SETL Newsletter # 160

J.T. Schwartz
November 13, 1975

An Algebra of Program Events Potentially Useful in a Debugging Language.

1. Introduction.

In debugging and also in reasoning about program behavior and correctness, one needs to use language describing program events, and of course if one proceeds informally this is no problem. But to make such language available either within an implemented debugging system or in a correctness-proving system, formalisation is necessary. This short newsletter will sketch the (rather simple) semantics and syntax of a formal language of program events, and will then go on to indicate the use which such a language might have in systematic debugging.

We choose to represent events by boolean-valued functions of two parameters:

f (now, prev),

where the parameters *now* and *prev* are both cycles (moments) during the execution of a particular program, and where we always have *prev* $\leq$ *now* (in the sense that *prev* is an earlier moment than *now*). If, speaking heuristically, f represents an event E (in general, E will have some certain time duration, which may however be as short as one cycle; and E can occur repeatedly) then f(now, prev) will have the value true for all moments *prev* prior to *now* at which the event E transpires.

An event can either be a *primitive program event* or a *composite*. An example of a primitive program event is 'at ℓ', where ℓ is a label or program point more generally; this event transpires when control reaches the label ℓ.

An example of a composite event is f_1 and f_2, where f_1, f_2 are both events. This transpires when both events occur simultaneously.

If it were always true that $f(n_1,p) = f(n_2,p)$ when $n_1 \geq n_2 \geq p$, then single-parameter boolean-valued functions instead of two-parameter functions could be used to represent events. As an example showing that this is not always the case, consider the composite event

$$g = \text{and } f;$$

where f is an event. This has the definition

$$g(n,p) = f(n,p) \text{ and } (n \geq \forall p' > p \mid \text{not } f(n,p')).$$

Other useful elementary and composite events are as follows.

Elementary events:

at ℓ: control is at the program point ℓ.
call r: control enters the routine r.
return r: control returns from the routine r.
e : e is an occurence of a boolean expression. As an event, it transpires when this expression, evaluated in its currently active environment, has the value true.
changes e: e is an occurence of a boolean expression. This event transpires when the value of e, evaluated within its currently active environment, changes.
within rb: control is within rb (a routine or block)
evaluated e: e is an occurence of an expression. This event transpires whenever e is evaluated.
assignto v: v is an occurence of a variable. This event occurs whenever an assignment with v as target is executed.

Composite events and their definitions:

g = not f:	g(n,p) = not f(n,p)
g = f_1 and f_2:	g(n,p) = f_1(n,p) and f_2(n,p)
g = f_1 or f_2, etc:	g(n,p) = f_2(n,p) or f_2(n,p), etc.
g = end f:	g(n,p) = f(n,p) and (n $\geq \forall p' > p \mid$ not f(n,p'))
g = start f:	g(n,p) = f(n,p) and (p $> \forall p' \geq 0 \mid$ not f(n,p))
g = after f:	g(n,p) = not (n $\geq \exists p' \geq p \mid$ f(n,p))
g = after(k) f:	g(n,p) = p $\geq \exists q_1, p_2, q_2, \ldots p_k, q_k \mid q_1 \geq p_2 \geq q_2 \geq \ldots \geq p_k \geq q_k$ and f(n,q_1) and not f(n,p_2) and f(n,q_2) and not f(n,p_2) and ... not f(n,p_k) and f(n,q_k).
g = f_1 before f_2:	n' = if n $\geq \exists p > 0 \mid$ (f_2(n,p) and p $> \forall p > 0 \mid$ not f_2(n,p)) then p else if f(n,n) then 0 else n; g(n,p) = f(n',p);
g = endings f:	g(n,p) = f(n,p) and not f(n,p + 1)
g = startings f:	g(n,p) = f(n,p) and not f(n,p - 1).
g = last f:	g (n,p) = f(n,p) and (not n $> \exists p' > p \mid$ f(n,p') and not f(n,p' - 1))
g = first f:	g(n,p) = f(n,p) and (not p $> \exists p' > 0 \mid$ f(n,p') and not f(n,p' +1)
g = last(k) f:	g(n,p) = f(n,p) and (# {p', n $\geq$ p' $>$ p | f(n,p') and not f(n,p' - 1)}) le k-1
g = first(k) f:	g(n,p) = f(n,p) and (#{p' | n $\geq$ p' $>$ 0 | f(n,p') and not f(n,p' + 1)}) le k-1
g = (k) last f:	first last(k) f
g = (k) first f:	last first(k)
g = f_1 following f_2:	g = f_1 and after f_2.

Note that last f is true during the last continuous period within which f is steadity true; that last(k) f is true in period p if there do not exist more than k continuous time periods including and subsequent to p (but prior to n) during which f is true, and that (k)last p is the k-th from the last time period prior to n during which f is true, if there exist k such periods; otherwise (k)last f degenerates to first f. The operations first, first(k), and (k)first can be described in a similar way.

If the operation cycle n is such that f(n,n) is true, then we shall say that f occurs at n.

The dictions that have been introduced can be compounded in obvious ways. Thus e.g., we can write

```
at ℓ and after(3) (in b following (2)last at ℓ)
```

to describe moments at which control returns to ℓ after having entered and left the block *b* at least three times since ℓ was last visited.

Dictions of this type are bound to be useful in informal debugging. They enable us to call for dynamic checks, program dumps,etc., at carefully specified program moments, as e.g. by writing

```
if at ℓ and after(3) (in b following (2)last at ℓ) then
                        print various_variables;;
```

or

```
assert (at ℓ and not after (in b following (2)last at ℓ)
                        implies some-proposition;
```

or

```
assert not some-event-thought-to-be-impossible;
```

etc. In the remainder of this newsletter, we shall outline a more systematic approach to this way of using the event-oriented dictions that have just been introduced.

2. 'Significant program events' and systematic debugging.

'Systematic debugging' may be defined as debugging which aims to make some substantial part of a full program correctness proof manifest, and which then goes on to check the Floyd assertions which thereby appear. A reasonable procedure to use in systematic debugging is as follows:

(a) Write out a careful but informal correctness proof for the program PR being examined. The following example (culled from Newsletter 155) illustrates the dictions which can be expected to appear in such a proof:' Note that one statement after the point at which we exit from the $\forall$n-iteration, assertion (K) reduces to (∈). On entry to the $\forall$n-iteration, (κ) follows from (∈),since allsorted = allsorted + halfsorted (1:ntaken) will just have been executed. Only the last statement of the *mergein* routine changes its *vect* argument, and from the form of this statement it is evident that on exit range(vect) has become the union of the entry of range(vect), plus {elt}... .

(b) Using the program event language described in the preceeding pages, produce formal phrases describing each of the events alluded to explicitly or inplicitly in the informal proof developed as step (a), and add these event descriptions to a 'significant events list'. Each significant event description should be accompanied by an auxiliary list of conditions to be checked when the event occurs.

(c) A program debugging system should be able to accept the significant events list and the attached condition lists built up during step (b). It should be able to use these lists during debug runs of the program PR, specifically to produce both diagnostic statements wherever any condition belonging to an auxiliary list is not met, and also after a run to produce a list of all significant program events which have never occurred. The debugging system should be capable of passing lists of occurrences from one debugging run to another; then when a final listing is generated, only significant events that have never occurred in any of a series of debugging runs need to be printed.