COMPASS: A Comprehensible Assertion Method

Staffan Bonnier[1] and Tim Heyer[2]

[1] Carlstedt Research & Technology AB (CR&T), Stora Badhusgatan 18-20, S-411 21 Göteborg, SWEDEN, stabon@carlstedt.se
[2] Real-Time Systems Laboratory, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, SWEDEN, timhe@ida.liu.se

Abstract. We present an approach for automatically generating relevant, focused questions to be asked during code inspection sessions. The method is based on Hoare-logic. 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 for the automatic derivation 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.

1 Introduction

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

The general goal of the project is to develop practical means for increasing confidence in software correctness. Our strategy is to provide semi-formal support both for the development of code, and for its inspection. The latter is achieved by supporting 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.

This article reports on ongoing work in the project. The purpose is to present the basic principles of the COMPASS method, to explain the rationale behind these principles in terms of the general goal, and to also present our plans for the continued development. The style of presentation is chosen rather to give a flavour of COMPASS, than to present its formal underpinnings.

The cornerstones on which COMPASS is based are (1) Hoares method for proving programs correct [14] (and hence, to a certain extent, Floyds intermediate assertion method [10]), (2) Dijkstras discipline for program development [6], and (3) Fagans work on code inspection [8].

COMPASS thus has its foundation in well-known theories for so called *assertional programming.* 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 procedure two special assertions called the *pre-* and the *postcondition* of the procedure, assertions may be used to specify the intended result of executing the procedure. Hoare introduced in 1969 [14] a logic for reasoning about assertions. The formulae of the logic are so called *Hoare triples* $\{P\}C\{Q\}$, where P and Q are assertions, and C is a piece of code. It is to be read "if C starts executing in a state where P holds, and if the execution of C 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 three phases: (1) Development of asserted code (i.e. code decorated with assertions), (2) Derivation of a set of *verification conditions,* using the proof rules of the programming language, and (3) A formal proof of the verification conditions, using axioms and proof rules for the domain over which the program variables range.

Dijkstra [6] noted that verifying code after it has been developed is not entirely realistic. Dijkstra instead proposed that code should be developed along with arguments for its correctness. For this purpose he suggested a discipline of programming, based on Hoare-logic, where one states the assertions the code is to establish, and then uses the assertions to guide the development of the code.

Both Hoares method (see e.g. [1]) and Dijkstras 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 by 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 (e.g. [18]). We believe this lack of understanding of Hoares and Dijkstras ideas is 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.

Our basic hypothesis is that a method which is more easily used in practice, and which remains to be partially mechanizable, may be achieved by relaxing the requirements on formal rigour in a controlled manner. The novel key idea in COMPASS concerns predicates ranging over the domain of program variables, and which hence are used for expressing assertions; COMPASS allows such predicates to occur without an associated formal definition (axiomatization). They are instead expected to be defined in normal prose in a special kind of 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, expressible e.g. as a comment in the code.

Our experience so far is that informal predicates (a) enables expressing assertions at a high level which makes the algorithmic contents of a program explicit, and (b) enables reasoning about assertions and verification conditions at a level which is suitable for man rather than machine. This should be contrasted to the case when a formal axiomatization is required for each predicate. Such axiomatizations do often have a non-obvious connection to the intuitive understanding of the property the predicate is to represent. As a consequence, they are both difficult to state and to reason about.

The resulting method thus simplifies the formulation of assertions in phase (1) of Hoares method. Furthermore, phase (2) is not dependent on the meaning of the predicates involved. Thus verification conditions sufficient for the correctness of the program are automatically derived. These conditions will in general themselves contain informal predicates, thus disabling the possibility of formal proof. We therefore propose that step (3) above be substituted for an inspection session. 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. Hence the somewhat loosely defined steps of Fagans code inspection method [8] are filled with a very concrete contents, specifying a highly structured and machine supported protocol for inspecting the code.

We are presently developing a tool for the automatic derivation of verification conditions, and for presenting these conditions in a way suitable to form a basis for code inspection. In order to evaluate our method in a real industrial setting, we have adapted it to the C programming language. Certain restrictions of the language are of course needed. Furthermore, for the safe use of the method, the programmer is expected to develop the asserted code in a way compatible with the discipline of Dijkstra.

