Development of Real-Time Hardware/ Software Systems

J. W. Layland
Communications Systems Research Section

This report presents a concentrated overview of the critical issues and tools for the development of real-time systems. Real-time systems are defined to be those which perform their actions in response to stimuli from outside themselves, and which must respond to these stimuli within fixed, predetermined time limits. A real-time system with many independent external stimuli almost certainly contains a large number of interacting asynchronous processes. From the viewpoint of the equipment surrounding this real-time system, these processes operate in parallel, and their operations are only partially ordered. A single process can be well represented by a flow chart which relates step-by-step exactly which action follows the last one. Multiple interacting asynchronous processes cannot be conveniently described by a flow chart of their combined operations, even though when taken individually each process can be depicted on a flow chart. However, each of the multiple asynchronous processes can be readily understood as a finite state machine, and the interaction between machines can be graphically represented by a state-transition net, or "Petri-net." This report develops the use of such nets for software and hardware design through description and example.

I. Introduction

The purpose of this report is to present a concentrated overview of the critical issues and tools for the development of real-time systems. Real-time systems are defined to be those which perform their actions in response to stimuli from outside of themselves, and which must respond to these stimuli within fixed, predetermined time limits. Typically there are many independent stimuli which require a response. Each stimulus causes the activation of, or creation of, at least one process within the system, which in turn will develop the responses required by the source of the stimulation. In software terminology, a process is a sequence of operations which is fully ordered, and has a well-defined start and end. A real-time system with many independent external stimuli

almost certainly contains a large number of interacting asynchronous processes. From the viewpoint of the equipment surrounding this real-time system, these processes operate in parallel, and their operations are only partially ordered.

A single process can be well represented by a flow chart which relates step-by-step exactly which action follows the last one. Multiple interacting asynchronous processes cannot be conveniently described by a flow chart of their combined operations, even though when taken individually each process can be depicted on a flow chart. The early conceptual developments (Ref. 1) which engendered the current raging fad of structured programming were aimed primarily at the taming of the complexities of software containing asynchronous processes. The more recent, more formalized development of structured programming (Refs. 2, 3) has emphasized the decomposition of single-process flow charts over the less tractable real-time problems. This report is one step of an attempt to bridge the gap between the underlying principles of structured programming and the problems of developing a working real-time system.

The act of solving a complex problem, or designing a complex system can be characterized as a hiding of locally irrelevant details, so that those details which are relevant to the locale of interest can be properly studied and interpreted. A significant fraction of such improvements as have been observed with structured programming may well be attributable to such hiding of detail as is induced, instead of to the rigorous application of the restricted control structures themselves.

In Part II we present some views on the process of solving complex problems or designing complex systems. Part III considers the problem of choosing a language for the description and/or implementation of real-time software. Part IV presents assorted advice on the implementation of real-time systems. Some large portion of Part IV consists of quasi-obvious common sense, yet it is this collection of obvious things that taken together represents the bulk of the increment in difficulty between nonreal-time and real-time systems.

Part V is an introduction to the syntax of the Petri net (Ref. 4), or state-transition net. Such nets describe the partial ordering of events within a finite-state machine, and as such can be well used to represent the interactions of the asynchronous processes in a real-time system. Part VI makes use of the transition nets to describe the behavior of a suitably substantial synthetic example system. This description includes both processes which are totally software and processes which operate both in hardware and software. Part VII and subsequent discussions contain additional examples of processes which are represented by transition nets, and also contain examples of the mechanization of transition nets as both software and hardware. They will be written in the future as additional experience is gained through the use of this system description tool.

II. On Problem Solving and System Design

Design problems, whether they are destined to be executed as computer software, logical hardware, or as some entirely unrelated material come in two varieties. They are either small enough that their entire character fits inside the problem-solver's head as a single chunk, or they are not. An example of a thing which is single-chunk could be "a transparent green glass marble." An example of a thing which is not single chunk could be "a thousand marbles rolling down a giant slide." The features which make this second example not a single chunk for most of us are that it contains a large number of identifiable things and that these things move in a quasi-independent way.

Differentiation between those things which are comprehensible as a single chunk, and those things which are not is subjective, and varies from person to person on the basis of training, experience, desire, mental capacity, etc. Single-chunk problems can be solved or implemented in an almost random fashion without undue difficulties. Larger problems must be either laboriously studied until they resemble single chunk problems (if this is possible), or dissected into smaller problems which are single chunk in nature, and which taken together solve the original problem. In order to make the subproblems single chunk when the original one was not, some information or design detail that is present in the problem as a whole must be hidden from the subproblem. Contrarily, each of the subproblems should contain detail that is hidden from the others. It is this systematic selective hiding of design details which makes solution of the larger problems possible. This problem-solving philosophy is similar to one which has been voiced by Parnas (Refs. 5, 6).

