Structured Inspections of Code*

M.H. van Emden†

Abstract

Cleanroom programming and code inspections independently provide evidence that it is more efficient to postpone the testing of code to a later stage than is usually done. This paper argues that an additional gain in quality and efficiency of development can be obtained by *structuring* inspections by means of an inspection *protocol*. The written part of such a protocol is prepared by the programmer before the inspection. It is modelled on Floyd's method for the verification of flowcharts. However, the protocol differs from Floyd's method in being applicable in practice. Structured inspections gain this advantage by not attempting to be a *proof*; they are no more than an articulation of existing forms of inspection.

With the usual method of structured programming it may be difficult to prepare the inspection protocol. On the other hand, Assertion-Driven Programming (of which an example is included in this paper) not only facilitates protocol preparation, but also the coding itself.

1 Introduction

Recent papers on code inspections [6, 25, 4] report improvement in both quality and productivity. Such a success leaves the reader with the burning question: What does one actually *do* at a code inspection? This paper describes one possible method. It is based on a method of program verification. As a result, we need to take immediate steps to avoid misunderstanding:

1. The method to be described *is not itself* verification, but *is derived from* a verification method.

2. The method to be described can equally well be based on any one of several verification methods. In fact, Dyer and Kouchakdijan [4], who, as far as we know, were the first to base an inspection method on verification, used the one of Linger, Mills, and Witt

*First edition, August 1991. Second edition, February 1993. To appear in *Software Testing, Verification, and Reliability*

†Computer Science Department, University of Victoria, P.O. Box 3055, Victoria, B.C., Canada V8W 3P6. E-mail: vanemden@csr.uvic.ca

[17]. Other possibilities are the methods of Hoare [12] or Dijkstra [3]. In this paper we use the method of Floyd [7], as it can be described very briefly. In this way we have been able to make the paper self-contained in this respect. Floyd's method applies to programs in the form of flowcharts.

Before explaining the technique it is necessary to review the software engineering methods that provide the necessary framework for inspections.

The software development process needs to be subdivided into stages. A difficulty is in finding an *effective* way to do so. One such way is described by Parnas and Clements [23]. Each of their stages corresponds to a *work product*. Other workers have elaborated on this work. For example, Hoffman [13] uses the following work products:

1. requirements specification
2. module guide
3. module interface specification
4. module implementation
5. test plan
6. test implementation

The work products are characterized by explicit criteria. Of course, such characterization is incomplete even in the best-run projects. We do not know how to determine just by looking at, for example, a module guide, whether the design embodied in it will not get us into trouble later. But many difficulties can be prevented by studying, discussing, and reviewing it carefully. Similarly, we do not yet know how to determine, just by looking at a module implementation, whether it satisfies its interface specification. The value of the approach of Parnas and Clements lies in their claim that it pays to apply explicit and well-defined criteria to each product so as to prevent, as much as possible, problems that could arise further down the line. Their philosophy is to invest up front so as to be amply repaid by better control over quality and cost.

Checking exit criteria by inspections. Fagan [6] pioneered the idea of testing for the exit criterion of any work product, not just code, by an *inspection*. An inspection according to Fagan is a formal procedure conducted by a team typically consisting of four programmers in various well-defined roles. The procedure differs from walk-throughs and similar reviews in that an inspection checks for criteria defined beforehand. The sole purpose of an inspection is to find errors. When one has been found, it is recorded and the inspection proceeds in search of another error. An inspection does not include an attempt to correct any error, nor any discussion of how it could have been prevented. One way to characterize the difference between an inspection and other meetings devoted to the examination of a work product is that *an inspection does not involve problem-solving.*

1.1 Advantages and limitations of code inspections

Code inspections according to Fagan and to Russell. Code inspections, as proposed by Fagan [6], and further reported on by Russell [25], consist of "paraphrasing" and checking for errors known to be likely from past experience. Both Fagan and Russell report data supporting the conclusion that code inspections are a cost-effective way of improving software quality.

As reported by Russell, code inspections are meetings of experts in the type of software being inspected at which one, the *Narrator*, who is not the author of the code, *paraphrases* the code. The other participants check whether the paraphrasing is accurate and whether it is a sufficient basis for confidence in the correctness of the code.

Clearly, the paraphrasing is the crucial activity from which the benefits of code inspections arise. The paper [25] gives no information about paraphrasing other than that the highest number of errors per hour of inspection was found at the slowest inspection rate reported: 150 lines per hour. An unpublished document from Russell's laboratory describes paraphrasing as follows:

> Finally, we get to the heart of a code inspection, which is the PARAPHRASING of the code, line by line, to reveal the function being implemented. This is THE crucial skill needed to do an effective inspection! Paraphrasing means expressing in English the meaning of one or more lines of code. The person doing the paraphrasing must continuously decide how many lines to express at once, but this number is generally never higher than two or three lines at a time, and can often be just part of a line, depending on code complexity. This is a very demanding mental exercise, and explains why code inspections should not be done for more than two hours at a time, and the maximum inspection rate should not exceed 100 to 150 lines an hour. At first glance this seems ridiculously slow, especially when compared to the execution rate when testing. However, no amount of real-time testing can provide the kind of code quality achieved by a top-notch inspection team executing code at 100 lines per hour.

It is surprising and encouraging that something as loosely defined as paraphrasing can be as effective in reducing the number of errors as Fagan and Russell have reported. Dyer and Kouchakdijan [4] have gone a step further and introduced a technique of program verification in both the writing of the code and its inspection. The reported result is not only improved code quality, but also increased productivity. The favourable results reported here depend on a suitable programming method:

> Designs that can be easily verified are favoured and ones that are difficult to verify should be considered suspect and redone for simplicity.

The approach of Dyer and Kouchakdijan to verification is based on Linger, Mills, and Witt [17]. Although this latter book contains much useful material, it does not build on the valuable techniques of the tradition represented by Floyd, Dijkstra, Hoare, Wirth, and

Gries. Apart from many papers, much of this material can be found in [3] and [9]. Floyd's method differs from those of Hoare and Dijkstra in being simpler to explain. But it is conceptually more closely related to these than to the approach of Linger, Mills, and Witt. It is the primary purpose of this paper to show how code inspections can profit from the work of Floyd, Dijkstra, Hoare, Wirth, and Gries.