The rest of this paper is organized as follows: In Sect. 2, the principles behind the COMPASS method are elaborated. This includes both the principles for code development and inspection. In Sect. 3, the method is exemplified in terms of the development and inspection of Quicksort. Section 4 presents and compares with related work. Finally, Sect. 5 contains conclusions and future work.

2 The COMPASS approach

The primary aim of the COMPASS approach is to give systematic support for generating relevant, focused questions to be asked during code inspection sessions. The questions should clearly reflect, and hence make explicit, the programmers intentions and thoughts during code development. Our method is based on Hoare logic. The novel key idea is the introduction of informal predicates, which (1) enables expressing assertions at a level which makes the algorithmic contents of a program explicit, (2) makes possible the automatic derivation of verification conditions, and (3) enables reasoning about assertions and verifying verification conditions at a level which is suitable for man rather than machine.

It should be clear that not any program nor any property is amenable to verification along the principles of this method. To start with we restrict attention to properties representable as relations between program variables. Furthermore, the code has to be well-structured in order for the method to be applicable. Our aim is to convey the discipline of Dijkstra along with our ideas of semi-formal assertions, and hence provide a guide for the development of correct code together with automatic support for its inspection.

2.1 The Language of Assertions

Before we go on, we will briefly describe the language in which to express the assertions and the notation which we use. The language of assertions is similar to that of *predicate logic.*

A **predicate** expresses that a certain relation holds between its arguments. Predicates constitute the core of assertions. Characteristic to our approach is that we do not formalize the meaning of such a predicate, i.e. we do not formally define what relation it represents. Instead we just require that an informal but precise definition of the predicate is supplied. This should typically be given as a special kind of comment in the code.

The **variables** that might occur in assertions are either *program variables* or so called *logical variables.* Program variables are the variables that occur in the program instructions considered. They are used in the same form as in the program itself. To each formal procedure parameter `p`, there is an associated logical variable `#p` which allows referring to the initial value of p (logical variables always starts with "#"). These variables are what enables us to state the relation between the value of a variable before and after the execution of a procedure.

In order to enable the application of **data abstraction**, we have introduced a special notation. A variable followed by "<>" represents the complete abstract data structure, i.e., both the contents and the structure of the elements. The notion of data abstraction is central to our method; The method encourages data abstraction, and assertions should be formulated in terms of objects of an abstract data type. In this way, verification of a high level algorithmic nature can be separated from the verification of low level invariants of the representation of the data type (such as e.g. non-corruption of the representing data structure).

Assertions are written as a special kind of comments in the code. To distinguish between the different kinds of assertions a unique tag is put directly at the beginning of the assertion, possibly followed by a label. The tags are `pre`, `pst`, `inv`, and `ast` for precondition, postcondition, loop invariant, and intermediate assertion respectively.

Also the **informal definitions** of predicates and functions are written as special code comments. The tag `ipd` stands for informal predicate definition and the tag `ifd` for informal function definition. The appropriate tag is followed by a predicate or function name and its arguments. The informal description follows as normal text. These informal definitions form the basis for the code inspection. Moreover, they allow simple consistency checking.

2.2 Program Development

Dijkstra recognized the practical problems involved in *post facto* verification of programs. He therefore suggested a discipline for developing the asserted code along with arguments for its correctness [6]. Since the program development method is described at length in the literature, e.g. [12], we will just give a brief overview.

Programming is 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 corresponding pre- and postconditions. Then, given the postcondition (and precondition), the aim is to develop a program that satisfies the postcondition. Two building blocks for a program are the alternative command and the loop construct:

- In order to invent an **alternative command** (e.g. an `if` or `switch` statement in C), a command C has to be found, that establishes the postcondition R in at least some cases. A boolean expression that is the weakest precondition for the command C and postcondition R can be used as a guard for the alternative command. This process has to be continued until the precondition implies that at least one guard is true.
- Given pre-, postcondition, and a loop invariant, a **loop construct** is 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 can be used as the guard.
  3. Finally, the loop body is developed in a way so that it improves towards termination while reestablishing the loop invariant.

