Report Number: WUCS-82-8

1982-02-01

Functional Specification of Distributed Systems

Gruia-Catalin Roman

A formal Distributed Systems Design Language (DSDL) is described. In DSDL, systems are described as nets of communicating processes. A net is defined by its processes, by the logical communications links between processes, and by the communications protocols. Each process in the net has its own local data, procedures that specify primitive operations over the data, and possesses the ability to exchange messages with other processes in the net. Some of these processes are used to model the system environment. The links identify the logical connections between processes. The way in which an individual link behaves in stipulated by... Read complete abstract on page 2.

Functional Specification of Distributed Systems

Gruia-Catalin Roman

Complete Abstract:

A formal Distributed Systems Design Language (DSDL) is described. In DSDL, systems are described as nets of communicating processes. A net is defined by its processes, by the logical communications links between processes, and by the communications protocols. Each process in the net has its own local data, procedures that specify primitive operations over the data, and possesses the ability to exchange messages with other processes in the net. Some of these processes are used to model the system environment. The links identify the logical connections between processes. The way in which an individual link behaves in stipulated by the communication protocol associated with the link. DSDL is introduced by means of a highly simplified annotated example representative of the nature of the language.

FUNCTIONAL SPECIFICATION OF DISTRIBUTED SYSTEMS

Gruia-Catalin Roman

Robert K. Israel

WUCS-82-8

February 1982

Department of Computer Science
Washington University
St. Louis, Missouri 63130

As appeared in Proceedings of the 1983 Conference on Parallel Processing, August 1983, pp. 503-505.

ABSTRACT

A formal Distributed Systems Design Language (DSDL) is described. In DSDL, systems are described as nets of communicating processes. A net is defined by its processes, by the logical communications links between processes, and by the communications protocols. Each process in the net has its own local data, procedures that specify primitive operations over the data, and possesses the ability to exchange messages with other processes in the net. Some of these processes are used to model the system environment. The links identify the logical connections between processes. The way in which an individual link behaves is stipulated by the communication protocol associated with the link. DSDL is introduced by means of a highly simplified annotated example representative of the nature of the language.

Acknowledgements: This work was partially supported by Rome Air Development Center and by Defense Mapping Agency under contract F30602-80-C-0284. The contribution of J. T. Love to the initial work in DSDL is also acknowledged.

Keywords: Distributed systems, specification languages.

INTRODUCTION

The potential for a major qualitative improvement in the effectiveness of systems development rests to a large extent on the availability of appropriate specification languages. While establishing the basis for precise communication, formal specifications also open the doors to extensive systematic (mental or automated) system design analysis techniques whose scope would ultimately include logical verification, performance checking, automatic generation of predictive models, and more. Such advances in system design technology are presumed to pave the way for powerful development tools which are very much needed at a time when system development productivity registers relatively minor increases and personnel costs are on the rise.

Specification languages have received considerable attention both in industrial and academic circles. Various proposals range in flavor from tables, standardized document formats, and graphic representations [ROSS77], at one extreme, to formal languages having well-defined syntax and semantics [ROBI77] at the other. The work on program specifications has largely dominated the field, both with respect to the attention received and level of formality. (The reader is referred to [LISK79] for a good survey of available formal program specification techniques.) This is in part due to the strong influence exercised by related research in the programming language area (CLU [LISK77], Alphard [WULF76], etc.).

Despite the considerable effort that has been expended in recent years in the study of parallel computation and distributed systems design, the specification of distributed systems continues to present designers with many unresolved problems [LISK79]. Some programming and program specification languages (Path-Pascal [CAMP79], DREAM [RIDD78], etc.), while able to express concurrency, are limited in their capacity to deal with distribution and restrict the designer's freedom to define arbitrary communication protocols. Furthermore, less formal approaches (e.g., RSL [BELL77]), while valuable from a pragmatical viewpoint, are only a temporary solution that sets the stage for future assimilation of theoretical results into the production environment. This paper reports on one effort to develop a formal Distributed Systems Design Language (DSDL) and the conclusions reached after experimentation with the language on several case studies. The hope is for this work to provide valuable insights that could affect the next generation of distributed systems specification languages.

