BIT 8 (1968), 174–186

A CONSTRUCTIVE APPROACH TO THE PROBLEM OF PROGRAM CORRECTNESS

E. W. DIJKSTRA

Abstract.

As an alternative to methods by which the correctness of given programs can be established a posteriori, this paper proposes to control the process of program generation such as to produce a priori correct programs. An example is treated to show the form that such a control might then take. This example comes from the field of parallel programming; the way in which it is treated is representative of the way in which a whole multiprogramming system has actually been constructed.

Key words: Algorithms, proof, correctness.

1. Introduction.

The more ambitious we become in our machine applications, the more vital becomes the problem of program correctness. The growing attention being paid to this problem is therefore a quite natural and sound development. As far as I am aware (see [1], [2], [3]), however, the problem has been tackled, posed roughly in the following form: "Given an algorithm and given specifications of its desired dynamic behaviour, prove then that the dynamic behaviour of the given algorithm meets the given specifications." After sufficient formalization of the way in which the algorithm and specifications are given, we are faced with a well-posed problem of some mathematical appeal.

In this paper I shall tackle the problem from the other side: "Given the specifications of the desired dynamic behaviour, how do we derive from these an algorithm meeting them in its dynamic behaviour?". For certain mathematical minds the latter problem will be less attractive (for one thing: the algorithm to be derived is not uniquely defined by the specifications given); it seems, however, to be of much greater practical value because, as a rule, we have to construct the algorithm as well.

This paper has been written because the approach seems unusual, while my collaborators and I have followed it very consciously and seem to have done so to our great advantage. I also publish it in the hope that it may serve as a partial answer to the many doubts evoked by our claim to have constructed a multiprogramming system of proven flawlessness.

In this paper I shall illustrate the method by deriving an algorithm meeting specifications, whose simplicity has been chosen in order to avoid an unnecessarily lengthy paper. In doing so I am running the risk of readers not believing in the practicability of the method when applied to large problems. To those I can only say first, that the art of reasoning to be displayed below is faithfully representative of the way in which we have actually designed a multiprogramming system with fairly refined management rules. Second, that it is my firm belief that by consistent application of such methods our ability to deal with large problems will improve. Third, that anyone who doubts the practicability of the method should try to apply it. Finally, that I know only too well that I can force no one to share my beliefs.

(The chosen problem is a synchronization problem as encountered in multiprogramming. Many of my readers will be unfamiliar with this type of problem and the article may therefore strike them as two articles merged into one: one dealing with multiprogramming and another dealing with the constructive approach. This shortcoming of the paper has been pointed out to me by various unofficial referees of its preliminary version: I agree with their criticism and apologize to my readers. I have, however, stuck to my multiprogramming example, for the general reader's unfamiliarity gives me a unique opportunity to illustrate the approach by treating a simple example, the solution of which is not immediately obvious to everyone. And this, I feel, illustrates the power of the approach more convincingly than treating a traditional problem.)

2. The demonstration problem.

For the purpose of demonstration I have chosen the following problem. We consider two parallel, cyclic processes, called "producer" and "consumer" respectively. They are coupled to each other via a buffer (in this example of unlimited capacity) for "portions" of information. In each of its cycles the producer puts a next portion into the buffer, in each of its cycles the consumer takes a portion from the buffer. The buffer is allocated in the universe surrounding the two processes; after introduction and initiation of this universe, the two processes are started in parallel, as indicated below by the bracket pair "**parbegin**" and "**parend**". It is also indicated that the activity of the producer as well as the activity of the consumer can be regarded at this stage as an alternating succession of two actions. It is understood that the actions labelled $P1$ (i.e. actual production) and $C2$ (i.e. actual consumption) are the time-consuming actions (probably synchronized to other processes) of which the possibility of parallel execution is of actual interest, while the actions labelled $P2$ and $C1$, in which portions are transmitted into or from the buffer (the only ones in which reference to the common buffer is made) will be very concise actions (some bookkeeping with pointers and links, say), the potential parallelism of which can be ignored if desired. We depart in our example from the (hopefully now self-explanatory) structure given below; here the actions invoked are to be considered as available primitives.