  The problem, how to discover the loop invariant remains. However, quite often one already has a certain algorithm in mind when developing the program. Writing down the loop invariant then should be rather simple. A more systematic way to develop the loop invariant is to weaken the postcondition by deleting a conjunct, replacing a constant by a variable, enlarging the range of a variable, or adding a disjunct.

The development method is presented here as described in Gries [12]. However, since we do not require all predicates to be formally defined, developing a program along with its assertions can be performed on a level more suitable for humans. An example of program development in COMPASS may be found in Sect. 3.

2.3 Code Inspection

Code inspection, as it is defined in [7], aims only at isolating faults in the code. Our approach takes 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 no errors are found, the code must be considered correct, i.e. verified, 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).

Individual Inspections. As soon as a function is implemented, the programmer can perform a code inspection to check the correctness of the function according to the assertions. It is not required that all subfunctions called by the function under investigation are fully implemented, as long as the interface specifications of the subfunctions in form of pre- and postconditions are known. Thus our approach enables the programmer to find faults in the software, or an argument for its correctness as soon as possible.

The inspection process itself consists of (1) The automatic derivation of verification conditions, and (2) An informal justification of why each condition holds. The verification conditions and the justifications given by the programmer are stored in a database. They may be used e.g. 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, possible faulty verification conditions might pass as correct. In order to exclude this source of uncertainty, group inspections should be performed.

Group Inspections. A group inspection is performed similar to the process first described by Fagan in [7] (see Sect 4.3). However, since our 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.

We propose an inspection group with three members. The participants and their different roles during the code inspection process are:

Moderator The moderator is responsible for organizing and moderating the inspection. Moreover, the moderator also participates in the discussion of the verification conditions.

Designer The designer contributes to the discussion of verification conditions with knowledge about the domain (in particular the abstract data types) and the intended behaviour of the program. She has to check whether the pre- and postconditions correspond to the intended function of the software. As the other members of the inspection group, the designer participates actively in the discussion of the automatically derived verification conditions.

Implementor The programmer who is responsible for coding the software according to assertion-driven programming (see Sect. 2.2). As the other members of the inspection group, the implementor participates in the discussion of the verification conditions.

The overall goal of our inspection process 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 consists of the following phases:

Planning phase Before the meeting the moderator distributes the asserted code.

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 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 behaviour of the program. If the assertions do not properly describe the intended behaviour, the inspection should be adjourned.
3. The group has to decide for each verification condition (which e.g. has been derived and stored during an individual inspection) whether the condition is true with respect to its informal interpretation (i.e. the interpretation determined by the informal predicate and function definitions). The verification condition together with the argument for its correctness or fault is stored in a database for later review.

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 behaviour 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 responsible programmers have to remedy all defects found. The non-valid verification conditions give a very delimited piece of code which contains the fault.
- 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 has to be redone. However, since the earlier results of the code inspection session are stored in a database, valid verification conditions do not need to be reevaluated.

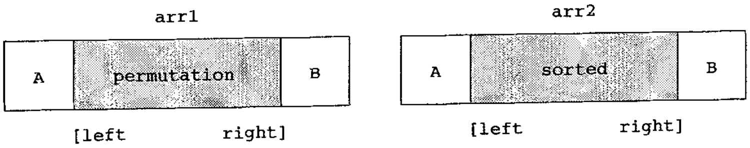
3 An Example

In this section we will give an example of how our method can be used. The program we will develop is an implementation of the sorting algorithm quicksort. 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.

Our starting point is to provide an interface specification, i.e. a procedure head, a pre-, and a postcondition. For quicksort we have no requirements on the precondition, while requiring in the postcondition that the array parameter is sorted when the procedure terminates:

```
void qsort(int v[], const int left, const int right)
/*pre true */
/*pst sorted(#v<>, v<>, #left, #right) */
```

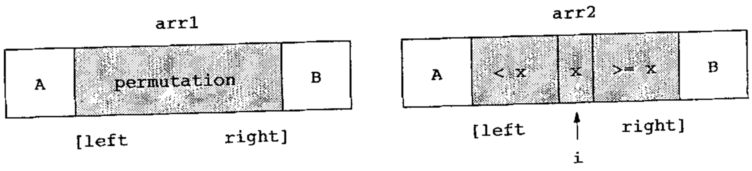
The predicate **sorted** is a typical example of an informal predicate, having only an informal definition. The relation expressed by **sorted(arr1, arr2, left, right)** is, that **arr2** is sorted in increasing order in the interval **[left, right]**, and that **arr1** and **arr2** are permutations in this interval while identical outside:



