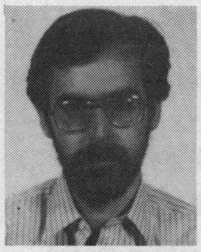
[46] ——, "Collision detection and avoidance in computer controlled manipulators," Ph.D. dissertation, Dep. Elec. Eng., California Inst. Technol., 1977.
[47] J. E. Warnock, "A hidden line algorithm for halftone picture representation," Dep. Comput. Sci., Univ. Utah, Rep. TR4-5, May 1968.
[48] J. Williams, "STICKS—A new approach to LSI design," S.M. thesis, Dep. Elec. Eng., Massachusetts Inst. Technol., 1977.
[49] T. Winograd, *Understanding Natural Language.* New York: Aademic, 1972.
[50] T. C. Woo, "Progress in shape modelling," *IEEE Computer*, vol. 10, Dec. 1977.
[51] F. F. Yao, "On the priority approach to hidden-surface algorithms," in *Proc. 21st Symp. Foundation of Comput. Sci.*, 1980, pp. 301-307.



Tomás Lozano-Pérez was born in Guantanamo, Cuba, on August 21, 1952. He received the S.B. degree in 1973, the S.M. degree in 1977, and the Ph.D. degree in 1980, all in computer science from the Massachussetts Institute of Technology, Cambridge.

In 1981 he joined the Department of Electrical Engineering and Computer Science, Massachussetts Institute of Technology, where he is currently an Assistant Professor. He has been associated with the M.I.T. Artificial Intelligence Laboratory since 1973. His research interests include robotics, geometric modeling, computational geometry, computer vision, and artificial intelligence.

Structured Specification of Communicating Systems

GREGOR V. BOCHMANN AND MICHEL RAYNAL

***Abstract*—Specification methods for distributed systems is the underlying theme of this paper. A model of communicating processes with rendezvous interactions is assumed as a basis for the discussion. The possible interactions by a process, and the interconnection between several subprocesses within a process are specified using the concept of ports, which are specified separately. Step-wise refinement of process specifications and associated verification rules are considered. The step-wise refinement of port specifications and associated interactions is considered as well. After the presentation of an introductory example, the paper discusses the basic concepts of the specification method. They are then applied to more complex examples. The step-wise refinement of ports and interactions is demonstrated by a hardware interface for which an abstract specification and a more detailed implementation is given. Proof rules for verifying the consistency of detailed and more abstract specifications are discussed in some detail.**

***Index Terms*—Communication processes, design verification, distributed system design, interface specifications, parallel processing, ports, specification consistency, specification language, specification methods, step-wise refinement.**

I. Introduction

MUCH work has been done in recent years in the area of design methods for distributed systems. This includes the development of languages for distributed systems, the choice of appropriate interaction mechanisms (message transmission, rendezvous interactions, remote procedure calls, etc.), communication protocol design for long distance and local computer networks, as well as for the communication between several VLSI components within a single computer system. As in the case of nondistributed software systems, the notion of step-wise refinement seems to be an important design tool for distributed systems. Some difficulty is encountered, however, if some sort of indivisible interaction primitives are assumed.

The specification method discussed in this paper indicates how the step-wise refinement of distributed systems may be described with the concept of process substructure and the concept of interactions that may be refined. The method is based on the concepts "process" and "port." A process is an entity that performs some data processing and is assumed to be the unit of specification. A port is a part of a process and serves for the communication of that process with its environment, i.e., other processes in the system. A process may possess several ports for communication with different parts of its environment. The specification of the properties of a process or port is given at an abstract level, in the sense that only the externally visible behavior of a process or port is described (i.e., its communication behavior), but not the way this behavior is realized by an internal structure of the process or port. Process and port implementations are specified separately as the elements for one step in the step-wise refinement of a system description.

Manuscript received January 14, 1982; revised July 6, 1982. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and IRISA-INRIA, France.

G. V. Bochmann is with the Departement d'Informatique et de Recherche Operationnalle, Universite de Montreal, Montreal, P.Q. H3C 3J7, Canada.

M. Raynal is with the ENST de Bretagne, BP 856, 29279 Brest-Cedex, France.

The specification of a port consists of two parts: a) an enumeration of the types of interactions that may occur at the port, and b) additional properties of the port which may in particular restrict the order and parameter values of the executed interactions.

The specification of a process also consists of two parts: a) the specification of the communication properties of its ports (as described above), and b) additional properties of the process which may in particular relate the interactions taking place at the different ports of the process. An implementation of a process is given by defining an internal structure of the process in terms of subprocesses and interconnections between their ports.

