USE OF DATA ABSTRACTION IN PROCESS SPECIFICATION

Franklyn T. Bradshaw, George W. Ernst and Raymond J. Hookway
Computer Engineering and Science Department
Case Western Reserve University
Cleveland, OH 44106

Abstract

Processes in computer systems are often intimately involved with various data abstractions that occur in the systems. The specification of such processes should state the nature of this involvement. This note focuses on a special kind of process, called a realization process, and its specification because it appears to capture the interplay between concurrency and abstraction in a natural way. A realization process has two important properties: it is transparent to the abstract space and it is serializable with processes at the abstract level. Transparency and serializability are the main specifications of realization processes. These concepts are illustrated by an example of the kind of realization processes found in actual computer systems which serves to motivate the discussion.

1. Introduction

Contemporary computer systems contain a variety of concurrent processes; some are quite different in nature from others. One kind of process is very similar to a procedure in the sense that an activation of the process causes some computation to start. When the computation finishes, this activation of the process is destroyed. For example, when a user requests that a program be loaded, typically a separate process is created for this purpose which does the link-editing and relocation, and then destroys itself, returning control to the user. Floyd/Hoare assertions seem to be a good method for specifying such processes. Hookway (1980) has used this method in developing complete formal specifications for a simple linking-loader.

Some processes appear to be quite different from those described above because they are designed to never terminate, but rather to loop forever. Unlike the procedure case, pre and post-conditions do not appear to be useful in the specification of such processes. They are usually created at system generation time which makes preconditions meaningless and since they never terminate post-conditions are not relevant.

These issues surfaced when we started doing research on formal verification of concurrent processes, several years ago. It since became obvious to us that such research could not make meaningful progress until we had a better understanding of what is involved in the specification of concurrent process. The reason is that formal verification is, by definition, proving that an implementation meets its specification. Hence, the kind of specification that is used can have a major impact on the verification rules. This note is only concerned with formal specification of concurrent processes even though our real goal is formal verification. The purpose of this note is to describe our approach at the intuitive level so that the basic concepts are not buried in technical details. The reader is referred to Ernst (1982) for a more technical treatment of the concepts in this note and formal verification.

2. Data Abstraction and Concurrency

Often processes in actual computer systems are intimately involved with various data abstractions that occur in the systems. The specification of such processes should state the nature of this involvement. Frequently, a process manipulates the respresentation of certain abstract objects but has no visible effect on them at the abstract level; i.e., the abstract values of the objects remain unchanged. Usually, such processes are of the non-terminating variety described in Sec.1.

For example, a front-end process takes input data and assembles it into blocks of an appropriate size before passing it on to the processes for which it is intended. But at a more abstract level, the front-end process is invisible because it has no effect on the values computed at the abstract level but only on the way data is passed between processes. This paper is primarily concerned with the specification of such processes.

To be more explicit, suppose that some abstract objects, a1, a2,... have been created and that these objects are collectively represented by objects r1, r2,... in the realization space. That is, the values of the r's determine the values of the a's. As usual some of the r's may themselves be abstract objects created by some lower level of abstraction, but all of the r's are used in the representation of one or more of the a's. We also assume that there are some procedures p1, p2,... that can access and modify the abstract objects. Of course, the p's accomplish this by accessing and modifying the r's which represent the a's.

The kind of process described at the beginning of this section has direct access to the realization objects and hence we will

call it a realization process. It manipulates the r's concurrently with processes that manipulate the a's. Of course, the latter is accomplished by calling the p's which in turn access and modify the r's. Hence, both the realization process and a process at the abstract level may be concurrently processing the r's.

The realization process is aware that one or more processes at the abstract level may be manipulating the a's by calling the p's. Of course, the realization process knows nothing about the nature of the abstract computation, but assumes that any sequence of legal calls on the p's can occur. A process at the abstract level, however, knows nothing about the realization process which must be transparent to the abstract space. In fact, usually there is a different implementation of the same abstract space in which there is no realization process. In such an implementation the a's remain the same as do the abstract specification of the p's, but different r's may be used and the p's manipulate them differently. Finally, there is no realization process manipulating the r's in the alternative implementation of the abstract space.

