# Use of the Concept of Transparency in the Design of Hierarchically Structured Systems

D.L. Parnas
Technische Hochschule Darmstadt
and
D.P. Siewiorek
Carnegie-Mellon University

This paper deals with the design of hierarchically structured programming systems. It develops a method for evaluating the cost of requiring programmers to work with an abstraction of a real machine. A number of examples from hardware and software are given as illustrations of the method.

Key Words and Phrases: hierarchical systems, bottom up design, levels of abstraction, synchronization primitives

CR Categories: 4.20, 4.30, 6.1, 6.20

## Introduction

The starting point of this paper is the goal of constructing systems with a hierarchical structure of the type first illustrated by E.W. Dijkstra in [1, 2]. Each level in such a system provides a virtual machine which hides (or abstracts from) some aspects of the machine below it. In designing such a system, we repeatedly face a question which a hardware designer faces only once: "How do I know that the instruction set provided by this machine is suitable for the programs which users will want to run upon it?" There is a risk in freezing the design of a level, the risk that we may force some inefficiency upon our final system. We may even eliminate some essential capability.

The purpose of this paper is to introduce a concept which appears to be useful in the design of hierarchically structured systems. For purposes of comparison, we shall review an approach which was suggested earlier, then introduce and illustrate the main concepts of this paper.

## The "Top Down" or "Outside in" Approach

Several papers [3, 4, 5] suggest that the solution to software design problems lies in beginning with a precise description of the desired system and deriving the internal structure from it. This would prevent design decisions which remove necessary capabilities and eliminate the risk of constructing a system with unexpected undesirable properties. The papers referenced were all concerned with providing simulation tools which could be used to verify that each decision was an adequate one. The approach was called "top down" or "outside in."

In this paper we shall refer to this approach as "outside in" rather than "top down" because the latter appellation often leads to a confusion of this approach with the levels introduced by Dijkstra [1]. The "outside in" approach and that of Dijkstra cannot be compared as they are addressing quite different questions. Dijkstra was not discussing the sequence in which design decisions were made, he was discussing the structure of the final product. Higher levels in Dijkstra's sense are not necessarily "closer to the outside" in our sense. Some low level features may appear on the "outside."

The "outside-in" approach has been discussed in several places (e.g. [6]) and found to involve a number of difficulties.

1. The necessary specification of the "outside" is often difficult to obtain. In addition to the obvious difficulty in making such design decisions, it is difficult to express those decisions precisely without implying additional, internal, design decisions.

2. The derivation of a design from such a specification is often not feasible. The set of possible internal structures for a given external specification is so large that one needs some additional constraints before a search can be begun. These constraints are usually information about the "inside" (e.g. the hardware).

3. In attempting to follow the "outside in" procedure it is quite easy to specify internal mechanisms which would simplify implementation of the desired outside but would themselves be impractical to implement.

4. It is difficult to apply this method if one is actually

designing a set of systems whose only description is "general purpose."[1]

5. As was pointed out in [7], the application of this method may result in a piece of software which is unnecessarily inflexible (see also [8]).

6. It is quite common to design software in a situation where the inside is already fixed (e.g. the hardware for an operating system, or the operating system for a piece of application software).

It is for these reasons that we have found it necessary to abandon the pure "outside in" approach and adopt some additional procedures which are actually of an "inside out" or "bottom up" nature. We do not propose the following as a procedure to be used instead of the "outside in"; we propose these as complementary approaches which must be used in some judicious combination according to the needs of the situation.

## "Transparency" of an Abstraction

We wish to consider a typical stage in a "bottom up" design process. We assume that we have a well defined lower level and are considering the design of the next highest level. The lower level may be either hardware or an intermediate level in our software design. We shall refer to either as the *base machine*. We assume that we are considering a proposal for a new abstraction to result in a new programmable machine which we shall refer to as the *virtual machine*.

We must determine the set of states which is possible for the base machine under arbitrary programs in the "language" of the base machine. Also of interest is the set of state *sequences* which can be obtained by arbitrary base machine language programs.

For any given implementation of our virtual machine we can determine a set of base machine states and sequences of base machine states which is obtainable by running programs written for the virtual machine.

If the virtual machine and its implementation were *completely transparent*, any base machine state and any sequence of base machine states which we could obtain by programming the base machine would also be obtainable by programming the virtual machine. In the more common situation, where some base machine sequences cannot be obtained by programming the virtual machine, we term the missing state sequences the *loss of transparency*.