This paper discusses the different aspects of this specification method. In Section II, an introductory example is given which illustrates the method and the concepts used. In Section III, the elements of the specification method and its underlying model are defined in more detail. The approach of step-wise refinement, in particular the refinement of port specifications, is discussed in detail in Section IV. Section V contains an overall discussion of the problems of design verification which allows to check the consistency between a process implementation with its specification.

II. An Example

The following example is given as an illustration for the specification method discussed in this paper. It introduces the main specification elements, which are processes and ports. These elements are more formally defined in Section III. The specification method favors a top-down design where first the specification of a process is given and then its refinement (implementation) in terms of communicating subprocesses is considered. Because of such possible refinements, a process is not necessarily sequential, but may involve several parallel activities.

A. The Specification of a Multiserver

Our example consists of a finite number of user processes which use the service provided by another process which is of type *multiserver.* Two operations are provided for interaction with the multiserver process. They are called *request* and *response.* A *request* contains a question which is answered by a *response.* For each user, a *request* interaction initiated by the user must always precede a *response* interaction.

The concept of a port is introduced to structure a process and its possibilities for interaction with the environment. Independently of its realization, a port is defined by a set of interactions and the constraints on their execution. For the example considered here we have the following port type definition.

```
port type service-access is
    operation request (X : question);
              response (Y : answer);
    constraint
    $(/\text{request})_i \rightarrow (/\text{response})_i \rightarrow (/\text{request})_{i+1}$
end service-access.
```

Here it is assumed that data type definitions are given for the parameters used in the port definition. They could, for example, be of the form

```
type question is array [1 .. 100] of char;
     answer is array [1 .. 200] of char.
```

The notation $(p/a)_i \dashrightarrow (q/b)_j$ specifies that the *i*th interaction of type *a* at the port *p* must precede the *j*th interaction of type *b* at the port *q*. For the specification of a port constraint, as in the definition above, the name of the port is implicitly given; therefore we simply write $(/a)_i$ for the *i*th interaction at the port. We note that this sequence-oriented notation is just taken as an example. The concepts discussed in this paper could as well be used when the constraints are specified by methods oriented towards programming languages or state machines [12].

In the following is considered a *multiserver* process which has a number of ports (one for each user process) through which the user processes obtain access to the question answering service. This service is realized by the interactions of type *request* and *response*, as explained above. Without going into more detail, we assume that the answer to a question is a function of that question, and that the interactions over the different ports do not interfere with one another. More formally, these properties may be specified as follows:

```
process type multiserver is
    users : array [user-id-type] of service-access;
  constraint
    for u in user-id-type holds
        $(\text{user}[u]/\text{response})_i . Y =$
FUNCTION-OF $((\text{users}[u]/\text{request})_i . X)$ end multiserver.
```

Here the notation $(p/a)_i . X$ denotes the value of the parameter *X* of the *i*th interaction of type *a* at the port *p*. We assume that the index data type of the array is defined as

```
type user-id-type is 1 .. n;
```

and the function *FUNCTION-OF* defines the answering service of the *multiserver.* This function is not specified here. The given constraint defines the noninterference between the different ports.

The nature of the specification of a process type is similar to the nature of a port type specification. [In the following we often simply say process (or port) for a process (or port) *type*.] In the case of a process specification, the set of declared ports of the process are associated with a constraint which must be satisfied by the executions of the interactions at the different ports. In addition, it is understood that the port constraints of the different ports are satisfied as well. For each port *users*[*u*], for example, the *i*th interaction *request* must precede the *i*th interaction *response.* Each process cooperating with the *multiserver* uses a port of type *service-access.* To show the flexibility of the specification method, we continue the explanation of the example by distinguishing two types of user processes, in particular, we assume that a *simple-user* process will only submit a subset of the possible questions. The user processes may be specified as follows:

```
process type user is
    service : service-access;
end user;
process type simple-user is
    service : service-access;
    constraint $(\text{service/request})_i . X$ in simple-question
end simple-user.
```

As before, the specification of a user process describes the interactions of a process with its environment. The specification of a *simple-user* includes an additional constraint for the parameter value of the *request* operation. Such restrictions are considered in [9]. In simpler cases, it could be specified by a data type definition of the form

```
        type simple-question is question range . . .
```

which is analogous to the definition of subtypes in ADA [16].

For the specification of a process type, only the interactions of that process are considered, but not the internal structure of the specified process. In the example above, nothing is said about the internal operations of the user processes. It is assumed here that they are really the sink for the information provided by the *multiserver* process. In reality, a user process may consist of a terminal handling process, a terminal and an operator that consults the database of the *multiserver* for some clients requesting information. The interactions between the operator and the clients is not defined in the specifications given in our example.

