SETL Newsletter # 145

P. Shaw
March 1, 1979

GYVE-oriented Inter-Process Coordination and Control Features for an Extended SETL. (SETLG)

(Preliminary Notes)

1. Introduction

This newsletter sets forth an initial proposal for the incorporation of GYVE-like capabilities, i.e., facilities for the coordination and control of multiple parallel processes, into SETL. The exposition which follows assumes general familiarity with the GYVE language described in the forthcoming doctoral dissertation of its author. The system to be described should provide powerful, sound, and fully protected mechanisms for the description of operating systems.

To summarize the approach which will be explained in more detail below: we propose to leave the SETL data types, aggregates, operations, and procedures unchanged; to add keyword and optional keyword syntax to procedures; to add a new storage class, global, and one new data type, pointer (to a global object). Global objects will be created and destroyed explicitly, and will be referenced by create-returned pointer's. In the extended system envisaged, all current SETL data will be process-local, and will be gargabe collected as currently; pointer's will be treated like SETL atoms and thus will be valid set members, tuple components, etc. In addition, we propose to add GYVE MODEs and COMPILATIONs to the extended SETL system, following the current SETL procedure syntax as closely as possible; to perform all protection-checking dynamically, as in current SETL; to omit GYVE aggregates and to add built-in modes and procedures for GYVE system objects and operations (which is also how they are presented in GYVE).

The resulting language, provisionally designated as SETLG, will be in close conceptual correspondence with GYVE, at least in regard to parallel process semantics; and operating systems specified in SETLG should transliterate in relatively mechanical fashion to GYVE. Note in this connection that SETLG will not really be a perfectly general 'operating system specification language', in which description of the totality of all conceivable operating systems is facilitated; rather, it will be made easy to write systems, functionally general, but all structured in the style which GYVE suggests.

The major loss which will be incurred in going from a GYVE to a SETLG version of an operating system is run-time efficiency. This will be balanced by a gain in ease of use, which of course reflects an underlying difference between SETL and GYVE objectives.

2. Syntactic and Semantic Facilities of SETLG.

a. Keyword and Optional Keyword Parameter Syntax

It is useful add the GYVE syntax for keyword and optional keyword parameters to SETLG, in the following manner shown in the following example (where as in GYVE brackets define optional parameters)

```
define execute(ps) runtime(t) usedtime(ut)
                   [reservelimit(rl)] requested_reserve(rr)
                   [port(pt)] [priority(pr)] [message(m)]
                        [post(pst)]
                        [interruptpriority(ip)]
                        [interruptmessage(im)]
                   result(r);
```

To see how useful this syntactic sugaring really is, compare

```
execute(ps,300,ut,,rr,,,,,,,r);
```

with

```
execute(ps)runtime(300) usedtime(ut)requestedreserve(rr)result(r)
```

As in GYVE we permit the keyword *k* attached to an optional procedure parameter to be used within the body of the procedure as a synonyn for 'the parameter designated by *k* has actually been supplied'.

b. Storage Classes

It is proposed that all of current SETL storage be left as-is, but defined to be strictly intra-process. Thus, SETL storage would correspond to GYVE Local and Private storage--- except that SETLG objects would, like SETL, be created implicitly, variable in size, and garbage collected. In particular, this means that any current SETL program could be invoked by a SETLG process.

Enforcement of the intra-process rule will be patterned after the GYVE enforcement except that it will be dynamic rather static. The essential aspects of this rule are:
(i) at the implementation level, we must ensure that assignments to base variables of shared instances are on a rigorous "by value" basis, no dangerous implementation level pointer shortcuts being allowed;
(ii) all process INITCALL arguments must be transmitted "by value" i.e., with copying of all data objects; this approach ensures that no process is able to obtain access to the private data of another process. An alternative technique of enforcement of the privacy rule is to stamp every object with the 'id' of the process which creates it, and to check every reference to guarantee that the referencing process is the same as the creating process; this seems a less desirable technique.

c. MODEs

This key structural mechanism of GYVE should fit into SETL quite easily. Figures 1a and 1b below gives a typical GYVE 'MODE' example, namely a "queue" object type, and Figures 2a and 2b show the corresponding SETLG version.