In the above we have defined transparency as a property of a triple consisting of the base machine, the virtual machine, and the implementation of the virtual machine on the base machine. In many cases, however, we can find that there is a loss of transparency for the virtual machine, base machine and any conceivable or likely to be used implementation. In such cases we shall

[1] We are indebted to C.W. Koot of NV Philips-Electrologica (Apeldoorn, The Netherlands), who was the first to point out to us the difficulties introduced when "general purpose" is included in the description of a future product.
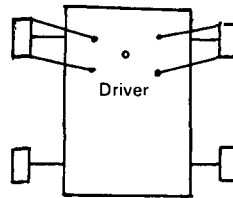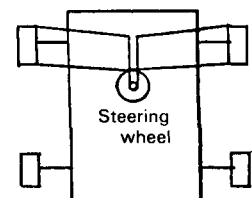
Fig. 1.



Fig. 2.



speak loosely of the transparency of the virtual machine for a given base machine.

In fact, in many cases we can ascertain a lack of transparency for a given virtual machine and any base machine likely to be considered. In those cases we can speak very loosely about the transparency of the virtual machine without reference to a specific base machine.

For the purposes of the present paper it is sufficient to rely on our intuitive understandings of what the properties of reasonable base machines and certain virtual machine propositions are. For many interesting software design problems there is no need to resort to formal models.

### Preliminary Example

The following example is intended to illustrate the concept of transparency and to make the point that a loss of transparency is often one of the goals of a design.

Figure 1 shows a diagram of a low level portion of a four wheeled vehicle. Note that each front wheel is connected to two strings and should a driver use such a vehicle, he would control the steering by pulling on a total of four strings.

It is probably feasible for well coordinated people to learn to use such a control mechanism, but it is certainly not convenient or pleasant. Figure 2 shows the addition of a higher level mechanism which uses the mechanism of Figure 1 to provide a more convenient virtual machine for the driver. The ropes have been wrapped around a steering wheel and attached so that now the vehicle can be controlled by the more easily learned mechanism of turning the wheel in the desired direction. If this is properly done, it is a very good abstraction from the real machine. (If it is not properly done, it may introduce all sorts of inefficiencies, including excessive tire wear and poor driving characteristics.)

The point of this example, however, is that even if this is done in an ideal way, the abstraction is not transparent in the sense just defined. Figure 3 shows some of the states which were possible with the lower level control mechanism. Positions (a) and (b) will be possible by the use of any reasonably designed steering wheel implementation. Positions (c) and (d) will no longer be possible with reasonable implementation. Very sharp turns (e) could be eliminated by some designs and permitted by others.

If the steering wheel were an abstraction proposed in a "bottom up" design process, we would ask that the

402

Communications
of
the ACM

July 1975
Volume 18
Number 7

Fig. 3.

(a)      (b)      (c)      (d)      (e)
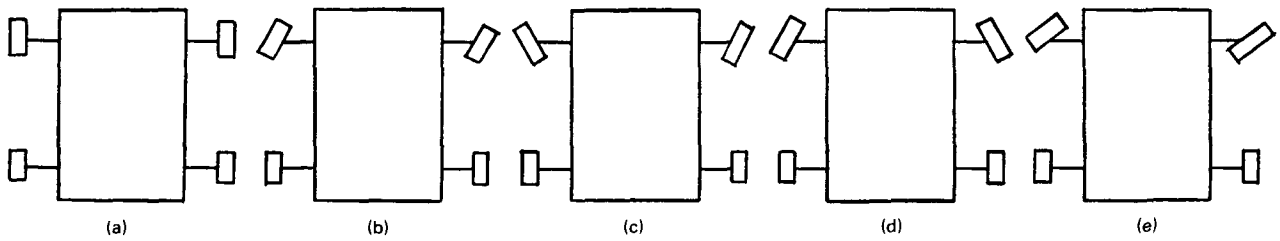
designer use the concept of transparency in evaluating the validity of the proposed design. In this particular case the lack of transparency with regard to (c) and (d) would be considered acceptable because situations in which those positions are useful are extremely rare. The lack of transparency for those cases can be considered a desirable feature of the abstraction; one of the purposes of introducing certain abstractions is to prevent the occurrence of undesirable states. The loss of (e) is more difficult to evaluate; it is undesirable, but it might be acceptable if the turning circle would be adequate anyway or if there was a cost decrease obtained by eliminating this extreme position.