The interconnection of processes of type *user*, *simple-user* and *multiserver*, as shown in Fig. 1, is called a system of cooperating processes. In general, such a system may again be considered a process at a higher level of abstraction. Some language elements are needed for specifying the interconnection structure of the processes. The system shown in Fig. 1 may be specified as follows:

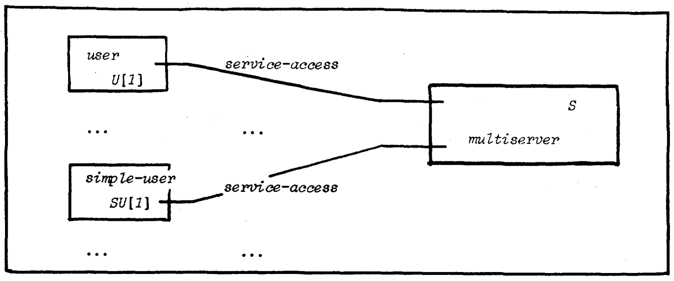


Fig. 1. A system of cooperating processes.

```
system is
    S : multiserver;
    SU : array [1 . . nSU] of simple-user;
    U : array [(nSU + 1) . . n] of user;
connection$_0$
    for u in 1 . . nSU : SU[u] . service = S . users[u];
    for u in (nSU + 1) . . n : U[u] . service = S . users[u]
end system.
```

An interconnection between processes is established by connecting their ports. (It is necessary, however, for the ports to be of the same type.) An interaction occurs when the two processes invoke the same type of interaction on the interconnected ports. A rendezvous kind of interaction mechanism is assumed, as discussed in more detail in Section III-A.

B. A Refinement of the Multiserver Process

It is important for a specification method to support the step-wise refinement of specifications. In the case of the specification method discussed in this paper, a decomposition method is used similar to the example shown in Fig. 1.

Continuing our example, we assume that a *multiserver* process consists of two subprocesses, as shown in Fig. 2: a *server* process that provides the answering service over *one* port, and a *multiplexer* process which provides the appropriate multiplexing between the different users.

The specification of each of these subprocesses may be given in the following form:

```
process type server is
    user : service-access;
    constraint
        $(\text{user/response})_i . Y$ = FUNCTION-OF $((\text{user/request})_i . X)$
end server;
process type multiplexer is
    singles : array [user-id-type] of service-access;
    multiplexed : service-access;
    constraint
        for s in user-id-type holds $\forall i \; \exists i'$
            $((\text{singles}[s]/\text{request})_i = (\text{multiplexed/request})_{i'}$
    and $(\text{singles}[s]/\text{response})_i = (\text{multiplexed/response})_{i'})$
end multiplexer.
```

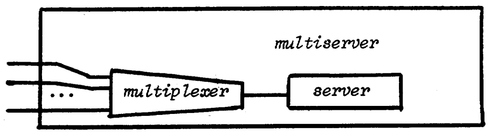


Fig. 2. The internal structure of a *multiserver* process.

In the case of the *server*, the constraint specifies the correspondence between the *i*th question and the answer. In the case of the *multiplexer*, the constraint specifies that all requests from a given user *s* are forwarded to the *service* process and that the obtained answer is returned to the user, without interference with the activity of the other users.

Given the above specification of the subprocesses, we may now give a refined specification of the *multiserver* process as shown in Fig. 2. The specification that follows defines instances of the *service* and *multiplexer* process types, and defines the interconnection between these subprocess instances:

<u>process implementation</u> imps <u>for</u> multiserver <u>is</u>

  M : multiplexer;

  S : server;

  <u>internal connection</u> *M* . multiplexed = *S* . user;

  <u>external connection</u> users <u>is</u> *M* . singles

<u>end</u> imps.

Here the abbreviation *users <u>is</u> M . singles* stands for <u>*for all*</u> *u <u>in</u> user-id-type : users*[*u*] <u>*is*</u> *M . singles*[*u*].

While a process type specification has the major parts "port declarations" and "constraints," a specification of a process implementation, which is a refinement of a given process type specification, has the major parts "declarations of the subprocesses" and the "interconnection of their ports." As the example shows, two kinds of interconnections are distinguished: The *external connections* specify how the ports of the process type specification are realized by the implementation, while the *internal connections* define connections between the ports of the subprocesses that are not visible at the higher level of abstraction (as represented by the process type specification).

It is clearly possible to define several different implementations for a given process type. They would differ from one another by the types of the subprocesses used and by the interconnection structure between the subprocesses. The same kind of external connections must be defined by all implementations, since they connect the external ports which are defined, together with their types, in the process type specification. This approach has certain similarities with the step-wise refinement of abstract data type specifications [24], [14], [34]. As in the case of such specifications, we have to address the problem of asserting the coherence between a process type specification and its implementation. We will come back to these questions in Sections IV and V. We note that the example above and most related work [10], [15], [26], [27], [18] assume that the same type of ports are used in the process type specification and its implementation. In Section IV, we consider the refinements of ports which eliminates this restriction of the specification method.

