Charles H. Sauer
Edward A. MacNair
Silvio Salza

A Language for Extended Queuing Network Models

Queuing networks are popular as models of performance of computer systems and communication networks. The Research Queueing Package, Version 2 (RESQ2), is a system for constructing and solving extended queuing network models. We refer to the class of RESQ2 networks as "extended" because of characteristics absent from most queuing models. RESQ2 incorporates a high-level language to concisely describe the structure of the model and to specify constraints on the solution. A main feature of the language is the capability to describe models in a hierarchical fashion, allowing an analyst to define parametric submodels which are analogous to macros or procedures in programming languages. RESQ2 thus encourages use of structured models to effectively evaluate complex systems.

Introduction

Models are used to estimate the performance of computing systems when measurement of system performance is impossible (*e.g.*, because the system is not yet operational) or impractical (*e.g.*, because of the human and machine resources required). Traditional, highly imitative simulation models are often as complex as the modeled system and thus suffer feasibility and practicality problems similar to measurement. Queuing networks have become important as performance models of computer systems because performance of these systems is usually principally affected by contention for resources. Relatively simple queuing network models can be used from the early design stages of a system on through system configuration and even system tuning to estimate system performance. In the design stages, relatively inaccurate estimates obtained from very simplistic models may still be quite sufficient for weeding out designs with potentially unacceptable performance. More realistic queuing models can be developed for later stages of system development which are sufficiently accurate and much less expensive than the alternatives of measurement or traditional simulation. The basic problems in using queuing network models are to (1) determine the resources and their characteristics which will most affect performance, (2) formulate a model representing these resources and characteristics, and (3) determine (algebraically, numerically, or by simulation) the values for performance measures (*e.g.*, mean response time) in the model. Recent issues of *Computing Surveys* (September 1978) and *Computer* (April 1980) are dedicated to the solution of queuing networks and the representation of computer and communication systems as queuing networks.

For queuing network models to be used effectively, appropriate software is necessary. A survey of queuing network software and a proposed set of design objectives are given in [1]. Based on our previous work on the Research Queueing Package (RESQ) [1-3], we have designed and implemented a language for construction of queuing network models. This second version of RESQ, RESQ2, incorporates results from programming methodology and hierarchical modeling [4, 5] to encourage users to produce well-structured models. This, in turn, enables users to effectively use queuing network models to evaluate the performance of large and complex systems where models of such systems would be unmanageable with previous software.

This paper focuses on the language aspects of RESQ2. However, we must first say more about queuing network

models to put the RESQ2 language in the proper context. We next give a brief discussion of queuing network models. Following that we describe the extended class of queuing networks allowed by RESQ. We are then able to discuss the design and implementation of the RESQ2 language.

Queuing network models

A typical queuing network model consists of a set of queues (corresponding to resources in the computer system) and a set of jobs (which correspond to processes in the computer system, users at terminals, messages sent from computer to computer, etc., depending on the system). The individual queues are usually described in terms of types of resources, numbers of units of resources, queuing (scheduling) disciplines, and probability distributions for the service times of jobs at the queues. The jobs are described by their individual characteristics (usually by homogeneous groups), by their routing from queue to queue (corresponding to the sequence of resource requirements in the system), and by their arrival processes (and departure procedures).

Much of the research on queuing network models has focused on methods for obtaining solutions, *i.e.*, performance estimates, from the models. Efficient numerical algorithms have been developed for networks with a *product form* solution [3, 6-8]. However, there are many system characteristics which preclude a product form solution, *e.g.*, priority scheduling or simultaneous resource possession. For models with these characteristics and more than a few queues and/or jobs, the only solution methods available are approximate numerical methods [5, 8, 9] and simulation. Specialized simulation techniques have been developed which apply to simulation of queuing networks [8, 10]. RESQ incorporates numerical methods and simulation. (Though RESQ includes simulation components, we do not consider RESQ to be a *simulation* language. Rather, we consider RESQ to be a *modeling* language. We make the distinction primarily because of the higher level of abstraction of RESQ elements, as compared to popular simulation languages, and also because of the nonsimulation solution methods provided in RESQ.)