The fundamental assumption behind our proposed "bottom up" approach is that the primitive mechanisms from which one builds a system have the ability to perform all the functions finally expected of the system. (If that is not true, the project is hopeless from the start.) If we evaluate each level by examining the loss of transparency as illustrated above and make certain that nothing desirable is lost, we may be assured that the upper levels will still have the desired capabilities.

The remainder of this paper will be devoted to examples from the field of computer systems.

## "Register" for Markov Algorithm Machine

Figure 4 is a specification of a module developed for use in a Markov algorithm interpreter or compiler. One can view this module as providing a virtual machine which has a register which has essentially the same capabilities as that in the idealized Markov algorithm machine. Characters may be inserted and deleted at any point in the string, etc. The one fundamental difference is that, because this is a specification for a real piece of software, there are limits to its capacity.

Informally, the four operations provided can be described as follows:

"LENGTH"     reveals the number of characters in the register.

"CHAR(I)"     gives the $I$th character in the register if $I \leq$ length.

"INSERT(I,J)"     places a new character at the specified point in the register.

"DELETE(I,J)"     removes a character in the register.

Fig. 4. Definitions.

INTEGER PROCEDURE: LENGTH
possible values: an integer $0 \leq \text{LENGTH} \leq 1000$
effect: no effect on values of other functions
parameters: none
initial value: 0

INTEGER PROCEDURE: CHAR (I)
possible values: an integer $0 \leq \text{CHAR} \leq 255$
parameters: I must be an integer
effect: no changes to other functions in modules
    if $I \leq 0 \lor I > '\text{LENGTH}'$ then a procedure call to a user written routine RGERR is performed (program cannot be assembled without such a routine).
initial value: undefined

PROCEDURE: INSERT(I, J)
possible values: none
parameters: I must be an integer
           J must be an integer
effect:
    if $I < 0 \lor I > '\text{LENGTH}' \lor J < 0 \lor J > 255$ then a subroutine call to a user written routine INSAER is performed (routine required).
    else LENGTH = $'\text{LENGTH}'$ +1 if LENGTH $\geq$ 1000 a subroutine call to user written function LENGER is performed.
    CHAR(K) =
       if $K \leq I$, $'\text{CHAR(I)}'$
       if $K = I + 1$, J
       if $K > I + 1$, $'\text{CHAR(K} - 1)'$

PROCEDURE: DELETE (I, J)
possible values: none
parameters: I, J must be integers
effect:
    if $I \leq 0 \lor J < 1 \lor I + J > '\text{LENGTH}' + 1$ then a procedure call to a user written routine DELERR is performed.
    else
       LENGTH = $'\text{LENGTH}'$ − J.
    CHAR(K) = if $K < I$ then $'\text{CHAR(K)}'$
              if $K \geq I$ then $'\text{CHAR(K} + J)'$

At first glance this appears to be a good design. In fact, it was used unsuspectingly and, for quite a while, the faults were not apparent to any of those involved in the project. The fault is easily noticed as a loss of transparency.

Such a module has many possible implementations. We list just a *few* of the more interesting or useful ones:
1. Register is an array. Access is by indexing; inserts and deletions require shifting.
2. Register is a one-way linked list. Access is by linear search counting for the $I$th item requested. Inserts and

deletions require list processing operations—no large shifts.

3. Register is a two-way linked list. Access is by search from either end or from the last point accessed. Insertions require list processing operations.

4. Register is a linked list with an "index" pointing to a number of points within the list to reduce searching.

5. Register is a linked list of small arrays. Most small changes can be done on a single small array as in implementation (1). Larger changes require addition or removal of one or more small arrays. (The small arrays might be machine words in which up to six characters are packed.)

Each implementation would be good under some set of operating conditions and costs (e.g. (1) is the minimal coding time version).