III. The Concepts Used for Specification

In this section we examine in more detail the concepts used for specification. This amounts to defining an underlying model for the specification method. In Section III-A, we first examine the concepts of processes and their interactions; then in Section III-B, we consider the concept of a port and the interconnection mechanism associated with it. In Section III-C, we introduce the concept of a role which is taken by a process in respect to a given port. Section III-D discusses the specification method based on processes, ports and their interconnection.

A. Processes and their Interactions

For the specification of a process type, the point of view of an external observer is considered. Only the externally visible behavior should be specified; a process is therefore defined by its possibilities of communicating with its environment, i.e., interacting with the other processes in the system.

An interaction between two processes occurs when both processes invoke the same type of interaction on ports that are connected. It is assumed that a rendezvous kind of interaction mechanism [15], [26] is used. The first process ready to execute the interaction must wait for the second to be ready. (The assumption of such an interaction mechanism simplifies the model without restricting its descriptive power; for example, a message queuing mechanism may be specified by the introduction of a buffering process.)

In this paper we do not make the assumption that interactions are necessarily atomic. In fact, we assume, as discussed in [22] that the execution of an interaction consumes some time. This assumption seems more realistic than an "event model" [19] where it is assumed, for specification purposes, that different interactions exclude one another in time. We note, however, that a transition between the two models may be defined: it is sufficient to define for each nonatomic interaction *p* two atomic events *begin*(*p*) and *end*(*p*). By specifying a total order over the events *begin*(p_i) and *end*(p_i), it is possible to define an arbitrary ordering of the interactions p_i, including the possibility of simultaneity [13], [32], [1].

Since the specification of a process type should only describe its externally visible behavior, the specification should define the possible interaction types and the order in which they may be executed. As shown by the example in Section II-A, the interaction types are defined by a name and the type of the exchanged parameters. (The reason for associating the interactions with ports will be discussed in Section III-B.) In general, the constraint part of the specification of a process type defines restrictions on the possible interactions and the possible execution order. The specification of such restrictions may take different forms, such as assertions on parameter values, expressions defining full or partial orders of execution sequences. Different specification techniques may be considered for this purpose. Whichever technique is chosen, it must be suitable for the two kinds of restrictions mentioned above. In the case that a total order of the interactions is specified, the specifi-

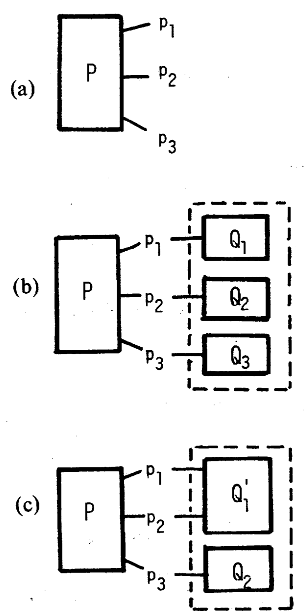


Fig. 3. A process and its environment. (a) Abstract view of the environment. (b) and (c) Different implementations of the same environment specification.

cation of a process type is analogous to the specification of a module or abstract data type as discussed in [30] and [4].

B. Ports and their Connections

The concept of a port [3], [33] allows to give some structure to the interactions of a process with its environment. The ports defined for a given process type may be considered as an abstraction of the environment of that process. Let us consider the example of a process P with three ports p_1, p_2 and p_3, as shown in Fig. 3.

The ports p_i represent an abstract specification of the environment of P, as it is seen by the process. The environment may be implemented in different ways, as shown in the figure. The specification of P, however, is the same for all these environments.

A port binds together a number of different interaction types and defines some constraints on their executions (see example of Section II-A). Similarly, a process, if it has several ports, defines some additional constraints that must be satisfied for the executions of interactions at the different process ports. Each interaction of the process is associated with a particular port.

For the specification of distributed systems, the notion of a port may also be used for distinguishing different points (in the distribution space) of the distributed system. The specification of a communication service, for example, is usually given in terms of a (distributed) process that provides a certain number of ports (called "access points") through which the user processes may access the communication service [6].

Apart from the above considerations, the main reason for the introduction of the port concept is its use in the refinement process. As already discussed in Section II-B, the concept of ports is essential for the specification of process implementations in terms of subprocesses. The interconnection of these subprocesses is in fact defined by establishing an identity relation between the ports of different subprocesses. The connected ports must be of the same type (i.e., allow the same type of interactions with the same port constraints). It is possible to define more complex connection patterns where the ports of more than two processes are identified by one connection [30]. An example of such a connection can be found in Section IV-C.