The design rules and restricted control structures of structured programming are intended to guide the dissection of a problem into subproblems which are single-chunk in nature, not only for the designer, but also for his managers, and for any future casual readers. The oft-quoted structure theorems (Ref. 2) tell us that we can reorganize any flow chart using only three control structures, no matter how complex it may appear. Flow charts, however, are single-process in nature and cannot

completely and/or conveniently represent all features of a multiprocess operation. Small details, like timing constraints or conflicts, don't fit and are left for prose commentary, or worse, are left to clutter the designer's head while he works on each of the pieces, because these details are not hidden from any of them. Because these most troublesome details of real-time software do not fit a flow-chart representation, the ritual of structured programming as it is conventionally preached cannot solve the real-time software designer's biggest problems, although it can assist him in handling those pieces of his problem which can be isolated as single processes.

We should remember, however, that some of the earliest work in the realm of structured programming was aimed directly at the methodical design of a system containing multiple asynchronous processes (Ref. 1). That these early efforts were successful is evidence that the principles which underly structured programming can be used to great advantage in real-time systems, even if some large part of the recently formalized superstructure cannot.

III. Programming Languages for Design and Implementation

We can segment the design of a system into two primary tasks. The first is to fully describe the actions of the system in its response to the external stimuli, be they human or electromechanical in origin. The second is to implement those actions within the available hardware/software resources. Again, we are hiding details, by considering first what is to be done, and then, separately, how it is to be accomplished. Actual development almost always requires an iteration between these two phases, because some of the actions which appear to be needed may be impossible to implement, or simply expensive, whereas some actions which are close in some sense to the needed action might be very much simpler to implement. The alternate actions could not be known to be acceptable without considering the overall response demands of the world outside the system being designed.

With some problems, and an appropriate programming language, a complete description of that problem can also be the implementation of its solution. The ease with which any task can be performed via a specific programming language actually seems to depend upon the extent to which that language is able to describe the problem to be solved, as opposed to implementing the solution to that problem. A near-classical example here is any numerical formula calculation, and one of the popular FORmula TRANslation languages. Such languages intrinsically hide a great quantity of implementation detail from the problem solver; and therein lies the power of the higher level languages (HLL) with respect to their proper domain. Outside of that domain, however, any specific higher level language may be no better than a machine assembly language (MAL), and may, in fact, be much poorer if the operations required to implement the required problem solution cannot easily be synthesized from the repertoire of that higher level language.

The power of a language with respect to a particular problem may be measured by the number of statements within that language which are required to implement the solution to that problem. For largely algebraic problems, a single FORTRAN statement may contain the equivalent of many tens of machine-assembly language instructions. On the other hand, to complement a bit in a data-structure may require ten FORTRAN statements and only two or three MAL statements. Both language classes would implement the operation, rather than describe it, and both would be machine-dependent in nature. Some other types of operations would require a comparable number of statements in either the MAL, or an HLL; some of these would be machine-dependent by the nature of the data structures involved, and in none of these would the implementation details be hidden from the designer. Thus, only in very special circumstances does any programming language hide enough details to be appropriate for describing the system design.

For most problems there is no uniformly most powerful language, and the choice of an implementation language (or languages) must be made on other considerations. Standardization upon a major HLL is one apparently reasonable way. However, using an HLL to implement operations for which it is poorly suited may require the implementer to know details of the implementation of the HLL itself, thereby greatly increasing the difficulty of his task. In these situations, the HLL has hidden the wrong machine details from the designer; the HLL is nontransparent to some specifically important elements of the machine's hardware capability.

A mixed arsenal of an algebraic HLL, used where appropriate, and the target computer's MAL can combine the best features of both. The MACRO capability that is present with many MALs, or a machine-independent MACRO preprocessor (Refs. 7, 8) can be used to locally extend the power of the MAL with respect to the problem at hand. It may, in fact, be quite desirable to use

MACROs to implement the restricted control structures of structured programming within the MAL and HLL, and maintain commonality between control statements of both.

Although most specific major HLLs seem presently to be a poor choice for the sole implementation language of a real-time system, the HLLs as a class are not yet ruled out. Several attempts have been made, usually in a university environment, to define an HLL specifically for the implementation of real-time, or systems software. None of these have as yet achieved widespread acceptance outside of their native centers, yet the achievements that are claimed are positive enough to warrant their serious consideration. BLISS (Ref. 9) is one such language, which was designed originally for the PDP-10, and has since been adapted for the PDP-11 minicomputer. BLISS is ALGOL-like in structure, yet was designed to require negligible software support at execution time and to allow the program designer great flexibility in the accessing of data. Because the method for accessing various data elements is specified as the data is declared, BLISS programs are not machine-independent, but can apparently be readily modified to transport them between machines. Two other relevant efforts are BCPL (Ref. 10), and the Graphical Automatic Programming system (Ref. 11), which is almost a language.

