SODA—a dual activity operating system

By Władysław M. Turski*

SODA is an operating system for a new Polish computer ODRA-1204. This paper gives an account of the basic principles of SODA structure and operation. SODA is an operating system for a small computer (without magnetic tapes) tailored to the needs of a research institute. A fundamental objective achieved by means of SODA is the simultaneous preparation and execution of programs.

(First received October 1967)

In order to understand the basic features and somewhat tortuous principles of the SODA, several words must be devoted to a brief description and history of ODRA–1204.

A standard ODRA–1204 is a small scientific computer whose main characteristics are collected in **Table 1.** Originally, this computer was designed as a general purpose computer with all features necessary for time-shared BDP applications. Later on, the Elwro Factory decided to introduce another computer specifically for BDP and to market a truncated version of the original machine as a scientific computer ODRA–1204. This bit of history explains the very unusual and imbalanced characteristics of the machine which has full interrupt and memory protection facilities, single I/O channel,† no tapes and relatively small backing store on magnetic drums.

The aim of the SODA designers was very simple—to build an operating system which, while making use of the advanced features of the computer hardware (which would have been idle in a traditional one-user-at-a-time running system), would not actually limit the throughput characteristics (which could easily happen if standard solutions were applied). At the same time, the designers aimed at an operating system reasonably well suited to the research institute environment in which the computer was to be used.

In other words, the SODA is a compromise between several antagonistic tendencies, subject to rather severe restrictions imposed by certain austerity of the hardware configuration. The conflicting tendencies are: (i) sustained high throughput, i.e. a development towards batch-processing of a sort, (ii) short turn-around time for small programs and frequently many compilations for a single production run, i.e. a development towards easily accessible on-line system, (iii) prevailing desire to utilise as fully as possible the interrupt, memory protection and autonomous data transmission features

† There is also another version of this computer which is installed with two I/O channels; for this machine there is an enlarged SODA II system. Differences between the two SODA's are minor and mostly implementational, as the second channel is not used for communication with the external world. These differences will not be treated here.

Table 1

Basic Information on ODRA–1204

Produced by	ELWRO Factory, Wrocław, Poland
Arithmetic	Binary, parallel, fixed and floating point
Word length	24 b
Memory core	4 K–64 K (standard 16 K)
cycle	6 μsec
drum	1–8 units of 64 K word each (standard 2 units)
Addressing	relative, indirect, address modification
I/O autonomous channel	up to 8 (standard 1)
paper tape punch	150 char/sec (standard 1 unit)
reader	300, 1000 char/sec (standard 2 units)
line printer	600 lines/min
control typewriter	10 char/sec (standard 1 unit)
Interrupt	on I/O operations, illegal memory access, certain arithmetic conditions
Memory protection	in blocks of 256 words, by address base and upper limit

* *Computation Centre, Polish Academy of Sciences, Warsaw, Poland.*

under the heavy restriction of one-channel and small array of I/O units. The practical solution was obtained by a systematic application of two basic principles:

(1) Logical separation of two activities performed by the computing system: preparation and execution.
(2) Strictly hierarchical organisation of the control structure within the operating system.

Regarding the first of these principles it may be justly observed that the two activities are not always clearly separable in everyday practice. An obvious example of the information processing task, where the separation is rather diffused, is running a program under an interpretative translation. In this case the preparation and execution are normally very much interweaved, in at least one sense: i.e. that of the time sequence in which units of action of each activity are undertaken. The same example may be considered from a different point of view, raising a highly pertinent question: what is the precise meaning of the stipulated division? Or even more basically: in an information processing task what are the steps constituting the preparation and what are the steps constituting the execution? Disregarding for the moment the precise semantic content of these two terms we may safely assume that in performing almost any information processing task, presented to a computing device in the form of a program and data, we shall intuitively differentiate between (i) actions taken by the computing device according to some previously stored program (or programs), such that the program of the presented task is regarded as data, i.e. operated upon, and (ii) actions taken by the computing device according to the (possibly modified) program of the presented task (and, possibly, some other previously stored programs).

Having recognised functional differences in these two ·ypes of activity, one would logically conclude that two separate operating systems are needed in order to organise an efficient supervision, particularly so since a given program may be many times in the preparatory processing before finally going to the execution. Hence, two operating systems of 1st level: SUGAR and SEKS were specified—the former to supervise the performance of all the preparatory steps and the latter to supervise the execution. (The names of the systems derive from Polish abbreviations: SUGAR—system for debugging, generating, adjustment and reservation, SEKS—system for execution.) These two systems, together with a third one—SAD, form the core of SODA. SAD is a supervising system of 0th level whose job is to converse with the human operator, supervise SEKS and SUGAR, and generally judge and decide about priorities and assignments of various elements of the hardware to the systems of 1st level.

Irrespectively of our initial intuitive definition of preparatory and executive actions, we are now in a position to say that in future by 'preparation' we shall mean actions supervised by SUGAR, and by 'execution' those supervised by SEKS.