C. The Role Concept

The concept of a port as explained above may appear incomplete. In fact, for an interaction that involves the transfer of parameter values from one process to another, it may be necessary for a complete specification to indicate which process determines the value. (We note that the interaction is executed when both processes invoke the interaction as a rendezvous; therefore both processes may, in principle, participate in the determination of the parameter values.) The role concept is introduced to make this aspect more precise: In the definition of an interaction, each parameter is associated with the name of a role, and only the process that has this role attribute associated with the port will have to provide the parameter value for the interaction in question; the other process is implicitly the receiver of the parameter value. When for the purpose of implementation two ports of different processes are connected, it is clearly necessary that the two connected processes have complementary roles for the connected ports. We note that this concept has certain similarities with the definition of protection attributes [17] which apply to operation names, instead of parameters as in our case.

To further explain the concept, we consider the following example:

```
port type p is
  role side1, side2;
  operation op1 (X by side1 : integer);
            op2 (Y by side2 : character);
            op3 (T by side1 : integer, U by side2 : Boolean);
  constraint . . . .
end p.
```

We suppose that two processes $Q1$ and $Q2$ have declared the ports $p1$ and $p2$ of type p with the roles *side*1 and *side*2, respectively, and suppose that the two ports are connected. In the case of a joint execution of the *op*1 interaction, the process $Q1$ will provide the integer parameter value and the process $Q2$ will receive it; in the case of *op*2 the actions of $Q1$ and $Q2$ are interchanged. In the case of *op*3, $Q1$ determines the integer value of the parameter T, while $Q2$ determines the boolean value of U.

It is important to note that the roles played by the processes in respect to interaction parameters is independent of which process initiates the interaction. For example, the execution of *op*1 could be initiated by $Q1$ or $Q2$. The distinction of an initiating process is usually made in a more detailed specification or an implementation of a system. For example, when guarded commands [11] are used for the description of the processes, the initiating process is usually the one for which the interaction is not part of a guard. (The initiation is usually in a deterministic context.) We note that in CSP [15], in contrast to ADA [16], the distinction between initiation and parameter determination does not exist: only the reception of interaction parameters may appear in guards. Several proposals for eliminating this restriction have been made [5], [33], and [7].

D. Specification of Ports and Processes

The example of Section II showed the similarity of the specifications for ports and processes: the specification of a port type consists of a list of interactions (possibly with roles) and constraints on their execution; the specificatin of a process type consists of a list of ports and constraints on the execution of the interactions that are defined for those ports. The latter constraints are sometimes called "global constraints" since they involve the interactions on several ports. In addition, the process satisfies for each of its ports, the constraints specified for the corresponding port type. These constraints are sometimes called "local constraints" since they are local to one port. The syntax used in this paper for the specification of process types adopts a kind of "ADA style." The syntax for a process type specification is as follows:

```
process type ⟨name⟩ is
        ⟨list of port declarations⟩;
        constraint
        ⟨port constraints on execution order⟩
end ⟨name⟩.
```

For the description of an implementation of a process type, as a step in the refinement process, the following syntax is used:

```
process implementation ⟨impl-name⟩ for ⟨name⟩ is
⟨subprocess declarations⟩;
internal connection ⟨interconnections of subprocess ports⟩;
external connection ⟨interconnections to the external ports⟩
end ⟨impl-name⟩.
```

In the remaining part of this section we give another example which illustrates the specification of processes with roles. The example is a process that provides some communication service with queuing between two ports $p1$ and $p2$. User processes that are connected to these ports may exchange messages in both directions through the service. A specification of the port and process type is given in the following.

```
port type sr-point is
    role caller, callee;
      operation send (X by caller : message);
                receive (X by callee : message);
        constraint
            $(/\text{send})_i \dashrightarrow (/\text{receive})_j$ or $(/\text{receive})_j \dashrightarrow (/\text{send})_i$
end sr-point;
process type service is
        p1(callee), p2(callee) : sr-point;
        constraint $(p1/\text{send})_i . X = (p2/\text{receive})_i . X$
            and $(p1/\text{receive})_j . X = (p2/\text{send})_j . X$
end service.
```

The constraints of the *service* specifies that the received messages are those that were sent, and that the order of reception is the same as the order of sending; the communication service is reliable, no messages are lost nor damaged. In the case that additional coordination is desired (such as rendezvous, limited buffering, etc.), supplementary constraints must be given. For instance, the additional constraints:

$(p2/\text{receive})_i \dashrightarrow (p1/\text{send})_{i+n}$

and $(p1/\text{receive})_j \dashrightarrow (p2/\text{send})_{j+m}$

would specify a maximal buffering of n in one direction, and m in the other.

We note that the declaration of the ports in the definition of the *service* process type includes a definition of the role of the *service* process at the ports. The specification for the port $p1$ implies that the connected user process provides the message parameter for the *send* interaction, and the *service* process for the *receive* interaction.