Graphical representations have been used for many years in the design of software systems. Simple flow charts are widely accepted as software documentation and software design documentation. The syntax of a programming language and the operations required to interpret it have been graphically represented in the form of a finite state machine (FSM) (Ref. 12). Similar FSM representations have been used to describe the interactions between a user and a computer operating system, and to design communications-handling software for a time-sharing operating system (Refs. 13, 14). The multiple asynchronous processes of a real-time system can each be understood as an FSM which interacts with the periphery equipment and with the other FSMs as it acts to produce the required responses. These interactions, and the partial ordering of actions of the FSMs, can be well represented graphically, even though not by the conventional flow chart.

One particular graphical FSM representation known as the Petri net (Refs. 4, 15), or state-transition net was developed to deal with asynchronous interactions between FSMs, and contains the operations necessary to describe the partial ordering of events, and the timing interactions of asynchronous software processes, as well as within hardware realizations of an FSM, or at the hardware/software interface. The syntax of the transition net representation is defined in Section V.

IV. Implementation of Real-Time Software

Three characteristics are desired for real-time software, that it: is consistent, is reasonably efficient, and meets appointed deadlines of execution. One of the last things a designer wishes to have is for the results of computation to vary from time to time, with no apparent change in input parameters and conditions. For a nonreal-time single-process computation, this can be assured by ensuring that all parameters and variables that are used by that process are preset to their proper initial condition at the start of the process. This requirement is obvious, yet it is one source of occasional errors. It is also obvious that all data used as input to a process must be valid when that process starts executing and not changed by another process until the using process has terminated. However, failure to satisfy this requirement is probably the most common error encountered in real-time systems. The problem is basically one of communication between, and synchronization of, intrinsically asynchronous processes. It appears as a race between events in logical hardware, as well as intermittent software errors.

Stimuli from the world surrounding our computer almost always appear as a logic signal at the interrupt portion of the computer's hardware at some particular point in time. In due course, the computer will respond to this interrupt signal by saving a small amount of information (the current instruction address, and perhaps some additional status) in a predetermined location in memory, and obtaining a new current instruction address for the interrupt subroutine associated with this signal. The computer interrupt hardware will also prevent a recurrence of this interrupt response until commanded otherwise. The next instruction executed will be the first instruction of the interrupt subroutine. The first opportunity for inconsistency is here. If all resources within the computer which are needed to service the equipment which initiated the stimulus have been saved by the computer's automatic response, we are free to service that equipment. If a resource is needed which has not been saved automatically, its current status must be temporarily saved before the resource is used within the interrupt subroutine, and then restored to its original condition after use and before returning to the process which was interrupted; the penalty for not doing so is lack of consistency in the interrupted process. Examples of resources whose state must be saved if they are to be used include hardware registers, arithmetic status bits, and

software registers which contain intermediate temporary results for subroutines called.

In the interest of efficiency, however, it is unwise to save and restore any resources which are not needed to service the requesting equipment, since these operations, while necessary for consistency on the resources used, represent a nonproductive overhead with respect to actual tasks. A strong case can be made for using a minimal MAL subroutine for at least the most frequent interrupts. By doing so, the resources needed can be made visible and controlled, thus restraining unnecessary overhead. Services requested by equipment with an interrupt signal can often be categorized with respect to the time available to perform them. Some must be performed immediately—otherwise there is no real excuse for generating the interrupt at all. Others could be deferred to another slower software process by buffering several requests together. Deferrable services for which no additional overhead is generated to allow them to be performed within the interrupt subroutine may as well be performed there. Deferrable services which would require additional overhead should be deferred if by doing so the overhead dictated by consistency requirements would be lessened.

Some simplification of process interfaces and concomitant reduction of overhead can often be achieved by anticipating the more complex parts of the required responses, and precomputing these when it is convenient to do so. These precomputed results can then be delivered via a "'mailbox"' to the using process when needed, or via a much-simplified interrupt-driven process to the external system hardware. In the implementation of software, results which are precomputed for the interrupt routines can vary greatly in character and extent. Their single common feature, and the feature which they share in common with deferred computations, is that they have been removed from the most time-critical of their possible points of action to a domain of (hopefully) lessened time criticality.

The second major category of consistency failures occurs at the interface between our interrupt subroutine and the software processes that perform those services that were deferred. This is the interface between cooperating sequential processes to which the synchronizing primitives of Dijkstra (Ref. 16) and the considerable following literature apply. The essential element for consistency here is that during no interval of time should more than one process be empowered to change the same location of memory. Areas into which a given process may store data should be privately owned by that process while it is empowered to store that data, and then deliver intact to whatever process will use that data. Semaphores are used for communication between processes, just as the interrupt signals were used for communication from the hardware to the software processes. Whether the manipulation of these semaphores is performed by synchronization primitives (Refs. 16, 17) or is implemented directly via increment/decrement and test instructions, they must, for the moment of their change, be made private to the process which is changing them.