The differences between the GYVE and the SETLG MODE syntax and semantics may be described as follows:

i. Mode header

We pattern the SETLG MODE syntax after the syntax of SETL procedures, but with the keyword "mode" replacing "define". We include provision for parameters (and also keyword parameters as described in Section 1). Note however that in practise SETLG MODEs would use fewer parameters than semantically similar GYVE MODEs, because some of the parameters of GYVE modes are generally used to state dimensions or string lengths for base variables, whereas in SETLG we will permit base variables to vary in length like other SETL variables. The other use of MODE parameters, which will still be needed in SETLG, is for the transmission of initial values.

ii. Creating MODE Instances.

This is done using a *create* statement:

create (*mode*) in (*account*) spacelimit (*nbytes*) set (*pointer*) [result(r)];

the *result* parameter is optional. The 'spacelimit' parameter, which does not occur in GYVE, defines the total amount of space, in implementation-defined 'byte units', which the mode instance is allowed to occupy. Base data objects of the instance will reside in, be manipulated in, and be removed by a garbage collector from a block of space never exceeding this size. Any operation which would violate this condition is said to cause a space fault, with consequences to be defined below.

iii. INIT entries

GYVE INIT entries are written

((INIT)): ENTRY; BLOCK END ENTRY;

Entries of this kind are always parameterless: they are automatically invoked when a MODE instance is created, and can

reference the MODE parameters. SETLG should provide this facility, probably with the syntax

initially
block
end initially,

like that currently used in connection with SETL procedures.

iv) Base Variables, Events

Here we can use a declaration giving a simple enumeration of base variable names,e.g.

base b, in, out, q;

Different modes should be permitted to use the same base variable names; base variables must be accessible (only) to the entries of the mode.

In GYVE, an EVENT is a special class of PRIVATE MODE; objects of this MODE can only be declared as base variables of some other MODE, i.e.. cannot be created by the normally available CREATE statement. In SETLG, we propose to declare variables of type EVENT by statements of the syntactic form

events $e_1, e_2, \ldots, e_n$;

Quantities of this type are automatically initialised to an empty condition; they are then addressable by the two primitives

block e; and wakeup e;

whose semantics are the same as in GYVE. This convention keeps each event object strictly local to some mode instance. We then propose to omit the GYVE 'PRIVATE MODE' construction; this construction is useful in GYVE for the data structuring facility it embodies, but not appropriate for SETLG, which like SETL approaches data structures in quite a different way.

v) Entries

A syntax like that of the present SETL 'define' statement can be used, e.g.,

```
entry insert(c) [result(r)];
```

The use of "entry" rather than "define" as a keyword will help readability. The other clauses of a GYVE entry header are the optional RETURNS and READER clauses. The SETLG syntactic convention can incorporate this information into a keyword, as ,e.g.,

```
entry p(l);
readerentry q(j);
readerentryf r(k);
```

d. Protection.

The most essential element of the GYVE protection scheme is that which prevents a pointer p to a mode instance from being assigned (or otherwise transmitted) to any variable x declared to be a pointer having more 'rights' than p. In SETLG we propose to provide the same protection dynamically. More specifically, every pointer p to an instance of mode M will carry with it a list of all the entries of M which are accessible through p. Then on every attempt to use an entry via p (in the syntactic form p.e(x) which we take over from GYVE) this list will be checked, and if e is not available through p the invocation p.e(x) will be treated as a fatal error leading to process termination. To allow p to be used to create a pointer p' to the same instance as p but with a shorter list of entries, we allow the construction $p' = p[e_1,\ldots,e_n]$, where $e_1,\ldots,e_n$ name the entries to p', all of which must of course be available as entries to p.

It is also convenient to provide the following primitives for use with modes and pointers:

p has e

returns true or false, depending on whether the entry e is accessible through p;

m has e

returns true or false, depending on whether the mode m has e as one of its entries; and

p isof m

returns true or false, depending on whether or not p points to an object of mode m. In addition, we allow the operations eq, ne for pointers as for blank atoms; and also gt, lt, ge, le, etc., with implementation-defined but consistent boolean values.