In DSDL, systems are described as nets of communicating processes. Each process in the net has its own local data over which it has sole control, procedures that specify primitive and indivisible operations over the data, and possesses the ability to exchange messages with other processes in the net. The behavior of the process specifies the order in which its procedures are invoked. Sequences of procedure invocations, also called event sequences, are allowed to execute concurrently within the process.

A net is defined by its processes, by the logical communication links, and by the communication protocols associated with the individual links. Among the processes of a net, some are used to model its

environment; they are called external processes. The links identify the logical connections between processes. Several processes may be associated with the same link and the same process may use several links. The way in which an individual link behaves is stipulated by the communication protocol associated with the respective link.

Several considerations have influenced heavily the nature of the DSDL: the emphasis on formality, the desire to promote the principle of separation of concerns, the need to support hierarchical specifications, and the aim toward generality. Formality is achieved through the use of set theoretical models for data representation, by employing predicate calculus in defining the procedures (using input/output assertions), etc. The principle of separation of concerns is reflected by the manner in which the definitions of the net and of the process are structured; they are meant to enhance the designer's ability to describe the system in terms of clean abstractions. Hierarchical descriptions of the system are enabled by the fact that processes may be refined into nets. Finally, the generality of the language is enhanced by its capacity to describe a variety of communication structures and protocols.

The language, as described here, is concerned only with the functional specification of distributed systems. However, the addition of performance specifications to DSDL is currently under investigation and is anticipated to share the direction adopted in [BOOT80, SMIT79, SANG79]. The ability to relate in a direct and simple fashion functional and performance aspects of distributed systems is expected to contribute to the enhancement of the designer's ability to choose objectively between alternate solutions based on performance analysis of the various candidates.

The next section introduces DSDL by means of a highly simplified annotated example representative of the nature of the language. While many of the language feature may still remain rather obscure after scanning the example, the subsequent section refers back to the example as an illustration for the definition of the language syntax and semantics, thus removing any ambiguities one might read into the example. The language definition is followed by a review of several open issues and preliminary research results.

