TOWARDS ISMs FOR OPSs*

Daniel M. Berry
Computer Science Department
School of Engineering and Applied Science
University of California, Los Angeles

1. Introduction

Information structure models (ISMs) [Weg70] have been applied successfully to the study of programming languages both in their definitions [LW69, Wlk68,70,Wgb70] and in proofs of correctness and equivalence of various implementation strategies [Luc69,JH70,JL71,McG71,Bry72].

This paper examines briefly the possibility of using ISMs to define operating systems (OPSs). We give a definition of the notion of ISM and indicate how to use them. Then we describe methods by which the process notion has been defined in the past in the hope that this will give some idea of the applicability of the ISM approach to the definition of OPSs.

2. ISMs

An ISM describes an abstract machine by giving a set of possible snapshots, i.e. abstract memory states, a set of initial snapshots, and a transformation which accomplishes the abstract machine's instruction execution cycle.

Definition: $M = (I, I_0, F)$ is an ISM iff
1. I is a countable set (of snapshots).
2. $I_0 \subseteq I$ (is the set of initial snapshots).
3. F is a transformation of the form $F: I \rightarrow P(I) = \{x | x \subseteq I\}$.

A computation is a sequence of snapshots starting with an initial snapshot. The progression from a snapshot to its successor is done by the transformation. To permit modeling of nondeterministic, asynchronous phenomena, e.g. interrupts and processes, the transformation maps a snapshot to a set of possible successors only one of which is chosen as the actual successor in a given computation.

Definition: Let $M = (I, I_0, F)$ be an ISM. Then the sequence $C = \langle S_1, \ldots, S_i, \ldots \rangle$ (nonempty and possibly infinite) is a computation in M iff
1. For all S_i in C, $S_i \varepsilon I$.
2. $S_0 \varepsilon I_0$.
3. For all S_i in C with $i>0$, $S_i \varepsilon F(S_{i-1})$.
4. C is not a proper initial subsequence of any other sequence satisfying 1, 2 and 3.

The last condition merely does not admit as a computation any subcomputation of another computation.

A computation C may be infinite (as most correct OPSs are) or finite. If C is finite, then there is a snapshot S_n in C such that $F(S_n) = \emptyset$ and S_n is the final snapshot in C.

One defines a system by giving an ISM defining an abstract machine which behaves like the system. The abstract machine accepts command language streams, programs in the languages supported by the system, and data all as input and simulates the system's behavior by having the transformation set up system structures and modify them according to directions from the input.

In order to give this ISM, an appropriate I, I_0 and F must be defined. The Vienna Definition Language [LW69] provides a syntactic metalanguage for defining I and I_0 and a semantic metalanguage for programming F. Very often, for informal or working purposes, pictures are used to describe I and I_0 and English is used to specify F. We will do the latter here.

One important thing has been learned about defining languages with ISMs. This is that a good method of defining something is in terms of a typical stylized implementation, if for no other reason than that this gives an accurate reflection of the defined object's properties. With this method it is hard to define into something more than is possible under an implementation.

3. Model of Multi-Process Systems

We now give the basic outlines of two closely related models of the process concept which have been used and which can be incorporated into an ISM definition of an OPS.

It is assumed for both of these that the snapshot contains two components:

1. The algorithm: a list of instructions which has the potential of growing as new programs are added. They may or may not be re-entrant. These may be made up of any of the machine instructions and primitives supported by the system.
2. The record of execution: consists of all modifiable data cells each of which is located by

*This research was supported by the National Science Foundation under Grants GJ809 and GJ28074.

a unique pointer, may be assigned to or retrieved from, and may be allocated and freed by whatever mechanism the system supports.

For both models of processes we follow the advice that a good way to define a concept is by some implementation of it. A process is usually implemented by having a data cell (sometimes called a process control block) which serves as a virtual processing unit. We call this cell the process. Its subcells correspond exactly to the registers of a processing unit. Because processes are data cells it is easy to permit a "potentially unbounded"* number of them.

The processes are supposed to be running in some semblance of parallelism. This is done by having a finite number of processing units (usually 1 or 2) multiplex themselves over the usually larger number of processes. A processing unit copies the state of a process into its registers and executes on behalf of this process until interrupted, whereupon the processing unit saves its state in that process and starts the cycle over with another process.

In the literature there have been at least two ways of modeling processes based on the implementation described above.

1. The processing unit is in the snapshot and the process-switching actually takes place [Wlk68].
2. No processing unit is in the snapshot and process-switching is buried under the transformation [Wlk69,Joh71,Bry71].

We describe both models by giving, for each, extensions to the basic snapshot form and the transformation.

3.1 Processing Unit Included in Model

3.1.1 Snapshot. There is an additional component to the snapshot, namely some finite set of processing units. Each of these processing units has the same internal structure as the processes described below.