GYVE has three protection-oriented mechanisms: access rights, sub-aggregate references, and restricted members and entries of compilations. For reasons to be explained below, we propose to abandon the second of these mechanisms in SETLG, but to keep the third. The access-rights mechanism in GYVE has two aspects: access to entries of mode instances, and read/write access to variables, and especially to procedure parameters. A mechanism restricting access to entries of mode instances is essential, and will be provided dynamically in the manner explained above. Read/write restrictions are by contrast matters of convenience and discipline. They can be provided in the following form in SETLG: a procedure parameter marked :write, as in

define rout (x:write, y:read, z);

will be initialised to Ω on procedure entry, rather than being initialised from the value of the corresponding actual argument (in the calling routine); a parameter marked :read will be initialised on entry (transmission being by value, as always in SETL), but not transmitted to the calling routine on sub-procedure return. For compatibility with the present SETL convention the rule .. applying if nothing is said will be :read/write (both read and write privileges), rather than :read as in GYVE.

e. Space Overflow.

Space overflow is a much more substantial problem in SETLG than in GYVE, since SETLG objects can vary in size in totally dynamic ways, and since the number of implementation level 'bytes' required for the storage of any particular SETLG object will generally not be obvious. Whereas in GYVE only CREATE operations can lead to a space overflow, in SETLG any number of other operations, e.g., tuple concatenation, set formation, element insertion, etc., can require space and hence cause overflow. To deal with this problem, we propose the following conventions:

i. The formation of objects such as sets and tuples (of arbitrary size) will be assumed to be carried out in a nominal 'system workarea', of potentially infinite size. Consequently, no such operation will generate a spacefault *while still in progress* (unless the operation has assignment side-effects; see below.)

ii. The instance or process to which an object belongs will be 'charged' for the space required to form a new object only upon, but instantaneously upon, the assignment of this new object to a base variable or to a local variable of the process. Any assignment will be triggered by an occurence of one of the operators '=' or 'is'. This convention serves nicely to 'localise' the points at which a space overflow fault is detected, the instance or process storage space in which

it seems to have occurred will be marked for garbage collection; this will determine whether the apparent fault is real. If not, execution can continue. If a fault occurs, the assignment which causes it to occur will be suppressed (but side effects which preceed the assignment in point of time will persist.) What happens after a space fault will depend on whether or not an *overflow fault clause* has been attached to the assignment statement which gives rise to the fault. Such fault clauses, which must always preceed the terminating semicolon of the statement to which they are attached, have the syntactic form

$$:\text{overflow}(\ell_1, \ell_2, \ldots, \ell_n),$$

where each ℓ_j is a label, and no more labels are supplied than are called for in view of the number of '=' and 'is' operators which occur in the immediately preceeding statement. An example would be

$$a = b + c \ \underline{\text{is}} \ d + e \ :\text{overflow}(\ell_1, \ell_2);$$

Each label in a fault clause refers to some assignment operation, and defines the point to which transfer is to be made if an overflow fault occurs at the assignment to which it refers. Any of the labels of an overflow fault clause may be omitted e.g., we may write $:\text{overflow}(\ell_1,,\ell_2)$; such an omission causes the corresponding overflow fault to be handled as a termination.

Note that very similar rules apply to 'existence faults', i.e., invocations p.e(x) of some entry of an instance which has been destroyed previous to said invocation; except that destroy fault clauses are attached not to assignments but to operations which invoke entries of mode instances, and are written as

$$:\text{destroyed}(\ell_1)$$

Note, in connection with the garbage collection operation which preceeds the occurence of a space fault, that every SETLG value is strictly local to some process or mode instance, i.e., none of the implementation level pointers which structure the value point from within an object belonging to one instance to an object or object part belonging to some other instance. This isolation is guaranteed by forming a complete copy whenever an object is passed as an argument from one instance or process A to another B and assigned to a variable of B. It is a consequence of this rule that the garbage collection operation which preceeds a space fault occuring in A only needs to process the space belonging to A, and not any larger part of the total memory area managed by the SETLG system.

