SETL Newsletter # 167 | J.T. Schwartz
April 14, 1976

A Variant SETL Implementation Incorporating 'Whenever' Dictions.

This newsletter will outline a way of implementing simulation-oriented dictions like those described in NL 157 in a fairly straightforward extension of the planned new (optimized) SETL implementation. At the same time, some dictional revisions will be suggested.

The (revised) semantics to be supported are as follows:

i. *Process* objects are introduced. These are treated essentially as pointers to internal environments, which can *execute*, and which can *wait* for particular conditions to arise before they execute. The components of a process are its *internal stack*, *instruction location index*, and *termination flag*.

ii. Static variables of one process may be modified by another. Specifically, if p is a process-valued expression, and v is a static variable, then p.v can be used within any other process to access the current value of v in the process p. Three special static variables, *initialvariable*, *processpriority* and *errorprocess* present in every process, will play special roles. As a syntactic convention, we assume these to be public variables of a module *processystem*; thus they can be aliased for convenience of reference.

iii. Processes can be copied. The operator which does this is

copy p.

The value of copy p is a new process p' all of whose variables (including internal stack and instruction location index) have the same value as they have in p. If p is terminated, then copy p is equal to p.

If p executes copy self the p' it creates will appear to have just completed the copy operation, and to have received p' as the value of this operation. If the process p is executing an await operation (see below), then its copy p' will be ready to execute, and will appear to be just about to execute the await operation.

iv. Two forms of await diction are provided. The simple await has the form

(1) await C;

and is an executable statement. The component await has the form

(2) await $(C_1, C_2,\ldots,C_n)$

and is a function. Here $C, C_1, C_2,\ldots,C_n$ are boolean-valued expressions whose evaluation should have no side effects, and which should access no global variables not appearing expliciting in $C, C_1,\ldots,C_n$ These statements suspend execution of the process executing them until C (resp. one of $C_1,\ldots,C_n$) is satisfied.

v. The nulladic primitive self returns the value of the process executing it.

vi. The primitive destroy erases the internal environment of a process (which may release other objects for garbage collection), and sets the process termination flag.

vii. The primitive restart p resets the instruction location counter of p to the first location of p's 'main program' code, and clear p's internal stack. Moreover, p is made ready (see below). This primitive cannot be applied to p if the terminate bit of p has been set. Note that this primitive is provided only for efficiency; it provides no essential semantic capability that cannot be duplicated using the copy primitive.

viii. The primitive eval acts on a vector $<f, x_1,\ldots,x_n>$ whose first component is a function of n parameters, and returns $f(x_1,\ldots,x_n)$ as its value.

ix. If an error occurs during the execution of a process p whose *errorprocess* variable has as its value a process q, then the termination flag of p is set, and the value of q. *initialvariable* is set to p.

x. The (purely syntactic) *subordinate* dictions introduced in NL 157 are retained.

<u>Backtracking dictions can be represented</u>.

It is worth noting that the semantic mechanisms that have just been described can be used to realise the generalized backtracking primitives match these of NL 166 closely enough. The monadic and dyadic <u>try</u> primitives of NL 166 can be represented as follows: Introduce a global variable called *flagvariable* (so that a call with this name will be present in every process) the dyadic <u>try</u> is

```
definef env try val;
flagvariable = false;
env. initialvariable = val;
env. flagvariable = true;
await flagvariable eq true;
return initialvariable;
end try;
```

The monadic <u>try</u> operator is

```
definef try val;
flagvariable = false;
env = copy self;
env. initialvariable = val;
env. flagvariable = true;
await flagvariable eq true;
return initialvariable;
end try;
```

