Invariance and Non-Determinacy [and Discussion]

E. W. Dijkstra, R. S. Bird, M. H. Rogers and O.-J. Dahl

Phil. Trans. R. Soc. Lond. A 1984 **312**, 491-499
doi: 10.1098/rsta.1984.0072

Phil. Trans. R. Soc. Lond. A **312**, 491–499 (1984) [491]
Printed in Great Britain

Invariance and non-determinacy

By E. W. Dijkstra
Burroughs, Plataanstraat 5, 5671 AL Nuenen, The Netherlands

Since the earliest days of proving the correctness of programs, predicates on the program's state space have played a central role. This role became essential when non-deterministic systems were considered. The first (and still best known) source of non-determinacy was provided by operating systems, which had to regulate the cooperation between components that had speed ratios that were beyond our control. Distributed systems have revived our interest in such configurations.

I know of only one satisfactory way of reasoning about such systems: to prove that none of the atomic actions falsifies a special predicate, the so-called 'global invariant'. Once initialized, the global invariant will then be maintained by *any* interleaving of the atomic actions. That solves the problem in principle; in each particular case, however, we have to choose how to write down the global invariant. The choice of notation influences the ease with which we can show that, indeed, none of the atomic actions falsifies the global invariant.

An example will be given and discussed.

An accident introduced me 32 years ago to automatic computing, a topic that has fascinated me ever since. As the years go by, I am beginning to appreciate the length of my involvement more and more, since I owe to it a very lively picture of a sizeable part of the history of the growth of a science. I have observed profound changes in our thinking habits, and I have found those observations interesting and instructive.

I do remember, for instance, one of my first efforts – in the mid 1950s – to come to grips with what we would now call 'repetition'. It was profoundly inadequate, and in the course of this talk I hope to explain to you why. Very operationally, I tried to deal with it as a recurrence relation: one instructs the machine to start with an initial value x_0 and to generate from there enough values from the sequence further defined by the recurrence relation

$$x_{i+1} = f(x_i).$$

Why did I do that? I think because I was glad to recognize something familiar, and in those days familiarity was more important than significance. The knowledge I had at the time was already sufficient to doubt the significance, but I do not remember doing so. You see, a well known concept was the 'order' of a recurrence relation, the Fibonacci sequence being given by the second-order recurrence relation

$$F_{n+2} = F_{n+1} + F_n;$$

but any programmer would implement this by

$$(A, B)_{n+1} = (A+B, A)_n,$$

i.e. a first-order recurrence relation! In short then, I should have already been suspicious. But it *was* the prevailing view in that decade: not only FORTRAN but even ALGOL 60 included only repetitive constructs of which the so-called 'controlled variable' was an essential ingredient.

My estimation is that the introduction of the so-called 'controlled variable' has delayed the development of computing science by almost a decade. I got very suspicious in the late 1960s when I discovered that the 'dyed-in-the-wool' FORTRAN or ALGOL programmer had been conditioned so as to be unable to design the elegant solution to what became known as The Problem of the Dutch National Flag. The fact that, in the 1970s, the Euclid algorithm for the g.c.d. of the positive integers X and Y was widely quoted as a paradigm, I can only explain by the circumstances that it is the simplest program that demonstrates so convincingly the inappropriateness of the notion of the 'controlled variable'.

I now write the Euclid algorithm in the form

```
|[x, y: int
 ;x, y := X, Y
 ;do x > y → x := x − y
   ▯ y > x → y := y − x
  od
]|
```

This is evidently a repetition in which there is no place for a 'controlled variable' counting something of relevance or controlling termination, or both.

* * *

Another incident – A. W. Dek's invention of the real-time interrupt – introduced me 25 years ago to non-determinacy. My first major concern was to show that saving register contents at program interruption and restoring them at program resumption could not be corrupted by the occurrence of a next interrupt. The arguments required were very tricky, so tricky as a matter of fact that I was not surprised at all when I found flaws in the designs of the interrupt facilities of later machines such as the CDC 165 and the IBM 360. I experienced the problems caused by the unpredictable interleaving as completely novel ones, not suspecting that, about a decade later, they would be tackled by the same techniques that would then be used for reasoning about repetitions.