Since queuing networks can be (and have been) characterized in a variety of ways, we informally define our characterization. A queuing network consists of (1) a set of nodes, (2) a set of queues, (3) a set of routing rules, (4) a set of jobs, and (5) a set of routing chains. The nodes are points in our network in the sense of nodes of a graph. The nodes provide places of residence for jobs (perhaps only instantaneous residence). Visits to a node by a job usually result in actions affecting the job and other network elements. A queue is a set of nodes and a set of rules which define the nature of the residence of jobs at the nodes, depending on the number and status of jobs at each of the nodes. The routing rules define the jobs' paths from one node to another, including rules for determining which path to take when there are several possible destinations for a job leaving a given node. A chain consists of corresponding subsets of the nodes, routing rules, and jobs of the network. The subsets of the nodes (routing rules, jobs) defined by the chains are disjoint. (The routing chains are described as "open" or "closed" depending on whether or not jobs may enter and leave the chains. Jobs entering or leaving a chain also enter or leave the network, respectively.)

Extended queuing networks

In order to facilitate more accurate representation of computer systems, the queuing networks of RESQ have been designed to include and naturally build upon the category of networks with product form solution. Some of the elements are obvious generalizations of product form elements, for example, queues with general (*e.g.*, priority) scheduling disciplines. Other generalizations of product form networks include (1) capabilities for marking jobs with information (such as message length for a job representing a message in a communication network) and (2) routing rules dependent on the current network state (*e.g.*, queue lengths) as well as the usual probabilistic routing rules.

In addition to allowing the above-described characteristics, which violate product form solution conditions, we provide in RESQ new network elements and refer to the resulting category of networks as "extended" queuing networks [11]. We restrict attention to the most important of these elements, the "passive queue." We refer to traditional queues as "active queues" and to the nodes of active queues as "classes." One of the limitations of a network consisting only of active queues is that a job can only hold one resource at a time. This contradicts actual systems in which the element modeled by the job requires several resources simultaneously. For example, a program requires memory as well as a CPU before it can be run, but most traditional queuing models ignore either memory contention or CPU contention. In extended queuing networks a job can hold resources at several passive queues and one active queue simultaneously.

A passive queue consists of a set of "allocate nodes," a set of "release nodes," a set of "create nodes," a set of "destroy nodes," and a pool of identical "tokens" of a resource. A job joins a passive queue when it arrives at an allocate node. Upon arrival the job requests one or more tokens. If sufficient tokens are available, the job receives

them and moves on to another queue of the network without delay. However, the job belongs to the queue from which it received the tokens as long as it holds the tokens. If insufficient tokens are available, the job waits until enough become available and then immediately moves on through the network after receiving them. When several jobs wait for tokens of a passive queue, they are allocated tokens according to a specified scheduling discipline. A job gives up tokens, and thus leaves the corresponding passive queue, when it is routed through a release node of the queue. The job passes through the release node instantaneously. Create nodes have no effect on the job, but do have the effect of adding new tokens to the pool. Destroy nodes are similar to release nodes but do not return the tokens to the pool. See Fig. 1.

The terms "active queue" and "passive queue" are intended to indicate the nature of the queue's effect on a job's use of a server or token, respectively, and of the relative dominance of the modeled resources. With an active queue the length of time a job holds a server is entirely determined by the characteristics of that queue and the jobs at that queue. With a passive queue the length of time a job holds a token is determined entirely by events at other queues.

Figure 2 shows a simplistic representation of a widely used model of interactive computer systems [12, 13]. The resources represented by active queues are the terminals, CPU, and I/O device(s). A passive queue is used to represent memory contention. After a think time at the terminal, a user keys in a command. A job representing the process executing the command requests memory. After receiving memory, the job alternates CPU and I/O activities until the command is finished. The job then releases its memory and returns to the terminal's queue for another thinking and keying time.

RESQ2

One can draw analogies between queuing network elements in the RESQ2 language and programming language elements, especially concurrent languages such as Concurrent Pascal [14]. We use identifiers to symbolically reference nodes, queues, chains, and other elements we have yet to define. We may think of nodes as elementary data types. Some kinds of nodes are associated with queues; thus we may think of queues as aggregate data types. There are kinds of nodes which are not directly associated with queues. We can think of the jobs as processes in a concurrent language, and we can think of the routing rules as part of the explicit control structure of our language. (The mechanisms associated with the various types of nodes form an implicit control structure analogous to semaphores and monitors in concurrent languages.)