Thus, an information processing task is functionally split into two types of activity, supervised by two separate operating systems: SEKS and SUGAR, which, in turn, are supervised by SAD. SAD's supervision of SUGAR and SEKS is an example of the application of the second basic principle of SODA. This principle is carried right down the entire system, in both its branches (cf. **Fig. 1**).

SUGAR, or to be more exact, a special program known as SUGAR Manager selects and supervises the work of a 2nd level Manager, which supervises, in turn, the performance of a selected Preparator.*

* By Preparator in this context we mean any of the fixed number of self-contained programs, stored permanently in the backing store, capable of accepting an externally coded program (via an input device assigned to the SUGAR branch) and producing a suitable internal form of this program.

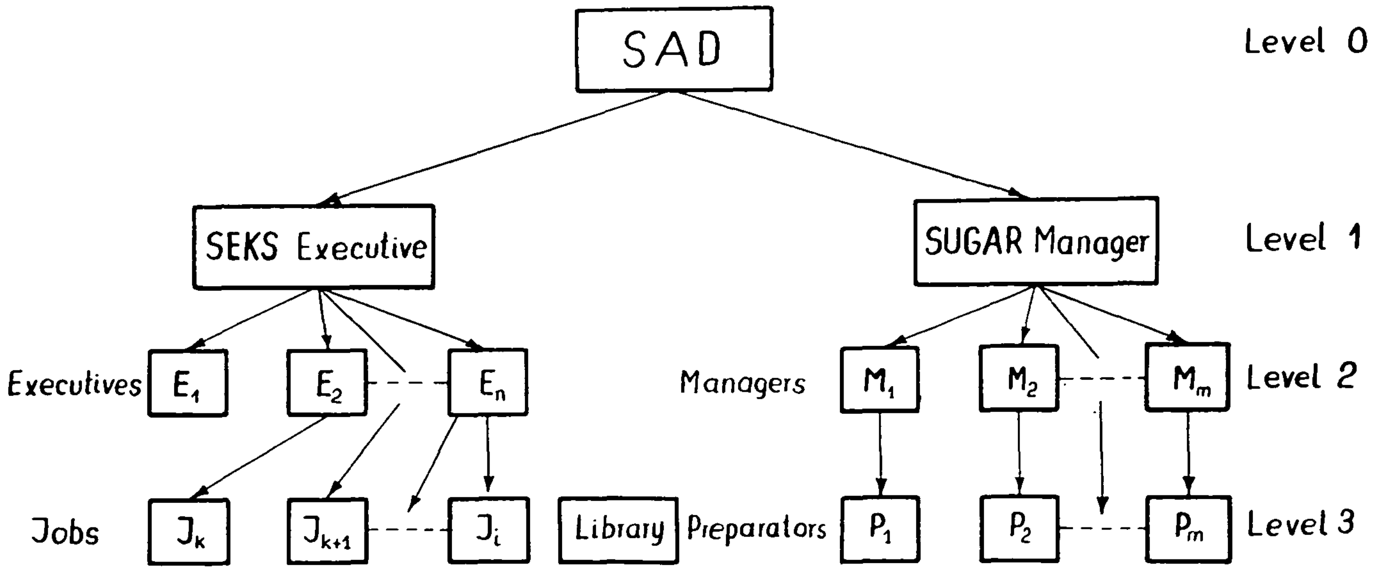

Fig. 1. Hierarchical organisation of control structure of SODA

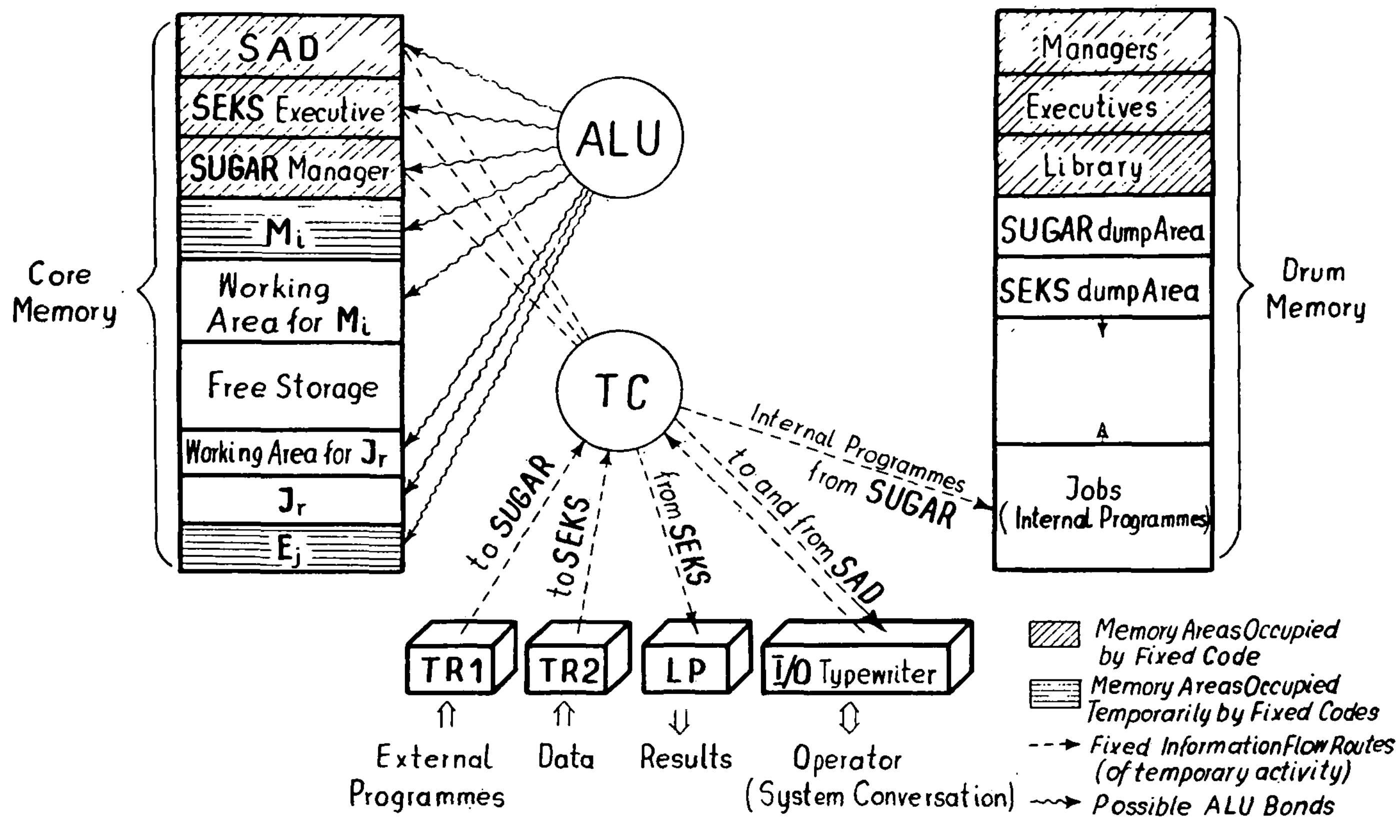

Fig. 2. Hardware distribution and sharing

SEKS Executive selects an appropriate 2nd level Executive. It should be observed that there is no one-to-one correspondence between Preparators (and their Managers) and Executives; different Preparators may produce internal jobs that will work under the same Executive, or one Preparator may produce jobs to be run under different Executives—e.g. a formally correct ALGOL program may be run under one Executive, whereas a formally incorrect program may be run under another one.

The selected Executive of 2nd level supervises the running of the SUGAR prepared program. The supervision encompasses the issuing of the static and dynamic hardware requests (i.e. requesting necessary core memory areas, backing storage and additional I/O equipment) and all run-type supervision and organisation (changing segments, servicing buffers and transmitting I/O requests in the form required by the more basic Executive).

Before explaining in some more detail the structure and performance of the individual SODA elements we should consider briefly the reasons underlying the selection of such a complicated hierarchy of control over the information processing.