I am, of course, referring to the technique of the so-called 'invariant' as illustrated in the following type of annotation of a repetition (assertions being written within braces)

```
{P}
do B → {P ∧ B} S {P} od
{P ∧ ¬B}.
```

In words: if assertion P, guard B and statement S are such that the additional validity of B guarantees that execution of S does not destroy the validity of P, then the whole repetition **do** B → S **od** will not destroy the validity of P, *no matter how often* the repeatable statement S is repeated.

Now, if we have several clauses B → S, none of which destroys the validity of P, the validity of P will not be destroyed *no matter how often and in what order* they are repeated. In other words, the pattern appropriate for reasoning about repetitions is straightforwardly able to cope with the non-determinacy that has to be absorbed by the operating system for, say, a multiprogrammed installation.

The technique has been used rather constructively in the design of the THE Multiprogramming System to derive the 'synchronization conditions' (i.e. guards) that would ensure, for instance, that no buffer would become emptier than empty or fuller than full. At the time we did not know the axiom of assignment; we only knew what it entailed for simple assignment statements, such as $n := n+1$, and equally simple predicates, such as $n \leqslant N$. This was in the first half of the 1960s.

In the second half of the 1960s the method was formalized for deterministic sequential programs by R. W. Floyd (1967) and by C. A. R. Hoare (1969). Floyd included proofs of termination, but addressed himself to programs that could be expressed by arbitrary flow charts. (This latter generality was not too attractive. In the control graph one had to select a set of so-called 'cutting edges', i.e. a set of edges such that their removal would leave a graph with no cycles, and to each cutting edge a proof obligation corresponded. The awkward thing is that for an arbitrary control graph the problem of determining a minimum set of cutting edges is most unattractive.) Hoare's subsequent contribution was twofold: on account of the structure of the axiom of assignment he definitely decided in favour of so-called 'backwards reasoning' – Floyd had left this choice open – and he tied the proof obligations in with the syntactic constructs for the flow of control. (Ironically, he confined himself to partial correctness, though the problem of finding a minimum set of cutting edges – which are required for termination proofs – had been reduced to triviality by the sequencing discipline he had adopted.) All this was synthesized in the early 1970s by myself, and my 'guarded commands', besides forming a basis for a calculus for the derivation of programs, introduced non-determinacy into conventional sequential programming.

Central to this game was the formal expression and manipulation of so-called 'assertions' or 'conditions', i.e. predicates that contained the coordinates of the program's state space as free variables, for example to derive for a program fragment the precondition corresponding to a given post-condition. (It is this direction of the functional dependence to which the term 'backwards reasoning' refers. In addition to a simpler axiom of assignment, the pragmatic advantages of backwards reasoning are twofold. It circumvents undefined values since for any program fragment the pre-condition is a total function of the post-condition, whereas the post-condition is, in general, a partial function of the pre-condition. Furthermore, the calculus includes non-determinacy at no extra cost at all.)

For the formulation and manipulation of these conditions, the predicate calculus became a vital tool; so much so, that during the last decade it became for many programming computing scientists an indispensable tool for their daily reasoning. (In passing I may mention my strong impression that those computing scientists may very well have been the first to *use* the predicate calculus regularly. Mathematicians, and even logicians, for whom, for instance, the facts that equivalence is associative, that disjunction distributes over equivalence, and that conjunction distributes over non-equivalence, belong to their active knowledge, are extremely rare; I have never met one. Without intimate knowledge of such basic properties of the logical connectives one can hardly be expected to be a very effective user of the predicate calculus; hence my strong impression. In retrospect I found rather shocking the conclusion that as far as the mathematical community is concerned George Boole has lived in vain.)