IV. Step-Wise Refinement of a Specification

The proposed specification method allows a step-wise refinement of specifications. The example of Section II shows on the one side, how the specification of the external behavior of a process or port can be given independently of an implementation, and on the other side, how a specification can be refined by giving a particular implementation. Such an implementation is given by defining a decomposition of the process into several subprocesses and a connection pattern between the ports of the subprocesses. This kind of step-wise refinement is considered in previous work [10], [27], [18]. It implies that the interaction primitives that are exchanged between the process and its environment are the same for the more abstract specification as for the more detailed refinement. We think that it is important to allow for a step-wise refinement of the interaction primitives as well. Therefore we discuss in Section IV-B an approach to the step-wise refinement of port specifications. The use of this approach is illustrated in Section IV-C by the example of an arbiter [31], [8], and the implications for the verification of the refinement step are discussed in Section V.

We note that the approach to refinement discussed in this paper can be used in as many steps of refinement as desired. As shown by the arbiter example of Section IV-C, this may lead to a level of detail which allows direct realization in hardware. If software implementation is sought, it seems convenient to convert at a certain level of detail to a specification by a programming language, which is used to implement the subprocesses obtained in the last step of the refinement. An example for this can be found in [28].

A. The Internal Structure of Processes

We give in this subsection some comments on the internal structure of a process implementation. First, there are two kinds of connections: internal and external ones (see example in Section II-B). The former interconnect ports of the subprocesses and are purely a matter of implementation choice. The latter specify which ports of the implementing subprocesses represent the (external) ports that are defined in the higher-level specification of the process. Therefore the same kinds of external connections will be found in each implementation of the process. Second, it is important to note that during the phase of the compilation of a process implementation description, it is possible to perform certain consistency checks. In particular, the port types and roles can be checked for each connection, as discussed in the Sections III-B and III-C. Another check would be to detect any port that has not been used for any connection.

B. Port Refinements

A port of a process represents an abstract interface of the process which defines the possible interactions and the constraints for their execution, as explained in Section III-B. A refinement of such a port, which must be defined for an implementation, brings up the following two questions.

1) Is a given interaction of the port, which is a primitive at the abstract level of the specification, also a primitive at the more detailed level of the implementation?

2 Is a given port composed of several subports? (In this case a nonprimitive interaction may involve several interactions at several different subports.)

To illustrate the first question, we consider the port specification that follows.

port type resource *is*

  role user, server;

  operation use-resource (X *by* user : question,

      Y *by* server : answer);

  constraint (/use-resource)$_i \dashrightarrow$ (/user-resource)$_{i+1}$

  end resource.

This port defines the interaction *use-resource* which represents a procedural access by a *user* process to a process playing the *server* role. The *user* process provides the *question* value of the first parameter and receives from the *server* the *answer* value of the second parameter. Comparing this port specification with the *service-access* port defined in Section II-B, we see that the latter may be considered an implementation of the former, where the *use-resource* interaction is implemented by the sequential composition of the interactions *request* and *response*. (The syntax for the specification of such a port implementation is demonstrated by the next example.)

To illustrate the second question (implementation with subports), and to give an example of the syntax for port implementations, we consider the port specification below.

port type reservation *is*

  role requesting, responding;

  operation reserve;

      release;

  constraint (/reserve)$_i \dashrightarrow$ (/reserve)$_{i+1}$ (1)

    and (/release)$_i \dashrightarrow$ (/release)$_{i+1}$ (2)

    and (/reserve)$_i \dashrightarrow$ (/release)$_i$ (3)

    and (/release)$_i \cdots\!\rightarrow$ (/reserve)$_{i+1}$ (4)

end reservation.

The port type *reservation* offers two types of interactions, *reserve* and *release*. The execution constraints are expressed using two temporal relations "$\dashrightarrow$" (precedes) and "$\cdots\!\rightarrow$" (may influence) [22]. (The expression "$A \dashrightarrow B$" means that the execution of the interaction A ends before the execution of the interaction B begins; and "$A \cdots\!\rightarrow B$" means that A has begun when B ends). These temporal relations are used in the specification of the constraints to indicate that the execution of the interactions may extend over a certain time period, and their execution may possibly overlap. In fact, the $(i + 1)$th *reserve* interaction may begin before the ith *release* interaction ends, but it may not end before the latter [see line (4)]. Lines (1)–(3) define the usual sequential ordering of interactions.

Let us now consider the port specification given below, which reflects the properties of a hardware circuit (i.e., a single wire) in a digital system.

port type circuit *is*

  role active, passive;

  operation up;

      down;

  constraint (/up)$_i \dashrightarrow$ (/down)$_i \dashrightarrow$ (/up)$_{i+1}$

end circuit;

The two interactions of the *circuit* are *up* and *down* (i.e., "set" and "reset" or set to 1 and set to 0) with the constraint that their execution alternates.

