <= numRoutesLockTab( ltab^ ) + 1;
*/
        {
          /* check if "route_a" and "route_b" share a track segment
             or point and if not, allow them to be used concurrently */
          route_b = route_b + 1;
        }
      route_a = route_a + 1;
    }
}
```

The definition of the predicate *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 <= "route1"-1 or X is "route1" and Y is <= "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^ and route_a^ < numRoutesLockTab( ltab^ ) + 1 and
    0 < route_b^ and route_b^ < numRoutesLockTab( ltab^ ) + 1;
  ensures
    result == shareElems( ltab^, route_a^, route_b^ );
*/
```

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

```
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;
```

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

```
public void generateExclusions( /*+const*/ LockTab ltab, ExclTab etab )
/*+
  requires
    emptyExclTab( etab^, numRoutesLockTab( ltab^ ) );
  ensures
    matchExclTab( ltab^, etab );
  modifies
    etab;
*/
{
  int route_a = 1, route_b;
  while( route_a < ltab.numRoutesLockTab() + 1 )
```

```
/*+
  invariant
    partMatchExclTab( ltab^, etab^, etab, route_a, 1 ) and
    0 < route_a and route_a <= numRoutesLockTab( ltab^ ) + 1;
*/
    {
      route_b = 1;
      while( route_b < ltab.numRoutesLockTab() + 1 )
/*+
  invariant
    partMatchExclTab( ltab^, etab^, etab, route_a, route_b ) and
    0 < route_a and route_a < numRoutesLockTab( ltab^ ) + 1 and
    0 < route_b and route_b <= numRoutesLockTab( ltab^ ) + 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 on page 99.

6.3. Inspection of the Table Generator

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

6.3.1. 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 questions regarding the method *generateLockings*:

```
VC#1  generateLockings :: TableGen -----------------------------------

Suppose
   1. validTopology( topo^ )
```

```
   2. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                    numPointsTopo( topo^ ), routes^ )

Then
   1. partMatchLockTab( topo^, ltab^, ltab^, 1 )
   2. 1 <= numTracksTopo( topo^ ) + 1

Does verification condition 1 hold?
```

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 holds since the number of track segments in the topology is always greater or equal to zero according to `validTopology`.

```
VC#2  generateLockings :: TableGen -----------------------------------

Suppose
   1. validTopology( topo^ )
   2. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                    numPointsTopo( topo^ ), routes^ )
   3. partMatchLockTab( topo^, ltab^, ltab, track )
   4. 0 < track
   5. track < numTracksTopo( topo^ ) + 1

Then
   1. track <= numTracksTopo( topo^ )

Does verification condition 2 hold?
```

In the method `generateLockings` the functions `trackLeft` and `trackRight` are invoked. The above question is to check if their preconditions hold before they are invoked. Obviously, the conclusion holds. The question could be automatically answered by an extended inspection tool.

```
VC#3  generateLockings :: TableGen -----------------------------------

Suppose
   1. validTopology( topo^ )
   2. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                    numPointsTopo( topo^ ), routes^ )
   3. partMatchLockTab( topo^, ltab^, ltab, track )
   4. 0 < track
   5. track < numTracksTopo( topo^ ) + 1
   6. ( trackLeft( topo^, track ) == 0 ||
        trackRight( topo^, track ) == 0 ) &&
```

```
        trackLeft( topo^, track ) != trackRight( topo^, track ) &&
        routeEmpty( ltab )

Then
     1. routeEmpty( ltab )

Does verification condition 3 hold?
```

In the method *generateLockings* the method *incRoutesByOne* is invoked. The above question is to check if the precondition of *incRoutesByOne* holds before it is invoked. The conclusion holds 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.

```
VC#4  generateLockings :: TableGen ----------------------------------

Suppose
     1. validTopology( topo^ )
     2. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                      numPointsTopo( topo^ ), routes^ )
     3. partMatchLockTab( topo^, ltab^, ltab, track )
     4. 0 < track
     5. track < numTracksTopo( topo^ ) + 1
     6. ( trackLeft( topo^, track ) == 0 ||
          trackRight( topo^, track ) == 0 ) &&
        trackLeft( topo^, track ) != trackRight( topo^, track ) &&
        routeEmpty( ltab )
     7. incRoutesByOneMod( ltab, ltab~2 )

Then
     1. matchTrackPoint( topo^, ltab~2 )
     2. 0 < numRoutesLockTab( ltab~2 )
     3. 0 <= numPointsTopo( topo^ )
     4. track <= numTracksTopo( topo^ )

Does verification condition 4 hold?
```

In the method *generateLockings* the method *findFromTrack* is invoked. The above question is to check if the precondition of *findFromTrack* holds before the invocation. Conclusion 4 clearly holds due to premise 5. Conclusion 3 holds directly because of premise 1. That the locking table *ltab~2* contains at least one train routes (conclusion 2) holds due to premise 7 (*ltab~2* has one more train route than *ltab*), premise 3 (*ltab* is the same as *ltab^* except that zero or more train routes have been added), and premise 2 (*ltab^* has zero train routes). Finally, *ltab~2* has the same number of track segments and points as *topo^* (conclusion 1) holds because of premise 2, 3, and 7.