The discovery reported by Dyer and Kouchakdijan [4] may have been made so late because of fundamental misunderstandings concerning the role of proofs in mathematics and in programming. Although [22] contains some clarifications, more is needed for a wider understanding of the issues. The secondary purpose of this paper is to contribute to such an understanding.

A problematical example. As emphasised by Dyer and Kouchakdijan, it is crucial for successful inspection that the author of the code follows a coding method that matches the method of the inspection. Hence this paper is as much about coding method as about inspection method. As an introduction, we present an example[1] of code that was produced according to a method that does not match the inspection method to be presented in this paper. Indeed we doubt whether the function is easy enough to be paraphrased to make the usual inspection method effective. Later, we will demonstrate a suitable coding method by rewriting the function. The example occurs in the standard C library of Berkeley Unix.

```
/*
 * Compare strings (at most n bytes):  s1>s2: >0  s1==s2: 0  s1<s2: <0
 */
strncmp(s1, s2, n) register char *s1, *s2; register n; {
   while (--n >= 0 && *s1 == *s2++)
      if (*s1++ == '\0') return(0);
   return(n<0 ? 0 : *s1 - *--s2);
}
```

1.2 Suggestions for improvement in code inspections

Inspection protocols. Inspections differ from other kinds of reviews in requiring less problem-solving, in being more routine, and in having a more predictable rate of progress. For inspections one can plan to proceed at 150 lines an hour. In other kinds of reviews the pace can vary so wildly that it does not make sense to plan on a certain number of lines per hour. To "inspect" suggests a certain basic and simple action: one checks whether something is the case or not. It is not trouble-shooting. In mathematics a similar usage prevails: when it is claimed that an assertion is verifiable "by inspection," this means that one can check it by merely looking at it; there should be no need to resort to pencil and paper, nor should one need to think about it for any length of time.

[1] In spite of the fact that software engineering is concerned with large amounts of code, there *should* be no need to apologize or otherwise draw attention to the small size of the example. What matters is that the method presented here is relevant and that the example is large enough to demonstrate the method.

The bodies of most functions are too complex to be validated "by inspection." The above example is, in spite of its brevity, no exception. Hence the code of a function should be decomposed into parts simple enough to be checked by inspection.

As acknowledged by Russell in the above quote, the paraphrasing of the code is a demanding exercise. Thus it seems reasonable to assume that the text of the paraphrase is recorded as a *comment in the code*. This can be done by the programmer before the inspection. Then these comments can serve as the atomic check points to be addressed at the inspection meeting.

This suggests that we regard the inspection meeting as the execution of a *protocol*, which consists of the code with prerecorded paraphrasing in the form of comments. The protocol says what is to be done at an inspection meeting. Thus the meaning of "protocol" here is not the one most common in computer science (according to the Webster dictionary, "a set of conventions governing the treatment and especially the formatting of data in an electronic communications system"), but a meaning that is used in science more generally: *"the plan of a scientific experiment or treatment"* (again according to Webster). For inspections as described by Fagan and by Russell, the protocol may consist only of checklists of errors to watch out for. But more probably it will also include paraphrases prerecorded as comments in the code. The inspection meeting then determines whether the comments are adequate as a basis for approval of the function as a whole and whether the code lives up to the comments.

The problem with some code (such as the example above) is that adequate comments are hard to find. *Such code should have been written from scratch with the criterion of inspectability in mind.* Most programmers find this difficult to do for the type of code most in need of inspection. The purpose of this paper is to present a method for structuring comments and code in such a way that they are an effective inspection protocol.

Structuring code for inspectability. Ideally, the exit criterion for the code of a module implementation is correctness, that is, conformance of the module's behaviour to its interface specification. The only kind of inspection that can *guarantee* this is a formal correctness proof. Much research, over a considerable period of time, has been devoted to formal verification of programs. Although there has been progress, this research has so far not affected the practice of software development.

The existence of a formal proof, which provides a guarantee for the absence of errors, we call a *strong* exit criterion for the work product in question. A code inspection depending on checklists and paraphrasing is a drastically weakened criterion. Yet the latter has contributed in practice much more to quality of code than the former. From this point of view it is tempting to investigate whether less drastically weakened versions of program proofs may yield even greater benefits in practice than inspections have already done.

There is a vast gap between the weak criteria on which code inspections are based and the strong criterion of formal verification. In the first, one concludes at best that no errors have been found. In the second, one concludes that none exist. At the moment, software development practice leaves this gap empty. In this paper we will show that a graduated spectrum exists between the weak and the strong criteria. The spectrum contains inspection protocols of practical value.

This graduated spectrum consists of correctness arguments that are informal and incomplete to various degrees. In theory, any lapse from perfect formality and complete coverage of all assumptions is enough to render an attempt at a correctness proof completely worthless. In practice, perfect formality and complete coverage are impossible. But in practice the existence of an informal and incomplete version of a valid correctness proof is very effective in decreasing the number of errors. In conjunction with testing, it leads to software that is as trustworthy as any.

In practice, informal and incomplete versions of a valid correctness proof can provide the structure that makes code easy to inspect. It provides us with the needed protocols for *structured* inspections.

2 An example of an inspection protocol

A specification of strncmp(). An effective inspection should verify that the code causes the intended function to be computed. Thus it is necessary first to specify what the intended function is.

In this section we present an inspection protocol for a function. As example we have chosen the same string comparison task addressed by the function shown in the previous section. Let us first review the convention for strings in the C programming language. Normally, strings are terminated by the null character `'\0'`. Thus a program comparing strings should handle such a terminator. If the program receives as parameter a pointer to a sequence that is not terminated by a null character, then it should terminate in a way determined by an additional parameter indicating how many characters should be examined at most.