The extensive use of the predicate calculus in program derivation during the last decade has had a profound influence, the consequences of which are still unfathomed. It turned program development into a calculational activity (and the idea of program correctness into a calculational notion). The consequences are unfathomed because suddenly we find ourselves urgently invited to apply formal techniques on a much greater scale than we were used to. It turns out that the predicate calculus only solves the problems 'in princple': without careful choice of our extra-logical primitives and their notation, the formulae to be manipulated have a tendency of becoming unmanageably complicated. As a result, each specific problem may pose a new conceptual and notational challenge. By way of illustration, I shall show an extreme example from the field of distributed programming; the example is extreme in the sense that almost all the manipulations of the derivation belong to the extra-logical calculus of regular expressions.

* * *

We consider a network of machines that can send messages to each other. Each machine is in one of three states, namely

n for 'neutrally engaged',
d for 'delayed', or
c for 'critically engaged'.

The objective is to ensure that at most one machine at a time shall be in state c. A critical engagment lasts for only a finite period and is immediately followed by a neutral engagement of the machine in question. Between a neutral engagement and the subsequent critical engagement a delay may occur in view of the requirement that at any moment at most one machine be critically engaged (called 'mutual exclusion'). The implied synchronization has to be implemented in such a manner that no delay lasts forever (called 'fairness').

We introduce a single **token**, either held by one of the machines or being sent from one machine to another. Mutual exclusion is then achieved by maintaining the truth of the predicate

a critically engaged machine holds the token.

The machines maintain this by (i) not initiating a critical engagement unless holding the token, and (ii) not sending the token to another machine while being critically engaged.

Furthermore each machine maintains

the machine holding the token is not delayed

by (i) skipping the delay upon termination of a neutral engagement while holding the token, and (ii) initiating a critical engagement upon receipt of the token while delayed. Fairness is therefore ensured when each delayed machine receives the token within a finite period of time. Consequently, there must be some means of passing the token from one machine to another. But we do not want the token to pass unless necessary. We therefore also need one or more **signals**, which are sent by delayed processes to indicate interest in obtaining the token.

The rest of this example deals with the control of the movement of the token and signals. To this end the machines are connected by links into a ring, of which the two circular directions are called 'to the left' and 'to the right', respectively. The token is sent to the left and signals are sent to the right. Each link connecting two neighbouring machines in the ring is in one of three states, namely

u for 'unused',
t for 'carrying the token to the left', or
s for 'carrying a signal to the right'.

The last two states are postulated to last for only a finite period of time.

The computation will be broken up into *atomic actions*. (An atomic action is the same type of idealization as the 'point mass' in physics.) Each atomic action is performed by one of the machines and involves a state change for that machine and for one or both of its adjacent links. There are four kinds of atomic action to be designed, those which take place:

(n) upon completion of a neutral engagement,
(c) upon completion of a critical engagement,
(s) upon arrival of a signal,
(t) upon arrival of the token.

(We need not bother about 'completion of a delay' since this will be subsumed by the arrival of the token; similarly the 'completion' of the state 'unused' for a link is subsumed in sending either the token or a signal over that link.)

Our invariant for the whole system is, loosely speaking, 'the ring is in a permissible state', but that is only helpful provided we have a very precise characterization of the set of permissible states. Instead of giving this characterization in advance as an invariant predicate, we shall derive the set of permissible states as the transitive closure of the atomic transitions, starting from a given initial state, namely: all machines neutrally engaged, all the links unused, and the token residing in one of the machines.

Immediately the question arises how to characterize sets of ring states. We shall represent a ring state as a string in which machine states and link states alternate, with the understanding that the left end of the string is adjacent to the right end.

We can now characterize a set of ring states by writing down a grammar for representative strings. In this example we shall use the grammar of 'regular expressions', though in fact we are concerned only with strings of fixed length (twice the number of machines).

