Behavior Research Methods & Instrumentation
1982, Vol. 14 (2), 254-263

SESSION XIV LABORATORY APPLICATIONS

PASTOR: A new schedule programming language

JUERG ELSNER
Institute of Toxicology, Swiss Federal Institute of Technology, Zurich, Switzerland

A new schedule programming language is presented, including its implementation on a PDP-11 running under RSX-11M. This language combines modern syntax concepts of PASCAL with the state notation of SKED. The present implementation includes event-driven activation for fast response time and shared code for minimal storage requirements, allowing the simultaneous control of many experiments.

In the many years of its existence, SKED[1] (Snapper, 1976) has enjoyed an ever increasing popularity in behavioral research. The main reason for this success seems to be its state notation, which is as close as it can be to the behavioral scientist's way of thinking. Little effort is needed to translate an experimental concept of a schedule into a SKED program. As SKED is a specialized language with a limited set of different instructions, it is easy to learn and its implementation may well be optimized.

The major problem one encounters in programming in SKED is the lack of structure within a composite output statement. The compiler's interpretation of one's program is often unpredictable. For the same reason, it is difficult to write a compiler for SKED, as its syntax does not follow some basic rules necessary for using standard compiler techniques.

Computer programming language developments have been very active in the past few years. Emerging from the structured program principles promoted by the PASCAL language (Jensen & Wirth, 1975), a multitude of new languages have been defined, some of which include process control features, such as MODULA (Wirth, 1980) ADA (Johnson, 1981), and FORTH (Dessy & Starling, 1980). These languages present the advantage of very concise syntax rules, forcing the programmer to have clear concepts of what he is doing and to write readable programs that document themselves.

These languages departed from traditional state notation concepts implemented in SKED by using the notion of parallel processes. A process is defined as a structured program module whose execution may be started or stopped by other processes. The program flow may be delayed at a point until one (and only one) specified event occurs (time or I/O). This way of thinking presents not only advantages. The communication between processes often becomes quite cumbersome, and the memory and programming overhead is generally high. High sophistication and training are also demanded from a programmer using these languages.

A state notation combined with the possibility of defining parallel state sets implicitly allow the definition of parallel processes in the same way as the modern languages enforce it. It seems to be an optimal compromise to combine SKED's limitations of a specialized language and its state notation with modern syntax concepts that allow structured programming down to the lowest language level. This compromise has been created in the PASTOR language, which also introduces several new features that allow one to use novel programming techniques.

THE PASTOR LANGUAGE

A sample program written in PASTOR is presented in Figure 1. One may note that most of the basic features contained in SKED have been maintained: A program is structured in state sets, which in turn are composed of states. The main difference from SKED lies in a concise PASCAL-like syntax. The analysis of this variable-interval schedule is left to the reader, as it is claimed that a PASTOR program documents itself. The following discussion of PASTOR's syntax and semantics may resolve any arising doubts.

The PASTOR Syntax and Semantics

Figure 2 represents the syntax rules of PASTOR using syntax definition techniques described by Wirth (1977). A schedule is composed of a schedule name, a device-type definition, and a sequence of blocks. A block is an optional series of declarations followed by a state set.

An identifier is defined in the same way as in

Copyright 1982 Psychonomic Society, Inc. 0005-7878/82/020253-11$01.35/0

Variable Interval Schedule

```
SCHED vie;
DEVICE sb;
RESP lever=LV1+LV2 END;
STIM llight=LL1+LL2; rnfenbl=VS1+VS2; rnf=VLV END;
CONST vilst =
    (26", 66",  1", 35", 20", 90", 42",  4", 32", 80",
     13", 61", 40", 53", 23", 10", 22", 72",  2",  5",
     24", 50",  7",  7",  9", 19", 57", 37",  3", 44",
      4",  2",  8", 18",135", 14", 30", 12", 34", 17",
     47",  6", 15", 27", 11",105", 11",  1", 16", 29");
     nvi=50; rnfdly=0.1"; sesstim=30'; maxrnf=50 END;
VAR lx,vival END;
SYNCH reinf END;

STSET sessctl;
  STATE session; start; lx:=0; SEND llight;
    PARALLEL
      AFTER sesstim DO SWITCH TO waitidle;
      WHEN reinf maxrnf TIMES DO SWITCH TO waitidle
    END
  END;

  STATE waitidle;
    IF (lever CLEAR) AND (rnfctl IN rnfidl) THEN stop;
    PARALLEL
      WHEN lever CLEAR DO SWITCH TO waitidle;
      WHEN rnf CLEAR DO SWITCH TO waitidle
    END
  END
END;

STSET victl;
  STATE viact; SET rnfenbl;
    lx:=lx+1; IF lx>nvi THEN lx:=1; vival:=vilst[lx];
    WHEN lever SET DO
    BEGIN
      SIGNAL reinf; CLEAR rnfenbl; SWITCH TO rwait
    END
  END;

  STATE rwait;
    WHEN lever CLEAR DO SWITCH TO viidle
  END;

  STATE viidle;
    AFTER vival DO SWITCH TO viact
  END
END;

STSET rnfctl;
  STATE rnfidl; CLEAR rnf;
    WHEN reinf DO SWITCH TO rnfact
  END;

  STATE rnfact; SET rnf;
    AFTER rnfdly DO SWITCH TO rnfidl
  END
END.
```

Figure 1. Sample schedule written in PASTOR. This schedule controls a variable-interval schedule with even reinforcement probability over time. The state set "sessctl" controls the session duration: After 30 min or 50 reinforcements, the state "waitidle" stops the schedule if there is no response and if no reinforcement is active; otherwise, it waits for such a situation. The state set "victl" controls the reinforcement interval: A new interval is started at the moment the subject releases the pressed lever. The virtual stimuli declared as "rnfenbl" mark the availability of the reinforcer in the event logging file. The state set "rnfctl" controls the delivery of a water reinforcement: Upon an internal synchronizing signal, "reinf" the valve is opened for .1 sec.