The avoidance of deadlock dominates much of the literature on multiple process computation. A deadlock is said to exist between two or more quasi-independent processes whenever all of them possess at least a part of the resources they need for completion, none of them possess all of the resources they need for completion, and none of them are willing and/or able to release those resources to allow another process to complete. In a committed real-time system, deadlocks should not only be avoided, but they should be designed out. Any process should possess all of the resources it needs to allow it to complete its activity before it becomes active—with NO exceptions.

Determining whether the appointed execution deadlines are all satisfied can only be done with certainty after implementation is complete. This is accomplished by means of a Ganttchart (Ref. 18) or time-occupancy diagram for all of the processes with deadlines through their critical time interval (Ref. 19). If reasonably large margins are included, a high confidence in meeting deadlines can be achieved by using estimates of process execution time once enough of the overall design is completed. If deadlines are not to be met, some revision is needed, which could be as simple as increasing efficiency by reorganization of processes to reduce overhead, or as involved as renegotiating system requirements, or acquiring a new computer. For a real-time system, the side effects from deadline problems can be minimized by designing and implementing first the processes that service the highest frequency and most time-critical interrupts, and then proceeding into the more mundane parts of the system.

V. The Petri-Net Representations

As observed earlier, it is convenient for the designer of a real-time system to conceptualize his system as an ensemble of finite state machines (FSMs) which operate on command and work together to produce the intended system responses, much as the musicians of an orchestra follow their own score yet interact time-wise to reproduce the effects intended. Each of the designer's FSMs needs

only to be concerned with what it is required to do to service the needs of the periphery equipment. Each of the FSMs assumes certain states as a result of interactions with the periphery equipment or with certain others of the FSMs; the future action of each FSM is governed by its current state and future inputs. The system designer needs a representation for the FSMs that can fully describe what they do in an unambiguous, concise way. This designer's representation must also be lucid enough to permit a system implementer to add such additional interactions as may be needed to integrate the FSMs together into one computer, to allocate the FSMs between several computers and supporting hardware, or to inform the designer that his dream can't be realized within the budget.

The FSM representation known currently as the Petri-net was introduced by Dr. C. A. Petri in 1962 to deal with the communication between automata (Ref. 4). It bears a significant generic relationship to earlier graphical network representations for FSMs; for example the Neural Networks of McCulloch and Pitts (Ref. 20). Petri nets currently form the core of a slowly growing literature concerning the analysis and exploitation of parallelism in computing hardware or software (Refs. 15, 21–30). The components of the original Petri net have the same basic appeal for representation of FSMs that the basic three structures of structured programming do for single-process computations: their syntax is exceedingly simple, yet is capable of concisely describing the interaction between cooperating sequential processes.

Formally, a Petri net is a directed graph with two types of nodes. Nodes represented as open circles are called locations. Nodes represented as solid bars are called transitions. Figure 1 is an example of a trivial transition net. A location is denoted to be occupied by placing a token, a solid dot, within that location, as in location B of Fig. 1. If the entire net is to be considered as one finite state machine, that machine's state is fully defined by a list of the occupied locations. Tokens move about the net under control of the transitions. A transition is enabled to fire whenever all locations which are on lines of the graph directed into that transition are occupied, and all of the locations which are on lines of the graph directed from that transition are empty. When a transition fires, the tokens are removed from all locations which lead into that transition, and tokens are placed in all locations which are fed from that transition. The firing of a transition is instantaneous. In Fig. 1, transition a is a source of tokens, and will supply a token to location A whenever A is empty. Transition d is a drain for tokens and will remove a token from location D whenever D is occupied.

In the current state of Fig. 1, only transition a is enabled. After a has fired, location A is occupied, and a is no longer enabled, but b is. After b has fired, C is occupied and c is enabled. Since A is now empty, a is also again enabled. After c and a have fired, locations B, D, and A are occupied, and transitions b and d are enabled. An oscillatory activity now ensues with transitions d and b firing to cause locations A, B, and D to empty and location C to be occupied; followed instantly by the firing of transitions c and a. The net result is a steady migration of tokens from a to d in synchronism with the oscillation between B and C.

In representing a software activity, it is convenient to consider the tokens within a net as independent asynchronous processes. The location which each process (token) occupies then represents the state of that process. The transitions through which the processes must pass represent points of interaction between processes which ensure proper synchronism between the processes. In parallel process terminology, transition c of Fig. 1 is a FORK operation from a single process in location C to two (now independent) processes in locations B and D. Transition b of Fig. 1 is a JOIN operation wherein two independent processes at locations A and B are merged into a single process at location C. Although time does not exist explicitly within a transition net, many of the processes which we wish to represent are time-consuming in their data operations, exclusive of the interprocess interactions. This form of time consumption can be embedded within the transition net representation by stretching the change from a location being empty to that location being occupied to include that time. We should view a location as being half occupied, or perhaps, undefined, during this time interval, as no further interactions with other processes appearing explicitly in the net are possible until this time-consuming activity is completed. Upon closer examination, this time-consuming activity which we represent as being within a specific location (process state) may itself be further decomposed as multiple asynchronous processes, or may be a single process which is representable by a conventional flow-chart.