The reasons are twofold. First reason is the economy of the main storage utilisation. In the proposed system only the SAD, SEKS Executive and SUGAR Manager reside permanently in the core memory (cf. **Fig. 2**). In addition, during normal work of the system, only one of many Executives and one of many Managers occupy space in the prime memory. This means that only necessary parts of the supervisory system are present in the core memory, which is quite obvious and not very new. Much more important is the second reason, the simplicity. By making the hierarchical structure very rigid we were able to define quite precisely the sort of information which is to be transmitted between levels (within a branch) irrespective of the internal structure, and the tasks and performance of individual objects occupying these levels. Hence, we were able, for example, to specify clearly what is the kind of information that any Executive receives from the SEKS Executive, and any Manager passes to the SUGAR Manager. Moreover, we were able to specify uniform forms of requests which various elements of the system pass on to the more basic levels, uniform forms of reporting, etc. The gain from all this is quite considerable, as will be clearly seen from two examples:

(1) It is possible to effect the inter-level communication entirely via fixed dimension tables.
(2) It became possible to formulate precise specifications for writing users' own Preparators and/or Executives to be included into the system.

In fact, the pragmatic importance of the latter could hardly be overemphasised. Since a Preparator may be for example a compiler, and an Executive—a matching run-time supervisor for the same language, the SODA system is capable to accept any new language, without slightest changes in the basic levels of control, provided that the compiler is written in such a way that it handles all necessary communication with other levels of control according to some clearly specified rules. Moreover, thanks to a high degree of standardisation and fine resolution of control levels it is very easy to run programs written externally in two or more languages.