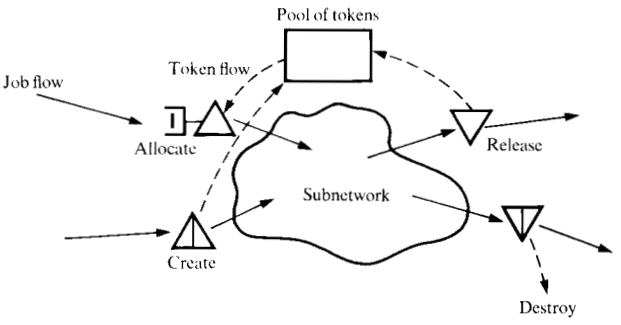

Figure 1 A passive queue.

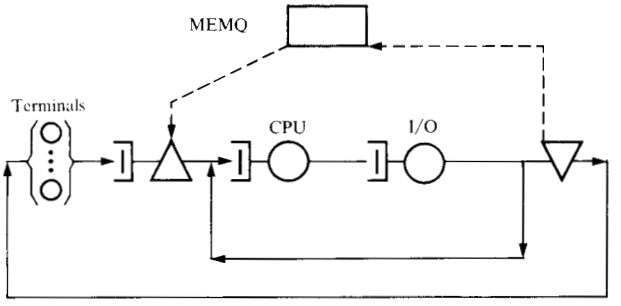

Figure 2 Computer system model.

In designing the first version of RESQ our focus was on the solution methods and the definitions of queuing network elements. With the second version our focus has been on the design of the language interface and its processor. We have designed a modeling language which attempts to incorporate the wisdom gained in the design of programming languages and structured programming while providing a gentle transition for users of the first version of RESQ.

The principal limitation of the first version is the inability to characterize relatively homogeneous network components (*e.g.*, queues, routing chains, subnetworks) once and then characterize their differences. Rather, one must define the details of each such component individually. The situation is somewhat analogous to programming in *machine* (not assembly) language, though not so extremely tedious because of the level of abstraction of the extended queuing network elements. The user is essentially forced to begin each model definition from scratch, being largely unable to build upon his or her own previous models or on others' efforts. (It is possible to retain the concepts of previous models, but usually not the machine definition of those concepts.) The situation was tolerable

```
QUEUE TYPE: pfcfs /*passive fcfs queue*/
    PARAMETERS-
        NUMERIC: tkns
        NODE: alloc(*) releas(*)
    TYPE: passive
    TOKENS: tkns
    DSPL: fcfs
    ALLOCATE  NODE LIST: alloc
                  AMOUNT(S): 1
    RELEASE NODE LIST: releas
END OF QUEUE TYPE PFCFS
```

Figure 3 Queue type definition.

```
QUEUE: memq
    TYPE: pfcfs
    TKNS: nmem
    ALLOC : anode(*)
    RELEAS: rnode(*)

QUEUE: memq
    TYPE: pfcfs: nmem ; anode(*) ; rnode(*)
```

Figure 4 Equivalent queue type invocations.

when we were developing a research prototype and our typical networks had only a few components; it is not tolerable for our own use in research on queuing network modeling technology, nor is it tolerable for application users who may have hundreds of queues and nodes in their models.

Thus our principal problem in designing the RESQ2 language has been to provide new structures to organize, and simplify the specification of, the data and control structures already present. The resulting features principally consist of what we call "the template facility." The templates allow the user to overcome the above-cited limitation of the first version of RESQ, the inability to simply characterize homogeneous network components. In addition, the templates allow for the benefits typically associated with structured and top-down programming, *e.g.*, a model can be successively refined by redefining template definitions, and/or by adding additional nested templates, a model can be studied at different levels of detail as appropriate to the reader, etc.

We define two kinds of templates: "queue types" and "submodels." Queue types correspond to data types in the sense of Pascal. Submodels correspond to macros and, to a limited extent, procedures. Queue types facilitate structuring and specifying the principal "data" of a model, its queues. Submodels facilitate structuring and specifying both the "data" and the "control" of a model. In addition to these templates, we provide for arrays of some kinds of elements, operations on these arrays, and some additional specialized data types. We first discuss templates.