We now develop the procedure in a stepwise refinement fashion, carefully stating along the way all assertions that we intend to hold. Initially we check whether there are actually more than one element to be sorted; if this is not the case we are done:

```
void qsort(int v[], const int left, const int right) {
   if(left<right){
      /* sort v[] inbetween left and right */}
/*pst sorted(#v<>, v<>, #left, #right) */}
```

As already mentioned, **qsort** sorts by first splitting the array around a splitting element, and then recursively sorting the two sub-arrays. We therefore introduce the (informal) predicate **partition** which states that the array is split around a certain integer **i**. Concretely, **partition(arr1, arr2, left, right, i)** expresses that **arr1** and **arr2** are permutations in the range **[left, right]** while identical outside, and that the following relation holds:



Assuming we have a procedure **split** which does the actual job of splitting the array, we may write the full definition of **qsort**:

```
void qsort(int v[], const int left, const int right) {
   int i;
   if(left<right){
      split(v, left, right, &i);
/*ast partition(#v<>, v<>, #left, #right, i) */
      qsort(v, left, i-1);
      qsort(v, i+1, right);}
/*pst sorted(#v<>, v<>, #left, #right) */}
```

The idea is that `split` should return the splitting position in the variable `i`, thus it is called with a reference `&i` to `i`. We are obliged to develop `split` in such a way that it establishes `partition(#v<>, v<>, #left, #right, i)` as its postcondition. It is furthermore only meaningful to apply `split` in case there is at least one element in the array to be partitioned. Thus we also need a precondition for `split`, expressing this assumption:

```
void split(int v[], const int left, const int right, int* i)
/*pre left<=right */
/*pst partition(#v<>, v<>, #left, #right, *i) */
```

From the definition of `qsort` and the interface specification of `split`, a number of verification conditions may (automatically) be derived. Their satisfaction implies the correctness of all assertions, that is, that each assertion holds each time execution reaches the position where it is placed. The following are the derived verification conditions:

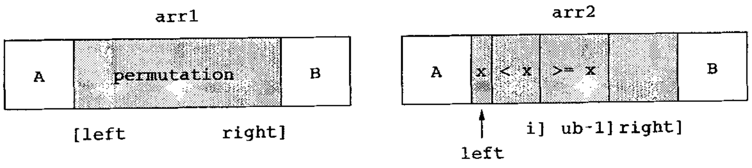
```
Suppose:
   1. #left>=#right
Then:
   A. sorted(#v<>,v<>,#left,#right)

Suppose:
   1. #left<#right
Then:
   A. #left<=#right

Suppose:
   1. #left<#right
   2. partition(#v<>, v<>, #left, #right, i)
   3. sorted(v<>, v'<>, #left, i-1)
   4. sorted(v'<>, v''<>, i+1, #right)
Then:
   A. sorted(#v<>, v''<>, #left, #right)
```

The first two conditions are clearly satisfied. Indeed, the second condition could be automatically discharged. Furthermore, using the informal definitions of `partition` and `sorted` it is not difficult to argue for the correctness of the third condition.

Let us now develop the function `split`. For the sake of simplicity we choose here the leftmost element in the array `v`, i.e. `v[left]`, as the splitting element.

Our general strategy then is as follows: We keep a position ub (for "upper bound") and an index i. The idea is that each of the elements v[left+1] to v[i] should be strictly less than v[left], and each of the elements v[i+1] to v[ub-1] should be greater than or equal to v[left]. The relation between the array v<>, and the indices i and ub should be kept invariant. That is, assuming it holds before executing the body of the the main loop, it also holds after its completion. By successively incrementing ub, while preserving the invariant, we eventually have ub=right+1. At this point we can simply interchange v[left] and v[i] and we are done.