It should once more be emphasised that it is not only the inter-level communication formats that are standardised, it is also the various hardware requests which are expressible in a standard fashion.

A general analogy to this system is that of standardised interface building blocks. Stretching this analogy explains why the control level resolution is so fine—in order to preserve high flexibility in such standardised environment one has to go for pretty small blocks.

Hardware utilisation and sharing

In this section it is convenient to consider our computing system as three separate systems:

(1) SAD—consisting of an I/O typewriter and certain fixed core area C_{SAD}.
(2) SEKS consisting of one input and one output device* and a fixed core C_{SEKS} and drum D_{SEKS} areas.
(3) SUGAR—consisting of one input device and fixed core and drum areas, C_{SUGAR} and D_{SUGAR}.

In the computing system there is also an Arithmetic and Logic Unit, ALU, and a transmission channel, TC. Unassigned areas of the core and drum memories will be denoted by FC and FD.

SUGAR and SEKS compete for the control over ALU and TC and try to get as much of the FC as they currently need. System SAD evaluates their request, communicates with the human operator and performs all book-keeping functions, of which we shall give no detail (they are more or less standard).

A system is considered *active* only if and when it exerts control over ALU. When there occurs one of a number of conditions the hardware generates an interrupt signal. The effect of this signal (for the purpose of our explanation) is such that ALU is taken away from the system which at this time was controlling it and given to SAD. If, at the time of the generation of the signal, ALU was being controlled by SAD itself, the signal is temporarily stored. Before releasing ALU from its control SAD checks whether there are any interrupt signals stored. As usual, the interrupt signal carries information of the particular condition which caused its emission. The system from which ALU was just taken away is said to be *halted*.

Occasionally, a system wants to transfer some information to or from its core area. In order to do so it requests services of TC. This request causes an interrupt and the system becomes *suspended* until it gets control over TC. It should be noted that:

* We are considering here the standard ODRA–1204 configuration.

(1) Only an active system may request TC.
(2) If the request comes from a system other than SAD it becomes momentarily halted.
(3) If the request comes from SAD it does not halt, but if TC is not available immediately, SAD enters a special waiting loop without releasing ALU (in fact, a TC request coming from SAD does not cause an interrupt signal; on the other hand any human operator's message to SAD generates such a signal and activates SAD, which enables it to get hold of TC as soon as the latter becomes available).

When SAD decides to give control of TC to a suspended system, this system enters the so-called *transfer mode*. A system in transfer mode is not active. At the end of requested information transfer, TC issues an interrupt signal and the system which was in the transfer mode becomes *waiting* (for ALU). For simplicity's sake, we shall say also that a system which was halted and did not become suspended is also waiting. Thus, on an interrupt it may happen that when SAD wants to release ALU either one or both remaining systems are waiting. In the former case ALU is given to the waiting system, in the latter—to SUGAR. Thus, SUGAR has higher priority, which is explained by the nature of its work, from which it follows that SUGAR is more likely than SEKS to make frequent TC requests and hence less likely to monopolise ALU.

Thus, the sharing of ALU between the three systems is organised as follows:

SAD gets it immediately when there arises a condition requiring its activity (the only possible exception occurs when the operator wants to send a message and TC is already busy with a transmission; in this case the possible delay is the duration of this single TC task which, in view of the lack of magnetic tapes, cannot be longer than a few seconds†).

SUGAR gets control over ALU whenever it is ready to make use of it, unless there occurred interrupt conditions which have to be taken care of first.

SEKS gets control over ALU when it is ready to make use of it and neither interrupt conditions nor SUGAR are waiting to be serviced.

In **Fig. 3** and **Table 2** we give an example of a sequence in which ALU and TC are allocated to particular systems together with the listing of corresponding systems statuses.

From the adopted policy of ALU sharing it follows immediately that for smooth running of both SEKS and SUGAR it is essential that SUGAR calls for relatively short bursts of ALU activity and frequent information transfers. It may be worth-while pointing out that it is generally assumed that a good balancing of the computing system utilisation may be achieved when BDP jobs are run against the background of 'scientific' computation. Since in the type of organisation for which

† And this only in cases when the full output buffer is being emptied onto the paper tape; if the line printer is used as an output device for SEKS, the delay may never be longer than a fraction of a second.