PASCAL: While an identifier may be as long as one desires, only the first six characters are significant. Numbers are unsigned 32-bit integers. Time is internally stored as such a number representing hundredths of seconds. For the program, a time magnitude may be represented as hours:minutes'seconds.fraction" (e.g., 2:16'57.45" or 5'30").

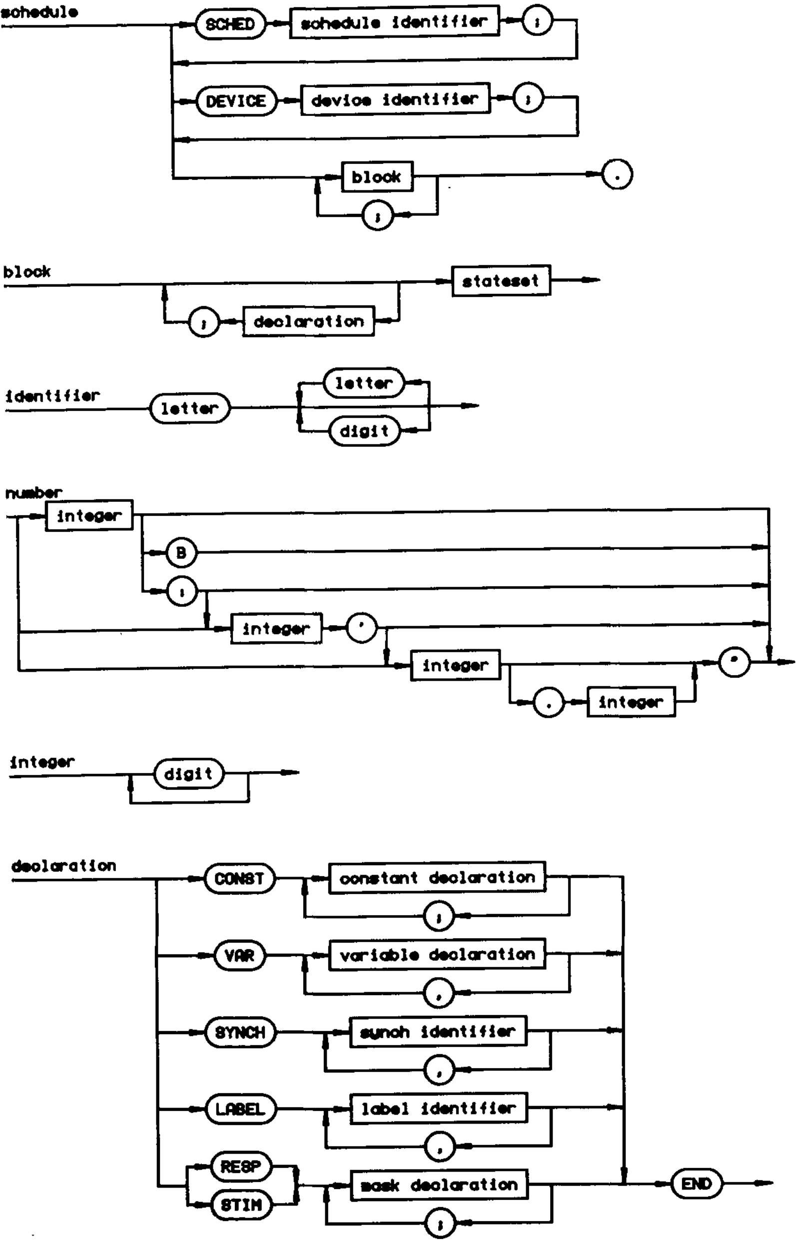
In the declaration section, all constants, variables, synchronization signals, labels, responses, and stimuli used in the program have to be declared by identifier. These declarations are globally valid; no local names are allowed. An internal synchronization signal (synch) has the same function as the z-pulse in SKED. A list of constants may be defined by the inclusion between parentheses of numbers separated by commas. Such a list may be referenced like an array. Only one-dimensional arrays may be declared as variables by indicating the array length within brackets. Response and stimulus masks are defined by combining predeclared mnemonics specific to the given device type or by the specification of octal mask words separated by commas indicating the selected bits in the I/O record described below.