Thus with every string there is associated a function `e` defined as follows:

> If a null character occurs in the string `s`, then `e(s)` is such that `s[e(s)] == '\0'` and `'\0'` does not occur in `s[0..e(s)-1]`. If no null character occurs in `s`, then `e(s)` is positive infinity[2].

An expression of the form `a[m..n]` denotes the set `{a[i] | m <= i <= n}`, which is, of course, empty if `m > n`. Such expressions are used in various combinations. For example, `a[m..n] == b[m..n]` means that for every integer `i` such that `m <= i <= n`, it is true that `a[i] == b[i]`.

The function to be implemented by the string comparison program is specified informally as follows:
`strncmp(s1,s2,n)` is

- negative if `pre(s1,n)` precedes `pre(s2,n)` in lexicographic order,

[2]If this term is unclear, then read instead "least transfinite ordinal." Of course, no program variable takes any result of `e` as its value. The function `e` only plays a role in assertions about the computation state.

- positive if `pre(s2,n)` precedes `pre(s1,n)` in lexicographic order, and
- zero otherwise;

where `pre` and "lexicographic order" still need to be defined.

The function `pre(s,n)` is the result of truncating the string `s` at the integer `n`, in the following way:

```
pre(s,n) == s[0..n-1] if s[0..n-1] does not contain '\0'
pre(s,n) == s[0..e(s)-1] otherwise
```

By using `pre` in the specification of `strncomp`, we can restrict the definition of lexicographic order to character sequences not containing `'\0'`, as below.

1. If string `s1` is a prefix of string `s2` and `s1 != s2`, then `s1` precedes `s2` in lexicographic order.
2. If string `s2` is a prefix of string `s1` and `s1 != s2`, then `s2` precedes `s1` in lexicographic order.
3. If `s1` and `s2` are such that neither is a prefix of the other, then there is a largest `i` such that `s1[0..i-1] == s2[0..i-1]`. The lexicographic order is then determined by the order between `s1[i]` and `s2[i]`.

It follows that `s1` and `s2` are equal if neither lexicographically precedes the other.

An inspection protocol for strncmp(). A first requirement for a successful inspection of `strncmp()` is an encoding of the algorithm in a suitable style. The listing below contains such an encoding, together with the remainder of the inspection protocol to be explained later in this section.

This encoding is of course only one of many possible. We ask readers disturbed by the appearance of a `goto` statement to withhold judgment until they have studied section 4, where we explain how this code has been obtained in a systematic fashion from the function's specification.

```
int strncmp(s1,s2,n) char s1[], s2[]; int n; {
    int ndx;
    /**A0: e1 == e(s1) && e2 == e(s2)**/
    ndx = 0;
A3: /**s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    if (ndx >= n)
       /**A4: s1[0..n-1] == s2[0..n-1] hence pre(s1,n) == pre(s2,n)**/
       return(0);
    /**A2: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2 &&
       ndx < n
```

```
    **/
    if (s1[ndx] != s2[ndx] || s1[ndx] == '\0' || s2[ndx] == '\0') {
       /*ndx large enough for return value to be determined right away*/
       /**A1: Let rslt == s1[ndx] - s2[ndx]:
          if rslt < 0
          then pre(s1,n) precedes pre(s2,n) in lexicographic order and
          if rslt > 0
          then pre(s2,n) precedes pre(s1,n) in lexicographic order and
          if rslt == 0
          then pre(s1,n) == pre(s2,n)
       **/
       return(s1[ndx] - s2[ndx]);
    }
    /**A5: s1[0..ndx] == s2[0..ndx] && ndx < e1 && ndx < e2**/
    ndx++;
    /**A6: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    goto A3;
    /***T: ndx has increased since goto A3 was last passed and n has not
        increased. ndx cannot increase beyond n without the function
        terminating.
    ***/
}
```

We propose to replace the paraphrasing used by Fagan and by Russell by a more structured alternative that we call an inspection *protocol*. It is more structured because it is partly written and consists of a number of items each of which is determined by the structure of the code and by the level of detail desired in the inspection. In addition to the written part of the protocol there is, for each inspection item, *ad lib* narrative that is spoken during the inspection meeting to explain the written protocol.

The written part of the protocol consists of commented, compilable statements. This is the part of the source code used to generate the executable. It is a work product in the sense of Parnas and Clements [23] and takes the place of their module implementation. Certain comments, delimited by `/**` and `**/`, or by `/***` and `***/`, are the additions to the code that make the text into the written part of the protocol. Comments in another form are not constrained to play a specific role.

Code inspection attempts to verify correctness. As in formal program verification, We distinguish two aspects: partial correctness and termination. Partial correctness with respect to an assertion $\mathcal{A}$ ensures that $\mathcal{A}$ holds true of the state of the computation when and if the function exits. By the state of the computation we mean the values of the variables manipulated by the function. A partially correct program may not exit; hence the need to inspect for termination as well.

Usually it is not possible to establish by inspection that the desired assertion $\mathcal{A}$ holds on exit without auxiliary assertions placed at strategic locations throughout the code. If these placements are done in a suitable way, it can be established by inspection that, when execution reaches any assertion, its truth is guaranteed.

Each placement of assertions in the code implies a well-defined number of inspection items. Each of these assumes that along the execution path followed so far all assertions hold true. The inspection item is then to verify that the extension of the execution path to the next assertion ensures that the latter holds true also. Because the inspection items are determined by the placement of the assertions, the items do not have to be listed explicitly. However, as this is an introduction to structured inspections, we do include the list of inspection items for the `strncmp()` example:

```
Inspection Items for Partial Correctness:
     Item A: A0 to A3
     Item B: A3 to A4
     Item C: A3 to A2
     Item D: A2 to A1
     Item E: A2 to A5
     Item F: A5 to A6
     Item G: A6 to A3

Inspection Items for Termination:
     Item T
```

This is a complete list, as every execution path can be decomposed into segments taken from this list.

So far we were concerned with the written part of the protocol. As for the narrated part of the protocol, we will only show a few examples here. The assertions are written in prose fortified by symbols.