LANGUAGE ILLUSTRATION

The purpose of this section is to introduce the reader to the various DSDL features by means of a simple annotated example. It is intended to provide a concrete reference point for the formal definitions of process and net described in the next section. A highly simplified version of a distributed banking system forms the basis for this illustration. Upper case words indicate language defined entities while lower case words denote terms selected by the designer.

```
NET egbank. /* EGBANK is a small bank having two branch offices in the city.
            /* Tellers from each branch office are authorized to create new
            /* accounts, to check the balance of one or more accounts, to
            /* make deposits and withdrawals from one account at a time, and
            /* to transfer money from one account to another. These activities
            /* are supported by a computer system consisting of three
            /* components that communicate with each other via messages.
            /* Each branch office interacts with a local process which, in
            /* turn, has access to a database located elsewhere.

  DEFINE office: PROCESS.     /* Definition of a class of processes.

      PARAMETERS.
        CONST id:      INTEGER;          /* Branch office identifier.
        CONST db:      PROCESS;          /* Process controlling the databank.
        CONST tty:     EXTERNAL PROCESS; /* Local data entry sources.
        CONST ttylink: LINK;             /* Connection to data entry sources.

      DATA.
        VAR   /* Request counter and message identifier.
              n:   INTEGER;  n>0;
        CONST /* Terminal identifiers.
              Terminals={z | 0<z<21 AND INTEGER(z)};
        CONST /* Definition of acceptable input commands, i.e., requests.
              Req      ={('create',c),('update',a,v),('transfer',a1,a2,v),
                       ('read',a[1],..,a[m]) | INTEGER(m) AND m>0 };

      INITIALIZATION.
        n = 1;

      PROCEDURE w:=format(z).   /* Message is formed for transmital to db.
        IN:     z MEMBER.OF (Terminals X Req) AND z=(trm,rq);
        OUT:    n'=n+1 AND w'=(id,n,trm,rq);
        EXPT:   w'=NIL;

      PROCEDURE w:=reply(z).    /* Reply from db is prepared for the teller.
        IN:     z=(id,no,trm,ans) AND no<n AND trm MEMBER.OF Terminals;
        OUT:    w'=(trm,ans);
        EXPT:   w'=NIL;
```

```
    BEHAVIOR.
      PARBEGIN
          BEGIN /* Terminal requests are accepted, formated, and sent to the
                /* database for processing.  Invalid queries are rejected.
              LOOP  {GET(z) FROM tty ON ttylink;   w:=format(z);
                     IF NOT.NIL(w) THEN SEND(w) TO db  ON switch;
                                   ELSE SEND(w) TO tty ON ttylink;}
          END.

          BEGIN /* Replies from the db are sent to the terminals.
              LOOP  {GET(z) FROM db ON switch;  w:=reply(z);
                     IF NOT.NIL(w) THEN SEND(w) TO tty ON ttylink;}
          END.
      PAREND.
END-DEFINE office.


EXTERNAL PROCESS tty1: UNDEFINED;
EXTERNAL PROCESS tty2: UNDEFINED;


PROCESS branch1: office.  /* Local processing for branch1.
    PARAMETERS.
      id     = 1;
      db     = database;
      tty    = tty1;
      ttylink= ttylink1;
END-PROCESS branch1.


PROCESS branch2: office.  /* Local processing for branch2.
    PARAMETERS.
      id     = 2;
      db     = database;
      tty    = tty2;
      ttylink = ttylink2;
END-PROCESS branch2.
```

```
PROCESS database.          /* Central databank.
    DATA.
      VAR   /* Input buffer for messages received from the branch offices.
            ibuffer  ={z | z SUBSET.OF (? X ? X ? X Req)};
      VAR   /* Output buffer for messages to be sent to the branch offices.
            obuffer  ={z | z SUBSET.OF (? X ? X ? X Ans)};
      CONST /* Definition of acceptable requests.
            Req      ={('create',c),('update',a,v),('transfer',a1,a2,v),
                       ('read',a[1],..,a[m]) | INTEGER(m) AND m>0 }
      CONST /* Definition of database replies.
            Ans      ={'error'} UNION (Accounts X Names X INTEGER)*;
      VAR   /* Databank containing account information.
            Records  SUBSET.OF (Accounts X Names X INTEGER);
      CONST /* Valid account numbers.
            Accounts ={z | 1000<z AND INTEGER(z)};
      CONST /* Set of all representable names.
            Names    ={z | CHARSTRING(z)};

    INITIALIZATION.
      UNDEFINED.

    PROCEDURE file(z). /* Branch query is filed for future processing.
      IN:     z=(id,no,trm,req);
      OUT:    ibuffer'=ibuffer UNION {z};
      EXPT:   NIL;

    PROCEDURE transaction. /* A query from the input buffer is processed
                           /* and the answer is placed in the output buffer.
      IN:     z=(id,no,trm,req) AND z MEMBER.OF ibuffer AND
                IF t=(id1,no1,trm1,req1) AND t MEMBER.OF ibuffer
                      THEN no<no1+1;

      OUT1:   /* Final buffer states.
              ibuffer'=ibuffer MINUS {z};
              obuffer'=obuffer UNION {(id,no,trm,ans)};

      OUT2:   /* The effect of creating an account for customer c.
              IF req=('create',c) THEN
                Records'=Records UNION {(k,c,0)} AND
                ans=(k,c) AND k=NEWACCT;

      OUT3:   /* The result of extracting information about m accounts.
              IF req=('read',a[1],..,a[m]) THEN
                IF (a[i=1,..,m],c[i],b[i]) MEMBER.OF Records
                  THEN ans=((a[1],c[1],b[1]),..,a[m],c[m],b[m]));
                  ELSE ans='error';

      OUT4:   /* The result of adding signed value v to account a.
              IF req=('update',a,v) THEN
               IF (a,c,u) MEMBER.OF Records AND u+v+1>0
                 THEN Records'=Records MINUS {(a,c,u)} UNION {(a,c,u+v)}
                      AND ans=(a,c,u+v);
                 ELSE ans='error';
```

```
        OUT5:   /* The effect of transfering positive amount v from
                /* account a1 to a2.
                IF req=('transfer',a1,a2,v) THEN
                  IF (a1,c1,u1),(a2,c2,u2) MEMBER.OF Records
                     AND u1-v+1>0 AND v>0
                    THEN Records'=Records MINUS {(a1,c1,u1)} MINUS {(a2,c2,u2)}
                                          UNION {(a1,c1,u1-v)} UNION {(a2,c2,u2+v)}
                         AND  ans=((a1,c1,u1-v),(a2,c2,u2+v))
                    ELSE ans='error';

        EXPT:   RESTART;

      PROCEDURE z:=retrieve. /* Processed query is removed from the
                             /* output buffer.
        IN:     w=(id,no,trm,ans) AND w MEMBER.OF obuffer;
        OUT:    z'=w AND obuffer'=obuffer MINUS {w};
        EXPT:   RESTART;

      PROCEDURE w:=branchid(z). /* The destination of answer contained in z
                                /* is determined and returned in w.
        IN:     z=(id,no,trm,ans);
        OUT:    w'=id;

      BEHAVIOR.
        PARBEGIN
            BEGIN /* Database queries are accepted and placed in the
                  /* input buffer.
                LOOP  {GET(z) FROM ALL ON switch; file(z);}
            END.

            BEGIN /* Database answers are removed from the output buffer
                  /* and sent to their sources.
                LOOP  {z:=retrieve;          w:=branchid(z);
                       IF w=1 THEN SEND(z) TO branch1 ON switch;
                       IF w=2 THEN SEND(z) TO branch2 ON switch;}
            END.
        PAREND.
END-PROCESS database.

LINKS.
   /* Definition of logical communication links.
      switch:   (branch1, branch2, database);
      ttylink1: (tty1, branch1);
      ttylink2: (tty2, branch2);
```

```
  COMMUNICATION.
     /* Definition of the communication protocol for each link.
        switch: PARBEGIN
                  BEGIN LOOP { branch1:SEND[1](z) to database;
                               database:GET[1](z);
                               {branch1:GO[1]; // database:GO[1];}  }
                  END.

                  BEGIN LOOP { branch2:SEND[1](z) to database;
                               database:GET[1](z);
                               {branch2:GO[1]; // database:GO[1];}  }
                  END.

                  BEGIN LOOP { database:SEND[1](z) to branch1;
                               branch1:GET[1](z) from database;
                               {database:GO[1]; // branch1:GO[1];}  }
                  END.

                  BEGIN LOOP { database:SEND[1](z) to branch2;
                               branch2:GET[1](z) from database;
                               {database:GO[1]; // branch2:GO[1];}  }
                  END.
                PAREND.

        ttylink1: UNDEFINED;
        ttylink2: UNDEFINED;

END-NET egbank.
```