It will turn out to be handy to give the machines one of two colours, either black (b) or white (w), and a machine state will be coded by prefixing one of the three states n, d or c, by one of the colours b or w. Blackness of a machine indicates that interest in the token exists to the left. The machine holding the token will be identified by writing its colour with the corresponding capital letter. Initially all machines being white, we can characterize the unique initial state by the regular expression

$$-\,(\mathrm{wn\ u})^*\ \mathrm{Wn\ u}\,- \qquad (0)$$

(Note: if a regular expression is used to characterize a set of ring states, we shall surround it by a pair of dashes. This implies, for instance, that (0) is equivalent to

$$-\,(\mathrm{u\ wn})^*\ \mathrm{u\ Wn}\,- \qquad .)$$

In state (0), only completion of neutral engagements is possible. For the time being we confine our attention to the more interesting case of such completions taking place in machines not holding the token and propose the transition

$$\mathrm{wn\ u} \rightarrow \mathrm{wd\ s} \quad , \tag{n. 0}$$

i.e. a white machine without the token completes its neutral engagement by becoming delayed and sending a signal over the link to its right. (Transition (n. 0) only caters for the situation that a ‘wn’ has a ‘u’ to its right.)

The transitive closure of (0) under (n. 0) is

$$-\ (\mathrm{wn\ u} \mathbin{\square} \mathrm{wd\ s})^{*}\ \mathrm{Wn\ u}\ - \tag{1}$$

in which $\square$ (which syntactically has been given the lowest binding power) should be read as ‘or’. Note that (1) is equivalent to

$$-\ (\mathrm{wn\ u} \mathbin{\square} \mathrm{wd\ s}\ (\mathrm{wn\ u})^{*})^{*}\ \mathrm{Wn\ u}\ - \quad .$$

For the arrival of a signal at a white neutral machine we propose the transition

$$\mathrm{s\ wn\ u} \rightarrow \mathrm{u\ bn\ s} \quad , \tag{s. 0}$$

i.e. the machine transmits the signal and blackens itself. The transitive closure of (0) under (n. 0) and (s. 0) is given by

$$-\ (\mathrm{wn\ u} \mathbin{\square} \mathrm{wd}\ (\mathrm{u\ bn})^{*}\ \mathrm{s})^{*}\ \mathrm{Wn\ u}\ - \quad .$$

Closing this further under

$$\mathrm{u\ bn} \rightarrow \mathrm{u\ bd} \tag{n. 1}$$

$$\mathrm{s\ wd} \rightarrow \mathrm{u\ bd} \tag{s. 1}$$

yields $\quad -\ (\mathrm{wn\ u} \mathbin{\square} \mathrm{wd}\ (\mathrm{u\ bn} \mathbin{\square} \mathrm{u\ bd})^{*}\ \mathrm{s})^{*}\ \mathrm{Wn\ u}\ - \quad ,$

which we record as

$$-\ \mathrm{H}^{*}\ \mathrm{Wn\ u}\ - \qquad \text{with} \tag{2}$$

$$\mathrm{H} = \mathrm{wn\ u} \mathbin{\square} \mathrm{Q\ s} \qquad \text{with} \tag{3}$$

$$\mathrm{Q} = \mathrm{wd}\ (\mathrm{u\ bn} \mathbin{\square} \mathrm{u\ bd})^{*} \quad . \tag{4}$$

We note that the grammars H, H H*, H* and H*Q (note the absence of dashes: these grammars correspond to sets of strings) are also closed under the four transitions considered so far. (The reader is not expected to see this at a glance: the formal verification of the above claim requires a short calculation.) Furthermore we note that under the transitions given so far, the transitive closure of the string wn u H* equals H H*.