• *Queue types*

In order to define a queue, many characteristics may need to be specified. These include (1) whether the queue is active or passive, (2) the number of servers or tokens, (3) the scheduling discipline, (4) the identities and types of nodes associated with the queue, (5) probability distributions to be associated with the nodes (*e.g.*, for service times or number of tokens requested), (6) whether the probability distribution values are to be scaled by the variables associated with the jobs, and (7) the priorities associated with each node if the scheduling depends on priorities. For active queues it is also necessary to specify detailed characteristics of each server, *e.g.*, service rates as a function of queue length. In the first version of RESQ, a number of specialized active queue types are provided for common cases so that many characteristics are set by default. However, no corresponding specialized passive queue dialogues are provided, and the specialized active queue types do not allow specification of characteristics which may be important for a particular model and may require specification of other characteristics which are the same for many queues. The RESQ2 queue type allows definition of a parameterized template and subsequent definition of queues by invocation of the template with parameters. Figure 3 shows a queue type for a passive queue which has parameters for (1) the number of tokens, (2) the associated allocate nodes, and (3) the associated release nodes. Upper case is used for reserved words and lower case for other information. (This is for clarity; the user may freely choose between upper and lower case.) All other characteristics are specified with the queue type definition, either explicitly or by default. Figure 4 shows two equivalent invocations of the queue type of Fig. 3. (Upper case is used in the first part of Fig. 4 for the names of formal parameters as well as for key words.)

• *Submodels*

The submodel allows parameterized definition of an entire subnetwork. For example, in a model of a computer communication network, there may be several similar computer systems. One can define a submodel generally representing the computer systems, then use copies of the submodel to represent individual systems. These invocations and their effects may correspond closely to the invocation and expansion of macros. We assume this is the case for the time being and defer discussion of an alternate kind of invocation. Submodel definitions and invocations may be nested within submodels. One may specify arrays of invocations of a submodel.

A submodel (*i.e.*, a subnetwork) can be characterized by the same four sets which we listed before in defining a network. However, necessary differences arise as we define the boundaries between the submodel and the model which contains the submodel. We would like the elements of a submodel to be hidden from the model containing the submodel, so that the person using the submodel needs little awareness of its internal characteristics.

Most of the nodes of the submodel can be hidden from the outside. However, those on the boundary, providing entrances to and exits from the submodel, must be visible. We provide synonyms for the entrances and exits, "input" and "output," respectively, so that one need not know the submodel's names for the entrances and exits. Usually a submodel will have exactly one entrance and one exit per "external" routing chain. An external routing chain is one with entrances and exits from the submodel and may be closed or open, depending on the invocation. An "internal" routing chain is hidden from the outside and specified closed or open in the submodel definition. We would like to encourage the one entrance/exit characteristic, and must enforce it in certain cases, so we only allow one "input" and "output" per chain. However, in some cases we must allow multiple entrances and/or exits for a chain, so we allow specification of a submodel's internal node names in routing rules of an invoking (sub)model. This is an error-prone provision, analogous to a "goto" leading outside of a procedure, and is designated by a special syntax. (It is also necessary to use a submodel's internal node names in requesting performance estimates for those nodes, but this is not similarly error prone.) All of the queues of a submodel can be hidden from the outside, except that their component nodes are accessible as just described (and except that their names must be used in requesting performance estimates). The routing rules of a submodel are completely hidden from the outside, even though these rules contribute to the routing chains which cross the submodel boundaries (the rules of external chains). Jobs in external chains are defined in the outermost definition of those chains. Jobs in internal chains are completely hidden from the outside. (There is an exception to this statement with regard to initialization of simulations; we shall not discuss the exception.)

In addition to describing the nodes, queues, routing rules, and jobs of a submodel, one must define its parameters. The types of the parameters are (1) numeric types as in programming languages, (2) specialized data types which we have yet to discuss, and (3) external chains. Since chains potentially cross the boundaries of many submodels, we need to have their identities available in all submodels which refer to them.