The example of Fig. 1 contains nodes with only 0, 1, or 2 inputs and 0, 1, or 2 outputs. The actual number of inputs and outputs is immaterial and can be arbitrarily increased as long as operation of the transitions follows the conventions previously described. However, since clarity of representation is a principal goal, it is wise to restrict the number of inputs and outputs of any node to as small a number as can completely represent the machine operation, preferably 4 or less.

Figure 2 shows two transitions with inputs from locations that do not fit within the basic operation of transitions as previously described. The open circle input is called an enabling input, and the solid circle is called an inhibiting input. A transition with an enabling input behaves identically to a transition with only normal inputs except that when that transition fires, the token which occupied the location which provided the enabling input is not removed, but remains to enable further firing of the transition. An inhibiting input is the converse of the enabling input, in that a transition which has an inhibiting input may not fire as long as that location which provides the inhibiting input is occupied. Both enabling and inhibiting input connections will be used in the examples which follow.

VI. An Example

Previous sections have described the syntax of transition nets in a simple manner which may make the nets appear to be at best an interesting toy with which to describe concurrency which is already under control. The more substantive examples of this section are intended to show that the nets are not only interesting, but are a useful tool as well. We have used transition nets to date in the design of several segments of intercomputer communication software (Ref. 31), producing nets of varying complexity, from some almost as simple as the example Fig. 1, to some which became unpleasantly cumbersome when all necessary detail was forced into view. As a graphical display of concurrent activity, the transition net provides a skeleton within which relevant questions are easily viewed. Answering these questions remains the designer's problem, as a good representation scheme does not automatically design a system, but induces a careful consideration of all pertinent aspects of the design. It should become evident in the following discussion that some amount of prose commentary is also needed by the designers to relate the featureless tokens of the transition net to the physical resources and processes of the system being designed.

Suppose we wish to design a system to perform real-time Fourier Analysis on a continuous analog waveform. Waveform parameters specify for us the rate at which the input data arrives, and the number of input data elements which must be collected together to allow calculations to begin. Because the input data arrives nonstop, a second collection of data will be being received while the first collection is being analyzed. Likewise, the results of the analysis of the first collection of data may be being output to some recipient device while the second collection is being analyzed, and while a third collection is being input. We have a feasibility constraint in that the analysis of the first collection of data must be complete when or before the input of the second collection is complete, and that output of the first collection of data must be complete when or before the input of the third collection is completed. If one or the other of these feasibility constraints is not satisfied, the processing of data will lag behind the influx of new data to be processed and cause eventual loss of that data, no matter what other actions are taken to avoid such loss.

Figure 3 is a transition net description of the processes which operate within this system. Within this net, tokens represent both processes, as before, and resources (buffer spaces) which initially occupy locations Q_1, Q_2, and Q_3. At the level of detail presented in Fig. 3, the processes each have two states, idle (In) and active (An). These three processes correspond to the three major actions required of our system: input (data), transform (data to results), and publish (results); all three are initially idle. Operation of the net begins when a token is placed in the enabling location E, and ceases gracefully when this token is removed, presumably by some higher level process, human operator, or other. It may be worthy of note that at the level of detail shown in Fig. 3, we no longer need to know that the data transformation is a Fourier analysis; it could be any buffered data transformation.

The three active-state location for the three processes are each time-consuming, and hence can be further decomposed, either by single-process flow chart, or by expansion as transition nets with greater detail. The three idle-state locations are each simple and not time-consuming. In presenting Fig. 3 we have assumed that this transition net both restates the physical realizability constraints stated above and describes the actions of a system which conforms to these constraints. The skeptical or confused reader may find it desirable to sequence through the operation of the net in Fig. 3, using the transition behavior rules given earlier, and verify that it performs as advertised.

We can view Fig. 3 either from its manipulation of the resources (buffers), or from the actions of the processes. The buffers enter active location A_1 where they are filled with raw data. They travel briefly through queuing cell Q_4 into location A_2 where the data they carry is transformed. They travel briefly through queuing cell Q_5 into location A_3 where they are emptied of data, and are then returned to the queuing cells Q_3-Q_2-Q_1 for reuse. The three processes represent an assembly line which works upon a three-bucket conveyor system. Process 1, the input process, takes empty buffers from Q_1, fills them, and places the filled buffer in Q_4. Process 2, the transform

process, takes filled buffers from Q_4, operates on them, and places them in Q_5 after transforming. Process 3 takes filled buffers from Q_5, empties them, and returns the emptied buffer to $Q_3 \rightarrow Q_2 \rightarrow Q_1$. Interface between the processes is along near-minimum lines.