Version 0:

```
begin initiate an empty buffer;
  parbegin
    producer: begin local initiation of the producer;
                P1: produce next portion locally;
                P2: transmit portion into the buffer;
                    goto P1
              end;
    consumer: begin local initiation of the consumer;
                C1: transmit portion from buffer;
                C2: consume new portion locally;
                    goto C1
              end
  parend
end
```

For the proper co-operation of the two processes as described above we must assume an implicit synchronization, preventing the consumer to try to take a portion from an empty buffer. In the following we shall refuse to make any assumptions about the speed ratio of the two processes and our task is to program the synchronization between the two processes explicitly. (The synchronizing primitives I intend to use for this solution will be described in due time.)

NOTE. For brevity I omit the fairly simple proof that the above problem is well-posed in the sense that a synchronization satisfying the above requirement does not contain the danger of the so-called "deadly embrace", i.e. one or more processes getting irrevocably stuck because they are waiting for each other. I do so because this proof is more concerned with the problem as posed than with the task of programming it and the latter is the true subject of this paper.

3. Formalization of the required dynamic behaviour.

Our first step is the introduction of suitable variables in terms of which we can give a more formal description of the specification of the required dynamic behaviour. As stated, the consumer should behave in such a way that it does not try to take a portion from an empty buffer. The first question is: how do we keep track of its emptiness? As a result of transmitting a portion into the buffer, the buffer becomes non-empty, as a result of transmitting a portion from the buffer the latter only becomes empty if its last and only portion has been taken from it. In other words, we can keep track of its emptiness (i.e. whether the buffer contains zero portions) provided that we can answer the question whether the (non-empty) buffer contains exactly 1 portion. Repeating the argument we conclude that the number of portions in the buffer is a vital quantity. Therefore we introduce an integer variable, "n" say, whose value has to equal the number of portions in the buffer. The rule to be followed this time is particularly simple: first, initiate the value of "n" together with the initiation of the buffer, so that the relation

$$\text{“}n = \text{number of portions in the buffer”} \qquad (1)$$

is satisfied to start with. From then onwards, adjust the value of the variable called "n" whenever the number of portions in the buffer is changed, i.e. when transmitting a portion into it or from it. As a result the relation (1) will always be satisfied.

From now onwards the three actions initiating or changing the buffer contents are regarded as actions including the proper operation on the variable called "n". To indicate this, we may write Version 1:

```
begin integer n;
    initiate an empty buffer including "n := 0";
    parbegin
        producer: begin local initiation of the producer;
                      P1: produce next portion locally;
                      P2: transmit portion into the buffer including
                              "n := n+1";
                          goto P1
                  end
        consumer: begin local initiation of the comsumer;
                      C1: transmit portion from buffer including "n := n-1";
                      C2: consume new portion locally;
                          goto C1
                  end
    parend
end
```

Thus we have achieved that the specification of the dynamic behaviour can be formulated by the requirement that the inequality

$$n \geqq 0 \tag{2}$$

will always be satisfied.

(REMARK. Already in the transition from the original Version 0 to Version 1 we can observe one of the origins of the efficiency of the constructive approach. If we had regarded the Version 1 as given but in addition to this wanted to identify the current value of n with the current number of portions in the buffer, we would have to observe its initiation and its adjustments, but in excess to this we would have to read the whole program in order to verify that no other operations on it can occur. In the present constructive approach we exploit the fact that the actions labelled P~~2~~ and $C2$ by definition do not refer to the buffer.)
1

4. Analysis of the formalized requirements.

We now proceed from Version 1 and requirement (2). The latter requirement is satisfied initially; we have only to synchronize the two processes in such a way that it *remains* satisfied.

From the fact that requirement (2) concerns the value of the variable called "n" only, it follows that the processes can only cause violation by acting on this variable, i.e. only via the actions labelled "$P2$" and "$C1$" respectively. Closer inspection of the requirement ("$n \geqq 0$") and the actions shows that the action labelled "$P2$" (including "$n := n+1$") is quite harmless, because

$$n \geqq 0 \text{ implies } n+1 \geqq 0\,,$$

but that the action labelled "$C1$" (including "$n := n-1$") may indeed cause a violation. More precisely, as

$$n \geqq 1 \text{ implies } n-1 \geqq 0\,,$$

the action labelled "$C1$" is harmless when initiated when

$$n \geqq 1 \tag{3}$$

while when $n = 0$ it would cause violation; under the latter circumstance it has to be postponed.

(REMARK. Our last conclusion is that the only possible harm is trying to make the buffer more empty than empty. Its obviousness here is a direct consequence of the simplicity of this example. The point is, that this conclusion could be reached by inspection of the formalized requirement (2) and the operations on the variables concerned. In the case of a more refined management, the requirements analogous to (2) are no longer a simple inequality and their analysis will really tell you all the danger points.)