As an aid to the management of SETLG objects of unpredictably variable size, we provide monadic operators size, sizel, and sizeleft. The size operator calculates the number of bytes needed to store a complete, fully independent, copy of its argument. The sizel operator calculates the number of bytes needed to store an independent'1-level' copy of its argument (where, e.g., a 1-level copy of a set duplicates the implementation-defined 'body' of the set, but does not duplicate its elements.) The sizeleft operator can be applied to a process or mode instance, and measures the amount of free space still available in it.

Figure 1a: GYVE definition of a'QUEUE'MODE

```
QUEUE: MODE(N:INT)LIMIT(L:INT);
     DECLS;     B:(L) CHAR(N);
                IN:INT:INIT(1); OUT:INT:INIT(1); Q:INT;
                FULL_EVENT:EVENT; EMPTY_EVENT: EVENT;
     END DECLS;
INSERT: ENTRY(C:CHAR) [FULL(L:LABEL)];
     IF Q = EXTENT(B)  THEN IF FULL THEN GOTO L;
                                    ELSE BLOCK(FULL_EVENT);
                                    END IF;
                       END IF;
     B(IN) = C;
     IN = IN // EXTENT(B) + 1; Q = Q + 1;
     WAKEUP(EMPTY_EVENT);
     END ENTRY;
REMOVE: ENTRY(C:CHAR:W) [EMPTY(L:LABEL)];
     IF Q = 0 THEN IF EMPTY THEN GOTO L;
                           ELSE BLOCK(EMPTY_EVENT);
                           END IF;
                END IF;
     C = B(OUT);   OUT = OUT // EXTENT(B) + 1; Q = Q - 1;
     WAKEUP(FULL_EVENT);
     END ENTRY;
PURGE: ENTRY; Q = 0; END ENTRY;
COUNT: ENTRY: RETURNS(INT):READER; RETURN(Q); END ENTRY;
SIZE: ENTRY: RETURNS(INT):READER; RETURN(EXTENT(B)); END ENTRY;
END MODE;
```

Figure 1b: Use of 'QUEUE' MODE

```
PQ: QUEUE;
C:CHAR(80);
CREATE(QUEUE(80)LIMIT(10)) IN(A)SET(PQ)RESULT(R);
---
PQ .INSERT('ABC');  PQ.INSERT('CDE')FULL(L);
---
PQ .REMOVE(C);  PQ.REMOVE(C)EMPTY(L);
```

```
I = PQ.COUNT;
PQ.PURGE;
```

Figure 2a: SETLG version of queue mode.

```
mode queue (sizelim);
     base buf, sizelimit;
     events fullevent, emptyevent;
initially
          buf = nult; sizelimit = sizelim;
end initially;
entry insert(c:read) [result(r)];
     if size c gt sizelimit then
                    terminate;
     else
                    b = b + <c>; overflow(ofl);
                    wakeup emptyevent;
     end if size;
ofl: if result  /* i.e., if 'result' parameter is supplied */ then
                    r = valuerepresentingoverflow; return;
     else
                   block fullevent;
     end if result;
end entry insert;
entry remove(c:write)[result(r)];
     if b eq  nult  then
          if result then
               r =  valuerepresentinginderflow; return;
          else
                    block emptyevent;
          end if result;
          else
                    c = b(1); b = b(2:);
          end if b;
```

```
end entry remove;
entry purge;  b = nult; end entry purge;
readerentryf count; return #b; end readerentryf count;
readerentryf sizelim; return sizelimit: end readerentryf sizelim;
end mode queue;
```

Table 2b: SETLG uses of the queue mode.

```
create (queue(40)) in (a) spacelimit(400) set (pq) result(r);
...
pq .insert('abc');
pq .insert({'abc', 'def'}) result(ok);
...
pq .remove(c);
pq .remove(c) result(isunder);
...
i = pq .count;
pq .purge;
...
```

f. Compilations

The GYVE "COMPILATION" object will be retained in SETLG. It is very convenient for namescoping, and quite useful for protection (via the restricted members and entries mechanism). Syntactically, it will be written as

```
compilation;
/* a list of SETLG define's and modes */
end [compilation];
```

In GYVE, restricted members and entries of a compilation are denoted by parenthesizing member, and entry identifiers, such as

```
P:    PROC(I:INT);
(P2): PROC(I:INT);
M:    MODE;
E:    ENTRY(I:INT);
(E2): ENTRY (I:INT);
```