We permit data cells to be allocated to processes - there can be an arbitrary number of these. Its components are:

1. ip (instruction pointer): points to the next instruction to be executed by the process (ing unit).

2. ep (environment pointer): points to a data cell in the record of execution which constitutes the root of a data structure containing the process(ing unit)'s variable data. Any cell may contain pointers to other cells so that a process may be able to access other cells through the cell to which its ep points. The set of cells that a process(ing unit) can access through a chain of pointers originating with its ep is called its accessing environment and is the only cells it can use. The ep is thus a stylistic representation of base registers or a process segment table.
3. stack: a set of temporaries (registers) arranged in stack form for conceptual convenience.
4. process-id: if this is a process, the process-id points to itself; if this is a processing unit the process-id points to the process it is executing for.
5. status: one of the following four statuses:
   - RUNNING means the process is loaded into a processing unit
   - READY means the process could be running if there were enough processing units
   - ASLEEP means the process is not able to run now for program-directed reasons, e.g. waiting for I/O. It can be made ready in future.
   - TERMINATED means the process is dead - it cannot ever again be made asleep, ready or running.

A typical snapshot is shown in Figure 1. There are three processes, Π_1, Π_2 and Π_3. Π_1 and Π_2 each have private* data regions in their environments C_1, C_2 and C_3 respectively. If the instructions that Π_1 and Π_2 are executing are re-entrant and the only data they access are in the cells pointed to by their ep's, then the two processors will not clobber each other's data in spite of the fact that they are executing the same instruction. Because of the pointers in C_1, C_2 and C_3, all three processes may access C_4. C_4 is thus a shared cell. Π_3 is RUNNING and thus the processing unit has been loaded from Π_3. The processing unit's process-id points to Π_3. The processing unit used to be an exact copy of Π_3, but since the copying, the processing unit has executed a bit; the ip of the processing unit is further along in the algorithm than the process's ip.

3.1.2 Transformation. The processing units are supposed to be running in parallel - although possibly at different speeds. Rather than actually trying to do things in parallel and take into account different rates, for each computation step

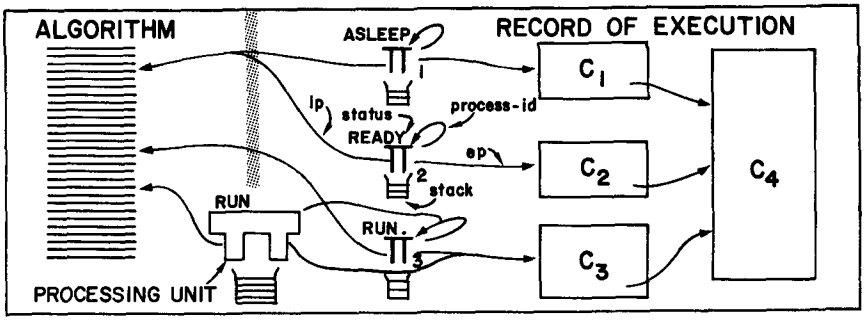

Figure 1.

*Actually bounded by finiteness of memory, but for all practical purposes, it's unbounded.

*i.e., no other process can access them.

the transformation nondeterministically picks a processing unit and has it go through an instruction execution-interrupt cycle. In this cycle it is nondeterministically determined whether to execute the next instruction or to interrupt. The transformation is as follows:

1. Pick any processing unit PU.
2. Flip coin.
3. If head then (PU does instruction execution)
   a. Fetch the instruction INST pointed to by PU's ip.
   b. Increment PU's ip to point to next instruction.
   c. Execute INST.
   d. End of transformation
4. If tail then (do process switch)
   a. Copy PU into process Π_1 pointed to by PU's process-id and change Π_1's status to READY.
   b. Select some READY process Π_2 (here defined nondeterministically, but can be by some scheduling algorithm).
   c. Change Π_2's status to RUNNING and copy Π_2 into PU.
   d. End of transformation.

At the end of the transformation we have a new snapshot.

3.2 No Processing Unit in Model

In this model the processing units are thrown out, and processes are considered capable of executing themselves. This necessitates throwing out the distinction between RUNNING and READY processes. Both are considered AWAKE.

3.2.1 Snapshot. The snapshot does not contain any processing units. There is still a potentially unbounded number of processes allocatable in the record of execution cells. The components of the process are almost as before:

1. ip, 2. ep, 3. stack, 4. process-id } as before

5. status: there are three:
AWAKE: all former RUNNING and READY processes
ASLEEP
TERMINATED } as before

A typical snapshot in this model is given in Figure 2. It represents the same state as in the snapshot for the other model.