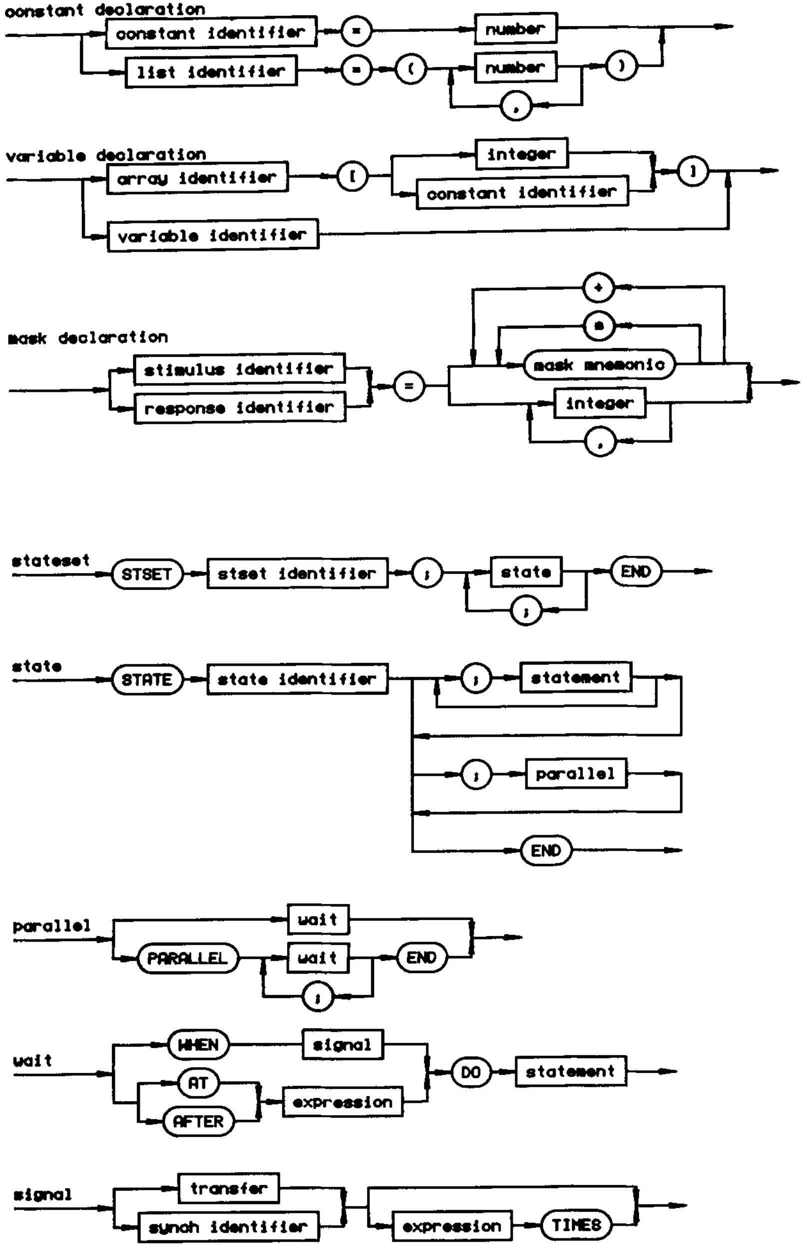
A state set consists of its name declaration followed by a number of states. A state is identified by its name followed by a sequence of statements that are executed immediately upon state entry. The possibility to write state entry statements allows for a state notation technique, as described in Martin and Conner, 1975. A stimulus condition may be set upon state entry instead of only upon state transition or as result of an event, as in SKED. A one-to-one relationship between a state and a stimulus condition may thus be established.

After this state entry segment, a state waits for an event, or for several different events, by structuring wait statements with a PARALLEL specifier. A state set may be blocked forever in the absence of any wait specification. Responses, stimuli, or synchs are awaited using the WHEN specifiers, absolute time of day, using AT, and time intervals, using AFTER. A state may await more than one occurrence of a signal by the specification of TIMES. This number of occurrences, as well as a time value, is given by a general expression evaluated just after execution of the state entry statements.

A change of stimulus conditions is treated like any other event. This allows for an additional triggering mechanism between state sets besides the synch pulses. The specifier SENT means that the whole response or stimulus pattern must correspond exactly to the declared one. If SET is specified, 1 or more of the bits have to be set, and CLEAR asks for all specified bits to be cleared in order to be recognized as the awaited event.

A statement must be labeled in order to be used by a GOTO statement. A statement may be jumped at in this way only if it is in the same program section as the GOTO statement (state entry section or wait section of the same state). Several statements are defined as they are in PASCAL. No user-defined procedures and functions can be declared in the actual version of PASTOR, so they have to be predeclared in the implementation. Additional statements are specific process control functions of PASTOR. A SEND statement copies the stimuli, overwriting any previously set or cleared stimuli. The specifier SET causes those stimuli that are set in the mask to be set, leaving the others unchanged; the speci-


schedule
SCHED
schedule identifier
;
DEVICE
device identifier
;
block
;
.
block
;
declaration
stateset
identifier
letter
letter
digit
number
integer
B
;
integer
'
integer
.
integer
*
integer
digit
declaration
CONST
constant declaration
;
VAR
variable declaration
,
SYNCH
synch identifier
,
LABEL
label identifier
,
RESP
STIM
mask declaration
;
END


constant declaration
constant identifier
=
number
list identifier
=
(
number
,
)
variable declaration
array identifier
[
integer
constant identifier
]
variable identifier
mask declaration
stimulus identifier
response identifier
=
+
*
mask mnemonic
integer
,
stateset
STSET
stset identifier
;
state
;
END
state
STATE
state identifier
;
statement
;
parallel
END
parallel
wait
PARALLEL
wait
;
END
wait
WHEN
signal
AT
AFTER
expression
DO
statement
signal
transfer
synch identifier
expression
TIMES