We can easily imagine having designed an abstract machine which contained operators which could be used for one of the above implementations. We refer to that machine as the *"base" machine*. On *any* likely base machine there will be simple sequences (e.g. a single store operation) which replace a single character in the register with another single character. These sequences involve no shifting in implementations (1) or (5) and no linked list operations in implementations (2)–(5). *These sequences cannot be evoked by calling the "virtual machine" operations defined above.* Thus, this design has a loss of transparency because there are sequences on the base machine which cannot be evoked by commands given to the virtual machine. Further, we see that the lack of transparency is undesirable because (1) the missing sequences are both harmless and useful, (2) the work they accomplish can only be performed by much more expensive sequences evoked by the higher level.[2]

The above loss of transparency can easily be corrected by the addition of the "alter" command specified in Figure 5. In our experimental project we did this during the project. Because of the "upward compatible" nature of the improvement, old programs continued to work but new ones could be written to be more efficient. In no case did we have to reveal the inner workings of a module to gain in efficiency.

Fig. 5.

```
PROCEDURE: ALTER(I,J)
possible values: none
parameters: I, J must be integers
effect:
    if I ≤ 0 ∨ I > 'LENGTH' ∨ J < 0 ∨ J > 255 then a subrou-
    tine call to a user written routine ALTERERR is performed.

    CHAR(K) = if K ≠ I then 'CHAR(K)'
              if K = I then J
```

For some time we considered the amended design to have the proper degree of transparency, but further reflection has indicated an additional problem. In most of the base machines there exist sequences which efficiently insert several characters at a given point in the register. For example, in implementation (1), if we wished to insert four characters, we could do so (on the base machine) by shifting the information right four places and then inserting the four characters. By calling the commands proposed, the base machine would *probably* perform four one place shifts instead of the single four place shift.

At this point there appear to be three fundamentally distinct solutions to this design problem. Each has advantages and disadvantages, and we are unable to make a general choice among them.

1. *A more sophisticated implementation.* The word "probably" occurs in the above paragraph because there do exist possible implementations which would not incur the loss of efficiency described. For example, "Insert" might be implemented so that it would not actually perform the insertions in the basic data structure until a call was made to insert at a different point. In this way the module could "store" commands until it had enough information to determine the most efficient way to perform the insertion series. Deletes are also possible in this way.

2. *String parameters.* We could modify the routines defined so that they accepted strings as parameters. In this way the insertion of a string could be specified as a single operation.

3. *Use of "open."* We could add an "open" instruction which would essentially mark a place in our register. Subsequent insert and delete operations would have the marked place as their implicit positional parameter. Modifications of the fundamental data structure could be postponed until a "close" command or another call of "open."

The first solution forces the module to make decisions which might not pay off. For example, such an implementation would be relatively slow if used for random insertions of single characters. The primary advantage of the first solution is that it has the same specification as the earlier solutions so that one could freely choose between a simple or a sophisticated implementation without changing the rest of the system.

The second solution's primary disadvantage is that it requires a more complex interface between the module and the rest of the system. Some format for the passing of string parameters must be agreed on. This is undesirable from the point of view of [9]. It might also result in a great deal of excess computation being done since strings might be assembled twice: once in the module and once in the parameter format. A good implementation in this direction is not impossible, but it certainly is difficult.

The third solution offers the greatest efficiency potential, but it is a little more revealing of internal struc-

ture. In a sense, this solution shifts the burden assumed by the module in solution (1) to the program which uses the module. Although all the solutions have situations in which they would be appropriate, this is probably the best "general" solution.

The above discussion permits us to discuss a fundamental "tradeoff" which exists between transparency and flexibility of a design. In the above examples we made the point that the lack of transparency intro-

duced was true for *all* reasonable implementations of the proposed design. There are, however, situations in which a proposed virtual machine would be adequately transparent for some base machines, but would have a distinct loss of transparency for others. A design which would increase the transparency for one machine may pose great implementation difficulties or inefficiencies for another base machine. We can offer no better advice than that the designer must be alert for such situations and be prepared to make a difficult decision.