3.2.2 Transformation. All AWAKE processes are considered to be executing in parallel. Therefore, the transformation nondeterministically selects an AWAKE process and has that process go through the instruction execution cycle:

1. Pick an AWAKE processor Π (here defined nondeterministically but can be by scheduling algorithm).
2. Fetch instruction INST pointed to by Π's ip.
3. Increment Π's ip to point to the next instruction.
4. Execute INST.
5. End of transformation.

At the end of the transformation a new snapshot has been produced.

3.3 Observations About These Models of Processes

1. The first model is a more accurate depiction of what happens in an actual implementation, but the second is a bit simpler to work with.

2. Creation of a process in both models is simply a matter of allocating a process cell and initializing it to appropriate values, i.e. an ip, ep, status, stack, and a process-id.

4. Observations About the Use of ISMs

1. The modeling is primarily qualitative rather than quantitative. ISMs are good for exposing the structure of objects but they have not yet been used for measurement. It is, however, possible to conceive of space-time estimates being made with ISMs in much the same way as with Turing machines.

2. The modeling can be as fine or as gross as desired and can ignore nonessential details; e.g., in our modeling of a process we were fairly detailed about the process and process selection, but we glossed over the rest of the snapshot and the instruction-execution phase of the transformation. This property is useful if one wishes to study a particular aspect of a system without being concerned about the rest of the system.

5. Future Research Suggestions

1. Define complete or large portions of OPSs.

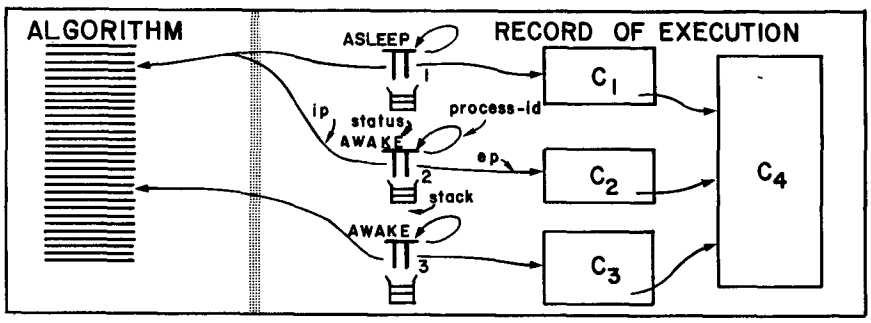

Figure 2.

2. Use proof techniques developed for proving equivalence of ISMs to help prove things about operating systems implementations.

3. Develop methods of measuring space and time requirements à la complexity theory with Turing machines.

6. Conclusions

The brief examples presented in this paper attempted to expose some of the issues involved in applying the ISM approach to the study of OPSs. It is hoped that the feasibility of this has been demonstrated and that avenues for further work have been suggested.

7. Bibliography

NOTE: DSIPL (pronounced "disciple") is Proceedings of the ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices, February, 1971. PAAP is Proceedings of the ACM Symposium on Proving Assertions About Programs, SIGPLAN Notices, January, 1972.

Bry71 Berry, D.M., Definition of the Contour Model in the Vienna Definition Language, TR-71-40, Center for Computer and Information Sciences, Brown University, April 1971.

Bry72 Berry, D.M., "The Equivalence of Models of Tasking", PAAP.

HJ70 Henhapl, W., and Jones, C.B., The Block Concept and Some Possible Implementations, with Proofs of Equivalence, IBM Lab. Vienna, TR25.104, 1970.

Joh71 Johnston, J.B., "The Contour Model of Block Structured Processes", DSIPL.

JL71 Jones, C.B., and Lucas, P., "Proving Correctness of Implementation Techniques", in Engeler, ed., Symposium on Semantics of Algorithmic Languages, Berlin: Springer-Verlag, 1971.

Luc69 Lucas, P., Two Constructive Realizations of the Block Concept and Their Equivalence, IBM Lab. Vienna, TR25.085, 1969.

LW69 Lucas, P., and Walk, K., "On the Formal Description of PL/I", Annual Review of Automatic Programming, 6:3, 1969.

McG71 McGowan, C., Correctness Results for Lambda Calculus Interpreters, Ph.D. dissertation, Cornell University, 1971.

McG72 McGowan, C., "A Contour Model Lambda Calculus Machine", PAAP.

Wlk68 Walk, K., et al., Formal Definition of PL/I, ULD Version II, IBM Vienna, 1968.

Wlk69 Walk, K., et al., Formal Definition of PL/I, ULD Version III, IBM Vienna, 1969.

Wgb70 Wegbreit, B., Studies in Extensible Languages, Ph.D. dissertation, Harvard University, 1970.

Weg70 Wegner, P., Information Structure Models for Programming Languages, TR-70-22, Center for Computer and Information Sciences, Brown University, September 1970.