Here are examples of what the Narrator may say at the inspection meeting.

- Concerning Inspection Item B, from `A3` to `A4`: As only a test intervenes between `A3` and `A4`, `A3` holds at `A4` as well as `ndx >= n`. Then at `A4` we have that `s1[0..n-1]` is a sub-sequence of `s1[0..ndx-1]` and similarly for `s2`. Hence, the first `n` characters are equal. By `A3`, these do not include a null character. Hence the function should return `0`.

- Concerning Inspection Item D, from `A2` to `A1`: As only a test is passed, everything that is true at `A2` is still true at `A1`. In particular, up to `ndx` the two strings are equal. At `A1` we know in addition that at `ndx` the elements are unequal or one is the terminating null character. Hence lexicographic order between `s1` and `s2` is determined by the order between the elements at `ndx`. Taking into account that the null character is the integer `0`, we see that the order encoded in the function value is obtained by subtraction of the elements at `ndx`.

- Concerning Inspection Item F, from `A5` to `A6`: Because of the intervening `ndx++`, every occurrence of `ndx` in `A5` should be replaced by `ndx-1` to obtain `A6`. This is indeed the case.

Goto statements are simplest to inspect. As such a statement is executed without change of state and without passing a test, the assertion before must simply imply the one after. Because of this property of the `goto` statement, the code is just as easy to inspect when the assertion before the `goto` statement is omitted. Hence the narrative:

- Concerning Inspection Item G, from `A6` to `A3`: As the intervening `goto` statement does not change the state, it is correct for `A6` and `A3` to be the same assertion.

3 An exit criterion for inspectable code

As explained above, we follow Parnas and Clements in subdividing software development in stages defined by work products, where each work product must satisfy well-defined exit criteria. Here we are concerned with structured inspection as a suitable exit criterion for module implementations, that is, for code. The previous section showed a structured inspection protocol for the example program. The protocol is based on Floyd's method of inductive assertions [7]. In this section we explain the method in general so that we have an exit criterion for code.

Floyd's method applies most directly to flowcharts. The correspondence between code and flowcharts is so close that we regard C programs as a textual representation of flowcharts, thus avoiding the need for graphics. But conceptually, we regard a flowchart as a graph where each node is a statement or a test. There is a directed edge between nodes wherever there is a direct transfer of control between the corresponding statements or tests. In addition to these, the flowchart has one node that does not correspond to any program text: this is the *entry node*, which has no incoming edge and a directed edge out to the first statement or test of the program text. Without loss of generality it can be assumed that the program is the body of a function. Then the return statements correspond to nodes in the flowchart without any edge going out of them. These we call *exit nodes*.

To some edges we associate assertions. These are logical formulas that are true or false for the state of the computation. This state can be interpreted as a vector of the values of variables accessible to the code under consideration.

Suppose two edges have associated assertions A_1 and A_2 such that there is one execution path from the first to the second. This implies, for example, that there is no loop in any execution path connecting these edges. The *verification condition* associated with these edges is then the statement that, if A_1 holds when control is at its edge, and the execution path is traversed to the other edge, then A_2 holds. The verification condition is valid if this statement is true.

A flowchart is *covered by* verification conditions if each execution path of the flowchart is a chain of verification conditions.

A useful property of a flowchart is that, if on entry the assertion associated with the entry edge is true, and if an exit edge is reached, then the assertion associated with that exit edge is true. Note that this property is stated in such a way that it is not guaranteed that an exit edge is reached. That is why the property is called *partial* correctness.

Floyd's method of inductive assertions is to prove partial correctness of a flowchart by proving the validity of a set of verification conditions that covers it. Manna has justified the method in a rigorous fashion [20].

Termination. A partially correct flow chart may not terminate. In Floyd's method, termination is argued independently of partial correctness. Basically, for each loop one shows that it can only be traversed a finite number of times. That leaves the problem of a simple criterion for recognizing loops at inspection time.

Loop templates (using the constructs `for`, `while`, `do ... while`) are of course recognized immediately. The other loops can be detected by a *goto* statement going backwards in the text. For each of these we verify whether it can only be executed a finite number of times. That excludes the possibility of nontermination due to loops formed by goto statements.

Floyd's method and structured inspections. As was seen in the previous section, a structured inspection is in principle Floyd's method of inductive assertions. Instead of assertions, there are comments delimited by `/**` and `**/` consisting of English, possibly supplemented by logical or mathematical formulas. Instead of verification conditions, there are inspection items. The only difference between these is due to the different nature of the assertions. Because of the informal nature of the assertions, we cannot say of the inspection items that they are valid in the same sense as the verification conditions. Also, for a verification condition to be valid, the first assertion should state all the assumptions. This is often not the case in an inspection item.

4 Assertion-Driven Programming

We have shown how inspections can be structured in such a way that an inspection becomes an execution of a protocol rather than an improvised paraphrase. To achieve this, the code has to be written according to a certain method.

Floyd's method allows one to prove a program correct, provided that one has a suitable set of assertions for the program. If such assertions don't come with the program, then, in all but the most trivial cases, it is difficult to find them. Such a task is similar to that of decoding an encrypted message when the key is not available: possible, but a lot of work.

As Dijkstra [2] pointed out, the difficulty is due to attempting to find assertions *afterwards*, without access to the programmer's thoughts. There *is* access to the programmer's thoughts when one *is* the programmer and one is about the write the code. *That* is the time to record the assertions. Structured inspections are perfectly practical, provided that a suitable method is followed. One such method is to compose the required assertions *before* the accompanying code. We call this *Assertion-Driven Programming*. It results in code structured for inspectability. In our experience, it does not make programming more difficult. On the contrary, it is a way to help discover the program, to eliminate "writer's block" for the programmer.

Floyd's method can be described concisely and in perfect generality. We have attempted this in the previous section. We do not see how to describe Assertion-Driven Programming concisely in the same general way. In this paper, we will therefore restrict ourselves to showing how to derive the inspectable version of `strncmp()` according to the method of Assertion-Driven Programming.