We have purposely avoided describing the concept of a realization process in terms of a particular language like Euclid or Ada so that the concept does not become entangled with the idiosyncracies of a language. We believe that any good concurrent programming language must have facilities for implementing abstractions and realization processes. This, of course, includes constructs like synchronization primitives but they do not appear to be the primary issue, at least not at this level of detail. Later in this note we will describe how realization processes fit into our concurrent programming language which is a derivative of Modula. But, next, we will give a concrete example of the kind of realization process that occurs in actual computer systems. This example will also clarify the roles of the a's, r's and p's.

3. An Example of a Realization Process

This example is a level of abstraction in the implementation of a file system. It allows a process to initiate a file operation and then continue with its execution. At a latter time the process can obtain the result of the file operation. This allows a process to overlap its execution with the exectuion of the file operations. Such file operations are a fairly low level of abstraction because a file read is split into two parts. However, it is definitely an abstraction because it does not deal with disk sector numbers, etc. Many systems have such file operations, e.g., the IO$ of the Sperry Univac 1100 Series Operating System (EXEC-8). Of course, our example will be much simpler than the Univac system, but it is based on the same basic idea.

Each file is an abstract object indexed by integers. The only other kind of abstract object in this example is a channel which is used in reading and writing files. To operate on a file a process must connect a channel to the file because the procedures which manipulate files have a channel as one of their parameters. For example, write(c,i,e) writes element e in the ith position of the file connected to channel c.

The realization space also has files and channels. A realization file is very similar to an abstract file, but a realization channel is quite different than an abstract channel. Intuitively, a realization channel is a buffer whose elements are read and write requests. For example, the procedure write(c,i,e) merely records a request for this write operation in the realization channel c which has a buffer for such requests. After the request has been put into the buffer, write returns to the process which called it. This process then continues even though the actual file update has not been made. In fact, at the abstract level it appears that the file has been updated because the abstract file has element e in position i after the call on write.

The implementation of this abstraction contains a realization process whose function is to update the realization files. This process continually cycles through the realization channels. Whenever it finds one that contains an outstanding request, it removes the request from the buffer in the channel and performs the actual operation on the realization file.

File reads are done in a manner similar to writes. A process requests a file read by executing readreq(c,i). This procedure call puts a request to read element i from the file connected to channel c in the request buffer of c, and returns to the calling process which then continues. At a later time the realization process performs the read on the realization file and puts the result in a special read buffer in the realization channel. The read operation is completed by executing readcomp(c,e). This procedure call removes the result of the actual file read from the read buffer of c and assigns it to e.

Fig.1 summarizes this example of data abstraction. The correspondence between the abstract and realization values of a file is that the realization file will become identical to the abstract file after all the outstanding updates are made on the realization file. These updates are stored in the request buffer of the realization channel connected to the file. Only one channel can be connected to a file at any time. If no channel is connected to a file its abstract value is the same as its realization value.

The read queue of a realization channel is the initial segment of the queue in the abstract channel. The remainder of the

Abstract Objects

files: Each file is a vector of elements of some predetermined type.

channels: Each channel contains the name of the file to which it is connected and a queue of the elements that have been read requested.

Realization Objects

files: Each file is a vector of elements of some predetermined type.

channels: Each channel contains

1. the name of the file to which it is connected;
2. a queue of the outstanding read and write requests;
3. a queue of the elements which have been read from the realization file.

Abstract Procedures

1. write(c,i,e) writes element e to the ith position of the file connected to channel c.
2. readreq(c,i) reads the ith element from the file connected to channel c and puts the result in the read queue of c.
3. readcomp(c,i) removes an element from the read queue of channel c and assigns it to e.
4. connect(c,f) connects channel c to file f.
5. disconnect(c) removes the connection between channel c and its file.

Figure 1. A summary of data abstraction in the file example.

abstract queue is the sequence of elements whose read requests are stored in the request buffer of the channel. The results of read requests occur in a channel's read queue in the order of their execution.

In presenting this example, we attempted to suppress as much detail as possible without losing its essential nature. An implementation of this example is reasonably involved, perhaps more so than the above description implies. Complete formal specifications for this example are given in Bradshaw and Ernst (1979) together with an implementation. The kind of detail suppressed in this note is that there must be a way to connect channels to files (given in Fig.1), and a way to create new files (not given in Fig.1). Processes must be synchronized. For example, the readcomp procedure must "wait" when the read queue is empty. This in turn causes the calling process to wait until the realization process performs the actual file read. The reader is referred to Bradshaw and Ernst (1979) for this kind of detail.