5. Consequence of the preceding analysis; the unstable situation.

In the previous section we have concluded that the action labelled "*C*1" is the only danger point. Having here only one consumer, we could have solved the problem logically by inserting just in front of it a wait cycle

"*C*0: **if** $n = 0$ **then goto** *C*0"

but our group refused to implement this busy form of waiting, because in a multiprogrammed environment it seems a waste to spend central processor time on a process that has already established that for the time being it cannot go on; furthermore this solution does not admit a straightforward generalization to, say, more consumers. Therefore we have implemented means—viz. the synchronizing primitives—by which a process can go to sleep until further notice (a sleeping process being by definition no candidate for processor time), leaving of course to the other processes the obligation to give this "further notice" in due time. This is so closely analogues to usual optimizing techniques that I proceed with this multiprogramming example in full confidence that the uniprogrammer will be able to apply similar considerations to his own tasks.

We see ourselves faced with the decision whether the action labelled "*C*1" should take place or not. Earlier we have seen that this decision depends on the current value of the variable called "n". Recently we have seen that under certain circumstances we refuse to regard this as a private decision of the consumer (this would imply the busy form of waiting) but wish to delegate it (via the mechanism of the further notice) to the producer. As long as it was a private decision of the consumer, inserting it at the right place in the consumer's text was a sufficient means to ensure that the decision was taken in accordance with the dynamic progress of the consumer. As soon as this decision may be taken by another process—in this example by the producer—the dynamic progress of the consumer becomes a question of general interest, in particular whether the consumer is ready for the next action labelled *C*1. We introduce a boolean variable, called "*hungry*" whose value has to indicate explicitly that the consumer's progress has reached the stage that the decision to execute or to postpone the action labelled "*C*1" is relevant.

To ensure that the variable called "*hungry*" has this meaning, we must

1) insert within the consumer's cycle the assignment "*hungry* := **true**" just in front of the statement labelled "$C1$";
2) include the assignment "*hungry* := **false**" as part of the action labelled "$C1$";
3) initiate in the universe the variable called "*hungry*" in accordance with the starting point in the consumer's cycle.

The variable called "*hungry*" is an explicit coding of the consumer's progress, analogous to the variable called "n", introduced as an explicit coding of the number of portions in the buffer. We arrive at Version 2:

```
begin integer n; Boolean hungry;
  initiate an empty buffer including "n := 0";
  hungry := false;
  parbegin
    producer: begin local initiation of the producer;
                 P1: produce next portion locally;
                 P2: transmit portion into the buffer including
                       "n := n + 1";
                     goto P1
              end;
    consumer: begin local initiation of the consumer;
                 C0: hungry := true;
                 C1: transmit portion from buffer including "n := n − 1"
                       and "hungry := false";
                 C2: consume new portion locally;
                     goto C0
              end
  parend
end
```

From relation (3) and the meaning of the variable called "*hungry*" we now deduce that the action labelled "$C1$" should take place whenever

$$n \geqq 1 \textbf{ and } hungry \tag{4}$$

becomes **true**, the action labelled "$C1$" itself causing (4) to become **false** again. In other words: we must see to it that (4) characterizes what we could call "an unstable situation", for as soon as it emerges it should be resolved by the action labelled "$C1$".

Having no permanently active observer that will give alarm whenever the unstable situation arises, we must allocate the inspection for the unstable situation (and, if found, its subsequent resolution by action "$C1$") somewhere in the sequential processes. The necessary and sufficient measure is to attach this inspection as an appendix to each action that may have generated the unstable situation from a stable one, thus pinning the responsibility to resolve the unstable situation down to the process that has generated it.

Some elementary logic applied to (4) tells us that this transition can only be effected by an action assigning the value **true** to the variable called "*hungry*" or by an action increasing the value of the variable called "*n*" (or by an action doing both, not occurring in this example). In terms of Version 2: the instability may be reached as a result of the action labelled "$C0$" (on account of "*hungry* := **true**") and of the action labelled "$P2$" (on account of "$n := n+1$"). So the action labelled "$P2$" —allocated in the producer—might get attached to it as an appendix the action labelled "$C1$", originally allocated in the consumer!

6. Interlude on synchronizing primitives.

At this stage of the discussion I must insert an interlude because I expect many a reader to be unfamiliar with the basic problems of programming parallel processes, the field from which our example happens to have been taken.