LANGUAGE DEFINITION

The presentation of the language is organized in a manner similar to that of the DSDL specifications except that the definition of a process is introduced first and is later used in the definition of the net. The discussion of the process consists of a formal statement of the semantics of the process and its component entities (data, procedures, and behavior) and an overview of the language features available for specifying processes. The net and its components (environment, processes, links, and communication) receive a similar treatment.

PROCESS DEFINITION.

The process is the basic functional unit of DSDL, and is similar to the guardian [LISK79] and the monitor [RIDD78] (except that internal parallelism is allowed). It serves to encapsulate data as in the abstract data type [GUTT77, WULF76], and has sole access to its own data. In addition, a process is able to receive and send information via messages as in [HOAR78]. In order to define the functionality of a process, a set of procedures are defined. They perform indivisible operations on data or communicate with the environment. The behavior of a process is then defined as the set of allowable sequences of procedure invocations within the process. The formal definition of the process is shown below.

DEFINITION.

A process p is defined as a five-tuple

$$p = (Dp, Tp, Rp, Sp, Bp)$$

where

$Dp = (Qp, Hp, Ip)$
with Qp denoting the set of data entities controlled by p, the data invariant Hp being a predicate over Qp, and the initialization Ip being a predicate defining the initial values for the data in Qp.

$Tp = \{z \mid z = (Ain(Dp, w),\ Aout(Dp, Dp', w, w'))\}$
with Tp representing a set of transformational procedures described by pairs of assertions; the input assertion is a predicate over the data owned by the process p, i.e., Dp, and the input values of the parameters w; the output assertion is over the old and the new values of the data and of the parameters.

$Rp = \{z \mid z = (\text{'true'},\ Aout(Dp, w'))\}$
with Rp representing a set of message receiving procedures whose (partial) meaning is given by an output assertion which describes the kind of values expected to be received from some other processes.