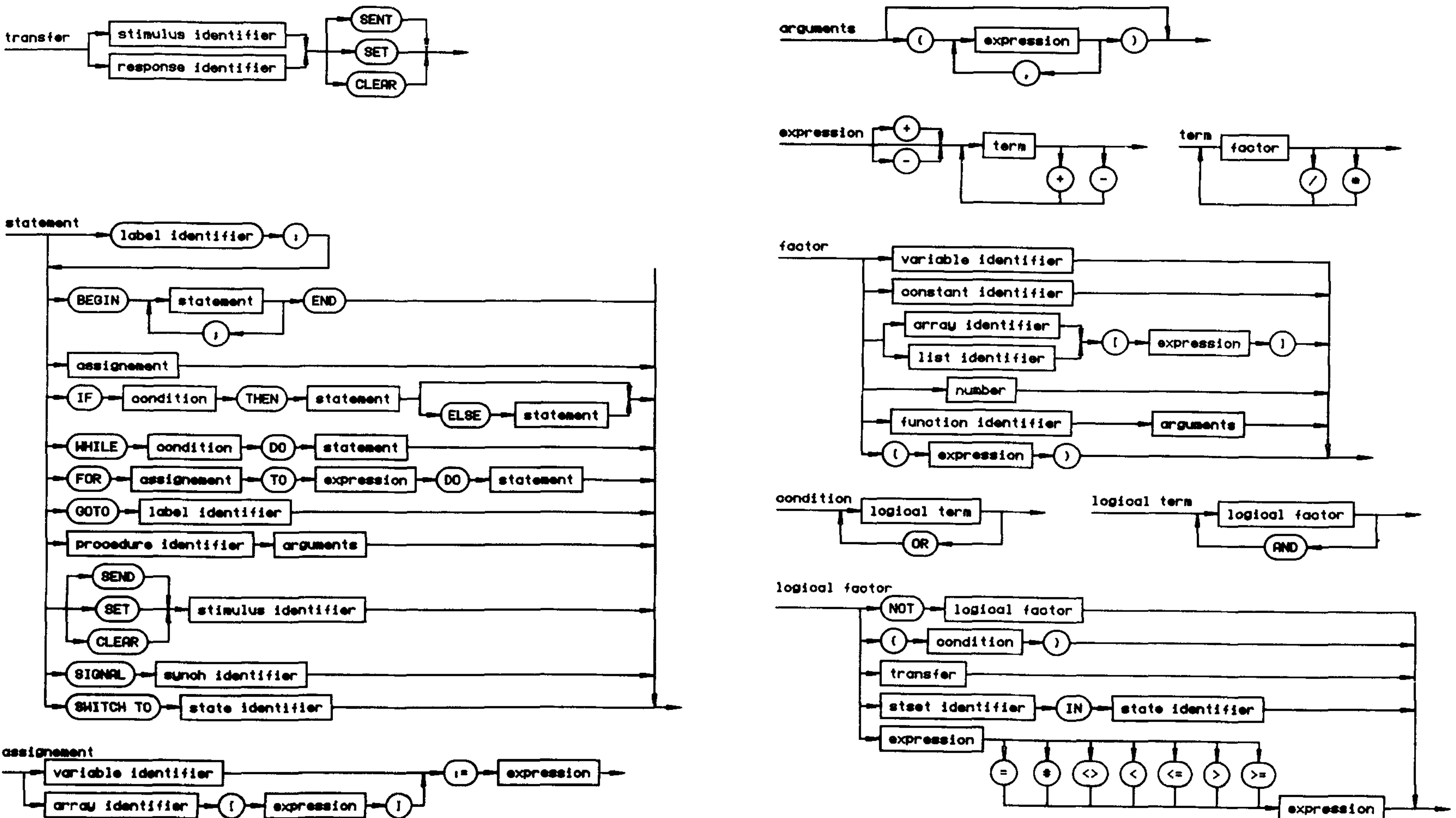


Figure 2. PASTOR syntax definition. Terminal symbols are framed by round windows, and nonterminal symbols are framed by rectangles. Each graph defines a non-terminal symbol labeled at the start of the graph path. A program's string may be created by following the arrows, having the choice at bifurcations to follow either direction.

```
predefined procedures

start  - start unit
stop   - stop schedule


predefined functions

time   - time of day in 100th of seconds
random - random number between 0 and 1000
```

Figure 3. Predefined procedures and functions of the actual version of PASTOR.

fier CLEAR causes them to be cleared. A synch may be created using a SIGNAL statement. A state switch to a state belonging to the same state set is executed through the SWITCH TO statement.

An expression is formulated as in PASCAL, with the limitation to the unsigned integer data type only. A condition is analogous to an expression. It may only be used in IF and WHILE statements, as no Boolean variables exist in the current version of PASTOR. The status of stimuli and responses can be tested in a condition, as can the active state of another state set.

Predefined Procedures and Functions

Figure 3 lists all implemented procedures and functions. The procedure "start" induces no other action than the lighting of a green status lamp on the unit's hardware controller in order to give information about a schedule's activity. A schedule's execution is actually started through the activation of a switch on the unit's controller, which is part of the experimental hardware in the laboratory. The logical "start" may be delayed until an event occurs (i.e., a specific time).

A scheduled "stop" extinguishes the status lamp and all other stimuli and sets a schedule dormant for this unit. In this condition, it does not recognize any events other than the unit's switch off, an action that stops the schedule. It may subsequently be restarted by switching it on. A schedule may be stopped any time by switching it off.

The function "time" gives the time of the day as a 32-bit value of .01-sec units since midnight, and "random" delivers a pseudorandom number between 0 and 1,000.

THE PASTOR IMPLEMENTATION

The development and implementation of PASTOR and all of its associated programs have been carried out on a PDP-11/34 running under the RSX-11M operating system.[2] Extensive use of the multitasking features of RSX-11M has been made. The transport of the actual implementation to other computers and operating systems may therefore not be straightforward. BIS, an interface system of our own design, briefly described in Elsner and Wehrli (1978), was used for the hardware communication with the experimental units. Also, problems may arise if other interfaces are to be connected.

The PASTOR Compiler

The compiler has been written in PASCAL, using top-down techniques described by, among others, Wirth (1977). The compiler structure consists essentially of a formal reproduction of the syntax definition of Figure 2 following certain rules also described in Wirth (1977).

Figure 4 shows the program flow for the creation of a schedule task based on a PASTOR source program. The compiler creates a MACRO program that consists of a collection of Polish routine addresses and MACRO calls. This program is assembled using the standard DEC MACRO-11 assembler. A general definition file describing table offsets, standard response and stimulus bits, and I/O function codes, as well as a macro library containing device-specific definition macros for the creation of the response and stimulus masks, are used in this step. The standard RSX-11M task builder links the created object with a general main program and resolves the Polish routine addresses using the symbol table of a resident PASTOR library. The task may subsequently be installed, run, and be attached to any number of experimental units of the appropriate type. A general indirect command file assists the program developer to perform automatically all steps from a program source to the active schedule task.