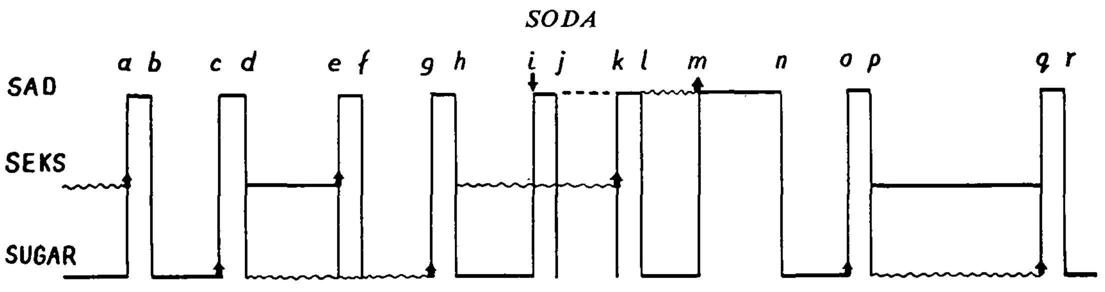
SODA
a b c d e f g h i j k l m n o p q r
SAD
SEKS
SUGAR
(Heavy line denotes ALU allocation, wavy line denotes TC allocation)
- an Interrupt Signal
- Operator's Signal

Fig. 3. Sequence of ALU and TC allocation

SODA was designed, BDP jobs are rare and far between, the BDP part of the scheme had to be replaced by some other information processing tasks of similar nature (prevalence of data transmission and retrieval over bulk computation). This was the original stimulus to consider separation of information processing jobs into their preparatory and executive phases.

TC sharing policy is very simple. Since a system (other than SAD) becomes inactive as soon as it requests the TC services the policy 'first request—first served' was a natural choice. The only exception is that SAD's requests have higher priority than those of other systems, which permits fast operator-system communication.

One more essential feature of the system should perhaps be indicated at this time: Since the number of I/O equipments and their distribution among the systems are fixed it was decided that the I/O buffers (of fixed capacity) be included into C_{SAD}, C_{SEKS} and C_{SUGAR}. These buffers belong to the 1st control level and are thus inaccessible to a programmer who operates on the 3rd control level. Hence, I/O mechanisms he uses are first transformed by the 2nd level Executive into the standard 1st level I/O requests structure and then executed in two steps: the actual input/output takes place on the 1st control level and necessary conversions and distributions are performed on the 2nd level by means of 'internal I/O procedures'. Finally, utilisation/generation of data takes place on the 3rd level. The number of pseudo-buffers and data structures of the 3rd level are of no concern to SODA.

Finally, let us discuss the memory sharing in SODA. Basic areas C_{SAD}, C_{SEKS}, C_{SUGAR}, incorporate all programs and tables pertinent to 0th and 1st level of control structure. Programs and tables of the 2nd level are permanently recorded in D_{SEKS} and D_{SUGAR}. In D_{SEKS} resides also the permanent library utilised by various programming systems employed in SODA; D_{SUGAR} houses all Preparators available in SODA.

Let us suppose that at a certain instant of time there are k jobs ready for the execution (i.e. internally coded, recorded in D_{SEKS}), SEKS controls currently a core area C^*_{SEKS}, SUGAR—C^*_{SUGAR}, and the execution of a current job has just finished. Suppose further that SAD has given to SEKS a command (which in absence of the operator's intervention is normal): 'execute next job'. SEKS Executive selects the highest priority job from $J_1, \ldots, J_k$ (by an analysis of a suitable portion of the Job Directory Table). From the internal label (see below) of the selected job the SEKS Executive determines the core requirements of this job. If it can be accommodated in C^*_{SEKS}, or more precisely in the C^*_{SEKS}—C_{SEKS} the execution of this job follows without further delay. If, however, these requirements exceed the core space currently available to SEKS, the system requests from SAD an additional core area. If this can be granted, SAD performs suitable reapportioning of the memory and the execution of the selected job follows. Otherwise, the SEKS Executive is instructed to select the second highest priority job, and so on.

If no job waiting for the execution can be accommodated, SAD informs the human operator of the situation. He then makes a decision by selecting one of two options made available by SODA. Either he stops SUGAR, in which case C^*_{SUGAR} is dumped onto D^*_{SUGAR} and C^*_{SUGAR}—C_{SUGAR} becomes available to SEKS, or he temporarily stops SEKS permitting SUGAR to finish its current task. The operator's decision depends, of course, on the circumstances, but generally the second option seems more promising, since if the SUGAR action is stopped there is no chance of it's being resumed before the full current load of jobs is discharged (no job from this load could be accommodated together with the dumped SUGAR task). Whereas, if the current SUGAR task is conducted to the end it may add a new job which may perhaps be executed by SEKS permitting at the same time initiation of a new task by SUGAR and thus avoiding a stalemate of a sort.

When a new SEKS task is a job requiring less than the current C^*_{SEKS} the difference is released and added to the current FC*.