Fig. 6. Simplified block diagram for the HP 2116.
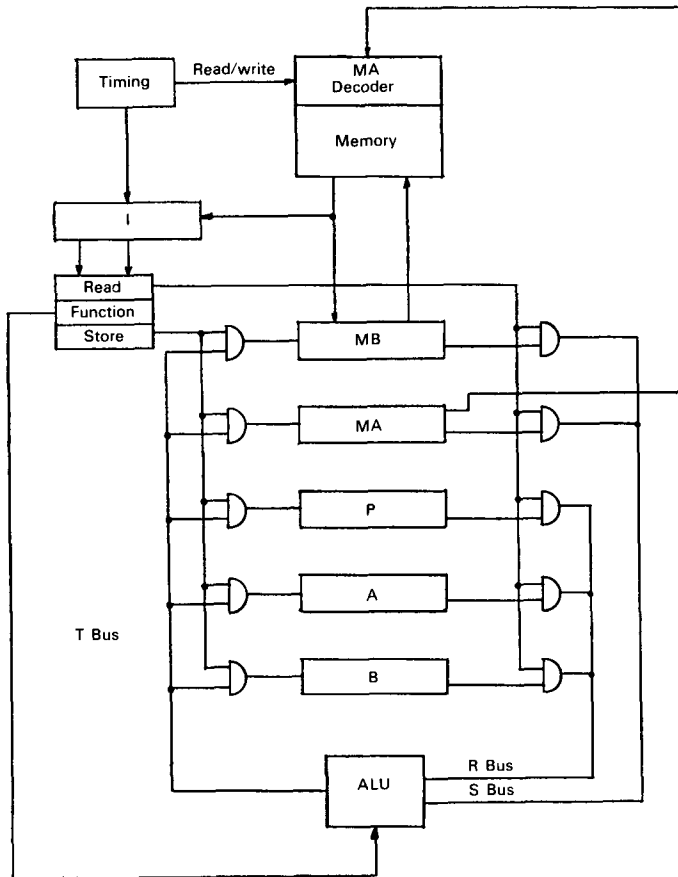


*Key*

A,B  General purpose registers
P    Program counter
MA   Memory Address register
MB   Memory Buffer register
I    Instruction register
ALU  Arithmetic and Logic Unit
—    16 bit wide data or control path

## A Hardware Example

As an example of a loss of transparency at the hardware level consider the Hewlett-Packard 2116. The HP 2116 is a 16-bit, general purpose minicomputer. A simplified block diagram is shown in Figure 6. The HP 2116 contains six registers: memory buffer (MB), memory address (MA), program counter (P), two accumulators or general purpose registers (A and B), and an instruction register (I).

The read/write memory cycle is divided into eight minor cycles. In each minor cycle one or more micro-operations can be performed. For example, the A register can be read to the R Bus during one minor cycle. A partial list of the micro-operations which can be performed in a minor cycle is given in ISP notation in Table I [15].

To see how these micro-operations may be combined to form a machine instruction, consider the timing diagram for the RAL (rotate A register left one bit) shown in Figure 7.

The ISP code describes the RAL instruction execution as follows:

RAL → (
    T0: (MB ← 0); next
    T1: (I ← 0); next
    T2: (I ← MB⟨15 : 10⟩); next
    T3: (R_Bus ← A); next
        (T_Bus ← R_Bus × 2); next
        (A ← T_Bus); next
    T6: (R_Bus ← P); (S_Bus ← 1); next
        (T_Bus ← R_Bus + S_Bus); next
        (P ← T_Bus))
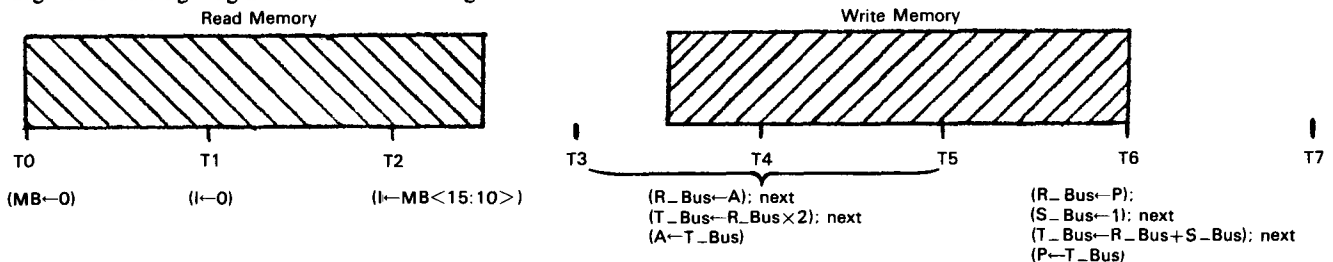
Fig. 7. The timing diagram for rotation of A register.