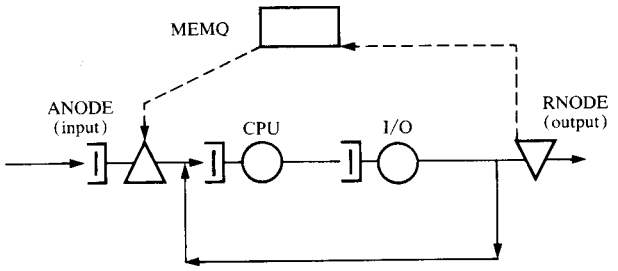

Figure 5 Submodel from Fig. 2.

```
SUBMODEL: ecsm /*extended computer system model*/
    PARAMETERS-
        NUMERIC: nmem cyclep
        CHAIN : chn
    QUEUE: memq
        TYPE: pfcfs
        TKNS: nmem
        ALLOC : anode
        RELEAS: rnode
    QUEUE: cpuq
        TYPE: ps /*Active queue with processor sharing scheduling*/
        CLASS: cpu
        WORK: .05 /*mean service time in seconds*/
    QUEUE: ioq
        TYPE: fcfs /*active*/ : io; .04
    CHAIN: chn
        TYPE: external
        INPUT: anode
        OUTPUT: rnode
        : anode→cpu→io
        : io→cpu rnode; cyclep 1-cyclep
        /* cyclep is probability of job cycling back to cpu*/
END OF SUBMODEL ECSM
```

Figure 6 Submodel definition.

Figure 5 illustrates a submodel taken from the model of Fig. 2. Figure 6 shows a RESQ2 definition of this submodel. Syntactically, there are three major sections present in Fig. 6, for definition of parameters, queues, and chains, in that order. It is assumed that queue types *pfcfs*, *ps*, and *fcfs* have been defined outside of the submodel. In general submodels, there would be additional sections for definition of identifiers, for definition of queue types, for definition of submodels nested within the submodel, for invocations of submodels, and for definition of other types of nodes.

Figure 7 shows a complete definition of the network of Fig. 2, using the two templates we have defined and presuming the previous definition of three other queue types. (Upper case letters are used in the definition of values of numeric identifiers as well as for reserved words and for-

```
NAME: csmtm /*computer system model with terminals and memory*/
    LIBRARY QUEUE TYPES: is pfcfs ps fcfs
    LIBRARY SUBMODELS: ecsm
    METHOD: simulation
    PARAMETERS-
        NUMERIC: nmem cyclep
    IDENTIFIERS-
        NUMERIC: cpop
            CPOP: 20
    QUEUE: termq
        TYPE: is /*"infinite" server*/
        CLASS: terms
        WORK: 15
    INVOCATION: comsys
            TYPE: ecsm: nmem; cyclep; chn
    CHAIN: chn
        TYPE: closed
        : terms→comsys.input
        : comsys.output→terms
        CHAIN POP: cpop
END OF MODEL CSMTM
```

Figure 7 Definition of computer system model.

```
SUBMODEL: ecsm /*extended computer system model*/
    PARAMETERS-
        NUMERIC: nchn nmem cyclep(nchn)
        CHAIN : chn(nchn)
    IDENTIFIERS-
        NODE ARRAYS: anode(nchn) cpu(nchn) io(nchn) rnode(nchn)
    QUEUE: memq
        TYPE: pfcfs
        TKNS: nmem
        ALLOC : anode
        RELEAS: rnode
    QUEUE: cpuq
        TYPE: ps /*Active queue with processor sharing scheduling*/
        CLASS LIST: cpu
        WORK DEMAND: .05
    QUEUE: ioq
        TYPE: fcfs /*active*/ : io; .04
    CHAIN: chn(*)
        TYPE: external
        INPUT: anode(*)
        OUTPUT: rnode(*)
        : anode(*)→cpu(*)→io(*)
        : io(*)→cpu(*) rnode(*); cyclep(*) 1-cyclep(*)
END OF SUBMODEL ECSM
```

Figure 8 Submodel definition with arrays.

mal parameters.) The model has two numeric parameters, *nmem* and *cyclep*, which are to be specified when the model is solved; thus they may be varied without retranslation of the model. The invocation of the submodel uses the short syntax corresponding to the bottom part of Fig. 4. Notice that references to the submodel in the routing are similar to references to nodes. In addition to the sections shown there could be a rather lengthy but straightforward section specifying characteristics of the simulation solution, *e.g.*, limits on the run length, nonstandard performance measures to be estimated, parameters of the regenerative method [8, 10], etc. We ignore that section and assume that the default simulation characteristics are appropriate.