```
VC#5  generateLockings :: TableGen ----------------------------------

Suppose
   1. validTopology( topo^ )
   2. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                    numPointsTopo( topo^ ), routes^ )
   3. partMatchLockTab( topo^, ltab^, ltab, track )
   4. 0 < track
   5. track < numTracksTopo( topo^ ) + 1
   6. ( trackLeft( topo^, track ) == 0 ||
        trackRight( topo^, track ) == 0 ) &&
      trackLeft( topo^, track ) != trackRight( topo^, track ) &&
      routeEmpty( ltab )
   7. incRoutesByOneMod( ltab, ltab~2 )
   8. routesAddedTrack( topo^, ltab~2, ltab~1,
                        numRoutesLockTab( ltab~2 ), 0, track )

Then
   1. partMatchLockTab( topo^, ltab^, ltab~1, track + 1 )
   2. 0 < track + 1
   3. track + 1 <= numTracksTopo( topo^ ) + 1

Does verification condition 5 hold?
```

The execution of the body of the loop has to reestablish the loop invariant. The previous and the following questions check if this is the case. The previous 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 yet (that is train routes have been added to the locking table). Clearly, conclusion 2 holds because of premise 4 and conclusion 3 because of premise 5. That `ltab~1` is the same as `ltab^` but with all train routes starting from the first `track` track segments added, holds due to 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).

```
VC#6  generateLockings :: TableGen ----------------------------------

Suppose

   1. validTopology( topo^ )
   2. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                    numPointsTopo( topo^ ), routes^ )
   3. partMatchLockTab( topo^, ltab^, ltab, track )
   4. 0 < track
   5. track < numTracksTopo( topo^ ) + 1
   6. trackLeft( topo^, track ) != 0 &&
      trackRight( topo^, track ) != 0 ||
      trackLeft( topo^, track ) == trackRight( topo^, track ) ||
```

```
        not routeEmpty( ltab )

    Then
       1. partMatchLockTab( topo^, ltab^, ltab, track + 1 )
       2. 0 < track + 1
       3. track + 1 <= numTracksTopo( topo^ ) + 1

    Does verification condition 6 hold?
```

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 obviously hold. Conclusion 1 holds because of premise 3 and 6.

```
    VC#7  generateLockings :: TableGen ----------------------------------

    Suppose
       1. track == numTracksTopo( topo^ ) + 1
       2. validTopology( topo^ )
       3. emptyLockTab( ltab^, numTracksTopo( topo^ ),
                        numPointsTopo( topo^ ), routes^ )
       4. partMatchLockTab( topo^, ltab^, ltab, track )
       5. 0 < track

    Then
       1. matchLockTab( topo^, ltab^, ltab )

    Does verification condition 7 hold?
```

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^` except that all train routes through the station have been added (according to `matchLockTab`) holds because of premise 3 and premise 1 (`track`= `numTracksTopo( topo^ )`+1).

Since all questions have been answered with *yes*, it can be concluded that the method `generateLockings` is correct with respect to the assertions.

6.3.2. 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 --------------------------------
```

```
Suppose
   1. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )

Then
   1. partMatchExclTab( ltab^, etab^, etab^, 1, 1 )
   2. 1 <= numRoutesLockTab( ltab^ ) + 1

Does verification condition 1 hold?
```

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

```
VC#2  generateExclusions :: TableGen --------------------------------

Suppose
   1. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )
   2. partMatchExclTab( ltab^, etab^, etab, route_a, 1 )
   3. 0 < route_a
   4. route_a < numRoutesLockTab( ltab^ ) + 1

Then
   1. 1 <= numRoutesLockTab( ltab^ ) + 1

Does verification condition 2 hold?
```

As for the outer loop, the invariant for the inner loop has hold before the first iteration. The above questions reflects that. The conclusion clearly holds, since the number of train routes in the locking table is at least zero (premise 1).

```
VC#3  generateExclusions :: TableGen --------------------------------

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

Then
   1. matchExclTab( ltab^, etab^, etab )

Does verification condition 3 hold?
```

The above question checks if the postcondition of the method ***generateExclusions*** holds 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 holds.

```
VC#4  generateExclusions :: TableGen --------------------------------

Suppose
   1. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )
   2. partMatchExclTab( ltab^, etab^, etab, route_a, route_b )
   3. 0 < route_a
   4. route_a < numRoutesLockTab( ltab^ ) + 1
   5. 0 < route_b
   6. route_b < numRoutesLockTab( ltab^ ) + 1
   7. not shareElems( ltab^, route_a, route_b )

Then
   1. route_a <= numRoutesExclTab( etab )
   2. route_b <= numRoutesExclTab( etab )

Does verification condition 4 hold?
```