We need primitives to control that processes may go to sleep or may be woken up. For this purpose we introduce

1) special purpose binary valued variables, called "semaphores". A semaphore may have the values 0 and 1. Semaphores are allocated in the surrounding universe and are initiated before the parallel processes are started. (Semaphores may be generalized from two-valued quantities to non-negative integers. In this article we do not do so, for our example is so simple that the generalized semaphore would provide a ready made solution!)

2) two special operations, called the P- and the V-operation respectively. The parallel processes shall access the semaphores via these operations only.

The P-operation on a semaphore can only be completed when the semaphore value equals 1. Its completion implies that the semaphore value is reset to 0. If a process initiates a P-operation on a semaphore with at that moment a value equal to 0, "the process goes to sleep, the P-operation remains pending on this semaphore".

The V-operation on a semaphore is only defined if its initial value equals 0. It will then set the semaphore to 1. If no P-operation is pending on this semaphore the V-operation has no further effect. If one or more P-operations are pending on it, the V-operation will have the further effect that exactly one of the pending P-operations will be completed (thereby resetting the semaphore to the value 0), i.e. the process in which this P-operation occurred is woken up again.

As a result a semaphore value equal to 1 implies that there are at that moment no P-operations pending on it.

The semaphores are used for two entirely distinct purposes; both standard usages will occur in the example.

On the one hand we have the so-called "private semaphores", each belonging to a specific sequential process, that will be the only one to perform a P-operation on it, viz. where the process might need to be delayed until some event has occurred: the semaphore values 0 and 1 at the initiation of the P-operation represent the situation that the event in question has not yet or has already occurred. As a rule the universe initiates private semaphores with the value 0.

On the other hand we have the semaphore(s) used for the implementation of so-called "critical sections", the executions of which have to exclude each other in time. Such critical sections can be implemented by opening them with a P-operation and closing them with a V-operation, all on the same semaphore with initial value 1. At each moment the value of such a semaphore for mutual exclusion equals the number of processes allowed to enter a section critical to it. The purpose of critical sections is to cater for unambiguous modification and interpretation of universal variables (such as "*n*" and "*hungry*" in our example).

Alternatively: at a certain level of abstraction we can visualize a single sequential process as a succession of "immediate actions"; the time taken to perform them is logically immaterial, only the states (as given by the values of the variables) observable in between the actions have on that level a logical significance. It is only when we shift to a lower level of abstraction and implement the actions themselves by means of (smaller) sequential sub-processes, that the intermediate states as well as their periods of execution enter the picture. And it is only at this lower level that "mutual exclusion in time" has a significance. In a single sequential process successive actions (now regarded as sub-processes) exclude each other in time automatically, because the next one will only be initiated after the preceding one has been completed. In multiprogramming the mutual exclusion at the lower level of abstraction is no longer automatically guaranteed and the fact that on the higher level we regard them as single "immediate actions" requires then explicit recognition. This is exactly what the critical sections cater for.

7. Resolution of the unstable situation and synchronization of the processes.

Our analysis of the unstable situation ended with the conclusion that the action labelled "*C*1", originally allocated in the consumer, will be attached as a conditional appendix to the actions that might generate the unstable situation, i.e. the ones labelled "*C*0" and "*P*2" respectively.

To pin the responsibility for the resolution of the unstable situation down to the process that has generated it, the latter one must be uniquely defined (which is not the case if the effective assignments "$n := 1$" as part of *P*2 and "*hungry* := **true**" as part of *C*0 are allowed to take place simultaneously) and it must have resolved the unstable situation before the other process may have discovered it. In other words, creation of the unstable situation and its subsequent resolution must be regarded as a single "immediate action" in the sense of the last paragraph of the interlude. We shall implement them by critical sections controlled by a semaphore, "*mutex*" say, that will be initiated with the value 1.

Finally, in Version 2 the sequential nature of the consumer guaranteed that each execution of the action labelled "*C*2" would be preceded by one execution of the action labelled "*C*1". This implicit sequencing can be made explicit with the aid of a private semaphore of the consumer, "*consem*" say, (to be initiated with the value 0) by concluding the action labelled "*C*1" with "*V*(*consem*)" and opening the action labelled "*C*2" with "*P*(*consem*)". The sequencing has to be made explicit because the action labelled "*C*1" may now occur as an activity of the producer.

After these considerations the final version of the program is given. For reasons of clarity and economy (of writing and thinking) the action labelled "*C*1" has been included in the body of a procedure declared in the universe.