Let us now look at the more interesting case that a signal arrives at the machine holding the token. The only way in which we can make the substring s Wn explicit in (2) is by adding the superfluous term Q s Wn u

$$-\ \mathrm{H}^{*}\ (\mathrm{Wn\ u} \mathbin{\square} \mathrm{Q\ s\ Wn\ u})\ - \quad ,$$

which we can close under

$$\mathrm{s\ Wn\ u} \rightarrow \mathrm{t\ wn\ u} \tag{s. 2}$$

by applying (s. 2) now as rewrite rule:

$$-\ \mathrm{H}^{*}\ (\mathrm{Wn\ u} \mathbin{\square} \mathrm{Q\ t\ wn\ u})\ - \quad .$$

As a result of the emergence of a new instance of wn u, this is no longer closed under the previous transformations. But because the ring is cyclic, we can rewrite this as

$$-\,\mathrm{H}^*\ (\mathrm{Wn\ u} \mid \mathrm{Q\ t}\ (\mathrm{wn\ u\ H}^*))\,-$$

and the closure of this yields

$$-\,\mathrm{H}^*\ (\mathrm{Wn\ u} \mid \mathrm{Q\ t\ H})\,- \quad . \tag{5}$$

Closing (5) under

$$\mathrm{Wn\ u} \rightarrow \mathrm{Wc\ u} \tag{n. 2}$$

obviously yields

$$-\,\mathrm{H}^*\ (\mathrm{Wn\ u} \mid \mathrm{Wc\ u} \mid \mathrm{Q\ t\ H})\,- \quad , \tag{6}$$

which is also closed under the inverse

$$\mathrm{Wc\ u} \rightarrow \mathrm{Wn\ u} \quad . \tag{c. 0}$$

With the introduction of the term Wc u we have created the possibility of a signal arriving at the critically engaged machine (which holds the token). Observing that in (6) the substring s Wc can only occur in Q s Wc u, adding this as a superfluous term, and applying the transition

$$\mathrm{s\ Wc\ u} \rightarrow \mathrm{u\ Bc\ u} \tag{s. 3}$$

as rewrite rule, we derive the closure

$$-\,\mathrm{H}^*\ (\mathrm{Wn\ u} \mid \mathrm{Wc\ u} \mid \mathrm{Q\ t\ H} \mid \mathrm{Q\ u\ Bc\ u})\,- \quad . \tag{7}$$

(Because we lack a full regularity calculus we did not apply it. It is instructive to know that, as a result, grammars (5), (6) and (7) are *not* fully closed.)

The introduction of the term Bc introduces a new form of critical engagement. When this terminates, we require that the token be sent to the left

$$\mathrm{u\ Bc\ u} \rightarrow \mathrm{t\ wn\ u} \quad . \tag{c. 1}$$

Since the resulting Q t wn u is subsumed by the preceding Q t H, (7) is closed under (c. 1) as well.

We leave to the reader the verification that (7) is also closed under the remaining three transitions, which enumerate how the token can arrive:

$$\mathrm{wd\ t} \rightarrow \mathrm{Wc\ u} \tag{t. 0}$$

$$\mathrm{u\ bn\ t} \rightarrow \mathrm{t\ wn\ u} \tag{t. 1}$$

$$\mathrm{u\ bd\ t} \rightarrow \mathrm{u\ Bc\ u} \quad . \tag{t. 2}$$

Since (7) tells us that t has a string Q to its left, which may end only in these three different ways, the construction of the closure and of the list of transitions that might be needed has now been completed.

Inspection of (7) shows that the condition of mutual exclusion is satisfied. It also enables us to convince ourselves that each delay will be of finite duration. For that purpose we associate with a delayed machine the string of (alternating) links and machines to its right, up to and including the machine that holds the token or the link that carries it. For that string we define k by

k = the number of elements in the string + the number of white machines in the string.

To begin with we observe that $k \geqslant 0$ and that none of the transitions increase k. We now convince ourselves that k decreases within a finite period of time, given that the states s, t and c are of finite duration: from (3), (4) and (7) we conclude that the delayed machine occurs in a Q;

(i) for a Q in H, the string contains an s, and (s. 0), (s. 1), (s. 2) or (s. 3) will decrease k;

(ii) for the Q in Q t H, (t. 0), (t. 1) or (t. 2) will decrease k;

(iii) for the Q in Q u Bc u, (c. 0) is inapplicable and (c. 1) will decrease k, and from (7) we conclude that this case analysis has been exhaustive. This concludes (the compact presentation of) our example.