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

```
VC#5  generateExclusions :: TableGen --------------------------------

Suppose
   1. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )
   2. partMatchExclTab( ltab^, etab^, etab, route_a, route_b )
   3. 0 < route_a
   4. route_a < numRoutesLockTab( ltab^ ) + 1
   5. 0 < route_b
   6. route_b < numRoutesLockTab( ltab^ ) + 1
   7. not shareElems( ltab^, route_a, route_b )
   8. allowRoutesMod( etab, etab~1, route_a, route_b )

Then
   1. partMatchExclTab( ltab^, etab^, etab~1, route_a, route_b + 1 )
   2. 0 < route_b + 1
   3. route_b + 1 <= numRoutesLockTab( ltab^ ) + 1

Does verification condition 5 hold?
```

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 holds according to the definitions of the involved predicates. Conclusion 2 and 3 clearly hold because of premise 4 and 6.

```
VC#6  generateExclusions :: TableGen ---------------------------------

Suppose
   1. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )
   2. partMatchExclTab( ltab^, etab^, etab, route_a, route_b )
   3. 0 < route_a
   4. route_a < numRoutesLockTab( ltab^ ) + 1
   5. 0 < route_b
   6. route_b < numRoutesLockTab( ltab^ ) + 1
   7. shareElems( ltab^, route_a, route_b )

Then
   1. partMatchExclTab( ltab^, etab^, etab, route_a, route_b + 1 )
   2. 0 < route_b + 1
   3. route_b + 1 <= numRoutesLockTab( ltab^ ) + 1

Does verification condition 6 hold?
```

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

```
VC#7  generateExclusions :: TableGen ---------------------------------

Suppose
   1. route_b == numRoutesLockTab( ltab^ ) + 1
   2. emptyExclTab( etab^, numRoutesLockTab( ltab^ ) )
   3. partMatchExclTab( ltab^, etab^, etab, route_a, route_b )
   4. 0 < route_a
   5. route_a < numRoutesLockTab( ltab^ ) + 1
   6. 0 < route_b

Then
   1. partMatchExclTab( ltab^, etab^, etab, route_a + 1, 1 )
   2. 0 < route_a + 1
   3. route_a + 1 <= numRoutesLockTab( ltab^ ) + 1

Does verification condition 7 hold?
```

Finally, the seventh question is generated to check that each iteration of the outer loop reestablishes the corresponding loop invariant. The first conclusion holds 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 hold due to premise 4 and 5.

Since all questions generated for the method `generateExclusions` have been answered with *yes*, it can be concluded that the method is correct with respect to the assertions.

7. Related Work

A great deal of work has been done in the field of software specification and verification, including program development, program verification, dynamic testing, and code inspection.

7.1. Program Development

As mentioned in Chapter 1, Hoare introduced a theory that provides the means for formally proving the correctness of a program with respect to given assertions added to the program after its development.

However, Dijkstra [Dij76] 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's logic (see Section 4.5). Later his ideas have been further developed by Gries [Gri81]. Another introduction to program development based on Dijkstra's discipline has been written by Backhouse [Bac86].

7.2. Formal Program Verification

There is a vast amount of literature on the subject of formally verifying the correctness of programs, mainly based on Hoare's Logic [Hoa69]. This approach has been extended in several ways to cover special programming language constructs, for instance pointers [Bij89], procedure calls [Hoa71], [Mar83], [GL80], [Bij93] recursive procedures [Hes93], data structures [Hoa72], and gotos [dB81].

7.2.1. SPARK

One of the newer and more successful approaches to formal software verification is the Ada subset called SPARK [CG90]. SPARK is a subset of Ada 83 that is extended by 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.

Mandatory annotations are required to perform extended static code analysis and comprise for instance the definition of used global variables and the definition of dependency relations, that is, a specification of which variables are imported and exported by a procedure and how they are related. The other kind of annotations, so called proof contexts, are used to introduce elements of formal specifications and proof obligations, for instance 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. SPARK mostly aims at low level properties, for instance the absence of run-time errors [GOC93], whereas COMPASS is more suitable for reasoning about the high level algorithmic content of a program.

7.3. Specification and Verification of Object-Oriented Programs

Object-oriented programming languages have received a great deal of attention during recent years. Hence, there has been a great deal of work in the field of specifying and verifying object-oriented programs.

7.3.1. Annotated C++ (A++)

In [CL90] 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++. The programming tool related to A++ determines to a certain extent statically that the conditions on the state of objects are satisfied.

7.3.2. Eiffel

In [Mey92a] Meyer stresses the importance of reliability for the 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 [Mey92b] 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. This supports the testing process but in practical applications it cannot guarantee the absence of errors because exhaustive testing in general is not possible. Moreover, since assertions are written in the programming language itself they may change the state of the execution and thus lead to a very different behavior of the software with monitoring activated or deactivated.