Similarly, when SUGAR finishes its current task and is ordered by SAD to procede with a new one, it may either release some area to FC*, be satisfied with the current C^*_{SUGAR} or request additional core area. If this request can be granted it is; if not, however, SUGAR's activity is temporarily stopped by SAD, and the operator informed (for record purposes only). SUGAR's activity is also stopped if there is no more space left in the

Table 2

Example of dynamic changes in systems status (as illustrated by Fig. 3)

	SAD	SEKS	SUGAR
		transmitting	active
a		(interrupt)	
	active	waiting	halted
b			
		waiting	active
c			(interrupt)
	active	waiting	halted
d			
		active	transmitting
e		(interrupt)	
	active	halted	transmitting
f			
		suspended	transmitting
g			(interrupt)
	active	suspended	waiting
h			
		transmitting	active
i	(operator's interrupt)		
	active	transmitting	halted
j			
	active (in waiting-loop)	transmitting	halted (waiting)
k		(interrupt)	
	active	waiting	halted (waiting)
l			
	transmitting	waiting	halted (waiting)
m	(interrupt)		
	active	waiting	halted (waiting)
n			
		waiting	active
o			(interrupt)
	active	waiting	halted
p			
		active	transmitting
q			(interrupt)
	active	halted	waiting
r			
		halted (waiting)	active

drum memory for storing any more of the internally coded jobs.

Thus SODA does not permit piling up of the prepared jobs at the expense of (even temporarily) stopping the execution. The reasons for such a decision are clear enough and we shall not elaborate this point any further.

Functions of SUGAR

Within the SUGAR branch there are three levels of control: SUGAR Manager, a Manager selected from the set $M_1, \ldots, M_m$, and a Preparator selected from the set $P_1, \ldots, P_m$.

SAD gives to the SUGAR Manager the command 'go on with next task' indicating at the same time whether the program of this task is to be read from the tape reader (TR) or from the typewriter. Since normally the reading is from the TR, we shall restrict our discussion to this case only. Each external program is preceded by the so-called *external label* (EL). The EL contains the following information, recorded in a standard form (i.e. independent of the language in which the external program is written):

(1) preparator specification
(2) required mode of preparation
(3) run-time specification and options.

The first item specifies the Preparator necessary to carry out the preparation of the given information processing task. This may be, for example, one of the translators and preprocessors contained in the system, or one of the correcting routines.

The SUGAR Manager checks whether the required Preparator is available in the system and whether it can be activated at this time. Each Preparator has its own specific hardware requirement (in the case of SODA it is mostly the required volume of the core memory). These are collected in a table available to the SUGAR Manager and from this table it is determined whether a given Preparator may be activated at a given time, i.e. with the present memory allocation. If the Preparator's core request exceeds the amount currently available to SUGAR, the SUGAR Manager requests an additional space from SAD, as discussed earlier. It should be noted that some Preparators may be segmented and their 2nd level Managers constructed in such a way that the amount of the core storage used during their performance may be variable within certain limits. It is possible, for example, that a compiler will perform satisfactorily when only one of its segments is present at a time in the core memory, but it will perform better (faster) if several segments may be accommodated simultaneously. In such cases the requirement table available to the SUGAR Manager contains two entries: the minimal amount of storage necessary for the Preparator's function and the maximal amount which can be reasonably used by this Preparator. If, by table look-up, the SUGAR Manager discovers such an option it consults SAD concerning what amount of core is to be allocated for the current task. SAD's decision is based on the consideration of the total amount available just now, and on the request of the next priority job for SEKS. As soon as the core memory is allocated for the current SUGAR task, the 1st level Manager brings an appropriate 2nd level Manager from the drum storage. This Manager becomes now a basic control level in the SUGAR branch. It supervises the work of the Preparator and propagates the necessary information through the remaining parts of SODA, via the interrupt mechanism. The last feature is strictly connected with the standard memory protection mechanism whereby addresses outside the designated area (comprising Manager, Preparator and working area) are treated off limits.

From the second item of the EL, passed down the control hierarchy, the 2nd level Manager determines the utilisation mode of the Processor. This is best explained by an example. If a Preparator is a compiler, a user may want to compile a program and have it run, or may only want it formally checked. In the first case the preparation consists of the production of the full internal form of the program, in the second case only a list of formal errors is prepared (these are hopefully absent in the first case).

The last item of the EL, i.e. the run-time specifications and options, are transformed from the standard form into a form appropriate for the Executive which will eventually supervise the execution of the internal job, and are incorporated into the *internal label* (IL) of this job.

The Preparators of the SODA system are not restricted by the system to do any particular kind of preprocessing. It is envisaged that among them there will be compilers, precompilers (programs producing internal forms suitable as data for interpreters), correctors (i.e. programs for correcting jobs already recorded in the internal form, or replacing one part of the external program by another, supplied separately), etc. Generally speaking, Preparators take their input from the TR and produce an output which is assembled on the magnetic drum as a new internal job. It is not, however, precluded that a part of input may be taken from the stack of prepared jobs. This option is restricted only to such jobs as were assembled internally under the 'to be corrected' mode of preparation. In such cases not only the specific internal code but also all necessary tables are recorded on the drum, and the IL carries a special mark indicating this fact. Taking into account the rather limited size of the memory this facility is in the present version of SODA restricted to very simple languages only (e.g. PLAN programs to be run in the interpretative mode).