$Sp = \{z \mid z = (Ain(Dp, w),\ \text{'true'})\}$
with Sp representing a set of message sending procedures

whose (partial) meaning is given by an input assertion which describes the kind of values expected to be sent to some other processes.

Bp SUBSET.OF (Tp U Rp U Sp)*
with Bp defining the process behavior given in terms of possible sequences of events, where each instance of a procedure invocation is treated as a primitive event.

In the EGBANK example, 'branch1', 'branch2', and 'database' are processes while 'office' is a process type used in defining the first two of the three processes in the net. The basic process definition takes two syntactic forms:

(1) PROCESS process.name;

DATA.
... definitions;
INITIALIZATION.
... initial values;

PROCEDURE ... definition;
...
PROCEDURE ... definition;

BEHAVIOR.
... description;

END-PROCESS process.name.

(2) PROCESS process.name:
process.type;

PARAMETERS.
... values;

END-PROCESS process.name.

Note: the process.type has to be declared through the use of the 'DEFINE' statement.

DATA.

The first element in the 5-tuple representing the process p is the data Dp, which belongs to that process. The data is defined as an ordered triple (Qp, Hp, Ip), as shown above. The first element, Qp, is a set of data entities controlled by p. Qp may be of arbitrary complexity and structure and its elements may be accessed only by procedures within the process. The second element, Hp, is the data invariant, which is a predicate describing the properties which must be possessed by the elements of Qp both before and after all data transformations. (See [WULF76] for a discussion of the abstract and concrete invariants used in Alphard). The purpose of the invariant is to provide an aid in checking for preservation of data consistency and to serve as a lemma in proofs concerning the process. The third element, Ip, is a predicate defining the initial values for each data entity in Qp.

The data controlled by some process appears in the "DATA" section of the process definition. Both variables and constants may be declared using statements whose syntax resembles Pascal. The variable declarations are placed side by side predicates that are taken to be parts of the invariant Hp (e.g., VAR n: INTEGER; n>0;). Because of the set theoretical approach to data representation adopted by DSDL, both variables and constants are either sets or elements of sets. Some sets are assumed to

be built-in (e.g., INTEGER) while others are constructed by enumeration (e.g., S={1,2,3}), by providing an intensional definition (e.g., S={z | 0<z<4 AND INTEGER(z)}), or by means of standard set operations (e.g., union, intersection, subtraction, cross-product, etc.). In addition to the set notation the standard mathematical notation for functions and relations is also available:

```
CONST f : INTEGER -> INTEGER;
      f(n) <=  IF n<1 THEN 1
                      ELSE n X f(n-1);
```

The use of the set theoretical notation is motivated not only by the desire to develop a simple but formal specification language, but also by a deliberate effort to promote a high level of abstraction which, free of unnecessary detail, allows the designer to concentrate on system level issues rather than the design intricacies of its components. Furthermore, most system designers are fairly familiar with set theory, a fact that makes it attractive both from the point of view of ease of use and with regard to the analizability of the specifications being generated.

PROCEDURES.

The process activities, data transformations and message exchanges, are defined by the procedures it controls. The transformational procedures, given in terms of input and output assertions, describe state changes and return values to be used as input parameters in subsequent procedure invocations. While these procedures are defined by the user of the language, the message exchanges are carried out by two built-in procedures (SEND and GET) whose semantics are stated in the communication section of the net definition. Consequently, their discussion is postponed for now.

The use of nonprocedural specifications in defining the meaning of the transformational procedures enhances the understandability of the process specification. Furthermore, by treating procedure invocations as primitive operations over the data the need for synchronization within a process is avoided in the same way as it is done in the monitor concept employed by concurrent Pascal [HANS77], but without prohibiting concurrency from occurring in the process.

Syntactically, the definition of the transformational procedures is straightforward: pairs of input ("IN:") and output ("OUT:") assertions are used to cover distinct cases; when an input assertion is followed by several output assertions (numbered or not) a conjunction between them is implied; an exception ("EXPT:") assertion may be provided to indicate the action to be taken in case all input assertions fail (e.g., NIL, RESTART, ABORT, etc.); standard predicate calculus is used in constructing the assertions (AND, OR, XOR, NOT, IF-THEN-ELSE, i.e., implication); finally, a name followed by a single quote denotes the value of a data item after the completion of the procedure.