7.3.3. Larch/C++

Another approach for the specification of C++ programs is Larch/C++ [CL94]. Larch/C++ is an interface specification language that enables a precise description of both the behavior and the C++ 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 the type but in terms of “abstract values”. An interface specification consists of a requires, an ensures, and a modifies clause. 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. The notation used to specify interfaces in COMPASS has to some extent been taken from the Larch/C++ approach. However, COMPASS is a semi-formal approach that supports the development of code and that enables the automatic generation of questions to be asked during code inspection whereas Larch/C++ is used for documentation purposes.

A detailed description of different approaches to the specification and verification of object-oriented programs can be found in the PhD thesis by Poetzsch-Heffter [PH97]. His thesis contains a formal framework for interface specifications and programming language semantics.

7.4. Dynamic Checking

Several approaches exploit code annotations to improve dynamic testing, for instance Robust C [FY93], APP [Ros95], Anna [LvH95], and C-Patrol [YB94]. Common to these approaches is that they extend the underlying programming language or introduce special kinds of comments to be written together with the code. A slightly different approach is used by ADLT [ADL95], where the (interface) specifications are not mixed together with the code; the specification is stored in a different file instead. However, the additional constructs can be used for instance for array index checking, range checking, or loop invariant checking.

In comparison to simple black box testing the above approaches improve error detection and decrease the necessary debugging effort to find the underlying fault. The assertions that have to be specified for applying the COMPASS method might be used in a similar way. However, in this case all used predicates have to be translated into executable code.

The major drawback of the above approaches is, that in practical applications none of the approaches can guarantee the absence of errors in the program under investigation because exhaustive testing in general is not possible. Moreover, extensive testing is very expensive.

7.5. Code Inspection

Code inspection was developed by Fagan in 1972 at IBM Kingston. It is a visual examination of code to detect errors in the code. A reader is paraphrasing the code 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. Still, what actually has to be done in an inspection session is only loosely defined. It is more or less up to the participants and thus, it is not clear how the inspection should be documented or repeated. Changes may require new inspections of large parts of the implementation. Moreover, since the code is not checked for all kinds of errors, the code might still be erroneous.

Nevertheless, in [Fag76] Fagan argues that design and code inspections increase the productivity and improve the final program quality. Ten years later, in 1986 [Fag86], Fagan suggests slight modifications to the inspection process and reports further industrial experiences that support his earlier results. In [Rus91] Russell describes similar experiences with the inspection in ultralarge-scale developments.

7.5.1. Assertion-driven Programming

One possible method that describes more precisely what actually has to be done in an inspection session was introduced by van Emden [vE92] in 1992. His code inspection method is based on Floyd's method for the verification of flowcharts [Flo67]. His basic idea was 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.

In order to obtain the annotated code, van Emden proposed a program development method, called *assertion-driven programming*. 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 [Dij76] method. However, van Emden's method does not produce code according to structured programming.

The major difference between van Emden's and the COMPASS approach is, that in COMPASS a *formal syntax* is combined with a partly *informal semantics* for the language of assertions. This enables automatic support of many kinds which is not possible in van Emden's approach: predicate transformation (and hence fewer assertions to specify), arithmetic simplification, and generation of verification conditions. Moreover, since COMPASS uses Dijkstra's development method, the code produced is in accordance with structured programming.

7.5.2. Cleanroom

The cleanroom method [Dye92] presents a practical approach to place software development under statistical quality control to deliver software with known and certified mean time to failure (MTTF). One of the major parts of the cleanroom method is the correctness verification that is carried out in form of so called *verification-based inspections*. These inspections are used for an independent confirmation of the correctness of the design. To be performed they 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, as presented in [Dye92] the design takes the form of annotated pseudo-code where the annotations are informal assertions written in natural language.

An advantage of using verification-based inspections is the organization of the questions to be asked during the inspection session. However, since the assertions do not have a formal syntax as in COMPASS it is not possible to automatically generate verification conditions that reflect the high-level algorithmic content of the design and are independent from actual text. The questions in accordance to the cleanroom method often have the form: "Does the behavior spec of line x satisfy the behavior spec on line y when the predicate on line z is true?". This means, the inspection team has to consider the effect of the design on the assertions themself.

8. Conclusion and Future Work

The COMPASS method presented in this thesis is based on Hoare's method for proving programs correct, Dijkstra's discipline for program development, and Fagan's work on code inspection. Both Hoare's verification method and Dijkstra's program development discipline are well established in academia. However, the methods are hardly known by industry. We believe this 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 hypothesis presented is that a method which is more easily used in practice but which still remains partially mechanizable, may be achieved by relaxing the requirements on formal rigor in a controlled manner. The novel key idea is the introduction of informal predicates, which, though not having a formal definition, may have a perfectly legal and unique interpretation. These informal predicates make it easier to express the required assertions and enable reasoning about assertions and verifying verification conditions at a level which is suitable for man rather than machine. Since a formal syntax is combined with an informal semantics, it is still possible to automatically generate verification conditions. The verification conditions provide questions to be asked during code inspection. The somewhat loosely defined content of the steps of Fagan's code inspection method is thus filled with a concrete content. Moreover, COMPASS does not only allow isolating faults in the code. COMPASS enables the generation of all questions relevant to establish the correctness of the code. Hence, if all questions are correctly answered with *yes* the inspected code may be considered correct with respect to the assertions (and the limitations described in Section 4.3). Furthermore, 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 code.