• *Substitutions*

There is a second (semantic) form of invocation of submodels, "substitution," which we referred to earlier. With the first form of invocation, though the model is defined hierarchically, it is treated as a horizontal entity as far as the solution portions of RESQ are concerned. However, a solution which recognizes the model hierarchy may be much more computationally efficient than one which does not. With a large model this difference in efficiency may determine whether it is practical to solve the model or not. Hierarchical solutions may be performed exactly for product form queuing networks [15] and limiting cases of other networks [16]. (Hierarchical solutions are the basis for many approximate solutions [4, 5, 8, 9, 12, 13].) One of the limiting cases is when subnetworks interact with each other relatively infrequently compared to the rate of activity within the subnetworks. In our example model, as *cyclep* becomes large, we approach this limiting case. In simulating that model there are many events at the CPU and I/O queues for each event at the terminals. The disparity in event rates may result in great computational expense in a horizontal solution, but a hierarchical solution is relatively inexpensive.

With the substitution form of invocation, the hierarchical structure of the model should be chosen with the hierarchical solution in mind. Yet one should be free of such considerations in formulating and defining a model, as far as possible. A simple approach to this paradox is to make liberal use of submodels in defining a model. Not only may this greatly improve the clarity of the model, it allows one to postpone decisions about hierarchical solution until one is ready to have the model solved. One can then use the substitution form of invocation as appropriate and the ordinary invocation elsewhere.

Syntactically, a substitution is similar to an invocation, but additional specifications may be included to characterize the (separate) solution of the submodel and the interface between the values obtained by that solution and the rest of the model. The solution and interface of substitutions is the case alluded to before where it is impractical to allow more than one entrance/exit per chain.

• *Special data types*

We currently provide two special data types, distributions and strings, which are analogous to data types in programming languages. In defining different copies of a model one may wish to vary the *form* of probability distri-

butions as well as their defining parameters; *e. g.*, one may wish to compare hyperexponential and gamma distributions as well as to compare different means and variances for a hyperexponential distribution. One would like to symbolically refer to such data elements as well. Identifiers (including parameters) of type "distribution" have values which are complete definitions of any probability distribution known to RESQ. Similarly, one may wish to symbolically refer to scheduling disciplines. Identifiers of type "string" have values which are the names of any scheduling disciplines known to RESQ. (We use the term "string" because we may generalize this type in the future.)

Arrays

We provide arrays in RESQ2 in the sense of programming languages. However, there are some subtle issues associated with arrays in this context with respect to which elements are appropriate for such aggregate definition and with respect to how the arrays may be manipulated. Before trying to discuss these issues, let us consider the motivation for arrays in RESQ.

The principal motivation, ignoring submodels for the moment, is to allow multiple parallel routing chains. The term "routing chain" is misleading to the extent that it suggests that the only distinction between chains is in the routing. Actually, different chains are often used to distinguish between groups of jobs which have essentially the same routing but differ in other ways. For example, one may wish to distinguish between interactive users doing "trivial" and "nontrivial" work, *e.g.*, those doing text editing and those running a compiler. The users doing trivial work typically have shorter think times as well as smaller computational demands than those doing nontrivial work. These differences are specified in the definitions of the corresponding nodes of the parallel routing chains. If the routing chains are completely parallel, then each queue has one node from each routing chain. (In a complex but parallel routing structure, a queue may have more than one node from each routing chain, but a queue has the same number of nodes from each routing chain.) Figure 8 shows the generalization of our example submodel to allow *nchn* parallel chains.

We began our RESQ2 design with a very general array provision. While implementing the translator, we recognized that the generality of our provisions led to quite difficult problems in providing diagnostics. We then restricted our provisions but believe that we have not significantly affected the expressiveness of the language. RESQ2 allows arrays of numbers, distributions, nodes, chains, invocations, and substitutions. Only arrays of numbers may have more than one dimension. The restrictions on our preliminary design include eliminating arrays of queues, reducing the number of dimensions on nonnumeric arrays, and eliminating iterative expressions except in definition of substitutions.

```
. . .
QUEUE ARRAYS: ioq(nchn)
NODE ARRAYS: ion(nchn;nio) . . .
. . .
QUEUE: do i=1 to nio: ioq(i)
      TYPE: fcfs
      CLASS: ion(*;i)
      WORK: .04
CHAIN: do i=1 to nchn: chn(i)
      TYPE: external
      INPUT: anode(i)
      OUTPUT: rnode(i)
      : anode(i)→cpu(i)
      : do j=1 to nio: cpu(i)→ion(i;j); prob(i;j)
      : do j=1 to nio: ion(i;j)→cpu(i); cyclep(i)
      : do j=1 to nio: ion(i;j)→rnode(i); 1-cyclep(i)
. . .
```