Our interest in this example is that it clearly illustrates the kind of realization processes found in computer systems. Usually they serve a very practical purpose. In our example, the realization process allows disk access time to be overlapped with the computation of processes at the abstract level. Fig.1 is a concrete example of the a's, r's and p's of Sec.2 (i.e., the abstract objects, realization objects and abstract procedures in Fig.1, respectively). The example clearly shows the involvement of the realization process with data abstraction. The next section discusses the nature of this involvement and its formal specification. From an intuitive point of view we know that the function of the realization process in the above example is to perform the actual file reads and writes. But how do we formally specify this intuitive idea?

4. Specification of Realization Processes

This section assumes that the a's, r's and p's are defined as in Sec.2. The general form of a realization process is B; while true do C, where B is a sequence of declarations followed by some statements which initialize variables. After "executing" B, the process executes C over and over again, never terminating. Since the realization process manipulates the realization objects r1, r2,..., they are global to B and C.

Typically, a realization process has two functions: (1) it must be transparent to the abstract space and (2) it must apply some transformation to the realization objects. The latter is usually related to some aspect of system performance. For example, often it is the responsibility of the realization process to remove and process entries in buffers. If it fails to do this, the buffers will become full which will block processes that add entries to the buffers. For this reason (2) is primarily concerned with termination. This note only deals with partial correctness (correct results when and if results are produced), and hence we will only consider (1) above.

To simplify matters, consider a single execution of C, the body of the unending cycle. This execution must satisfy the following property:

> For each abstract object a1, a2,... its value after executing C must be the same as its value before the execution.

We call this property transparency because it states that C has no visible effect at the abstract level. To see what this property entails, note that the values of the r's determine the values of the a's. The specification of the abstraction describes this relationship between the a's and the r's. An execution of C modifies the values of some of the r's and these new values of the r's determine the new values of the a's. Transparency requires the new and the old values of the a's to be the same.

We assume that a process at the abstract level can execute concurrently with a realization process; hence, both can concurrently process the r's. This implies that there must be some kind of synchronization between the two processes so that, for example, they don't attempt to simultaneously modify the same component of the same r. The usual kind of synchronization primitives should be quite adequate for this purpose.

A more subtle problem is that manipulating abstract objects concurrently may produce intermediate states whose values are not well defined. Consider invoking one of the p's with abstract object ai as an actual parameter. The input and output values of ai will satisfy the specifications of the procedure, assuming that its implementation is correct. However, half way through the execution of the procedure, the r's will have intermediate values whose correspondence to the abstract value of ai may be undefined, or at least different than either its input or output value. At this point, the body C of the realization process may concurrently start to manipulate one of the r's which is part of the representation of ai. The difficulty is that transparency requires C to preserve this intermediate value of ai which from an intuitive point is uncomfortable at best.

Due to this difficulty we require that the body C of the realization process must be serializable with processes at the abstract level. A set of processes is serializable if any concurrent execution of them is equivalent to some sequential execution. This concept is based on the notion of serializability in data base systems research where it has become an important concept; see, for example, Ullman (1980). Our interest in serializability is that it appears to be a property of realization processes. Hence, in addition to the usual kind of process synchronization, a realization process must contain sufficient additional synchronization to satisfy serializability. Of course, processes at the abstract level must also be serializable but this should be a result of the data encapsulation provided by abstraction. That is, processes at the abstract level can only manipulate the realization objects by calling the p's. Consequently, we require that the bodies of the p's and the body C of the realization process be serializable. This causes the realization process to be serializable with all processes at the abstract level.

Transparency and serializability constitute the major portion of the specification of a realization process. The remaining specifications have to do with invariants of the realization objects. We will not deal with this aspect of specification here because it is widely discussed in the literature (e.g., see the module invariant in Ernst and Ogden, 1980).

So far we have only considered a single realization process, but several of them may be concurrently manipulating the same r's. The bodies of these realization processes must be transparent to the abstract space, and together with the bodies of the p's, they must be serializable.