Retrospective remarks

In the calculations presented, the machines themselves have remained anonymous. We could have numbered them from 0 to $N-1$, but invite the reader to try to visualize what our invariant would have looked like, had we used quantifications over machine subscripts! It would have been totally unmanageable. (Not only did we leave the individual machines anonymous, but even their number is not mentioned in the analysis: for a ring of N machines, only the strings of length 2N that belong to the grammar (7) are applicable. A fringe benefit is that very small values of N do not require special analysis. To pay for these benefits, we have the trivial obligation to show that no transition changes the number of machines.)

After the decision to try to use regular expressions, it took me several iterations before I had reached the above treatment. My first efforts contained errors, due to my lack of experience in using the 'regularity calculus' for deriving a transitive closure under rewrite rules. The lack of experience was made more severe by the fact that the same language can be characterized by many different regular expressions: for instance, (a ▯ b)*, (a ▯ b ▯ a b)* and (a ▯ b a*)* are all equivalent. In the beginning I experienced this great freedom as a nuisance, but now I think this was naïve, since precisely these language-preserving transformations enable us to massage a regular expression into a form suitable for our next manipulation. Equivalences lie at the heart of any practical calculus.

Finally, it took me quite some time before I discovered the proper abbreviations to introduce. (H and Q, easy to defend in hindsight, could have been chosen much earlier, had we had more familiarity with the regularity calculus.)

Conclusion

I mentioned that, owing to the calculational approach to program design, each specific problem may pose a new conceptual and notational challenge. The example given has been included to give the reader some feeling for the forms that challenge may take. I called the consequences unfathomed, the reason being that the machines executing our programs are truly worthy of the name 'general purpose equipment' and that, consequently, the area that calls for the effective application of formal techniques seems to have no limit.

The example illustrating the use of the regularity calculus was developed under the critical inspiration of the Tuesday Afternoon Club, and of F. E. J. Kruseman Aretz in particular. It was a privilege to write half of the above with A. J. M. van Gasteren looking over my shoulder.

References

Floyd, R. W. 1967 Assigning meanings to programs. In *Proc. Symp. Appl. Math.* (ed. J. T. Schwartz), vol. 19, pp. 19–31. Providence, Rhode Island: American Mathematical Society.

Hoare, C. A. R. 1969 An axiomatic basis for computer programming. *Commun. Ass. comput. Mach.* **12**, 576–580.

Discussion

R. S. Bird (*Programming Research Group, Oxford University, U.K.*). What intuitions lay behind the invention of the coloured states black and white?

E. W. Dijkstra. Clearly the receipt of a signal, i.e. the obligation to send or transmit the token, has to be recorded. Under such circumstances I find 'colouring' objects a handy metaphor. It makes it easy to visualize and to talk about, and, furthermore, one can start colouring without knowing how many colours will eventually be needed. I remember the design of a mark-scan garbage collector in which reachable nodes changed during the marking phase from white to black via the intermediate colour grey. I used the same metaphor in the 1950s, when I designed an algorithm for the shortest path. Clearly the metaphor suits me.

M. H. Rogers (*School of Mathematics, University of Bristol, U.K.*). Does Professor Dijkstra hold out any hope for automating the procedure of choosing suitable global invariants, at least for some range of programs?

E. W. Dijkstra. No, not much. The example I showed is in this respect telling: the choice of notation was already critical.

O.-J. Dahl (*Institute of Informatics, Blindern, Oslo, Norway*). I notice that in this development the invariant is an end result, not an initial idea. Can Professor Dijkstra comment on that? When is this an appropriate mode of development?

E. W. Dijkstra. The reason to derive the invariant as I went along was probably twofold. First, it was too complicated to be guessed or postulated. Second, I wanted this time to have the strongest invariant so as not to have the program cater for situations that could not arise.