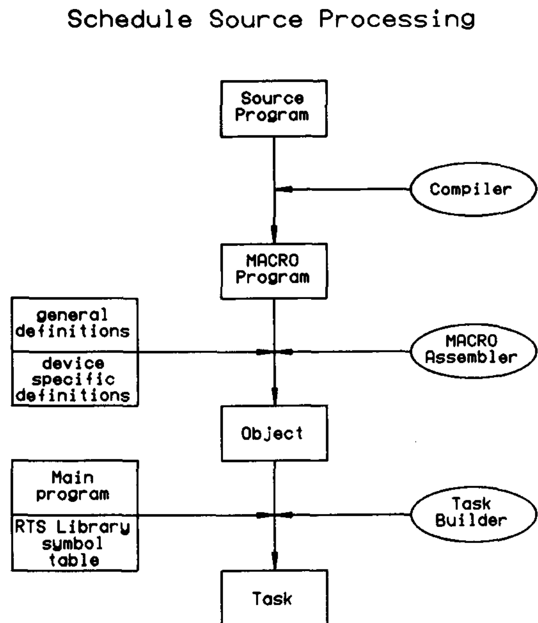


Figure 4. The program flow for the creation of a schedule task based on a PASTOR source program. The ellipses mark processing programs of which only the compiler is specific to PASTOR. The PASTOR compiler is written in PASCAL. The MACRO assembler and the task builder are those provided by Digital Equipment Corporation. The modules at the left of the graph are general to all schedules and define the run-time environment, as well as device-specific data.

The Software Environment

Figure 5 shows a possible task configuration and the respective links that may be active at one time. Protocol tasks control the activity of whole sets of experiments structured in sequences of animals belonging to specified groups performing a session according to the actual date. These protocol tasks are written as indirect command files processing a protocol definition file created with programmed assistance. Different schedules may be active for different units belonging to the same experiment protocol. There is only one schedule task per schedule type controlling all linked units.

One may notice in the syntax description of PASTOR that no I/O statements are defined other than those for interaction with the experimental units. All slow processes are excluded from a schedule task, in order to maintain a high responsiveness. The creation of sequential files containing experimental events is committed to special event logging tasks, one for each unit, linked with its respective unit and running with a lower priority than the schedule task. Such a task is activated by switching the unit on. It stores all ongoing data transfers from and to the unit into a sequential event file until the switch is turned off. An event file is identified by the unit number, the date, and a sequence number. This identification is associated by the protocol task to the corresponding session, animal, group, experiment, and so on.

Integral event logging presents the advantages of avoiding all prejudice concerning the analysis of an experiment and of allowing the investigation of aspects one may not have thought of at the time of experiment planning. This is particularly important in behavioral toxicology. In this discipline, the type of effect a new substance may present is generally unknown beforehand. The amount of data produced in the course of an experi-

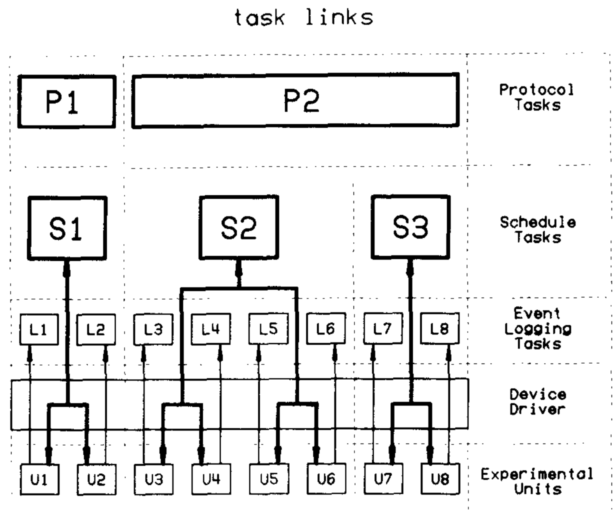


Figure 5. An example for a possible task-link configuration as it may be active in the multitasking environment of the RSX-11M operating system. The protocol tasks supervise the activity and linkages of the schedule and event logging tasks and establish logical connections of ongoing experiments with the protocolar structures (i.e., association of a log file to a session of an animal in a given group).

BIS driver QIO functions

QIO$ IO.LIO,...,<[tn1,tn2],tevf,[ast]>	link task to unit I/O
QIO$ IO.UIO,...,<[tn1,tn2]>	unlink task from unit I/O
QIO$ IO.UAL,...	unlink all tasks from unit I/O
QIO$ IO.RVB,...,<ibuf,isz>	read next record from buffer
QIO$ IO.RIO,...,<ibuf,isz,obuf,osz>	read last I/O
QIO$ IO.WVB,...,<obuf,osz>	send data to unit
QIO$ IO.CTI,...,<id,timl,timh>	mark delta time
QIO$ IO.CTA,...,<id,timl,timh>	mark absolute time
QIO$ IO.DTI,...,<id>	remove clock queue entry

ast = asynchrounous system trap routine address
ibuf = input buffer address
id = clock queue entry identification
isz = input buffer size in bytes
obuf = output buffer address
osz = output buffer size in bytes
tevf = trigger event flag
timh = time in 100th seconds (higher significant part)
timl = time in 100th seconds (lower significant part)
tn1 = task name in RADIX-50 (first three characters)
tn2 = task name in RADIX-50 (second three characters)

Figure 6. The BIS driver queue-I/O functions. A link between an installed task and a unit results in all data transfers from and to the unit to be buffered in a ring buffer; the selected event flag of the linked task is set upon a transfer, and the data may be read by the task using a read command. A task is attached if an AST routine address is given. An attached task gets the data in the stack through the asynchronous system trap mechanism. Only attached tasks may use the mark-time functions.

Response Record Format

byte count	
hour	minute
second	100th second
0 + unit number * 2	control bits
0 + 15 response bits	
.	
.	
.	