Figure 9 Queue array with iterative definition.

```
. . .
SUBMODEL: io /*i/o subsystem*/
      NUMERIC PARAMETERS: nchn
      CHAIN PARAMETERS: chn(nchn)
      NODE ARRAYS: ion(nchn)
      QUEUE: ioq
      TYPE: fcfs: ion; .04
      CHAIN: chn(*)
            TYPE: external
      INPUT: n(*)
      OUTPUT: n(*)
      /*no routing rules defined within the submodel*/
END OF SUBMODEL IO
INVOCATION: iosys(nio)
            TYPE: io: nchn; chn
CHAIN: chn(*)
            TYPE: external
INPUT: anode(*)
OUTPUT: rnode(*)
: anode(*)→cpu(*)
: cpu(*)→iosys(*).input; prob(*;*)
: iosys(*).output→cpu(*); cyclep(*)
: iosys(*).output→rnode(*); 1-cyclep(*)
. . .
```

Figure 10 Equivalent submodel with invocation.

The basic claims in our redefinition of RESQ2 array facilities were that the RESQ2 user would use arrays to characterize relatively homogeneous groups of structured data, and for that reason any explicit "DO loop" structures could be more appropriately expressed implicitly. As an example, suppose we wish to add a parametric number of I/O queues to our example submodel. Figure 9 shows how this might be done using arrays of queues and iterative expressions and Fig. 10 shows how with an alter-

nate approach using an array of invocations of a simple submodel. We claim that the approach of Fig. 10 is much clearer, and thus less error-prone, than the approach of Fig. 9. Since *nchn* and *nio* might well be parameters for the entire model, to be supplied when the model is solved, we also claim that it is quite difficult for a translator to give diagnostics for Fig. 9, but it is not so difficult for a translator to give diagnostics for Fig. 10.

• *Implementation*

The components of the implementation of the first version of RESQ may be partitioned into the user interfaces and the solutions. The solutions are implemented entirely in PL/I. There are three interface modes, with each interface mode implemented in both PL/I and APL. (We provide the interfaces in APL as well as PL/I to satisfy users who are unwilling to deal with environments other than APL. The APL interfaces give the user the appearance that RESQ is implemented in APL only [17].) The original interface mode in the first version of RESQ is a set of four interactive dialogues for defining, solving, listing, and changing a model. Such dialogues are an effective educational tool. However, the constraints of these dialogues limit the effectiveness of experienced users. A simple alternative to the interactive mode is to allow users direct access to procedures which define network characteristics. Unfortunately, the procedural mode in the first version of RESQ requires considerable sophistication and attention to detail on the part of the user. The third interface mode in the initial version of RESQ is the "dialogue file." The intent of this mode is to give the user a language for defining queuing networks and a processor for that language, alleviating limitations of the other two modes. A model definition (a "program") in this language is a transcript of a dialogue which *could have* been used to define the model. (The RESQ2 language is based upon this dialogue file mode, with upper case in our examples corresponding to prompts and lower case corresponding to responses.) Simple modifications to the interactive prompter for the model definition dialogue allow its use as a processor for the dialogue files. However, though this prompter has excellent error handling characteristics in interactive mode, these characteristics are useless in the dialogue file mode (*i.e.*, the prompter may reject an erroneous reply in interactive mode and repeat the corresponding prompt; in dialogue file mode there is no reply for the repeated prompt).

It was our intent that the implementation of RESQ2 use the solution portions of the previous version with relatively minor modifications. Thus the processing of a model in the above language should result in a model definition similar to that produced by the first version of RESQ. Further, the noninteractive mode, *i.e.*, the dialogue file, is intended to be the principal interface mode. An interactive prompter is provided for education of new users; we believe we have eliminated the need for a procedural interface mode, as discussed below. The language is designed so that simple parsing techniques, *e.g.*, recursive descent, may be used. (Recursive descent parsing here has the advantage that much of the translator can also be used for the interactive prompter.) However, two features of the language prevent the translator from directly producing the desired model definition. First, the provision for model parameters prevents even knowledge of the size of the model (number of nodes, queues, chains, etc.) before those parameters are defined. Second, the provision for hierarchical solutions (substitutions) means that solution of a model may entail separate solution of many submodels.