Development of an inspectable version of strncmp(). Here are some decisions about the form of `strncmp()`. The content comes later.

```
int strncmp(s1,s2,n) char s1[], s2[]; int n; {
    int ndx,rslt;
    /**A0: e1 == e(s1) && e2 == e(s2)**/
    ...(1)
A1: /**if rslt < 0
       then pre(s1,n) precedes pre(s2,n) in lexicographic order and
       if rslt > 0
       then pre(s2,n) precedes pre(s1,n) in lexicographic order and
       if rslt == 0
       then pre(s1,n) equals pre(s2,n)
    **/ return(rslt);
}
```

Although it does not amount to much, part of the function's code is in final form. To achieve this, we cheated: we have already declared the variable `ndx`, although the need for it only becomes apparent in the next step. The task remains to find the code to be substituted for `...(1)` is to make assertion `A1` true.

The final assertion `A1` looks too hard to achieve by one or a very small number of atomic statements. This suggests we need at least one other assertion; one that is less ambitious than `A1`. To find such an assertion, observe that the definition of lexicographic order implies that, if we have an index `ndx` such that

```
A2: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2 && ndx < n
```

then the order of `s1` and `s2` is the order between `s1[ndx..e1-1]` and `s2[ndx..e2-1]`. But of course, if `A2` holds and `ndx >= n`, then we have `pre(s1,n) == pre(s2,n)` and `strncmp()` should exit immediately, returning `0`.

Thus the assertion `A2` serves as a "bridge" between `A0` and `A1`. If `ndx` is sufficiently small, we are at one end of the bridge and `A2` is true independently of `s1` and `s2`. If `ndx` is sufficiently large, we are at the other end of the bridge and the value of the result `rslt` can be determined right away. Thus we start off making `A2` true by assigning a sufficiently small value to `ndx` and then, maintaining truth of `A2`, we try to make `ndx` large enough so that the lexicographic order can be determined in a single step.

Hence `...(1)` should be replaced by

```
...(2)
/**A2: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2 &&
       ndx < n
**/
...(3)
```

If we make `ndx` equal to `0`, then `A2` holds no matter what the strings `s1` and `s2` are. So it is useful to include `ndx = 0` in the code that is to replace `...(2)`. At that point we also have to take care of the possibility that the comparison is already complete because of the value of `n`. Hence we replace `...(2)` by:

```
    ndx = 0;
A3: /**s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    if (ndx >= n) {
       /**A4: s1[0..n-1] == s2[0..n-1] hence pre(s1,n) == pre(s2,n)**/
       rslt = 0; goto A1;
    }
```

In `...(3)` we need to find out whether `ndx` is large enough for the lexicographic order to be determined only on the basis of `s1[ndx]` and `s2[ndx]`. If this is the case, then the value to be returned can be determined by a few atomic statements. Otherwise, `ndx` has to be incremented and then the situation `A3` holds again. Thus `...(3)` is replaced by

```
if (s1[ndx] != s2[ndx] || s1[ndx] == '\0' || s2[ndx] == '\0' {
   /*ndx large enough for return value to be determined right away*/
   rslt = s1[ndx] - s2[ndx];
   /**if rslt < 0
      then pre(s1,n) precedes pre(s2,n) in lexicographic order and
      if rslt > 0
      then pre(s2,n) precedes pre(s1,n) in lexicographic order and
      if rslt == 0
      then pre(s1,n) equals pre(s2,n)
   **/
   goto A1;
}
/**A5: s1[0..ndx] == s2[0..ndx] && ndx < e1 && ndx < e2**/
ndx++;
/**A6: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
goto A3;
```

So far we explained how to get an inspectable *end* product. There is no need to go through the above development at an inspection meeting. These are the considerations going through the author's head when writing the code.

Putting the various pieces together we get:

```
int strncmp(s1,s2,n) char *s1,*s2; int n; {
    int ndx,rslt;
    /**A0: e1 == e(s1) && e2 == e(s2)**/
    ndx = 0;
A3: /**s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    if (ndx >= n) {
       /**A4: s1[0..n-1] == s2[0..n-1] hence pre(s1,n) == pre(s2,n)**/
       rslt = 0; goto A1;
    }
    /**A2: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2 &&
   ndx < n
    **/
    if (s1[ndx] != s2[ndx] || s1[ndx] == '\0' || s2[ndx] == '\0') {
       /*ndx large enough for return value to be determined right away*/
       rslt = s1[ndx] - s2[ndx]; goto A1;
    }
    /**A5: s1[0..ndx] == s2[0..ndx] && ndx < e1 && ndx < e2**/
    ndx++;
    /**A6: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    goto A3;
A1: /**if rslt < 0
       then pre(s1,n) precedes pre(s2,n) in lexicographic order and
       if rslt > 0
       then pre(s2,n) precedes pre(s1,n) in lexicographic order and
       if rslt == 0
       then pre(s1,n) equals pre(s2,n)
    **/ return(rslt);
}
```

The use of the variable `rslt` is an artifact of the development method. We would therefore submit as work product for inspection the following version, where we use the `return` statement instead.

```
int strncmp(s1,s2,n) char *s1,*s2; int n; {
    int ndx;
    /**A0: e1 == e(s1) && e2 == e(s2)**/
    ndx = 0;
A3: /**s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    if (ndx >= n) {
       /**A4: s1[0..n-1] == s2[0..n-1] hence pre(s1,n) == pre(s2,n)**/
       return(0);
    }
    /**A2: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2 &&
   ndx < n
```

```
    **/
    if (s1[ndx] != s2[ndx] || s1[ndx] == '\0' || s2[ndx] == '\0') {
       /*ndx large enough for return value to be determined right away*/
       /**A1: Let rslt == s1[ndx] - s2[ndx]:
          if rslt < 0
          then pre(s1,n) precedes pre(s2,n) in lexicographic order and
          if rslt > 0
          then pre(s2,n) precedes pre(s1,n) in lexicographic order and
          if rslt == 0
          then pre(s1,n) equals pre(s2,n)
       **/
       return(s1[ndx] - s2[ndx]);
    }
    /**A5: s1[0..ndx] == s2[0..ndx] && ndx < e1 && ndx < e2**/
    ndx++;
    /**A6: s1[0..ndx-1] == s2[0..ndx-1] && ndx <= e1 && ndx <= e2**/
    goto A3;
    /***T: ndx has increased since goto A3 was last passed and n has not
        increased. ndx cannot increase beyond n without the function
        terminating.
    ***/
}
```

Pointer arithmetic for strncmp(). We have used array notation for `strncmp()` because that seemed to make the assertions easier. If pointers are preferred, then a version using these can be obtained by translation. If this route is followed, then the translation has to be included in the inspection.