Perhaps the best approach for SETLG would be to add the keyword 'restricted' at the end of the header line of an entity, as in

```
define p(i);
define p2(i) restricted;
mode m;
entry e(i);
entry e2(i) restricted;
```

g. Checking for instance existence, entry availability, etc.
All checking in SETLG will be performed, as in SETL, dynamically. Eg., upon the call

```
pq .insert('cde') result(r);
```

the following runtime checks will made:

pq must be a pointer to an instance;
the mode of that instance must have an entry named "insert";
this entry must have a single positional parameter, and a keyword parameter group named "result" with a single parameter;
the "insert" entry must have no return value ---i.e., is not an "entryf";the "insert" must have no other required keyword parameters. Furthermore, during execution of the "insert" entry, all references to parameters, and of course to other variables also, will be checked at time of reference.

Note that all of these checks can give rise to run-time errors which are impossible in GYVE. All such errors can be handled by TERMINATEing as is done in GYVE for possible run-time errors such as subscript range error.

Many of these checks can be made more efficient by precompilation. For example, all the tokens encountered as entry names during the compilation of a complete set of programs can be collected in a single comprehensive table and converted into integers, thus substantially speeding up the search which must be performed when a named entry e of an instance pq is called.

h. Aggregates

It is suggested that GYVE aggregates be ommited entirely from SETLG. Justification is as follows:
GYVE aggregates are storage aggregates, and hence rely heavily upon data descriptions and declarations, whereas SETL aggregates are aggregate values. These two approaches to aggregates would be difficult to combine, and would probably lead to an unpleasantly redundant system of data aggregates. GYVE aggregates are basically oriented towards efficiency: i.e., toward realising savings in storage and in global address-table space by collecting numerous small objects into a single global object; and toward the reductions of global address-table references and the elisions of existence checks which the GYVE "attach" mechanism make possible. Such savings are less meaningful in the specification-oriented, deliberately "inefficient" SETL environment.

i. Valves; 'Quiesce' and 'Dequiesce'. Other built-in system modes.

The GYVE 'VALVE' mechanism allows pointers to be made inoperative even after they have been issued, and subsequently to be made operative again if desired. This construction can be carried over to SETLG with little change in syntax and none in semantics. We propose the syntax

create (valve(n)) in (acct) set (pv) [result(r)];

for the creation of a valve able to hold n pointers, and

link(p) through(pv) set(p2) [result(r)];

for the operation which subordinates *p* to *pv*, creating the object *p2* which may be used in place of p while *pv* remains open. To shut *pv* and to reopen it, we write

quiesce(*pv*) [result(x)];

and

dequiesce(*pv*) [result(x)];

respectively. Note that if *pv* is destroyed, the object p2 becomes permanently unusable.

The quiesce and dequiesce operations apply to other mode instances also. If an instance is quiesced, no entry may be made to it (though processes which have already entered are allowed to complete). To quiesce an instance, one needs access to *all* its entries; the same rule applies to destroy.

The pointer, global object, and MODEs mechanisms allow system objects and operations like those of GYVE to be defined for SETLG. Indeed, the system objects of GYVE itself are explicitly defined by a 'built-in compilation'. (Of course, the procedures, modes, and mode entries of that "system compilation" occasionally use 'very primitive' operations which are unavailable in the "overt" language, such as an operation of the "test and set" kind for defining primitive atomicity of process; moreover, they occasionally reference system components which are inaccessible in the "overt" language, such as global address tables). In much the same way, the system objects and operations of SETLG would be defined by a built-in compilation. SETLG, like GYVE, should permit reference to the modes and procedures of that compilation by simple name. Thus SETLG statements such as:

```
define p(a);
...
create(process space(20000)) in(a) set(ps) result(r);
...
initcall(f(3) for(j)) in(ps) result(r);
...
create (port posts(1) messagelimit(10)) in(a)set(pt) result(r);
...
execute(ps) runtime(500) usedtime(ut)
     port(pt) priority(1) message(13)
           interruptmessage(im) interruptpriority(pr)
     reservelimit(rl) reserverequest(rq)
     result(r);
```

will be available.