Stimulus Record Format

byte count	
hour	minute
second	100th second
1 + unit number * 2	control bits
0 + 15 stimulus bits	
.	
.	
.	

Time Record Format

byte count	
hour	minute
second	100th second
0 + unit number * 2	
1 + identification	

Figure 7. The data record formats. The first 4 bytes after the byte count contain the event's daytime at the precision of .01 sec. The number of response- or stimulus-data words in one data record may be set individually for each unit, according to the needed number of signals. This number, as well as the associations of the individual bits with their signals, is defined in the MACRO library of device-specific definitions shown in Figure 4. The different types of records are identified by Bit 0 of the first and second data word.

ment is considerable and demands sufficient mass storage space.

Besides the schedule and event logging tasks shown, any number of other tasks may run and be linked to the units in order to provide for special functions such as the real-time inspection of an experiment's activity on a terminal screen, or the hardware testing of an inactive unit.

The Device Driver

A very flexible device driver has been developed for the Behavioral Interface System (BIS) in order to allow the implementation of above mentioned links and functions. Figure 6 lists the QIO functions for the BIS driver. A task may link or unlink itself or other installed tasks to a unit through the function LIO, UIO, and UAL, respectively. Any number of tasks may be linked to a unit, and at the same time, any number of units may be linked to one task. If a linked task is not active upon occurrence of a data transfer from or to a unit, it is activated by such an event (as, for example, by switching the unit on).

The driver notifies a task about a new data transfer from or to a linked unit by setting an event flag, indicating that the data are stored in a ring buffer within the driver. These data may be read by the use of the function RVB.

An attach is a special link that is created by the specification of an asynchronous system trap (AST) routine address. Only one task per unit may be attached. An attached task gets I/O information through an AST (a fast task activation and data transfer mechanism implemented in RSX-11M). If a schedule task is attached, a write protection is erected against other tasks, in that only privileged tasks are allowed to send data to a unit through the function WVB. Unattached units are not protected.

The special functions CTI, CTA, and DTI may be used only by an attached task. A time event is scheduled by using these functions. The same entry point and data handling mechanisms are used for time events as for all other events.

Figure 7 represents the record format of a data transfer between a task and a unit. It is inherent in BIS that the number of 16-bit words per data transfer may be selected individually for each unit. Thus each record is assorted with a byte count that reflects the length of a transfer. Each record contains the time of day in a resolution of .01 sec. The first data word after the time contains the transfer direction indicator bit, the unit number, and some general control bits, such as the status of the unit's switch. The additional words contain device-

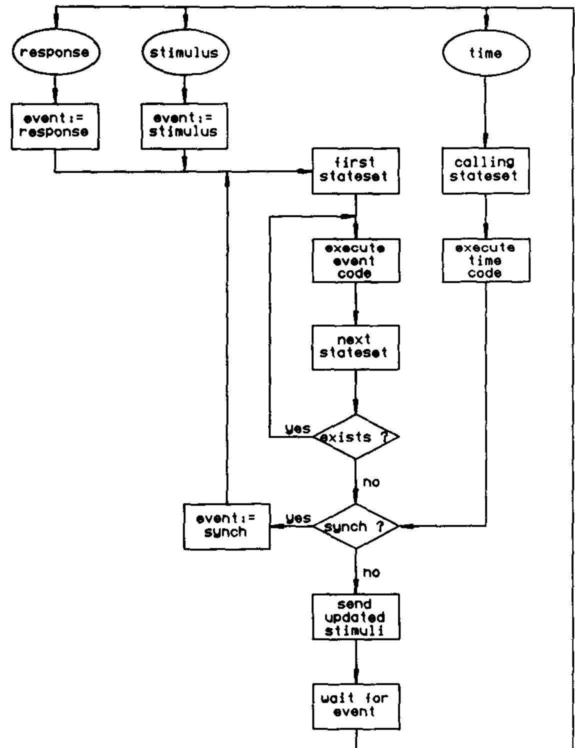


Figure 8. Event processing algorithm. The program sets itself into the correct event-type condition by establishing the corresponding state table's offset. A processing pass consists of executing the corresponding event code of all state sets' active states. If synchronizing signals (synchs) are created during a pass, a synch-event pass is executed. Synch pass iterations may not exceed a certain maximum.

specific bits whose meaning is described in the MACRO library mentioned above. The time record format is analogous to the data records. It identifies itself by an uneven second data word, something excluded in BIS data, which use BIT 0 of additional words as a continuation indicator.

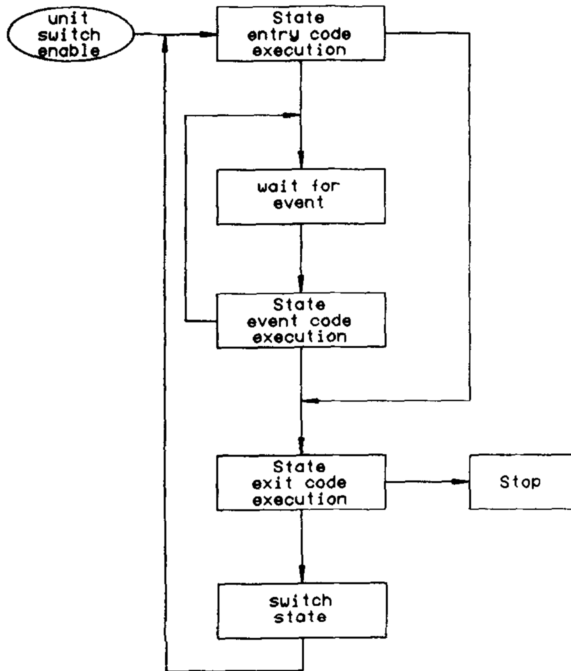


Figure 9. The processing of a state. A state switch or a stop may both be scheduled in the entry section of a state or in its event section.