A prototype tool to support the code inspection has been implemented and several small examples and a few larger once have been developed and inspected according to the principles of the COMPASS method. Although it is to early for a final assessment, our experience so far shows that informal predicates make it easier to express assertions at a high level which makes the algorithmic content of a program explicit. Moreover, the notion of informal predicates enables reasoning about assertions and verifying programs at an appropriate level. However, to successfully apply the COMPASS method in practice it seems to be required to carefully introduce the potential user in the underlying principles especially the discipline of programming. Furthermore, extended and improved tool support seems indispensable.

A major area of possible extensions of the COMPASS method is the interaction with the user. As the method is presented here, the user has to specify assertions and to answer questions in a form of predicate logic. However, this might not be the easiest way for an average programmer to express her intentions. An alternative may be to write the specification of a program (that is the assertions) in tabular form as presented by Parnas [PMI94] instead. Parnas reports in his article that the tabular notation allows "to describe a program in a more readable manner" compared to the conventional notation. Another approach to improve the interaction with the user is to develop a natural language interface. Since assertions and questions in COMPASS are allowed to contain informal predicates, that is predicates that are defined in natural language, it should be possible to develop this kind of support.

Another area of further developments of the COMPASS method concerns object-orientation. As outlined in Section 4.3.3, there are still open questions regarding the formal specification and verification of object-oriented programs. Though, this is not directly related to our intentions with the COMPASS approach it might be of interest to investigate if the semi-formal approach can be used to help overcoming these challenges.

Moreover, we intend to evaluate the COMPASS method further. As mentioned our experience so far has been promising. Our industrial partner ABB Industrial Systems has stated that, even though it is too early for a final evaluation, it seems apparent that "COMPASS is a simplification compared to traditional methods". To be able to further evaluate the COMPASS approach, we are planning to develop a new inspection tool that supports a larger portion of the Java programming language and that provides an improved user interface.

The COMPASS method as described in detail in this work is concerned with the development and verification 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 for instance [Lut92]). Therefore, it might be of great interest to study whether our general idea – to decrease the requirements on formality while still keeping a sufficient level of rigor – may be applied to higher level designs. We believe this may be a way to increase industrial acceptance of formally based development methods and will therefore develop and asses such a semi-formal specification and design method by combining, adapting, and extending existing formal methods. The method to be developed should help avoiding or detecting several types of errors which today survive until testing is commenced (or later) and it should be possible to smoothly integrate the method into a typical industrial development process of today.

Bibliography

[ADL95] Sun Microsystems Inc. and Information-technology Promotion Agency. *ADL Translator User's Guide: Getting Started with ADLT*, December 1995.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Bac86] Roland C. Backhouse. *Program Construction and Verification*. Prentice Hall, 1986.

[BH97] Staffan Bonnier and Tim Heyer. COMPASS: A comprehensible assertion method. In *TAPSOFT '97: Theory and Practice of Software Development*, Lecture Notes in Computer Science, pages 803–817. Springer-Verlag, 1997.

[Bij89] A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–205, September 1989.

[Bij93] A. Bijlsma. Calculating with procedure calls. *Information Processing Letters*, 46(5):211–217, July 1993.

[BMR95] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[CG90] Bernard Carré and Jonathan Garnsworthy. SPARK: An annotated Ada subset for safety-critical programming. Presented at TRI-Ada, 1990.

[CL90] Marshall P. Cline and Doug Lea. The behavior of C++ classes. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[CL94] Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, pages 39–49, October 1994.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, third edition, 1987.

[dB81] Arie de Bruin. Goto statements: Semantics and deductive systems. *Acta Informatica*, 15:385–424, 1981.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dye92] Michael Dyer. *The Cleanroom Approach to Quality Software Development*. John Wiley & Sons, 1992.

[Fag76] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(1):182–211, 1976.

[Fag86] Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposiom in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[FY93] David W. Flater and Yelena Yesha. Extensions to the C programming language for enhanced fault detection. *Software-Practice and Experience*, 23(6):617–628, June 1993.

[GL80] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, 2(4):564–579, October 1980.

[GOC93] Jonathan Garnsworthy, Ian O'Neill, and Bernard Carré. Automatic proof of the absence of run-time errors. In *ADA: Towards Maturity*, pages 108–122. IOS Press, 1993.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Han96] Kirsten Mark Hansen. *Linking Safety Analysis to Safety Requirements - Exemplified by Railway Interlocking Systems*. Ph.d. thesis, Technical University of Denmark, Department of Information Technology, August 1996.