To express the invariant as an assertion in the program code, we introduce an informal predicate pre-partition. The predicate pre-partition(arr1, arr2, left, right, i, ub) expresses that arr1 and arr2 are permutations in the range [left, right] while identical outside, and that the following relation holds:



Given the above informal predicates pre-partition and partition, the first skeleton of split thus looks as follows:

```
void split(int v[], const int left, const int right, int* i)
/*pre left<=right */ {
   int ub;
   /* establish the invariant */
   while(ub<=right){
/*inv pre-partition(#v<>,v<>,#left,#right,*i,ub) and ub<=#right+1 */
      /* preserve the invariant while incrementing ub */}
   /* interchange v[left] and v[*i] */
/*pst partition(#v<>,v<>,#left,#right,*i) */ }
```

Establishing the invariant is easy. It is sufficient to set *i to left and ub to left+1. To enable incrementation of ub without loosing satisfaction of the invariant, we have to consider two cases: in case v[ub]$\geq$v[left] the invariant is reestablished by incrementing ub. In the other case, when v[ub]<v[left] we do need to do some more work in order to reestablish the invariant. Noting that, according to the invariant, v[*i+1]$\geq$v[left] and v[*i]<v[left], we may conclude that the invariant will be reestablished, provided that we increment *i, and then (after this incrementation) interchange v[*i] and v[ub] before we increment ub. As will be seen, this argument corresponds exactly to the argument for the verification conditions generated from the final code. We may now write the complete procedure, including also the interchange which is the last step:

```
void split(int v[], const int left, const int right, int* i)
/*pre left<=right */ {
   int ub=left+1;
   int temp;
   *i=left;
   while(ub<=right){
/*inv pre-partition(#v<>,v<>,#left,#right,*i,ub) and ub<=#right+1 */
      if(v[ub]<v[left]){
         *i=*i+1;
         temp=v[*i]; v[*i]=v[ub]; v[ub]=temp;}
      ub=ub+1;}
   temp=v[*i]; v[*i]=v[left]; v[left]=temp;
/*pst partition(#v<>,v<>,#left,#right,*i) */}
```

The four verification conditions generated from this code and the assertions are the following:

```
Suppose:
   1. #left<=#right
Then:
   A. pre-partition(#v<>, v<>, #left, #right, #left, #left+1)

Suppose:
   1. pre-partition(#v<>, v<>, #left, #right, *i, ub)
   2. ub<=#right
   3. v[ub]<v[#left]
Then:
   A. pre-partition(#v<>, v'<>, #left, #right, *i+1, ub+1),
      where v'=v except for v'[ub]=v[*i+1], v'[*i+1]=v[ub]

Suppose:
   1. pre-partition(#v<>, v<>, #left, #right, *i, ub)
   2. ub<=#right
   3. v[ub]>=v[#left]
Then:
   A. pre-partition(#v<>, v<>, #left, #right, *i, ub+1)

Suppose:
   1. pre-partition(#v<>, v<>, #left, #right, *i, #right+1)
Then:
   A. partition(#v<>, v'<>, #left, #right, *i)
      where v'=v except for v'[#left]=v[*i], v'[*i]=v[#left]
```

The first condition corresponds to the first establishment of the invariant, and is clearly satisfied. The second and third condition correspond to the preservation of the invariant, and are satisfied according to the argument provided above along with the code development. Finally, the fourth condition is satisfied due to the fact (coming from `pre-partition`) that `v[#left]` is strictly greater than each element in `v[#left+1],..,v[*i]`, while being smaller than or equal to the elements in `v[*i+1],..,v[#right]`.

4 Related Work

A lot of work has been done to increase confidence in software correctness, including program verification, dynamic testing, and code inspection.

4.1 Program Verification

There is a vast amount of literature on the subject of formally verifying the correctness of programs, mainly based on so called Hoare-logic suggested by Hoare in 1969 [14]. This approach has been extended in several ways to cover special programming language constructs, e.g. pointers [2], procedure calls [15,17,3], recursive procedures [13], and gotos [5].

One of the newer and quite successful approaches to formal software verification is the Ada subset called SPARK [4]. 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 behaviour and partly to ensure simplicity of formal language definition and proof arguments.

Mandatory annotations are required to perform extended static code analysis and comprise e.g. 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, e.g. pre-, postconditions, loop invariants, and intermediate assertions for procedures.

Since SPARK has a formally defined semantics, formal program verification is possible and supported by the SPARK Examiner. This tool checks the conformance of a program to the rules of SPARK, carries out a flow and information analysis of the code, and supports formal verification.