Another interesting point is that during the preparation no external output is produced.* The whole output goes to the magnetic drum, and external output is always considered as an execution. Hence, if the preparation consists in producing a list of formal errors in an external program, the list will be prepared in the SUGAR branch and recorded on the drum as an internal job to be 'run' under standard printing routine as the Executive. The execution, of course, consists in printing this list out.

The results of the preparation are thus always recorded in the backing store, at the addresses indicated to SUGAR by SAD. This record is supplied with the IL, containing some of the information originally recorded in the EL and also some new information generated in the course of preparation. Having finished the preparation, the 2nd level Manager reports to the SUGAR Manager. The latter records the current state of the SUGAR system and the changes which the last preparation has introduced into the state of the whole system (new occupancy limits in the drum storage, new job added, etc.) and reports to SAD. SAD takes notice of these changes by adjusting the system state tables in such a way that the updated status information is deposited on these levels where it will be needed in performance of the next assigned task, and then informs the operator of the just finished preparation. This information may also contain a brief description of deviations (if any) from the expected course of preparation (e.g. if a program was to be compiled and then run, but the Preparator found it formally incorrect, the operator gets a message 'program (name) formally incorrect').

* Except, possibly some remarks to the operator, e.g. when a program has been submitted written in a language for which no translator is available.

The operator may now give to the newly created internal job a priority (a number from the range 0–99). If no priority is explicitly given, the job gets the priority 50. It should be observed that, if the preparation was abnormal, the operator may request a very high priority output of the preparation results by giving to this 'job' a suitably high priority value.

Functions of SEKS

Within the SEKS branch there are also three levels of control: SEKS Executive, an Executive selected from the set $E_1, \ldots, E_n$ and a job program. The difference with respect to the SUGAR branch consists in that whereas in SUGAR all active programs are pre-recorded, in SEKS the basic program—job—is (at least partially) derived from the user's input.

The SEKS Executive, having received from SAD a command to proceed with the next job, selects it from a table of jobs currently available in the backing store. This selection is normally done either according to the priority, or a job associated with the particular name is selected. The latter happens if the operator requests of SAD the execution of a given job rather than letting it choose the job automatically. Under abnormal conditions (lack of sufficient core space) the system works according to the options described earlier.

Having selected the job, the SEKS Executive brings to the core memory the IL of this job. This contains the following information:

(1) 2nd level executive request
(2) run-time specifications and options
(3) library specifications.

The first item allows an appropriate Executive to be brought from the backing storage (unless it is left there from the previous execution). This Executive then assumes full control of the execution. From the second part of the IL it determines the necessary core memory volume and reports it (via SEKS Executive) to SAD. If the job is written in a segmented form this request may again be of the alternative form and, similarly to what we described above, it is up to the Executive to supervise the replacement and distribution of segments during the execution. Other run-time specifications and options are specific for each of the Executives and will not be discussed here. (They include the specification of the output mode, on-line communication via the typewriter and so on). Finally, the library specification part informs the executive which routines from the executive library are to be incorporated or made available to the job program.

Since the organisation of the SEKS branch is very similar to that of SUGAR we could stop our discussion at this point. As, however, the example of the interpretive translation was mentioned before, we shall briefly describe the interpretive execution of a job.

An external program intended for such an execution has been already preprocessed by a suitable Preparator and transmitted to the backing store in a form of machine coded representation of the external form. This representation is (depending on the programming language) a more or less simple binary coded version of the source program. An appropriate Executive brings from the executive library the interpreter (or an initial segment of it) and an initial part of the code representing the program. Now, the interpreter becomes the active control level and treats the binary coded program as an 'internal input'. Naturally, the interpreted, and immediately executed, statements of the program may cause a physical input to take place; this is then performed through the SEKS tape reader.

We shall make one more comment, though it may appear superfluous. When an internal job is being executed all memory references outside the currently assigned core area (i.e. the lowest three segments in Fig. 2) cause an interrupt. Under certain circumstances this may be very advantageous (in fact, this has been extensively exploited to indicate the necessity of changing the mode of execution of a job, a technique which will be described separately), but always it gives full protection against any damage a non-tested user's program might do to the system.

The functions of SAD

As we mentioned earlier the main functions of SAD are:

(1) handling the operator communication
(2) supervising SEKS and SUGAR activities
(3) systems book-keeping.

The second and third of these functions either were described, or are so very standard as to be of no interest. Thus, we shall restrict ourselves to a brief discussion of the first function.

SAD performs the overall automatic control over the entire SODA system. It has been recognised from the very beginning, however, that the limitations imposed by the hardware will often cause contingent situations to occur, such that an automatic control, though possible in principle, would require very sophisticated decision making. Since this would be highly undesirable on (at least) two counts: (i) complex programming consuming too much of the valuable core space, (ii) inherent lack of flexibility of such solution, it has been decided to leave the decisions in such cases to the human operator. Departing from this decision it became quite clear that it would be relatively easy to allow the operator to overrule any decision made by SAD. In fact, the operator could be considered as a —1 level of control in the SODA hierarchy. It is occasionally called from below (i.e. from SAD) but, contrary to all other systems, it is (at least one would hope) permanently active and may intervene at wish.