In the following we give an implementation of the *reservation* port type by two subports R and A of type *circuit*. The implementation is based on the so called "four-cycle signaling scheme" which is illustrated in Fig. 4.

The subport R carries the *reserve* and *release* requests (implemented by the *up* and *down* interactions, respectively) from the process with the *requesting* role to the process with the *responding* role, while A returns the acknowledgments of

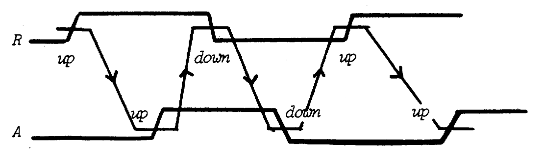


Fig. 4. Four-cycle signaling scheme.

the *responding* process. The order in which these signals may be invoked is defined by the four-cycle signaling scheme, as shown in Fig. 4. (We note that a temporal logic description of this scheme may be found in [8].) A formal notation for the description of this port implementation is given below. It also demonstrates a possible syntax for such specifications.

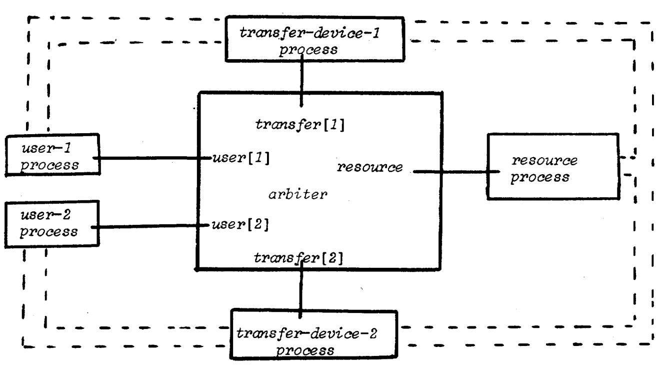


Fig. 5. Structure of the system with an *arbiter*.

<u>*port implementation*</u> four-cycle <u>*for*</u> reservation <u>*is*</u>

R (active=requesting), A (active=responding) : circuit;

<u>*constraint*</u>

$(R/\text{up})_i \dashrightarrow (A/\text{up})_i \dashrightarrow (R/\text{down})_i \dashrightarrow (A/\text{down})_i \dashrightarrow (R/\text{up})_{i+1}$

<u>*mapping*</u> reserve <u>*is*</u> R . up $\dashrightarrow$ A . up;

<u>*mapping*</u> release <u>*is*</u> R . down $\dashrightarrow$ A . down

<u>*end*</u> four-cycle.

The *constraint* part of the specification defines the four-cycle signaling scheme, and a *correspondence* part which defines the implementation of the interactions by giving for each interaction of the *reservation* port a mapping into a sequence of interactions of the subports R and A. The declaration of the subports also defines the relation between the roles of the subports and the roles of the port. We note that this relation is not the same for the two subports, since R carries information from the *requesting* to the *responding* process (the *active* ole is *requesting*), while A carries information in the opposite direction (the *active* role is *responding*). We note that the mapping definition which defines the correspondence between the port and subport interactions is particularly simple in this example. In general, it may involve sets of subport interaction sequences, and/or mapping of parameter values similar to the mappings found in the stepwise refinement of abstract datatypes [34], [14].

The above example demonstrates the refinement of a port into two subports of simpler structure. It is an example of a step-wise refinement process that leads from a more abstract port specification to a more detailed specification that shows how the abstract port may be implemented. Such port refinements are important for the design of communicating systems for the following reasons.

1) The communication between the different parts of a system is usually first described by defining an "abstract interface" which defines the logical interactions that may take place [6]. Subsequently, during the system implementation, this abstract interface is refined by giving an implementation in the form of software procedure calls, interprocess communication primitives, or hardware constructs.

2) The *constraint* part of the port type, or port implementation specification gives a complete view of the constraints that are imposed on the communication between two processes at the given level of detail. We believe that a separate specification of these constraints leads to a clearer structure of the system specification.

3) We note that the port refinement may lead to a specification level that is directly implementable in hardware, as the above example demonstrates. (The *reservation* port is implemented by two hardware *circuits.*)

Once an implementation of a port is described, the question rises whether the implementation is correct. This is discussed in Section V-C. Before, however, we give a more complete example for the refinement of a process specification, using the port implementation given above.

C. An Example of Refinement

The following example serves two purposes: a) to illustrate the use of port refinements as discussed in Section IV-B, and b) to serve as an example for the discussion of verification in Section V.