Active-state location A_3 contains within it the interaction with an asynchronous external device—the device upon which the transformed data is to be written—and should be instructive to decompose further. One feasible decomposition appears as Fig. 4. The initial state of this process is as shown. Upon activation, since we are doing output from the computer, the buffer is segmented into primitive units of accessibility (words) and saved in the queuing locations $WQ_1 \ldots WQ_n$. The process itself appears in location BSY which enables the setting of interface location (logic signal) STC. This transfers process initiative to the device which should respond by setting interface signal RSP. Since BQ_3 is nonoccupied while WQ_n is occupied, the word in WQ_n is transferred into the byte storage locations BQ_1, BQ_2, and BQ_3. The byte in BQ_3 is then transferred along with process initiative to the device via interface signal RDY. The device is expected at this point to return process initiative via $\overline{\text{RSP}}$, and will have the process initiative returned to it via $\overline{\text{RDY}}$. The signal STC has remained throughout this activity, so the four-phase cycle at the interface can begin again. The bytes remaining in BQ_2, and BQ_3 are transferred to the device with process initiative via signal RDY each time initiative is returned via signal RSP. If BQ_3 is unoccupied when the process initiative returns, a word is fetched from WQ_n into BQ_1, BQ_2, and BQ_3. If both BQ_3 and WQ_n are unoccupied when process initiative returns, the entire buffer has been written and the process activity ceases.

The active process described in the paragraph above still has a large number of open options for implementation. The interface to the device has been fixed by design, but the interface between hardware and software has not. Those readers who are familiar with the Deep Space Network standard 14-line interface (Ref. 32) will probably recognize from the signal names that Fig. 4 represents the data output mode of the 14-line standard interface adapter (SIA). A full SIA description is possible and will be generated in the future. There are at least three feasible places, which have been used in various SIA implementation, for the hardware/software interface to appear in Fig. 4: (A) at the device interface, (B) on the word-transfer path between WQ_n and the BQs, and (C) on the buffer-transfer path into the WQ_n's.

The main point of this discussion is that the active process description in Fig. 4 is complete from a functional design standpoint and works equally well in the description of hardware machine actions as in describing software actions that are best represented as finite state machines.

VII. Concluding Remarks

We have aired in this article a design concept for real-time hardware/software systems and a representation with which to describe the timing interactions of a real-time system. The design viewpoint is one of interacting finite-state machines, each performing its particular functions when resources and other enabling conditions permit. The representation is the Petri net, or state transition net. The article opens with a general discussion of real-time system design, and design rationale; then proceeds to define the transition nets and use them in an example to describe both hardware and software actions.

Although very simple, the transition net representation described herein is complete enough to aid in the development of real-time software, and it appears also to be adequate for performing resource allocation analyses for systems of asynchronous processes. As described here, the representation is not stand-alone but requires the addition of prose commentary to relate features of the real system to their manifestation in the transition net. The references contain some generalizations of transition nets which attempt to be stand-alone representations. We should, in the future, evaluate how successful these attempts have been. There is also a strong temptation to enrich the syntax of the representation. Such enrichment is at least in part self-defeating, since a syntactically rich representation scheme adds its own complexity to that of any system being represented.

References

1. Dijkstra, E.W., "The Structure of THE Multiprogramming System," *Comm. of the ACM*, May 1968, pp. 341–356.

2. Mills, H.D., "Mathematical Foundations for Structured Programming," IBM Document FSC72-6012, Federal Systems Div., IBM, Gaithersburg, Md., February 1972.

3. Tausworthe, R.C., "Standardized Development of Computer Software," to be published.

4. Petri, C.A., "Kommunikation mit Automaten," Bonn, Germany, 1962. English translation available as Supplement 1 to Rome Air Development Center, TR-65-377, January 1966 (AD630125).

5. Parnas, D.L., "On the Criteria to Be Used in Decomposing Programs Into Modules," *Comm. of the ACM*, December 1972, pp. 1053–1058.

6. Parnas, D.L., "A Technique for the Specification of Software Modules, With Examples," *Comm. ACM*, May 1972.

7. Waite, W.M., "A Language Independent Macro Processor," *Comm. ACM*, July 1967, pp. 433–440.

8. Waite, W.M., "The Mobile Programming System: STAGE2," *Comm. ACM*, July 1970, pp. 415–421.

9. Wulf, W.A., et al., "BLISS, A Language for Systems Programming," *Comm. ACM*, December 1971, pp. 780–790.

10. Richards, M., "BCPL, A Tool for Compiler Writing and Systems Programming," in *AFIPS Conference Proceedings*, Spring Joint Computer Conference 1969.

11. Kossiakoff, A., and Sleight, T.P., "A Programming Language for Real Time Systems," *AFIPS Conference Proceedings*, Fall Joint Computer Conference 1972, pp. 923–942.

12. Resnick, M., and Sable, J., "INSCAN, A Syntax Directed Language Processor," *Proceedings of ACM 23rd National Conference 1968*, pp. 423–432.

13. Birke, D.M., "State Transition Programming Techniques and Their Use in Producing Teleprocessing Device Control Programs," Second Symposium on Problems in the Optimization of Data Communication Systems, October 1971, pp. 21–31.