405

Communications
of
the ACM

July 1975
Volume 18
Number 7

Table I. A Partial List of Micro-operations for the HP 2116

| Read | Micro-operations |
|------|------------------|
| | S_Bus ← MB |
| | S_Bus ← MA |
| | S_Bus ← 1 |
| | R_Bus ← P |
| | R_Bus ← A |
| | R_Bus ← B |
| Store | Micro-operations |
| | MB ← 0 |
| | MB ← T_Bus |
| | MA ← T_Bus |
| | P ← T_Bus |
| | A ← T_Bus |
| | B ← T_Bus |
| | I ← 0 |
| | I ← MB⟨15 : 10⟩ |
| Function | |
| | T_Bus ← R_Bus ∧ S_Bus |
| | T_Bus ← R_Bus ∨ S_Bus |
| | T_Bus ← R_Bus + S_Bus |
| | T_Bus ← R_Bus × 2 |
| | T_Bus ← R_Bus / 2 |

The base machine for the HP 2116 can perform a combination of the micro-operations listed in Table I during one minor cycle. Eight minor cycles can be "stacked" together to form a machine instruction. Note, however, there are some physical limitations imposed by the structure of the base machine. First, the data read from memory during the current memory cycle isn't available until half way through T2. This effectively limits instruction execution to T3–T7. Also for data to be entered into memory it has to be in the MB by the middle of T3. The bus structure also limits some operations. For example, the A and B registers cannot be used during the same minor cycle because they both are connected to the R Bus. Finally some sequence of operations might be essentially a no-operation (NOP) such as ((R_Bus ← A) next; (T_Bus ← R_Bus ∧ S_Bus)). Since there is no store operation the A register remains unchanged.

When we look at the instruction code provided to the user we find that some of the micro-operation sequences which were possible at the base machine level cannot be obtained by sequences of machine instructions.

Consider, for example, the shift-rotate instruction group. In addition to the restrictions imposed by the base machine structure the following manufacturer imposed rules apply to all instructions in the group:
1. Minor Cycles T3, T4, T5 are used for instruction execution. The other minor cycles are used for housekeeping chores such as instruction decode, incrementing program counter, etc.
2. All shifts and rotates take place in T3 and T5.
3. All skip conditions are checked during T4. If the skip condition is met a flag is set so that two is added, instead of one, during the update of the program counter.

Since the machine instruction set allows at most two

one bit shifts per instruction, two machine instructions are required to perform a multiply by eight. The base machine can perform the multiply by eight in one machine instruction as indicated by the following ISP.

RAL8 → (
    T0: (MB ← 0); next
    T1: (I ← 0); next
    T2: (I ← MB⟨15 : 10⟩); next
    T3: (R_Bus ← A); next
        (T_Bus ← R_Bus × 2); next
        (A ← T_Bus); next
    T4: (R_Bus ← A); next
        (T_Bus ← R_Bus × 2); next
        (A ← T_Bus); next
    T5: (R_Bus ← A); next
        (T_Bus ← R_Bus × 2); next
        (A ← T_Bus); next
    T6: (R_Bus ← P); (S_Bus ← 1); next
        (T_Bus ← R_Bus + S_Bus); next
        (P ← T_Bus))

As another example of a loss of transparency consider a memory reference instruction. The instruction in Figure 7 was a register reference instruction and could be executed in one major cycle time. In contrast, a memory reference instruction requires at least two major cycle times: the first to fetch the instruction, the second to fetch the operand. During the instruction fetch major cycle of every memory reference instruction the address portion of the memory word is loaded into the memory address register. This can occur any time after T2 when the instruction is known to be a memory reference instruction. During this time a predesignated register could be added to the address portion of the memory reference instruction. Thus base-displacement (using one of the two accumulator registers as a base register) or relative addressing (using the program counter as the added register) could be performed by the base machine. The ISP for the fetch portion of a memory reference instruction using base-displacement addressing is as follows.

Fetch → (
    T0: (MB ← 0); next
    T1: (I ← 0); next
    T2: (I ← MB⟨15 : 10⟩); next
    T3: (R_Bus ← A); (S_Bus ← MB⟨9 : 0⟩);
        next
        (T_Bus ← R_Bus + S_Bus); next
        (MA ← T_Bus))

Whereas the multiply by eight sequence of micro-operations would be relatively cheap to add to the machine language level machine (add some extra decoding to select an unused bit pattern as the op-code) the cost of enhanced addressing modes may be higher. An alternate design using the same base machine might use a limited memory reference class of instruction (e.g. Load, Store) with enhanced addressing modes and a large

class of register reference operations. Yet another design would use double words for memory reference instructions. The first word could contain the op-code and addressing information, the second the address portion. It is not clear which of these three virtual machines is more desirable.