Again we emphasize that the above comments only address the issue of partial correctness; deadlock and other kinds of non-termination are considered outside the scope of this note.

5. Realization Processes in Modula

This section describes how realization processes can be incorporated into Modula (Wirth, 1977). This serves to make the ideas in the previous sections more concrete and also points out some of the ways that realization processes interact with other language mechanisms.

We have extended Modula by adding a facility for formal specifications to the language (Ernst and Ogden, 1980). Fig.2 is a module in our extended Modula which contains a realization process. In the following, Modula refers to our extension of the language. The module in Fig.2 implements an abstract object M of type T0. It also defines an abstract type AT and a procedure q, all of which can be referenced from outside the module. MPost(M) specifies the initial value of M which is computed when the module is declared. Our convention is that the variables referenced by specifications are enclosed in parenthesis after the identifier which names the specification.

The local module variable lv is used to represent the abstract object M, and CM(M,lv) specifies the relationship between M and its representation.

There are 5 major components to the declaration of an abstract type like AT. The second line in the declaration of AT specifies what an instance as of AT looks like abstractly. Such an abstract object is represented by a variable rs of type T3 which is allocated when an instance of AT is declared. Although rs is used exclusively for the representation of as, part of lv may also be used in the representation of as. (Usually lv is a large structured variable.) The correspondence of as to its representation is given by CT(as,rs,lv). ITDec and CTDec are procedures which are executed when an instance of AT is allocated or deallocated,

```
module M:T0;

   define AT,q;
   exit assertion MPost(M);

   var lv:T1;
   correspondence CM(M,lv);
   invariant IM(lv);

   abstract type AT;
      abstract structure as:T2;
      realization structure rs:T3;
      correspondence CT(as,rs,lv);
      invariant IT(rs);
      initialization ITDec;
      cleanup CTDec;
   end AT;

   procedure q(var vp:AT);
      use var M;
      use realization var lv;
      entry assertion Qpre(vp,M);
      exit assertion QPost(vp,M,#vp,#M);
      Qbody
   end q;

   realization process rp;
      use var lv;
      B;
      while true do C;
   end rp;

   MBody; rp

end M;
```

Figure 2. A module that contains a realization process.

respectively.

The procedure q has an instance of AT as its only parameter, but it also modifies the abstract global variable M as specified by the second line in the declaration of q. Its body can access and modify the realization object lv as well an object of type T3 used in the representation of vp. The exit assertion specifies the I/O relation of the procedure. In this statement #vp refers to the input value of the parameter while vp refers to its output value. Similarly, # is used to differentiate between the input and output values of M.

The realization process rp has access to the global variable lv as indicated by the second line in the declaration of rp. The remainder of rp is as described in the last section.

The body of the module is executed when the module is declared. The call on rp after MBody causes its execution to commence concurrently with the process in which the module is declared.

This very brief description of Fig.2 is not intended to describe the module in detail but rather to point out some of the major components of the module so their interaction can be discussed. We also note that the module in Fig.2 is not intended to be general. For example, usually there will be more variables local to the module. However, all such variables can be used in the same way as lv, and thus lv is intended to typify local module variables. Similarly, a module can define more abstract types and procedures, and a module can also contain several realization processes. The purpose of q, AT and rp in Fig.2 is to typify all instances of these constructs.

One of the most important features of the kind of data abstraction in Fig.2 is that several different abstract objects can use a single variable as part of their representations. For example, lv is used to represent the abstract object M. But lv can also be used as part of the representation of the different instances of AT. Of course, the procedures that manipulate abstract objects must be able to access their representations.

The ability to have the representations of several abstract objects "share" a single variable is important because certain implementation techniques apparently rely on this capability. As usual there is a price for such flexibility. Procedures which manipulate one abstract object have access to the representation of others and hence can "side-effect" them while otherwise operating correctly. For example, a call on q may correctly manipulate M and its parameter, but may side-effect a different instance of AT because QBody has access to lv which may be used in the representation of all instances of AT.

Hookway (1980) has developed verification rules which prohibit such side-effects. Using these rules to verify a procedure like q in Fig.2, one must prove that no instance of AT except vp is modified by q. The absence of such side-effects must explicitly be proven because it is not enforced by other mechanisms like scoping rules.