To continue the example, let us translate to a version using pointers instead of array indices. After omitting the assertions we are left with the following code:

```
int strncmp(s1,s2,n) char s1[], s2[]; int n; {
    int ndx;
    ndx = 0;
A3: if (ndx >= n) return(0);
    if (s1[ndx] != s2[ndx] || s1[ndx] == '\0' || s2[ndx] == '\0')
       return(s1[ndx] - s2[ndx]);
    ndx++; goto A3;
}
```

Using the convention of the C programming language that the array name is also a pointer to the array element with index `0`, we can change `s1[ndx]` to `*s1` (similarly for `s2`) provided that we accompany the statement `ndx++` with increments applied to the array names `s1` and `s2`. For a reason that will be clear shortly, we change the test `ndx >= n` to the equivalent `n - ndx <= 0`. This gives:

```
int strncmp(s1,s2,n) char *s1,*s2; int n; {
    int ndx;
    ndx = 0;
A3: if (n - ndx <= 0) return(0);
    if (*s1 != *s2 || *s1 == '\0' || *s2 == '\0') return(*s1 - *s2);
    ndx++; *s1++; *s2++;
    goto A3;
}
```

Now `ndx` is clearly superfluous: we can count just as well by decrementing `n`. Hence:

```
int strncmp(s1,s2,n) char *s1,*s2; int n; {
A3: if (n-- <= 0) return(0);
    if (*s1 != *s2 || *s1 == '\0' || *s2 == '\0') return(*s1 - *s2);
    *s1++; *s2++;
    goto A3;
}
```

5 Related work

As explained in the introduction to this paper, the first to combine inspection with verification were Dyer and Kouchakdijan [4]; this paper is to be regarded as a companion to cover another approach to verification and correctness-oriented programming.

We have taken Floyd's work [7] on the verification of flowcharts as starting point. The inspection protocols differ in being informal and incomplete in the statement of assumptions and therefore do not qualify as proofs. In making this change we have reversed the historical development up to Floyd's work: Goldstine and Von Neumann [8] already used assertions for checking out flowcharts, without, however, the ambition of providing *proofs* of correctness.

Hoare [12] is well-known for a verification method similar to Floyd's. A difference is that Hoare's is intended for structured programs rather than for flowcharts. As we find Floyd's method more easily adaptable to Assertion-Driven Programming, we have preferred it.

The main idea of Assertion-Driven Programming is formulated in Dijkstra [2], from which we quote:

> When concern for correctness comes as an afterthought, so that correctness proofs have to be given once the program is already completed, the programmer can indeed expect severe troubles. If, however, he adheres to the discipline of producing correctness proofs as he writes his program, he will produce program and proof with less effort than just the programming alone would have taken.

Dijkstra preferred to apply this principle in an unimplemented programming language [3], using weakest preconditions as framework for reasoning. We applied it in a conventional language using Floyd's method as framework for reasoning [26]. At the time we called it "programming with verification conditions"; now we prefer "Assertion-Driven Programming" because it describes the method better.

An important component of Assertion-Driven Programming is "stepwise refinement." The name originates in Wirth [27], although the most compelling exposition of the principle occurs in the earlier paper by Dijkstra [1]. Formalizations of the method are given by Hehner [11] and by Knuth in his "literate programming" [16].

A quarter of a century of research in formal program verification has not had an impact in practical software development. The best explanation that we know of is given by DeMillo, Lipton, and Perlis [22]. They observe that the practical purpose of a proof is to give confidence in correctness. Because of this they cast doubt on the use of formality to the degree of being machine-checkable. This leads to the larger question of why this degree of formality is considered so important. The issue has been analyzed by McCloskey [21], who identifies emphasis on formality in mathematics with analogous trends in other sciences and uses the term "modernism" for the larger phenomenon. McCloskey advocates a balance of modernism with "rhetoric." An example of rhetoric in the sense of McCloskey are the social processes described by DeMillo, Lipton, and Perlis [22]. Structured inspections are another example.

6 Discussion

Structured Inspections, Assertion-Driven Programming and Structured Programming. The reader may have noticed that a program that is written to be inspectable by a structured inspection is not necessarily a "structured program." One of the hallmarks of structured programming is the use of loop templates, using language constructs such as `for`, `while`, or `do...while`. Although such loops can be verified by means of assertions, We find that in Assertion-Driven Programming it often happens that loops "grow naturally." By this we mean that in writing the code an assertion is reached that holds true elsewhere in the text. It is then easiest, both from a program-construction point of view as well as from a verification point of view, to write a `goto` statement to the other assertion. If that assertion occurs earlier, then a loop has been created. An example of this occurred in the derivation of the `strncmp()` program.

Of course there are situations where one remembers a complete solution to a subproblem that occurs in writing a program and that this solution is remembered in terms of a loop template. For example, although we are inclined toward Assertion-Driven Programming, we use a loop template when we need to print the elements of an array.

The present proposal for structured inspections is based on Floyd's verification method via Assertion-Driven Programming. Structured inspections can also be based on other verification methods. Dyer and Kouchakdijan [4] have used Mills's method [17] as basis. Other possibilities are the methods of Hoare [12] or Dijkstra [3]. Assertion-Driven Programming shares the advantage of the methods of Hoare and Dijkstra of helping to discover the algorithm in addition to providing a verification. But Floyd's method has a simpler theory than the ones of Hoare and of Dijkstra. Floyd's assertions have the additional advantage (pointed out by Halpern [10]) that they can often be converted to executable code and then they provide a powerful aid in debugging. Such conversion is supported by widely available software, for example, the `assert()` macro of the C library in Unix.