BEHAVIOR.

The behavior of a process is defined as the set of all allowable sequences of events within a process, where an event is defined as an

invocation of a procedure. The following constructs are available for the behavior specification:

```
PARBEGIN concurrent begin-end blocks PAREND

{ event.sequence1 // ... // event.sequence.n }

BEGIN list of events or event.sequences END

{ event.sequence1 ... event.sequence.n }

IF condition THEN event.sequence1 ELSE event.sequence2

CASE (condition1) --> event.sequence1
       ...
     ()           --> default.event.sequence
ENDCASE

WHILE (condition) --> event.sequence

LOOP event.sequence

DOUNTIL (condition) --> event.sequence
```

where an event.sequence is

- a procedure invocation followed by a semicolon (e.g., z:=f(a);),
- a list of event.sequences between braces (e.g., {z:=f(a); g(a,z);}),
- a list of concurrent event.sequences (e.g., {{z:=f(a); g(a,z);} // h(a,b);}), or
- any sequence of events described by one of the flow of control constructs listed above.

Because of the nature of distributed processing, in general, and due to the fact that at higher levels of abstraction processes represent entire networks, the availability of both concurrency and nondeterminism in describing the process behavior is essential. The PARBEGIN-PAREND and the concurrent event sequences provide the mechanisms needed to express concurrency, while the CASE construct allows one to support nondeterminism. One last thing to be mentioned here is the scope rules for the variables used in the behavior section of the process definition: their type need not be declared because it is determined by the procedure definitions. The scope rules are the same as in all block structured languages and a mere listing of the variables after the BEGIN or PARBEGIN is required. As an exception, variables that are not declared explicitly are associated with the block in which are first used.

NET DEFINITION.

In order to specify a distributed system, the concept of a net is included in DSDL. A net is defined as a set of independent, concurrent processes which communicate among themselves by means of messages [BELL77, FELD79, RIDD78]. Messages are sent over abstract communications paths

called links. The behavior of these links, along with the behavior of each process, determines the behavior of the net as a whole. The formal definition of the net is given next.

DEFINITION.

A net n is defined as a four-tuple

$$n = (P, P', L, C)$$

where

$P = \{ p \mid p \text{ is a process} \}$

P' SUBSET.OF P
with P' representing a set of processes used to model the environment of the system.

$L = \{ l \mid l \text{ SUBSET.OF } P \}$
where a link l is defined by the set of processes that may use it.

C : L --> POWERSET.OF ((UNION OVER ALL p OF (Sp UNION Rp))*)
with
C(l) SUBSET.OF (UNION OVER ALL p MEMBER.OF l (Sp UNION Rp))*
i.e., C(l) establishes the link behavior which determines the set of allowable sequences of send and receive events over the link.

The first two elements in the 4-tuple describing the net are the set of processes P and a set P' such that P' SUBSET.OF P. Each of the processes in the set P is an independent unit, and is defined formally in the manner illustrated in the previous subsection. The process is used to model a single processing/storage element in a concurrent and possibly distributed system. These processes represent logical functional units, and not physical processors. Hence, an actual implementation of a process may be split across several real processors or share a single processor with several other processes. As part of the concurrent system, in most cases one or more processes are used to model the environment. These environment processes comprise the set P'.

The last two elements in the net 4-tuple are the set of links L and the communications protocol C. The links connect receiving ports of processes on the link to sending ports of other processes on the link, and represent the available paths of communication within the net. Each link is a logical communications path. Hence, a link may represent a large number of physical connections (such as paths through a packet switching network) or simply a message buffer in shared memory. For each link l a set C(l) of allowable sequences of send and receive type events in the processes that it connects is defined. This set essentially describes the communications protocol on the link, and will be referred to as the link behavior. The events used to define each link behavior are all of the send and receive type events in the processes which it connects, and the link behavior itself is specified using constructs introduced earlier for

defining a process behavior.

ENVIRONMENT.