In addition to the translator and the existing solution portions, we also implemented what we call the "expansion processor," which takes the model definition produced by the translator, obtains the model parameters, expands the submodel invocations and (repeatedly) calls upon the solution portions. When the entire model has been solved, the expansion processor provides the requested performance measures.

It would be convenient for the user who wishes to construct pre- and post-processors for a model to be able to embed an entire model definition in a language such as PL/I. This might be a much more convenient approach than a procedural interface. However, this suggests potentially difficult problems for the translator. It is our belief that relatively little communication is required between pre- and post-processors and a model definition, that communication of parameters (numeric, distribution, and string) to a model and of performance measures from a model is sufficient. Thus we have provided communication of such information between PL/I and APL programs and the expansion processor. This consists of a few simple procedures and is much more convenient for the user than a full procedural interface.

Summary

We have described the main features of the RESQ2 language for queuing networks. It is too early to characterize user experiences, but preliminary reactions are quite encouraging. We fully expect that RESQ2 will significantly simplify performance modeling of computing systems and that it will make feasible models which previously could only be conceived.

References

1. C. H. Sauer and E. A. MacNair, "Queueing Network Software for Systems Modeling," *Research Report RC-7143*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1978. [*Software Pract. Exper.* **9**, 5 (1979).]

2. C. H. Sauer, M. Reiser, and E. A. MacNair, "RESQ—A Package for Solution of Generalized Queueing Networks," *Research Report RC-6462*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1976. *(Proc. 1977 National Computer Conf.)*
3. M. Reiser and C. H. Sauer, "Queueing Network Models: Methods of Solution and Their Program Implementation," *Current Trends in Programming Methodology, Volume III: Software Modeling and Its Impact on Performance*, K. M. Chandy and R. T. Yeh, Eds., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978, pp. 115-167.
4. J. C. Browne, K. M. Chandy, R. M. Brown, T. W. Keller, D. F. Towsley, and C. W. Dissley, "Hierarchical Techniques for Development of Realistic Models of Complex Computer Systems," *Proc. IEEE* **63,** 966-975 (1975).
5. K. M. Chandy and C. H. Sauer, "Approximate Methods for Analysis of Queueing Network Models of Computer Systems," *Computing Surv.* **10,** 263-280 (1978).
6. K. M. Chandy and C. H. Sauer, "Computational Algorithms for Product Form Queuing Networks," *Commun. ACM* **23,** 10 (1980).
7. M. Reiser and S. S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," *Research Report RC-7023*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1978 [*J. ACM* **27,** 2 (1980)].
8. C. H. Sauer and K. M. Chandy, *Computer System Performance Modeling: A Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, in press.
9. C. H. Sauer and K. M. Chandy, "Approximate Solution of Queueing Models of Computer Systems," *Research Report RC-7785*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1979. [*Computer* **13,** 4 (1980).]
10. D. L. Iglehart, "The Regenerative Method for Simulation Analysis," *Current Trends in Programming Methodology, Volume III: Software Modeling and Its Impact on Performance*, K. M. Chandy and R. T. Yeh, Eds., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
11. C. H. Sauer and E. A. MacNair, "Computer/Communication System Modeling with Extended Queueing Networks," *Research Report RC-6654*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1977.
12. R. M. Brown, J. C. Browne, and K. M. Chandy, "Memory Management and Response Time," *Commun. ACM* **20,** 153-165 (1977).
13. Y. Bard, "The VM/370 Performance Predictor," *Computing Surv.* **10,** 333-342 (1978).
14. P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
15. K. M. Chandy, U. Herzog, and L. Woo, "Approximate Analysis of General Queuing Networks," *IBM J. Res. Develop.* **19,** 43-49 (1975).
16. P. J. Courtois, *Decomposability: Queueing and Computer System Applications*, Academic Press, Inc., New York, 1977.
17. E. A. MacNair and C. H. Sauer, "Multiple Language (APL and PL/I) Interfaces for Queueing Network Software," *Research Report RC-7535*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1979.

Received December 26, 1979; revised May 16, 1980

C. H. Sauer and E. A. MacNair are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. S. Salza, who had been on sabbatical leave at the Research Center when this work was done, is at the Centro di Studio dei Sistemi di Controllo e Calcolo Automatici del Consiglio Nazionale delle Ricerche of Rome, Italy.