The use of text such as `/** ... **/` that looks to the compiler like a comment, yet has a special meaning in contexts other than compilation, is borrowed from the work of Luckham and von Henke on annotated Ada [19, 18]. The role of the specialized comments is, however, different here.

Are structured inspections proofs? For the formalist, a proof is a sequence of theorems, each of which is either an axiom, or is obtained from theorems earlier in the sequence by one of a set of formally defined (hence machine-checkable) inference rules. Each theorem occurring in the sequence is then proved by the proof. Usually the last theorem in the proof is the one of interest.

The formality of the language in which the theorems are expressed and of the inference rule make it possible to verify by machine whether a purported proof is valid. If the inference rules used are sound, then the validity of the theorems in the proof is guaranteed.

According to this point of view, the inspection protocol proposed here is not only not a proof, but is of no value whatsoever. For the formalist, something is either a perfect, machine-checkable proof, or it is nothing. But then, according to this point of view, no working[3] proof in mathematics proves anything: all are informal and all omit assumptions.

Although formal proofs have not turned out to be useful in mathematics, research in program verification has aimed at formal proofs only. It has so far not had any impact on software development practice. In [22], DeMillo, Lipton, and Perlis consider why this is the case. They point out that in practice it is important that a responsible person gains confidence in the correctness of a program. A verification program (itself unverified, formally or otherwise) that takes the program to be verified as input, produces an enormous amount of unintelligible output with the word "proved" at the end does not increase confidence. Mathematicians believe certain theorems not only because of the informal proofs that accompany them, but also because of a social process in which theorem and proof are scrutinized. In 1977, DeMillo, Lipton, and Perlis were pessimistic about the prospects of any kind of program validation by means of proofs. Their opinion was based on the assumption that the type of social process that is effective in mathematics is not available in software development.

But inspections *are* such a social process. By making the inspection protocol an informal version of one of the verification methods proposed for programs, Dyer and Kouchakdijan have made the validation procedure that works well in mathematics available in software development.

Although the adoption of the method of mathematicians leads to valuable improvements in software, one should be aware of an important difference between programs and mathematical theorems. In mathematics one may prove a theorem, say, about all sets of a certain type of mathematical object. When it later turns out that the theorem is false for

[3]This qualification is intended to exclude proofs given only as a methodological demonstration.

the empty set, this is not regarded, in mathematics, as a failure or even as detracting anything from the value of the theorem. Such an incident is typically regarded as an editorial matter to be corrected by inserting the word "nonempty" in a suitable place.

In programming the situation is different. A similar error in safety-critical software can lead to disaster. This difference is sometimes taken for a reason why the mathematician's method is of no value in software development. We have shown how the mathematician's method can be used to make code inspections more effective. Even without this improvement, inspections have already proved valuable. In code inspections, we may have to be more pedantic than mathematicians.

The niggling details often overlooked, or even disregarded, by the mathematician can be taken into account in inspection protocols. In addition, guidelines for selection of test cases, such as discussed by Howden [14], concentrate on the kind of "pathological" case that a mathematician may dismiss as uninteresting; another reason to combine structured inspection protocols with testing.

Towards a process of higher quality. Pietrasanta [24] described the discrepancy between best and average practice in software development. According to this survey, average practice injects about 60 defects per thousand lines of code and relies on testing to remove as many of these as possible. This turns out to be about 50, so that 10 defects per thousand lines of code are released. Best practice, which emphasizes defect prevention by means of inspections, achieves a lower injection rate in combination with a higher detection rate, resulting in about .1 defects per thousand lines of code released. Thus, best practice delivers quality which is about two orders of magnitude greater than average.

Another study, from Lockheed Missiles and Space, quoted by Pietrasanta [24] reports a nine-to-one improvement in code quality combined with a five-to-one improvement in productivity. This comparison is between projects operating at SEI[4] Maturity Levels One and Three. During the years covered by this study, and over all organizations, the Level One is typical; Level Three is exceptional.

These data indicate a large discrepancy between best and average practice. The main difference seems to be the use of inspections. What makes inspections so effective? Russell [25] and the originators of the Cleanroom method [5], in which inspections play a crucial role, agree on the importance of *not allowing the programmer to execute the code.* There is reason to believe that this goes far in explaining the discrepancy between best and average practice.

In average practice it is accepted that testing is the main method for removing defects. When working in this mode, there are two stages of testing: a preliminary, informal one by the programmer and a formal stage by the quality assurance group. The confidence of the programmer in the code depends to a significant extent on not being able to find failures during this preliminary testing stage. In this mode it is normal to find failures the first few times any code is run.

[4]The Software Engineering Institute, which defined maturity levels for software development organizations; see [15].

With inspections, and in the absence of any kind of execution before inspection, the programmer's confidence *can only be based on reasoning about the code.* And inspection can only be passed when such reasoning is acceptable to others as well. In this mode it is normal that code runs without failures the first time and from then on. This higher standard has been claimed to be possible by Linger, Mills and Witt [17]. We surmise that it has been attained in the subsequent projects conducted according to "Cleanroom" principles [5] where the highest quality levels were reported and in which these authors played a leading role.

The challenge is then to achieve this higher level of quality for most programmers who at present work in teams performing at the level of average practice. An effective way to do so is to reduce inspections to the checking of well-defined small items with the assurance that the list of items is complete. This is only possible without loss of programmer productivity if the production of such a list is not an additional chore, but is a help in the programming process itself. Both these advantages are obtained from Assertion-Driven Programming and the corresponding structured inspections.