THE PASTOR RUN-TIME SYSTEM

The processing of an event by a schedule task is represented in Figure 8. The execution of all event codes, including time events, synch events, state entries, and exits are strictly event driven (no polling) and are called over identical routines. If an external event occurs, all state sets are serviced in sequence. A time event is associated to a specific state set's state and does not elicit any direct action for the other state sets. If the event processing creates internal synchronization signals, they are processed in a pass through all state sets. A pass counter monitors the number of such synch processings elicited by a single event and inhibits any further iteration after a maximal number (presently 10).

Figure 9 shows how a state is processed from initialization to exit. Part of the state entry code is under program control. The other part is determined by the wait statements: The event counters are reset, and the absolute times or time intervals are initialized. If a state switch or a stop occurs in any of the state sections, the state is left by executing its exit code, mainly canceling its still hanging time requests. A stop resets the active states of all state sets to the initial state.

PASTOR's Code Structures

Figure 10 presents the schedule task structure. A resident library contains all necessary Polish routines and other subroutines. Figure 11 lists all currently implemented Polish routines. These routines are written in reentrant code, in order to be used by all active schedules in common.

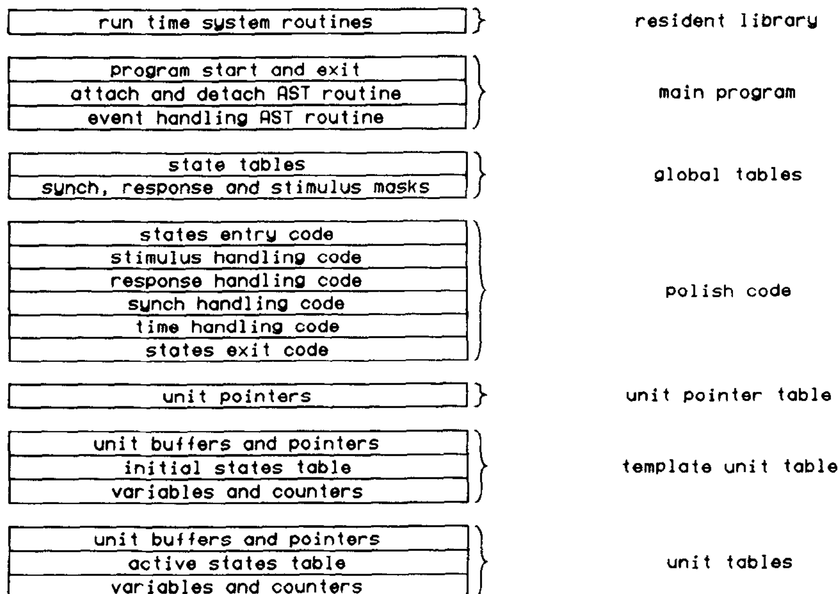


Figure 10. A schedule's task structure. The resident library contains Polish routines and other subroutines written in reentrant code to be used by all active schedules in common. Per schedule and unit type, only one task is needed. It handles all units attached to the schedule. The task is divided in pure and impure code. An impure unit table is created dynamically as a new unit is attached to the schedule.

The main program consists of program entry and exit sections and two AST routines.

The attach and detach AST routine is triggered by two separate tasks (SAT for attach and SDT for detach) that allow the operator or an indirect command file to establish a link between a schedule and a unit. These tasks communicate with the schedule task through standard RSX-11M task communication mechanisms. Upon an attach command, this AST routine extends the task size by the amount needed by a unit table, copies the template unit table into the new memory area, relocates the pointers, and enters the table into the unit pointer table.

The event handling AST routine dispatches all incoming events and time interrupts to the appropriate attached unit table and calls the main event handling routine in the resident library.

For each state in the schedule, there exists a state table, which is represented in Figure 12. A state table

polish mnemonics

```
$PSAD - add stack value into stack value
$PSAN - "and" two boolean values on stack
$PSAT - mark absolute time
$PSAV - convert address on stack into value on stack
$PSCS - clear stimuli
$PSDT - mark delta time
$PSDV - divide stack value into stack value
$PSDZ - decrement variable and test if zero
$PSES - set stimuli equal to mask
$PSEQ - compare two stack values if equal
$PSEX - exit polish mode
$PSGE - compare stack values if greater or equal
$PSGT - compare two stack values if greater than
$PSIC - test if mask bits clear in input
$PSIE - test if input value equal mask
$PSIS - test if any mask bits set in input
$PSIX - convert index and address on stack into address
$PSJC - jump conditionally if false
$PSJM - jump unconditionally
$PSLA - load address into stack
$PSLC - load constant into stack
$PSLD - load variable value into stack
$PSLE - compare two stack values if less or equal
$PSLO - convert offset to address and load into stack
$PSLT - compare two stack values if less than
$PSMC - move constant into variable
$PSML - multiply stack value into stack value
$PSMV - move variable into variable
$PSNE - compare two stack values if not equal
$PSNG - negate stack value
$PSNT - "not" a boolean value on stack
$PSOC - test if mask bits clear in output
$PSOE - test if output value equal mask
$PSOR - "or" two boolean values on stack
$PSOS - test if any mask bits set in output value
$PSRC - test if mask bits clear in response
$PSRE - test if response value equal mask
$PSRS - test if any mask bits set in response
$PSRT - remove time queue entry
$PSSB - subtract stack value into stack value
$PSSR - store stack value into address on stack
$PSSS - set mask bits into output
$PSSZ - set synch signal
$PSTR - switch to state
$PSTS - test if stateset in state
```

Figure 11. List of currently implemented Polish routines.

state S:	entry code pointer
	response handling code pointer
	stimulus handling code pointer
	synch handling code pointer
	exit code pointer

Figure 12. The state table format. Each state of a schedule is defined by a state table. They are pointed to by the unit's active state table entries. A state table entry points to its corresponding event code entry point.