The additional 'environment tree' related mechanisms of NL 166 are not necessary here, though of course they can be provided simply by including appropriate code in an appropriate group of routines used to represent the NL 166 primitives. Thus the primitives described in the preceeding pages are more powerful than those of NL 166. However, it must be noted that the NL 166 primitives are designed to be efficiently implementable in the context for which they are intended, namely one in which new environments are frequently generated and then abandoned after a few cycles of exploratory execution when they are seen to represent dead ends. In such a situation, one wants to minimize environment entry/exit costs, which is done by relating each environment *env* to a 'parent' *env'*, and representing *env* as a set of differential changes to *env'*. The efficient realization of this idea imposes semantic restrictions, in particular, we assume in NL 166 that the parent *env'* cannot be executed while it still has undestroyed descendants *env*. In the more general semantic framework described in the present newsletter all environments are independent, which at the implementation level probably implies that the internal stack of an environment *env* will be copied completely whenever we execute the primitive <u>copy</u> *env*.

<u>Implementation Considerations</u>.

The semantic structure that has just been described can be implemented in much the same way as standard SETL, except that process stacks must be kept as vectors in a garbage-collected area; moreover, stores to variables that appear in <u>await</u> statements (we shall call these 'await variables') must be handled in a special way. A plausible implementation is as follows: the system will maintain a *ready list* of processes able to execute (in the sense that they are not currently poised at an unsatisfiable <u>await</u>).

The ready list will be always kept sorted into order of diminishing priorities, and may consequently be represented by some data structure, e.g., balanced trees, particularly suitable for this purpose. With every await variable a *process list* will be associated; every process not on the ready list which is awaiting a change in the value of the variable will appear on this list. With each process p we will associate a *referencing item* list; all process list items representing p will be referenced on this list. Conversely, each process list item ℓ will point back to the referencing item i which refers to ℓ.

Whenever the value of an await variable v is changed, all the non-ready processes p appearing on its process list will be made ready; a *ready bit* will be set in each such p, the process list of v will be made null, and the referencing item of p's referencing item list which points to the process list item that points back at p will be deleted. Execution will then continue with the ready process of highest priority. When a process begins to execute, its referencing item list *ril* will be examined, all process list items located by the items of this list will be deleted, and *ril* will be cleared.

The restart primitive has an obvious implementation.

None of these interpretation rules are particularly problematical. The garbage collector's marking activity should begin with the stack vectors of all the processes on the system ready list; to make all other processes easily available, it may be worth maintaining a more comprehensive non-terminated process list.

A remark on optimization.

Objects will have to be copied more often in the multiple process environment that has been described than would be necessary in a monoprocess environment.

A reasonable scheme might be to copy x on each assignment b = x (or 'incorporation', e.g., b = {x}, for which the variable b belongs to a different environment than x. If this rule is followed, copying on other assignments and incorporations will be governed by standard monoprocess rules. However, it would be interesting to derive more efficient copy rules, which it is probably not hard to obtain using standard global optimization techniques.

<u>Languages of Mechanism</u>.

It is worth noting that the semantic primitives outlined above can be used to capture much of the semantics of what might be called 'language of mechanism', albeit in a way somewhat lacking in polish. More specifically, we can represent mechanisms with active parts by using a separate process to represent each of the elementary parts of the mechanism; interconnections between parts can be established by transmitting appropriate groups $\{p_1,\ldots,p_n\}$ of processes as parameters to other processes p which need to be connected to $p_1,\ldots,p_n$. 'Subassemblies' out of which more elaborate mechanisms are to be created can be represented by subroutines which accept external connections as parameters and which generate the processes which represent the internal elementary parts of a subassembly. The following linkage convention can be employed by a process p wishing to make use of a mechanism (e.g., to calculate some quantity which p requires):

a. *p* will call an 'activating' subroutine *ar*. *ar* will transmit parameters to appropriate entry processes of the mechanism.

b. *ar* will then <u>await</u> the appearance of the desired result within the mechanism; note that this activates the mechanism, whose component processes can execute at lower priority than *p*.

c. When the result has appeared, *ar* will call a mechanism-associated restart routine, which will re-initialize all the parts of the mechanism for subsequent use. Then *ar* will return.