The operator's intervention is achieved by his choosing one of a set of commands which he may type on the typewriter. We have seen that such messages are attended to by SAD practically immediately as they appear. We have already discussed two major subjects of the operator's decisions; below we list several others:

(1) It may happen that the execution of a current job is not terminated within the allotted compute-time interval (this is discovered automatically by SAD). The operator uses his discretion whether to dump this job (into the SEKS dump area of the backing store) and proceed with a next SEKS job or give this job an additional 'grace period'.
(2) Temporary changes in I/O equipment allocation to SODA subsystems.
(3) Changes of the priorities of jobs waiting for execution.
(4) Overruling the priority scheme and indicating by name the next job for SEKS.
(5) Overruling certain parts of the selected job Executive, e.g. initialisation of a job from an auxiliary entry point.
(6) Bringing jobs for execution from the SEKS dump area and restarting them.
(7) Deleting jobs from the waiting for the execution list and from the SEKS dump area.
(8) Changing the mode of execution of a job (e.g. instead of the requested run producing a binary coded version of the job on a tape, ready for the later input through a standard binary input Preparator).
(9) Requesting an alarm output of the binary coded 'image' of the core memory and/or backing store.
(10) Direct input to arbitrary hardware locations (i.e. setting the memory, operator's switches, programmable switches, accumulators, etc.).

SAD contains a set of relatively simple routines, one for each operator's command. These are activated on the decoding of the operator message. The messages may be either control commands which perform the tasks listed above or information requests which ask SAD to produce records of the system utilisation past and/or present, hardware distribution, jobs states, etc.

Periodically, SAD produces full listing of all statistical tables it keeps. This may not be suppressed by the operators, and, when suitably bound, forms the log-book of the computing system.

Acknowledgements

The SODA system has been designed in the Computation Centre of the Polish Academy of Sciences under an agreement with the Elwro Factory. The author is indebted to many people with whom he has discussed certain aspects of the system and who contributed to its implementation and in particular to Mr. J. Maroński who patiently transferred many half-baked ideas into a set of flow and state diagrams.

The lack of the customary reference listing should not be understood as an intended indication of this paper's absolute originality. Quite the contrary, the paper owes so much to so many sources that it would not be practical to list all of them.

Book Review

Machine Intelligence 2, edited by E. Dale and D. Michie, 1967; 252 pp. (Edinburgh: *Oliver and Boyd,* 70s.)

This book contains the collected papers of the Second Machine Intelligence Workshop held at Edinburgh University in the summer of 1966. Compared with the First Workshop, held a year earlier, there are fewer papers, 14 as compared with 17; also there is a slight, yet noticeable shift in fields of interest. This Workshop shows more emphasis on programming languages, and their basic properties. Four authors (besides the Edinburgh group) appear for the second time; these are Cooper, Murray and Elcock, and Foster. It is quite interesting to note the 12 months progress. The Edinburgh group also shows progress in its heuristic programming methods, and gives a reference manual for the on-line programming language it has developed—POP-2.

Undoubtedly the best way to give the flavour of the Workshop is to quote the titles of the papers, these are: 'Semantics of Assignment', by R. M. Burstall; 'Some Transformations and Standard Forms of Graphs with Application to Computer Programs', by D. C. Cooper; 'Data Representation—the Key to Conceptualisation', by D. B. Vigor; 'An Approach to Analytic Integration using Ordered Algebraic Expressions', by L. I. Hodgson; 'Some Theorem Proving Strategies Based on the Resolution Principle', by J. L. Darlington; 'Automatic Description and Recognition of Board Patterns in Go-Moku', by A. M. Murray and E. W. Elcock; 'A Five Year Plan for Automatic Chess', by I. J. Good; 'BOXES: an Experiment in Adaptive Control', by D. Michie and R. A. Chambers; 'A Regression Analysis Program Incorporating Heuristic Term Selection', by J. S. Collins; 'A Limited Dictionary for Syntactic Analysis', by P. Bratley and D. J. Dakin; 'POP-1: an On-line Language', by R. J. Popplestone; 'Self-improvement in Query Languages', by J. M. Foster; 'POP-2 Reference Manual', by R. M. Burstall and R. J. Popplestone.

In other subjects the value of an annual conference held in the same pleasant surroundings is well established, and a worthwhile tradition soon grows up. It seems that the Edinburgh Workshops will take this role for Machine Intelligence.

One complaint—in these days of the information-explosion the appearance of papers which are not headed by an abstract is wrong, and surely doubly wrong in the Information Sciences themselves.

J. J. Florentin (London)