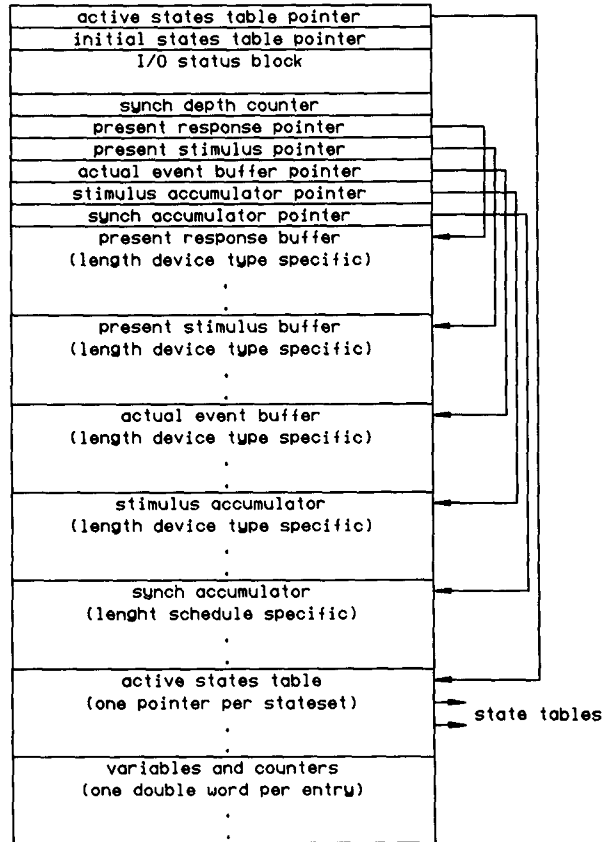


Figure 13. The unit table format. Each unit attached to a schedule is governed by such a table containing the impure data. The fixed-format table header contains pointers to variable-length buffers in the table body.

is a list of the entry points of the events' Polish code. The event type to be processed is coded as offset in this table, through which the proper code of each active state is found.

All stimulus, response, and synch masks are stored and labeled in a mask definition table, referenced by the Polish code.

The Polish code section is separated into its respective event types. If a state waits for several events of the same type in parallel, they are serviced in sequence as they appear in the program.

The unit pointer table has one entry for each unit

in the system. At the time a unit is attached and a unit table is created, the respective entry in the unit pointer table points to the unit table.

A unit table is structured as represented in Figure 13. It contains all impure data of the units. For each table entry, there exists a pointer in the header. An additional entry points to the template initial states table for the restart of a schedule. The header also contains the I/O status block for I/O error handling and the synch depth counter, which counts the synch handling iterations.

The present response and stimuli buffers serve to test the active status of responses and stimuli in a condition statement. The event buffer contains the responses, the stimuli, the synchs, or the time event identification, whichever is appropriate. In the stimulus accumulator, all stimulus changes occurring during an event processing are monitored. The updated buffer is sent, if changed, to the unit just before entering again in wait status. The synch accumulator stores all synchronizing signals created during one state set pass. At the end of the pass, the accumulated signals are copied into the event buffer and cleared.

The active states table has one entry per state set. It contains the pointer to the state table of the active state of this state set. A state switch updates this pointer.

The last impure data section contains the variables and counters. They are referenced in the Polish code by offset, rather than by absolute address.

CONCLUSION

The described software system is field proved. It has been employed in the present form for over 1 year in our laboratory for the execution of experiments in behavioral toxicology. It handles the activity of two residential mazes and eight Skinner boxes working generally day and night in several experimental protocols. These use a variety of different schedules, including an overnight autoshaping procedure with performance-controlled schedule switching.

The run-time system has very low memory requirements. The variable-interval schedule presented in Figure 1 needs, besides 800 words of resident library, 992 words of working memory plus 164 words/unit. An event logging task needs about 900 words, of which a great deal is needed by the block buffer. Many more experimental units may be added to the system without substantial reduction of responsiveness. The only limit is the available memory and the number of 128 units/interface system.

A whole set of modular programs performs special functions, such as the planning of experimental protocols, the testing of the hardware, the real-time observation of the activity in an experiment on a screen, the data processing of the event sequences, the storage of reduced data in structured data banks, and the graphical representation of the data and their statistical analysis. These applications, including program development and text editing, are executed in parallel to the experiments on the same computer without inducing any noticeable effect upon the efficiency of the experiment control. The response latency of the software is generally within 10 msec and seldom exceeds 20 msec.

Taken as a whole, this software system presents an efficient and practical tool for the everyday tasks of a behavior laboratory performing routine work and dynamic research.

REFERENCES

Dessy, R. E., & Starling, M. K. *Fourth generation languages for laboratory applications.* American Laboratory, February 1980.

Elsner, J., & Wehrli, R. Interface systems in behavioral research. *Behavior Research Methods & Instrumentation,* 1978, **10**, 259-263.

Jensen, K., & Wirth, N. *PASCAL—User manual and report.* New York: Springer, 1975.

Johnson, R. C. Special report: Ada, the ultimate language? *Electronics,* February 10, 1981, 127-132.

Martin, R. A., & Conner, R. J. Computer control of operant oriented laboratories. *Behavior Research Methods & Instrumentation,* 1975, **7**, 151-156.

Snapper, A. G. An introduction to state notation and SKED. *Behavior Research Methods & Instrumentation,* 1976, **8**, 69-72.

Wirth, N. *Compilerbau.* Stuttgart: Teubner, 1977.

Wirth, N. *MODULA-2.* Zurich: ETH, Institut fur Informatik, 1980.

NOTES

1. SKED is a trademark of State Systems, Inc.

2. PDP-11 and RSX-11M are trademarks of Digital Equipment Corporation.