SPARK mostly aims at low level properties, e.g. the absence of run-time errors [11], whereas COMPASS is suitable for reasoning about the high level algorithmic contents of a program.

4.2 Dynamic Checking

Several approaches exploit code annotations to improve dynamic testing, e.g. Robust C [9], APP [18], Anna [16], and C-Patrol [22]. 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 [20], 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 e.g. 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 our 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.

4.3 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 [7] Fagan argues that design and code inspections increase the productivity and improve the final program quality. Ten years later, in 1986 [8], Fagan suggests slight modifications to the inspection process and reports further industrial experiences that support his earlier results. In [19] Russell describes similar experiences with the inspection in ultralarge-scale developments.

One possible method that describes more precisely what actually has to be done in an inspection session was introduced by van Emden [21] in 1992. His code inspection method is based on Floyd's method for the verification of flowcharts [10]. 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 [6] method. However, van Emden's method does not produce code according to structured programming.

The major difference between van Emden's and our approach is, that we combine a *formal syntax* and 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 we use Dijkstra's development method, the code produced is in accordance with structured programming.

5 Conclusions and Future Work

The COMPASS method introduced in this paper is based on Hoares method for proving programs correct, Dijkstras discipline for program development, and Fagans work on code inspection. Both Hoares verification method and Dijkstras 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.

Our hypothesis is that a method which is more easily used in practice, and which remains to be partially mechanizable, may be achieved by relaxing the requirements on formal rigour 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 we combine a formal syntax with an informal semantics, it is still possible to automatically derive verification conditions.

The verification conditions constitute questions to be asked during code inspection. The somewhat loosely defined contents of the steps of Fagans code inspection method are thus filled with a very concrete contents. Moreover, COMPASS not only allows isolating faults in the code; The inspected code may be considered correct with respect to its assertions, if no errors are found.

Our short-term goals are to further refine the COMPASS method in cooperation with our industrial partner ABB ISY. We plan to complete the implementation of the tool support, and to evaluate the COMPASS method in a real software development project. For the latter it is necessary to lift certain restrictions presently imposed on the C-language, and to develop tutorials for the use of the method.

In the long term we intend to study whether our general idea - to decrease the requirements on formality, while still keeping a sufficient level of rigour - may be applied also to higher level specifications. We believe this may be a way to increase industrial acceptance of formally based development methods.

Acknowledgments

This paper reports on initial results of the project "Verification Automation in Software Development". The project is carried out in cooperation between ABB Industrial Systems and Linköping University. It is part of the competence center ISIS (Information Systems for Industrial Control and Supervision), financially supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

We are most grateful to Ulf Hammar and Stefan Frennemo at ABB Industrial Systems for remarks that have led to several improvements of the COMPASS approach. We would also like to thank Dr. Feliks Kluzniak for most useful comments on a draft version of this paper.

References

1. 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.
2. A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–205, September 1989.
3. A. Bijlsma. Calculating with procedure calls. *Information Processing Letters*, 46(5):211–217, July 1993.
4. Bernard Carré and Jonathan Garnsworthy. SPARK - an annotated Ada subset for safety-critical programming. Presented at TRI-Ada, 1990.
5. Arie de Bruin. Goto statements: Semantics and deductive systems. *Acta Informatica*, 15:385–424, 1981.
6. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
7. Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(1):182–211, 1976.
8. Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.
9. 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.
10. 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.
11. 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.
12. David Gries. *The Science of Programming*. Springer-Verlag, 1981.
13. Wim H. Hesselink. Proof rules for recursive procedures. *Formal Aspects of Computing*, 5:554–570, 1993.
14. C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12(10):576–80, 583, October 1969.
15. 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.
16. 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.
17. Alain J. Martin. A general proof rule for procedures in predicate transformer semantics. *Acta Informatica*, 20:301–313, 1983.
18. David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
19. Glen W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, 1991.
20. Sun Microsystems Inc. and Information-technology Promotion Agency. *ADL Translator User's Guide: Getting Started with ADLT*, December 1995.
21. Maarten H. van Emden. Structured inspection of code. *Software Testing, Verification and Reliability*, 2:133–153, 1992.
22. 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.