## An Unsolved Transparency Problem from the Operating System Area

The following example is a problem which we consider to be an important unsolved research problem.

One of the most difficult items in the programming of an operating system is the coordination and synchronization of many concurrent activities. The handling of interrupts (the hardware device available for coordinating concurrent activities) is very difficult for a programmer and likely to introduce errors. For this reason, several operating system designers have introduced an abstract machine for which interrupts no longer exist. Instead, the machines are provided with "process synchronization primitives" which can be used to allow synchronization and communication between several cooperating processes which are, at least conceptually, operating asynchronously and in parallel. Among the better known of these are those of Dijkstra [1, 10], Saltzer [11], and P.B. Hansen [12, 13]. If all process synchronization at all levels (except the lowest which implements the primitives) are to be handled in terms of the primitives, their transparency is an extremely important issue. The loss of any of the fundamental abilities to coordinate concurrent activities would seriously interfere with the usefulness of the operating system.

It is difficult to make a precise determination of the transparency of such primitives because we do not have a precise expression of the essential capabilities of the base machine. We can, however, discuss two of the mentioned primitive systems with respect to a "typical" interrupt system. For both cases some lack of transparency can be shown, but the question of "undesirable" lack of transparency remains a matter of opinion.

Consider first the following situation: We wish to have two cooperating administrative units operating in parallel at least part of the time. One of them is primarily computation and occasionally determines that it needs certain records from the disk. Fortunately, it determines the name of the record it needs well in advance of the time that it must have the record in order to continue. It sometimes determines the names of many records (e.g. 10 or 12) simultaneously. In those cases it must process the records one at a time (an error would be introduced if two were processed at once), but the order in which they are processed is irrelevant. The other process (or perhaps a group of processes) can care for the finding of the records on the disk and bringing them to core. The computational process will proceed until it needs one of the records requested, and if it is not avail-

able, will then wait for it. The disk handling process or processes should bring the records to core in an order unpredictable by the computational process. For optimum use of processing resources, etc., we should like to see the computational process send *one* message to the others with the names of the requested records but receive a "signal" as *each* record arrives so that it will not have to wait for all the records to arrive before beginning its work.

On any reasonable base machine it would be possible to set up such signaling (using the primitives from the THE system, for example). Using the primitives used by Hansen and his colleagues in the RC4000 system [12] we *cannot* set up such conventions. That system has a restriction on interprocess communication so that there is a reply for every message (1:1). In this way the computational process must either send 12 messages or wait for a single reply. (An even more expensive possibility is to send one message, wait for reply, then receive 12 messages and send 12 replies.) The fact that there is a lack of transparency is clear; whether or not it is an undesirable one is a matter of opinion. Hansen has stated [14] that the restriction was introduced as a means of detecting certain common errors and that the restriction was not significant in the situations for which the system was intended.

Another lack of transparency in [12] results from a decision to transmit an eight character message with each synchronization signal. Thus sequences on the base machine with simply synchronization but without such a message are not available through the virtual machine or nucleus. This was a decision based on knowledge that, in the intended application areas, synchronization without communication of a message would not be needed. Apparently the system was not intended to be able to handle teletype communication on a character at a time basis at the nucleus level. It would be unfortunate if each character arriving were handled with an eight character message and similar reply; some lower level mechanism must be used.

It is interesting to note that the primitives used by Dijkstra in THE do not have this particular lack of transparency. From another point of view it is possible to make certain programming errors with those primitives that would be detected by the RC4000 system nucleus [14].

The authors of this paper believe the transparency of Dijkstra's primitives is an open question; in fact, it is a question which required careful definition. We have seen statements of the problem which would yield a negative answer [16]. On closer investigation, it appeared that the statement of the problem eliminated solutions which would be acceptable on practical grounds [17]. The heart of the difficulty lies in our ability to reassign operating system tasks among processes (e.g. to increase the number of processes) to avoid an apparent limitation of the primitive scheme. Since we abstract from the concept of interrupt, supply the synchronizing primi-

tives, and introduce the concept of process simultaneously, the set of achievable computations is very hard to characterize.