In general, the nature of the environment with which a system is intended to interact affects the design not only with respect to the functionality that needs to be supported but also in terms of the workload characteristics. Systems having identical functionality may exhibit significant differences in design complexity due to the distinct assumptions made about their environment. Consequently, a system specification can hardly be considered complete unless these, often hidden, assumptions are made explicit. In DSDL, processes declared to be "EXTERNAL" encapsulate the nature of the environment. However, whenever the environment plays only a marginal role in the specification of the system, the external processes and the links that connect them to the system may be declared to be undefined. Under such circumstances, the environment is presumed to provide the system with "appropriate" messages on demand and to immediately accept all messages generated by the system.

PROCESSES.

The processes that form a net are defined in the manner already discussed above. It must be added, however, that processes at one level of the specification may represent abstractions of entire nets to be identified later. DSDL allows one to state this fact through the use of the attribute "REFINEMENT.OF" as in the example below:

```
NET netname; REFINEMENT.OF pname.
...
END-NET netname.
```

where the net "netname" is identified to be a refinement of the process "pname". Furthermore, any entity in the net may be declared to be a refinement of some entity in the process as long as consistency is preserved. Unfortunately, general consistency proof techniques are still under investigation and ad-hoc methods are used instead.

LINKS.

Each link is defined by its unique name, the processes that may use it, and a description of its behavior given in the section on communication. More than two processes may have access to the same link and the same two processes may have more than one link in common. The motivation for this approach is to be found in the desire to enable the description of arbitrary interconnection structures. Furthermore, because links are logical and not physical in nature, a link may be later refined as a net that implements the behavior of the respective link. A packet switching net, for instance, may be described first as a link between all the nodes it services and may be subsequently refined to include the switching nodes and their protocols.

COMMUNICATION.

For every link in the net, a behavior description has to be included in the communication definition section. The link behavior defines the communication protocol associated with the respective link, i.e., the semantics of the GET and SEND commands. In defining the link behavior, the designer may use the same the same means of specification as in the

description of a process behavior except that the set of events that may be involved is restricted to

- invocation of a receiving procedure
  (e.g., branch1:GET[1](z) from database;)

- invocation of a sending procedure
  (e.g., branch1:SEND[1](z) to database;)

- resumption of processing after the invocation of a sending or receiving procedure (e.g., branch1:GO[1];)

in processes associated with the respective link.

(Note: the number between the square brackets serves the purpose of matching resumption of processing events with corresponding invocations of sending and receiving procedures.)

In the definition of the behavior of the link called "switch", for instance, the following BEGIN-END block appears:

```
BEGIN LOOP { branch1:SEND[1](z) to database;
             database:GET[1](z);
             {branch1:GO[1]; // database:GO[1];}  }
END.
```

It establishes the fact that once a SEND is invoked, the issuing process waits for completion of the corresponding GET and that no GET is invoked by the database unless the corresponding SEND has been issued first. Moreover, after exchanging the message "z", both processes may resume processing in no particular order. Similar protocols are described for the other message exchanges occurring over the link "switch".

The behavior of the net as a whole is determined by the behaviors of its processes and links. The net behavior is formally defined as the set of all sequences of events that have the property that are consistent with the local behavior of each of the processes and links:

$$B = (Bp1 \,\%\, \ldots \%\, Bpn) \,\%\, (C(l1) \,\%\, \ldots \,\%\, C(lm))$$

where

B is the net behavior
Bpi is the behavior of process i (for i=1,...,n)
C(lj) is the protocol for link j (for j=1,...,m)

and

% is the synchronized shuffle operator defined as follows

$$V \,\%\, W = \{ v \,\%\, w \mid (v \text{ MEMBER.OF } V) \text{ AND } (w \text{ MEMBER.OF } W) \}$$
$$a \,\%\, b = \{ab, ba\}$$
$$a \,\%\, a = \{a\}$$

CONCLUSIONS

There are four key features that make DSDL an attractive candidate for a distributed system design language: high degree of formality, designer specified communication protocols, non-procedural character, and strong emphasis on separation of concerns.

The high degree of formality is achieved through the use of set theory and predicate calculus. Both are a common place background for recent computer science graduates and for most experienced system designers. Furthermore, sets and set operators are already present in languages such as Pascal while predicate calculus is central to current work in artificial intelligence. Thus, it appears that future attempts to support and analyze DSDL may require initially no development of new technology but use of already available expertise in the computer science field.