14. Bjorner, D., "Finite State Automaton — Definition of Data Communication Line Control Procedures," *AFIPS Conference Proceedings*, Fall Joint Computer Conference 1970, pp. 477–491.

15. Holt, Anatol, et al., *Final Report on the Information Systems Theory Project*, RADC-TR-68-305, Rome Air Development Center, New York, 1968.

16. Dijkstra, E.W., "Cooperating Sequential Processes," *Study Notes From Techniche Hogschule Eindhoven*, 1965. Reprinted in *Programming Languages*, F. Genuys (Ed.) Academic Press, New York, 1968.

17. Hansen, P.B., *Operating Systems Principles*, Prentice-Hall, Inc., New York, 1973.

18. Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," *Journal ACM*, July 1967, pp. 439–465.

19. Liu, C.L., and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal ACM*, January 1973, pp. 46–61.

20. McCulloch, W.A., and Pitts, W., "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133; also Minsky, M., *Computation: Finite and Infinite Machines*, Chapter 3, Prentice Hall, Inc., New York, 1967.

21. Baer, J.L., "Models for the Design, Simulation, and Performance of Distributed-Function Architecture, *Computer* (March 1974), pp. 25–30.

22. Rose, C.W., "LOGOS and the Software Engineer," *AFIPS Conference Proceedings*, Fall Joint Computer Conference 1972, pp. 311–323.

23. Glaser, E.L., "Introduction and Overview of the LOGOS Project," *COMPCON 72 Digest*, pp. 175–178, 191–192.

24. Heath, F.G., and Rose, C.W., "The Case for Integrated Hardware/Software Design," *COMPCON 72 Digest*, pp. 179–182.

25. Bradshaw, F.T., "Some Structural Ideas for Computer Systems," *COMPCON 72 Digest*, pp. 183–186.

26. Rose, C.W., Bradshaw, F.T., and Katzke, S.W., "The LOGOS Representation System," *COMPCON 72 Digest*, pp. 187–190.

27. Patil, S., and Dennis, J.B., "The Description and Realization of Digital Systems," *COMPCON 72 Digest*, pp. 223–227.

28. Nutt, G.J., "Evaluation Nets for Computer Systems Performance Analysis," *AFIPS Conference Proceedings of the Fall Joint Computer Conference 1972*, pp. 279–286.

29. Noe, J.D., and Nutt, G.J., "Macro E-Nets for Representation of Parallel Systems," *IEEE Transactions on Computers*, August 1973, pp. 718–727.

30. Misunas, D., "Petri Nets and Speed Independent Design," *Comm. ACM*, August 1973, pp. 474–481.

31. Layland, J.W., "Software for Multicomputer Communications," in *The Deep Space Network Progress Report 42-26*, pp. 145–154.