[Hes93] Wim H. Hesselink. Proof rules for recursive procedures. *Formal Aspects of Computing*, 5:554–570, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12(10):576–80, 583, October 1969.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Sumposium on Semantics of Algorithmic Languages*, Lecture Notes in Computer Science, pages 102–116. Springer-Verlag, 1971.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[Lut92] Robyn R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 126–133, 1992.

[LvH95] David C. Luckham and Friedrich W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1995.

[Mar83] Alain J. Martin. A general proof rule for procedures in predicate transformer semantics. *Acta Informatica*, 20:301–313, 1983.

[Mey92a] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Mey92b] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[PH97] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Habilitationsschrift, Technische Universität München, Fakultät für Informatik, January 1997.

[PMI94] David Lorge Parnas, Jan Madey, and Michal Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Rus91] Glen W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, 1991.

[SIC95] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 1995.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[vE92] Maarten H. van Emden. Structured inspection of code. *Software Testing, Verification and Reliability*, 2:133–153, 1992.

[YB94] Hwei Yin and James M. Bieman. Improving software quality with assertion insertion. In *Proceedings of the IEEE International Test Conference*, pages 831–839. IEEE, 1994.

A. Topology.java

```
public class Topology
/*+
  invariant
    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^ );
    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^, tracks^ );
    ensures
      addTracksMod( this^, this, tracks^ );
    modifies
      this;
  */
  {
    num_tracks = num_tracks + tracks;
  }
```

```
public void addPoint( /*+const*/ int base, /*+const*/ int plus,
                      /*+const*/ int minus )
/*+
  requires
    pointEmpty( this^ ) and
    openTracks( this^, base^, plus^, minus^ );
  ensures
    addPointMod( this^, this, base^, plus^, minus^ );
  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^ );
*/
{
  return num_tracks;
}

public int numPointsTopo()
/*+
  ensures
    result == numPointsTopo( this^ );
*/
{
  return num_points;
}

public int pointBase( /*+const*/ int point )
```

```
/*+
  requires
    0 < point^ and point^ <= numPointsTopo( this^ );
  ensures
    result == pointBase( this^, point^ );
*/
{
  return point_table[ point - 1 ][ 0 ];
}

public int pointPlus( /*+const*/ int point )
/*+
  requires
    0 < point^ and point^ <= numPointsTopo( this^ );
  ensures
    result == pointPlus( this^, point^ );
*/
{
  return point_table[ point - 1 ][ 1 ];
}

public int pointMinus( /*+const*/ int point )
/*+
  requires
    0 < point^ and point^ <= numPointsTopo( this^ );
  ensures
    result == pointMinus( this^, point^ );
*/
{
  return point_table[ point - 1 ][ 2 ];
}

public int trackLeft( /*+const*/ int track )
/*+
  requires
    0 < track^ and track^ <= numTracksTopo( this^ );
  ensures
    result == trackLeft( this^, track^ );
*/
{
  return track_table[ track - 1 ][ 0 ];
}

public int trackRight( /*+const*/ int track )
/*+
  requires
    0 < track^ and track^ <= numTracksTopo( this^ );
```

```
    ensures
      result == trackRight( this^, track^ );
  */
  {
    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;

*/

}
```

B. LockTab.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^, points^, routes^ );
    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^ );
    ensures
      incRoutesByOneMod( this^, this );
    modifies
      this;
  */
  {
    num_routes = num_routes + 1;
  }
```

```
public void copyRoute( /*+const*/ int from, /*+const*/ int to )
/*+
  requires
    0 < from^ and from^ <= numRoutesLockTab( this^ ) and
    0 < to^ and to^ <= numRoutesLockTab( this^ );
  ensures
    copyRouteMod( this^, this, from^, to^ );
  modifies
    this;
*/
{
  int i = 0;
  while( i < num_points )
/*+
  invariant
    pointsPartlySame( this^, this, from^, to^, i ) and
    0 <= i and i <= num_points;
*/
    {
      point_table[ to - 1 ][ i ] = point_table[ from - 1 ][ i ];
      i = i + 1;
    }
  i = 0;
  while( i < num_tracks )
/*+
  invariant
    tracksPartlySame( this^, this, from^, to^, i ) and
    0 <= i and i <= num_tracks;
*/
    {
      track_table[ to - 1 ][ i ] = track_table[ from - 1 ][ i ];
      i = i + 1;
    }
}

public void lockTrackForw( /*+const*/ int route, /*+const*/ int track )
/*+
  requires
    0 < route^ and route^ <= numRoutesLockTab( this^ ) and
    0 < track^ and track^ <= numTracksLockTab( this^ );
  ensures
    lockTrackForwMod( this^, this, route^, track^ );
  modifies
    this;
*/
{
  track_table[ route - 1 ][ track - 1 ] = 1;
```