That it is reasonable to try for a level of control in coding at which any failure is an exception rather than the rule should be apparent from the following analogy. In developing safety-critical code, the work of the programmer and of the inspection team is analogous to a high-wire act in a circus. Testing the code is analogous to the presence of a safety net in the high-wire act. Just as testing may not reveal all errors, so also may the safety net fail to save a fall.

When testing reveals an error in code approved by inspection, this is analogous to a fall by the acrobat. Such a fall is a serious matter, even when the safety net prevents bodily harm. Every fall occasions an investigation into its cause: Was the equipment in good working order? Was the acrobat in good physical and mental condition? Were there problems in personal relationships? And so on. Similarly in aviation, not only accidents are subject to formal investigation, but also incidents; for example, not only collisions, but also near-misses.

In the same way, when testing finds an error in inspected code, this should be regarded as a serious and exceptional incident. Testing the code should play the same role as the safety net. An error found in a test should call for an inquest. As we have seen, there is a great deal of discretion in the level of detail of the inspection protocol used. It is impracticable and unnecessary to elaborate the assertions to total formality and complete coverage of assumptions. But it is not clear how far to relax formality and completeness. The occurrence of an error in inspected code may mean that relaxation has been carried too far. But it could also reflect on the competence of the inspectors and it could also reflect on whether they work well as a team. These are the issues to be addressed in the inquest occasioned by failure at test.

The technique described in this paper confirms part of the Cleanroom software development process. The technique suggests a process for safety-critical code with the following features:

1. The programmer prepares the code as part of a structured inspection protocol ac-

cording to the method of Assertion-Driven Programming.

2. The programmer does not run the code.

3. The code is subjected to a structured inspection.

4. In case the inspection is successful, the code is tested.

5. If an error is found, an inquest is conducted into the shortcomings of the process.

At the prevailing level of process maturity, test results are only used to improve the product. At the level of best practice, test results are used primarily to improve the process. It is not taken for granted that the defective product is patched ("debugged").

Only for safety-critical software? We have alluded to the needs of safety-critical software to justify structured inspections. The question arises whether structured inspections should *only* be used for safety-critical software. We believe that this is not the case.

In the first place, a combination of the Cleanroom discipline with structured inspections may not be a more expensive development method. Russell found that his admittedly expensive inspections lead to overall savings by reducing even more expensive debugging. The Lockheed study quoted by Pietrasanta [24] reported a ninefold improvement in quality *in combination with* a fivefold improvement in productivity. Structured inspections may well pay for themselves in the same way.

In the second place, it is not clear how to decide what software is safety-critical. For example, is a compiler included? Compilers used in developing safety-critical software certainly are. How can one know in advance whether a compiler will be used in such a way? Engineering computations are often safety-critical; scientific computations are not. How do we know whether our routine for finding eigenvectors is for scientific calculations only?

The third reason for not wanting to restrict structured inspections to patently safety-critical applications is that the required skills come with a discipline and an attitude that cannot be switched on or off at will. A team that is able to work in this way may not be willing to work at a lesser level. If they do, they probably cannot get back when needed.

Do we then imply that *all* software should be developed by the Cleanroom discipline with structured inspections? There are of course exceptions. If one is writing the umpteenth sequel to the Doom of Zorc, or a program of which the sole purpose is to display winged toasters on the screen, then we have no recommendation to make. We believe this approach should be considered for all *serious* software, whether patently safety-critical or not.

7 Acknowledgments

Two anonymous referees have taken considerable trouble in criticizing an earlier draft. We have taken advantage of many of their remarks to improve the paper. Many thanks to Mantis Cheng, Daniel Hoffman, Paul Strooper, and Clifford Walinsky for discussions on this topic or suggestions for improvement. Generous support was provided by the British Columbia Advanced Systems Institute, the Institute of Robotics and Intelligent Systems, the Canadian Institute for Advanced Research, the Laboratory for Automation, Communication and Information Systems Research, and the Natural Science and Engineering Research Council of Canada.

References

[1] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.

[2] E.W. Dijkstra. Concern for correctness as a guiding principle for program composition. In J.S.J. Hugo, editor, *The Fourth Generation*, pages 359–367. Infotech, 1971.

[3] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[4] M. Dyer and A. Kouchakdijan. Correctness verification: Alternative to structural software testing. *Information and Software Technology*, 32:53–59, 1990.

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

[6] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

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

[8] H.H. Goldstine and J. von Neumann. Planning and coding problems for an electronic computing instrument. In A.H. Traub, editor, *Collected Works of John Von Neumann, volume 5*, pages 80–235. Pergamon Press, New York, 1963. This paper was written in 1947.

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

[10] Mark Halpern. *Binding Time.* Ablex, 1990.

[11] E.C.R. Hehner. do considered od: A contribution to the programming calculus. *Acta Informatica*, 11:287–304, 1979.

[12] C.A.R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[13] D.M. Hoffman. Software inspection, verification, and testing: A hybrid approach. In *Proc. Pacific Northwest Software Quality Conf.*, pages 390–402, 1990.

[14] W.E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, 1980.

[15] W.S. Humphrey. *Managing the Software Process.* Addison-Wesley Publishing Company, 1989.

[16] Donald E. Knuth. Literate programming. *Computer Journal*, 27:97–111, 1984.

[17] Richard C. Linger, Harlan D. Mills, and Bernard I. Witt. *Structured Programming: Theory and Practice.* Addison-Wesley, 1979.

[18] David C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs.* Springer-Verlag, 1990.

[19] David C. Luckham and Friedrich W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, 1985.

[20] Z. Manna. The correctness of programs. *Journal of Computer and Systems Sciences*, pages 119–127, 1969.

[21] Donald N. McCloskey. The rhetoric of economics. *The Journal of Economic Literature*, 21:481–515, 1983.

[22] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems. *Communications of the ACM*, 22(5):271–280, 1979.

[23] David Lorge Parnas and Paul C. Clements. A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, 1986.

[24] Alfred Pietrasanta. A strategy for software process improvement. Workshop at Pacific Northwest Software Quality Conference, 1991.

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

[26] M.H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, SE-5(2):148–159, 1979.

[27] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.