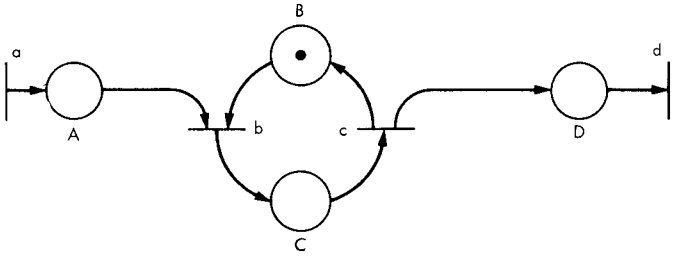

Fig. 1. Trivial transition net example

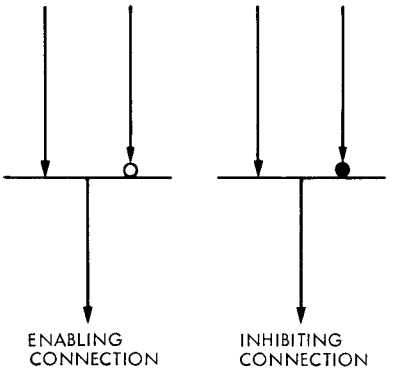

Fig. 2. Special transition—connections

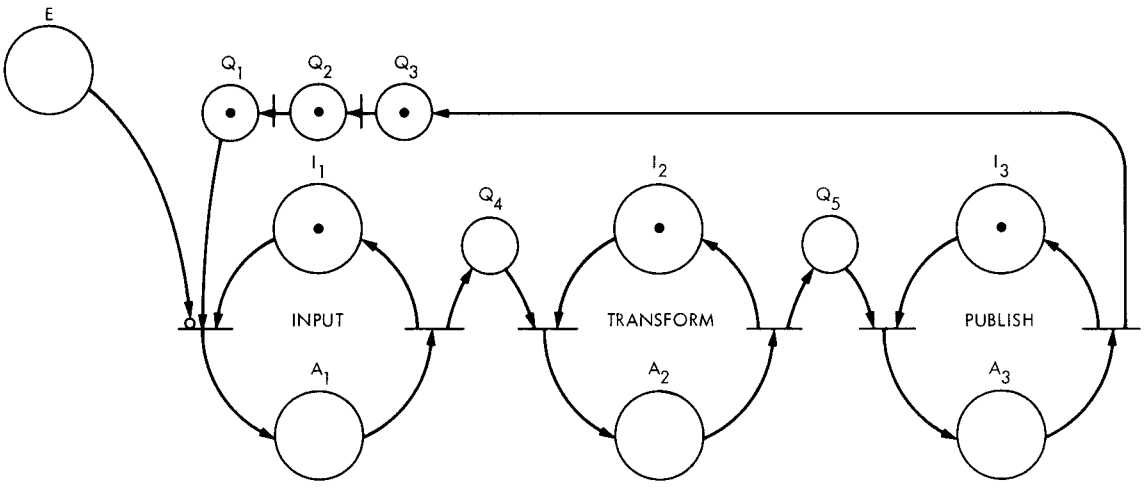

Fig. 3. Example system

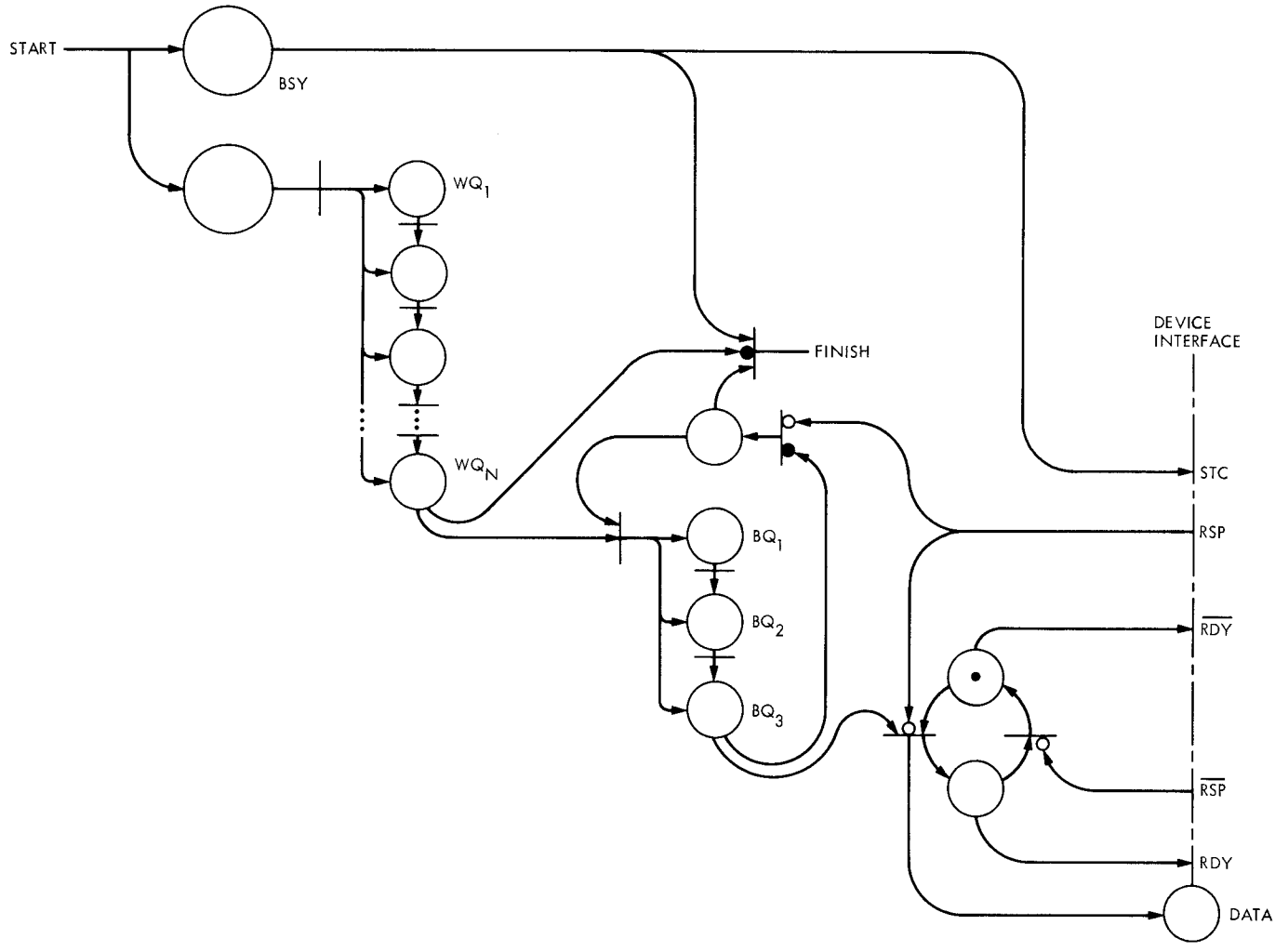

Fig. 4. Expanded net for active output process (A3)