The freedom of specifying arbitrary communication protocols is an essential feature for any distributed system design language. One can hardly expect a designer to have to have to limit the design space due to specification language limitations. Moreover, evaluation of alternate communication protocols is an important design activity and ought to be supported in a straightforward manner, i.e., one should be able to alter the communication protocols without having to consider changes in the other aspects of the system specifications.

Advances in proofs of the correctness of sequential programs have been based on the non-procedural nature of I/O assertions. It is our conjecture that the specification of distributed systems could benefit from the use of I/O assertions for the description of presumedly sequential activities within a process, and from the extension of the non-procedural type of specifications to behavior specification. (Efforts to accomplish the latter goal have not been completely successful and, consequently, the current definition of DSDL describes both behavior and communication in a procedural manner.) Furthermore, non-procedural specifications avoid the addition of extraneous detail during the different stages of the design and specification.

It is a generally accepted fact that hierarchical design is useful in reducing the overall complexity of large systems. Further reductions in the complexity of the specification are achievable by imposing some appropriate structure over each level of the hierarchical specification. In DSDL, this is achieved by applying the principle of separation of concerns in such a way as to assist in the logical verification of the specification at that level. The solution is to structure the design specifications based on correctness and self-consistency proof dependencies. In DSDL, for instance, one would first prove the preservation of specified invariants over the data local to a procedure. Proofs about data may then be employed as lemmas in proofs about the procedures; proofs about procedures may be used as lemmas in proofs about the processes, etc. The dependence of the higher-level entities on the lower-level entities creates the opportunity for simplification of proofs about the system as a whole through the use of hierarchically structured proofs.

DSDL has been exercised on several small problems. These exercises were useful in demonstrating the language's power of expression. Nevertheless, there are many unresolved issues. First of all, a meaningful evaluation of the language demands its use on a real-life project of adequate complexity. Second, future advances in the study of the formal aspect of the language must be carried out in preparation for potential incorporation of DSDL in a computer-aided design system. As of now, a definition for consistency between levels has been proposed but proof strategies need to be developed. Finally, a complete system specification language ought to include also the capability to define processors and their characteristics, the rules for allocating processes among processors, and performance specifications. Research in these areas is currently under way and its results will be reported elsewhere.

REFERENCES

[BELL77] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.

[BOOT80] Booth, T. L. and Wiecek, C. A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," IEEE Trans. on Soft. Eng. SE-6, No. 2, pp. 138-151, March 1980.

[CAMP79] Campbell, R. H. and Kolstad, R. B., "Path Expressions in Pascal," Proc. 4th Int. Conf. on Soft. Eng., pp. 212-219, 1979.

[FELD79] Feldman, J. A., "High Level Programming for Distributed Computing," CACM 22, No. 6, pp. 353-368, June 1979.

[GUTT77] Guttag, J., "Abstract Data Types and the Development of Data Structures," CACM 20, No.6, pp. 396-404, June 1977.

[HANS77] Hansen, B., The Architecture of Concurrent Programs, Prentice-Hall, 1977.

[HOAR78] Hoare, C. A. R., "Communicating Sequential Processes," CACM 21, No. 8, pp. 666-677, August 1978.

[LISK77] Liskov, B., et al, "Abstraction Mechanisms in CLU," CACM 20, No. 8, pp. 564-576, August 1977.

[LISK79] Liskov, B., and Berzins, V., "An Appraisal of Program Specifications," Research Directions in Software Technology, P. Wegner, editor, MIT Press, pp. 276-301, 1979.

[RIDD78] Riddle, W. E., et al, "Behavior Modeling During Software Design," IEEE Trans. on Soft. Eng. SE-4, No. 4, pp. 671-678, July 1978.

[ROBI77] Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs," CACM 20, No. 4, pp. 271-404, April 1977.

[ROSS77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 16-34, January 1977.

[SANG79] Sanguinetti, J., "A Technique for Integrating Simulation and System Design," Proc. Conf. on Simulation, Measurement and Modeling of Computer Systems, pp. 163-172, August 1979.

[SMIT79] Smith, C. and Browne, J. C., "Modeling Software Systems for Performance Predictions," Proc. Computer Measurement Group X, pp. 321-341, December 1979.

[WULF76] Wulf, W. A., London, R. I., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Soft. Eng. SE-2, No. 4, pp. 243-265, December 1976.