The realization process in Fig.2 has access to lv but not to the unique instance of T3 associated with an instance of AT. In general, realization processes can only access that part of the representation of abstract objects which is stored in local module variables. However, modifying these variables may modify the abstract objects which they represent.

For the reasons given in the last section the realization process should possess the transparency property. To verify this requires a proof because the realization process has the ability to modify abstract objects. Such modifications would be considered undesirable side-effect similar to those that procedures like q might produce. In fact, in many respects the body C of the

unending loop looks like a procedure that manipulates no abstract object and has no side-effects either. The point of this discussion is that a good data abstraction facility must allow representations to share memory. The specification and verification of such abstractions provides almost all of the mechanisms that are needed for specifying and verifying the transparency of a realization process.

The realization process, unlike the rest of the module, has no explicit specifications associated with it. The reason is that all realization processes have the same specification, i.e., transparency. This is similar to the absence of side-effects in procedures which is an implicit specification whose verification requires an explicit proof. Serializability is a requirement of all module procedures as well as the realization process and hence is an implicit specification. In fact our specification language is not strong enough to express transparency and serializability, explicitly. But logically sound verification rules will require all modules to possess these properties.

6. Conclusions

The main contribution of this note is the concept of a realization process which possesses the property that it has no visible effect on abstract objects even though it manipulates their realizations (i.e., representations). In addition to this transparency property, a realization process is serializable in the sense that any concurrent execution of its body with the bodies of procedures that manipulate abstract objects, is equivalent to some sequential execution of them. Transparency and serializability are the major specifications of realization processes and any implementation of them that meets these (and other minor) specifications are considered to be correct. A more formal development of these ideas can be found in Ernst (1982).

We believe that it is important to isolate realization processes as a special subclass of processes because they possess some common properties which other kinds of processes do not possess. An additional complication is that transparency and serializablity are "meta-properties" in the sense that they cannot be expressed in our specification language. This implies that verification rules must deal with them in a special way.

Some processes look much more like procedures than realization processes, as discussed in Sec.1. Floyd/Hoare assertions appear to be a good way to specify such processes. However, this method does not seem to be at all appropriate for realization processes. Our studies indicate that the processes in many computer systems fall into one of these two categories. There may be other classes of processes, but we have not been able to isolate any, yet.

Our focus on realization processes is the thing that differentiates our research from other research in this area. For example, Owicki and Gries (1976) and Apt, et al (1980) both develop verification rules for a very general class of concurrent processing. Hence, their rules cannot focus on the interaction between abstraction and concurrency like this note does. The research that appears to be most similar to ours is Owicki (1979) because it also focuses on the interaction of abstraction and concurrency. Even so, it is very different from this note because it does not isolate realization processes as possessing special properties but rather deals with a more general kind of concurrency.

References

1. Apt, K.R., Francez, N. and de Roever, W.P., A Proof System for Communicating Sequential Processes, ACM Trans. on Programming Languages and Systems, July, 1980, pp.359-386.

2. Bradshaw, F.T. and Ernst, G.W., Formal Specifications of a Layer in a File System, Report No. ESCI-79-1, Computer Engineering and Science Dept., Case Western Reserve Univ., 1979.

3. Ernst, G.W., A Method for Verifying Concurrent Processes, Report No. CES-82-1, Computer Engineering & Science Dept., Case Western Reserve Univ., 1982.

4. Ernst, G.W. and Ogden, W.F., Specification of Abstract Data Types in Modula, ACM Trans. on Programming Languages and Systems, Oct., 1980, pp.522-543.

5. Hookway, R.J., Verification of Abstract Data Types whose Representations Share Storage, Report CES-80-2, Computer Engineering and Science Dept., Case Western Reserve Univ., 1980.

6. Owicki, S.S., Specifications and Proofs for Abstract Data Types in Concurrent Programs, Program Construction, Lecture Notes in Computer Science, Vol. 69, 1979, pp.174-198.

7. Owicki, S.S. and Gries, D., Verifying Properties of Parallel Programs: An Axiomatic Approach, Communications of the ACM, 1976, pp.279-284.

8. Ullman, J.D., Principles of Database Systems, Computer Science Press, 1980.

9. Wirth,N., Modula: A Language for Modular Multiprogramming, Software--Practice and Experience, Jan., 1977, pp.3-35.