From a practical point of view, the ability to stop a process which is not executing a synchronization primitive seems available on the base machine, seems essential, and seems to be missing with Dijkstra's primitives. All attempts to go beyond this statement have failed to date. This example is included in the hope that others will see fit to investigate it further.

## "Suggestive Transparency"

One example of a lack of transparency which resulted in a performance difficulty occurred in the design of virtual memory mechanisms. Usually the virtual machine provided no means of indicating to the mechanism that a segment contained useless information. As a result, many old save areas and similar useless items were moved between core and backup store.

This is one of many situations in which a weaker form of transparency is important. It is often necessary that a mechanism be able to receive *suggestions* about certain base machine sequences although the virtual machine user is not able to cause those sequences. The user of a virtual memory mechanism should be able to *suggest* removal of a segment by indicating that he will not need it again. He must not be able to cause such removal since there may be other users of the segment or the optimal time for removal may not occur until later.

## "Misleading Transparency"

A related problem occurs when the design of the virtual machine suggests that certain virtual machine programs are efficient although they are actually expensive on the base machine. A virtual memory mechanism which simulates a very large random access memory is an example of such a design. To use such a virtual machine efficiently one must have certain additional information. It is often possible and preferable to design a virtual machine in which the expensive sequences are either impossible or difficult to evoke.

## Outside in and Bottom up Procedures in Combination

Advocation of design from the outside in is based on the engineering rule that one should not begin to design an object that is not fully specified. It is difficult to reject this precept. Whenever one begins to build an object with only a muddy view of what it will be, one gets a muddy object.

The difficulties with the outside in approach come because of a number of peculiar characteristics of software engineering.

1. The economics of the industry are such that one is seldom designing a single object; we are usually designing a family of related objects. (Only a proper subset of that family will actually ever exist.)
2. Because of our limited experience with man-machine symbiosis it is often impossible to specify the outside before construction and not want to change it afterwards. As was pointed out in [7] the outside in procedure often adds difficulties in such a change.

In software we begin with a specification of the *family* of objects one wishes to construct. The technique described in [18] allows one to describe parameterized families of objects, but the members must be highly similar items. To describe a broad family of objects we must describe a set of lower level mechanisms which will be common to all members. The family being designed consists of all possible "tops" for that lower level structure. It is at this point that the concept of transparency becomes important. By use of this concept we may assure ourselves that the class of tops which can be built upon the lower level structure includes the family of objects that we set out to design.

References
1. Dijkstra, E.W., The structure of the "THE" operating system, *Comm. ACM 11*, 5 (May 1968), 341–346.
2. Dijkstra, E.W. Notes on structured programming. Report of the Technische Hoogschool Eindhoven, Eindhoven, The Netherlands.
3. Parnas, D.L., and Darringer, J.A. SODAS and a methodology for system design. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press Montvale, *N.J.*, pp. 449–474.
4. Zurcher, F.W., and Randell, B. Multi-level modeling—A methodology for computer system design. Proc. IFIP Cong. 1968.
5. Parnas, David L. More on simulation languages and design methodology for computer systems. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 739–743.
6. Gill, S. Thoughts on the sequence of writing software. In *Software Engineering*, report of a conference in Garmisch, Germany, Oct. 1968.
7. Parnas, D.L. Information distribution aspects of design methodology. Proc. IFIP Cong. 1971.
8. Braden, et al. An implementation of MVT. UCLA report.
9. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM* (to appear).
10. Dijstra, E.W. Cooperating sequential processes. Report of Technische Hoogschool Eindhoven, Eindhoven, The Netherlands.
11. Saltzer, G. Traffic control in a multiplexed computer system. MIT Thesis.
12. Hansen, P.B. The nucleus of a multiprogramming operating system. *Comm. ACM 13*, 4 (Apr. 1970), 238–241.
13. Hansen, P.B., RC4000 Reference Manual. Regnecentralen, Copenhagen, Denmark.
14. Hansen, P.B., private discussions.
15. Bell, C.G., and Newell, A. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, 1971.
16. Patil, S.S. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Proj. MAC, Computat. Structures Group Memo 57, Feb. 1971.
17. Parnas, D.L. On a solution to the cigarette smoker's problem (without conditional statements). *Comm. ACM 18*, 3 (Mar. 1975), 181–183.
18. Parnas, D.L. A technique for the software modules specification with examples. *Comm. ACM 15*, 5 (May 1972), 330–336.