```
}

public void lockTrackBack( /*+const*/ int route, /*+const*/ int track )
/*+
  requires
    0 < route^ and route^ <= numRoutesLockTab( this^ ) and
    0 < track^ and track^ <= numTracksLockTab( this^ );
  ensures
    lockTrackBackMod( this^, this, route^, track^ );
  modifies
    this;
*/
{
  track_table[ route - 1 ][ track - 1 ] = 2;
}

public void lockPointMinus( /*+const*/ int route, /*+const*/ int point )
/*+
  requires
    0 < route^ and route^ <= numRoutesLockTab( this^ ) and
    0 < point^ and point^ <= numPointsLockTab( this^ );
  ensures
    lockPointMinusMod( this^, this, route^, point^ );
  modifies
    this;
*/
{
  point_table[ route - 1 ][ point - 1 ] = 2;
}

public void lockPointPlus( /*+const*/ int route, /*+const*/ int point )
/*+
  requires
    0 < route^ and route^ <= numRoutesLockTab( this^ ) and
    0 < point^ and point^ <= numPointsLockTab( this^ );
  ensures
    lockPointPlusMod( this^, this, route^, point^ );
  modifies
    this;
*/
{
  point_table[ route - 1 ][ point - 1 ] = 1;
}

public boolean trackForwLocked( /*+const*/ int route, /*+const*/ int track )
/*+
  requires
```

```
      0 < route^ and route^ <= numRoutesLockTab( this^ ) and
      0 < track^ and track^ <= numTracksLockTab( this^ );
    ensures
      result == trackForwLocked( this^, route^, track^ );
  */
  {
    return track_table[ route - 1 ][ track - 1 ] == 1;
  }

  public boolean trackBackLocked( /*+const*/ int route, /*+const*/ int track )
  /*+
    requires
      0 < route^ and route^ <= numRoutesLockTab( this^ ) and
      0 < track^ and track^ <= numTracksLockTab( this^ );
    ensures
      result == trackBackLocked( this^, route^, track^ );
  */
  {
    return track_table[ route - 1 ][ track - 1 ] == 2;
  }

  public boolean pointMinusLocked( /*+const*/ int route, /*+const*/ int point )
  /*+
    requires
      0 < route^ and route^ <= numRoutesLockTab( this^ ) and
      0 < point^ and point^ <= numPointsLockTab( this^ );
    ensures
      result == pointMinusLocked( this^, route^, point^ );
  */
  {
    return point_table[ route - 1 ][ point - 1 ] == 2;
  }

  public boolean pointPlusLocked( /*+const*/ int route, /*+const*/ int point )
  /*+
    requires
      0 < route^ and route^ <= numRoutesLockTab( this^ ) and
      0 < point^ and point^ <= numPointsLockTab( this^ );
    ensures
      result == pointPlusLocked( this^, route^, point^ );
  */
  {
    return point_table[ route - 1 ][ point - 1 ] == 1;
  }

  public int numTracksLockTab()
  /*+
```

```
    ensures
      result == numTracksLockTab( this^ );
  */
  {
    return num_tracks;
  }

  public int numPointsLockTab()
  /*+
    ensures
      result == numPointsLockTab( this^ );
  */
  {
    return num_points;
  }

  public int numRoutesLockTab()
  /*+
    ensures
      result == numRoutesLockTab( this^ );
  */
  {
    return num_routes;
  }

  public boolean routeEmpty()
  /*+
    ensures
      result == routeEmpty( this^ );
  */
  {
    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.  The table is initialized such that
    all possible train routes do not contain any 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;

*/

}
```

C. ExclTab.java

```
public class ExclTab
{

  private boolean[][] excl_table;
  private int num_routes;

  public ExclTab( /*+const*/ int routes )
  /*+
    ensures
      emptyExclTab( this, routes^ );
    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^ and route_a^ <= numRoutesExclTab( this^ ) and
      0 < route_b^ and route_b^ <= numRoutesExclTab( this^ );
    ensures
      allowRoutesMod( this^, this, route_a^, route_b^ );
    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^ and route_a^ <= numRoutesExclTab( this^ ) and
```

```
      0 < route_b^ and route_b^ <= numRoutesExclTab( this^ );
    ensures
      result == routesAllowed( this^, route_a^, route_b^ );
  */
  {
    return excl_table[ route_a - 1 ][ route_b - 1 ];
  }

  public int numRoutesExclTab()
  /*+
    ensures
      result == numRoutesExclTab( this^ );
  */
  {
    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

    Train route "route1" and train route "route2" may be used
    concurrently according to exclusion table "etab".

  end define;

  define numRoutesExclTab( etab ) informally

    Returns the number of train routes in exclusion table "etab".