NOTE 1. The program given below does not pretend to be the only, or the best or the most economical solution. It pretends to be a correct solution. It is general in the sense that more complicated similar problems can be solved along the same pattern.

NOTE 2. As announced (in section 2) the potential parallelism of the action "transmit portion into the buffer" and "transmit portion from buffer" could be ignored if desired. This indeed has happened, as the actions only occur within critical sections. The generalization to more producers, more consumers etc. is now straightforward.

NOTE 3. The step from Version 2 to the Final Version is rather large: for various unofficial referees of the preliminary version of this paper the

Final Version still came straight from "The Magician's Box". I can understand their feelings, I have not succeeded in remedying the situation, "in buffering the shock of invention".

Final Version:

```
begin integer n; Boolean hungry; semaphore mutex, consem;
  procedure resolve instability if present;
  begin if n ≧ 1 and hungry then
    begin transmit portion from buffer; n := n − 1;
      hungry := false; V(consem)
    end
  end;
  initiate an empty buffer; n := 0;
  hungry := false; mutex := 1; consem := 0;
  parbegin
    producer: begin local initiation of the producer;
                P1: produce next portion locally;
                P2: P(mutex);
                    transmit portion into the buffer; n := n + 1;
                    resolve instability if present;
                    V(mutex);
                    goto P1
              end;
    consumer: begin local initiation of the consumer;
                C0: P(mutex);
                    hungry := true; resolve instability if present;
                    V(mutex);
                C2: P(consem); consume new portion locally;
                    goto C0
              end
  parend
end
```

The above is as faithful a reproduction as I can give of the kind of reasoning we applied in the construction of a multiprogramming system, albeit interlaced with explanatory paragraphs, covering the insights we had already gained at an earlier stage by just thinking about the problems involved in the programming of parallel processes. At the end, when we were all familiar with this type of problem, the reasoning needed to derive the program from specifications much more complicated than the present example, used to be given on a single page or less.

8. Concluding remarks.

First, one can remark that I have not done much more than to make explicit what the competent programmer has already done for years, be it mostly intuitively and unconsciously. I admit so, but without any shame: making his behaviour conscious and explicit seems a relevant step in the process of transforming the Art of Programming into the Science of Programming. My point is that this reasoning can and should be done explicitly.

Second, I should like to stress that by using the verb "to derive" I do not intend to suggest any form of automatism, nor to underestimate the amount of mathematical invention involved in all non-trivial programming, on the contrary! But I do suggest the constructive approach sketched in this paper as an accompanying justification of his inventions, as a tool to check during the process of invention that he is not being lead astray, as a reliable and inspiring guide.

Third, I am fully aware that the style of reasoning I have applied, though possibly appealing to some, might easily appal others. For this difference in taste I blame them as little as they should blame me. I can only hope that they will find a constructive style satisfactory to them.

Finally, I should like to point out that the constructive approach to program correctness sheds some new light on the debugging problem. Personally I cannot refrain from feeling that many debugging aids that are en vogue now are invented as a compensation for the shortcomings of a programming technique that will be denounced as obsolete within the near future.

Acknowledgements.

Acknowledgements are due to my closest collaborators C. Bron, A. N. Habermann, F. J. A. Hendriks, C. Lightmans and P. A. Voorhoeve, for by working in the way they did, they convinced me of the practicability of the constructive approach when faced with a large problem.

Acknowledgements are also due—although they may be unaware of the fact—to Peter Naur and my colleague Gerhard W. Veltkamp. To the former because he convinced me that something should and could be done regarding program correctness, to the latter for his inspiring faith in my efforts, his patience when listening to me and his unrelenting, pityless criticism whenever I indulged in disguising sloppy reasoning by a verbal show.

REFERENCES

1. Robert W. Floyd, *Assigning Meanings to Programs*, Proceedings of Symposia in Applied Mathematics, Volume 19, *Mathematical Aspects of Computer Science*, pg. 19–32, American Mathematical Society, 1967.
2. John McCarthy and James Painter, *Correctness of a Compiler for Arithmetic Expressions*, Technical Report No. CS38, April 29, 1966, Computer Science Department, Stanford University.
3. Peter Naur, *Proof of Algorithms by General Snapshots*, BIT vol. 6, 1966, pg. 310–316.

DEPT. OF MATHEMATICS
TECHNOLOGICAL UNIVERSITY EINDHOVEN
EINDHOVEN, THE NETHERLANDS