  end define;
```

```
*/

}
```

D. TableGen.java

```
public class TableGen
{

  public void findFromPoint( /*+const*/ Topology topo, LockTab ltab,
                             /*+const*/ int route, /*+const*/ int track,
                             /*+const*/ int point )
  /*+
    requires
      validTopology( topo^ ) and
      matchTrackPoint( topo^, ltab^ ) and
      0 < route^ and route^ <= numRoutesLockTab( ltab^ ) and
      0 < track^ and track^ <= numTracksTopo( topo^ ) and
      0 <= point^ and point^ <= numPointsTopo( topo^ );
    ensures
      routesAddedPoint( topo^, ltab^, ltab, route^, track^, point^ );
    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 ) );
```

```
        ltab.lockPointMinus( new_route, point );
        findFromTrack( topo, ltab, new_route, point,
                       topo.pointMinus( point ) );
      }
    }
  }
}

public void findFromTrack( /*+const*/ Topology topo, LockTab ltab,
                           /*+const*/ int route, /*+const*/ int point,
                           /*+const*/ int track )
/*+
  requires
    validTopology( topo^ ) and
    matchTrackPoint( topo^, ltab^ ) and
    0 < route^ and route^ <= numRoutesLockTab( ltab^ ) and
    0 <= point^ and point^ <= numPointsTopo( topo^ ) and
    0 < track^ and track^ <= numTracksTopo( topo^ );
  ensures
    routesAddedTrack( topo^, ltab^, ltab, route^, point^, track^ );
  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^ ) and
    emptyLockTab( ltab^, numTracksTopo( topo^ ), numPointsTopo( topo^ ),
      routes^ );
  ensures
    matchLockTab( topo^, ltab^, ltab );
  modifies
    ltab;
*/
{
  int track = 1;
```

```
  while( track < topo.numTracksTopo() + 1 )
/*+
  invariant
    partMatchLockTab( topo^, ltab^, ltab, track ) and
    0 < track and track <= numTracksTopo( topo^) + 1;
*/
    {
      if( ( topo.trackLeft( track ) == 0 || topo.trackRight( track ) == 0 )
          && topo.trackLeft( track ) != topo.trackRight( track )
          && ltab.routeEmpty() ){
        ltab.incRoutesByOne();
        findFromTrack( topo, ltab, ltab.numRoutesLockTab(), 0, track );
      }
      track = track + 1;
    }
}

public boolean shareElems( /*+const*/ LockTab ltab,
                           /*+const*/ int route_a,
                           /*+const*/ int route_b )
/*+
  requires
    0 < route_a^ and route_a^ < numRoutesLockTab( ltab^ ) + 1 and
    0 < route_b^ and route_b^ < numRoutesLockTab( ltab^ ) + 1;
  ensures
    result == shareElems( ltab^, route_a^, route_b^ );
*/
{
  boolean sharet = false, sharep = false;
  int track = 1, point = 1;
  while( track < ltab.numTracksLockTab() + 1 && !sharet )
/*+
  invariant
    shareTrack( ltab^, route_a^, route_b^, track, sharet ) and
    0 < track and track <= numTracksLockTab( ltab^ ) + 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 )
/*+
  invariant
    sharePoint( ltab^, route_a^, route_b^, track, sharep ) and
```

```
    0 < point and point <= numPointsLockTab( ltab^ ) + 1;
*/
    {
      sharep = ( ltab.trackForwLocked( route_a, point ) ||
                 ltab.trackBackLocked( route_a, point ) )
        && ( ltab.trackForwLocked( route_b, point ) ||
             ltab.trackBackLocked( route_b, point ) );
      point = point + 1;
    }
  return sharet || sharep;
}

public void generateExclusions( /*+const*/ LockTab ltab, ExclTab etab )
/*+
  requires
    emptyExclTab( etab^, numRoutesLockTab( ltab^ ) );
  ensures
    matchExclTab( ltab^, etab^, etab );
  modifies
    etab;
*/
{
  int route_a = 1, route_b;
  while( route_a < ltab.numRoutesLockTab() + 1 )
/*+
  invariant
    partMatchExclTab( ltab^, etab^, etab, route_a, 1 ) and
    0 < route_a and route_a <= numRoutesLockTab( ltab^ ) + 1;
*/
    {
      route_b = 1;
      while( route_b < ltab.numRoutesLockTab() + 1 )
/*+
  invariant
    partMatchExclTab( ltab^, etab^, etab, route_a, route_b ) and
    0 < route_a and route_a < numRoutesLockTab( ltab^ ) + 1 and
    0 < route_b and route_b <= numRoutesLockTab( ltab^ ) + 1;
*/
        {
          if( !shareElems( ltab, route_a, route_b ) )
            etab.allowRoutes( route_a, route_b );
          route_b = route_b + 1;
        }
      route_a = 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 train routes 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 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
```

```
    pair of train routes that does not shares 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 <= "route1"-1 or X is "route1" and Y is <= "route2"-1.

  end define;

*/

}
```