1) *The abstract specification:* We consider the example of an *arbiter* which assures mutual exclusion to the access of a shared resource between two user processes. As in [31], [8], from where this example is taken, the *arbiter* is associated with two *transfer-device-i* (i = 1, 2) processes which transfer the operational parameters from the two respective users to the resource. The communication between the *arbiter* and the other processes in the system is realized through ports of type *reservation* using the interactions *reserve* and *release.*

As shown in Fig. 5, the *arbiter* communicates with the other processes through the five ports *user*[1], *user*[2], *transfer*[1], *transfer*[2] and *resource.* In respect to the first two ports, it plays the *responding* role, while it plays the *requesting* role in respect to the others. When a user process executes a *reserve*

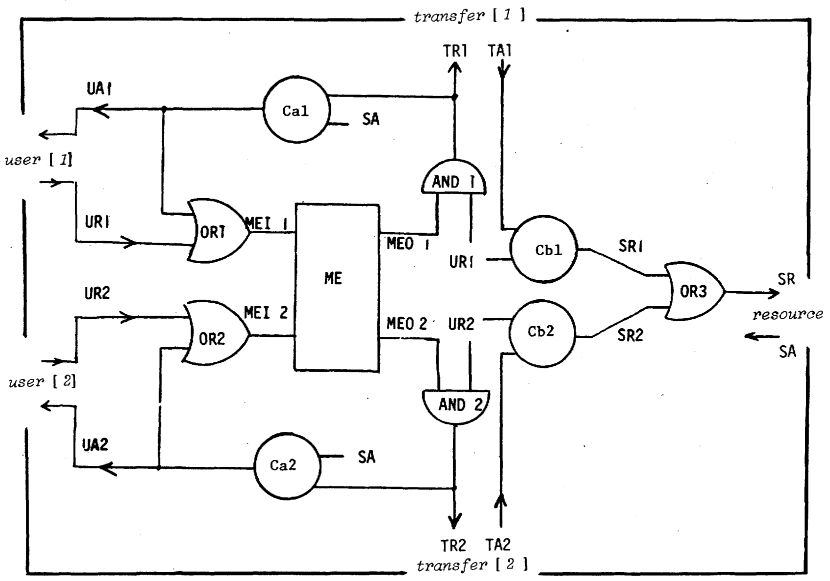


Fig. 6. Internal structure of the *arbiter*.

interaction the arbiter establishes a rendezvous between the user, the resource, and the corresponding transfer process, such that the latter may transfer the operational parameters from the user to the resource (as indicated in the figure by the pointed lines). A similar rendezvous occurs during a *release.*

This informal description may be defined more precisely with our specification language as follows:

```
process type arbiter is
        user (responding),
        transfer (requesting) : array [1 . . 2] of reservation;
        resource (requesting) : reservation;
constraint
for i in 1 . . 2 holds
    ((user[i]/reserve)_j ↔ (transfer[i]/reserve)_j and          (1)
    (user[i]/reserve)_j ↔ (resource/reserve)_j and              (2)
    (transfer[i]/reserve)_j ↔ (resource/reserve)_j and          (3)
        . . . similarly for release . . .);
for i in 1 . . 2 holds
    (i ≠ j implies not
    ((user[i]/reserve)_k ⇢ (user[j]/release)_k'                 (4)
    and (user[j]/reserve)_k' ⇢ (user[i]/release)_k))            (5)
end arbiter.
```

This specification uses the abbreviation "$A \leftrightarrow B$" which is equivalent to "$A \dashrightarrow B$ and $B \dashrightarrow A$" which means that A and B overlap in time. Lines (4) and (5) of the specification define the mutual exclusion property, while lines (1)–(3) define the three-way rendezvous during the *reserve* interaction of a user.

2) *Refined Specification—Implementation:* The implementation of the *arbiter* process considered here uses as subprocesses elementary hardware components. The components used are the combinational circuits *AND* and *OR*, the more complex Muller C-elements (for which the output remains constant until both inputs assume the opposite value), and a primitive mutual exclusion element *ME*. The interested reader may find more details in [31] and [8]. Here we concentrate our discussion on the subprocess and connection structure of the implementation, and the refinement of the *reservation* ports. The structure of the implementation is shown in Fig. 6.

The implementation uses two ports of type *circuit* to realize a *reservation* port of the abstract specification. For example, *UA*1 and *UR*1 implement the *reservation* port *user*[1]. The correspondence between the pairs of *circuits* shown in Fig. 6, and the abstract *reservation* ports is defined by the *four-cycles* implementation discussed in Section IV-B. We note that the circuits *URi*, *TRi* and *SR* carry requests (for the *TRi* and *SR* the *arbiter* plays the *active* role, whereas for the *URi*, it is the user), while the other circuits *UAi*, *TAi* and *SA* carry the corresponding acknowledgments.

More formally, the implementation may be defined as follows:

```
process implementation arb-impl for arbiter is
    OR : array [1 . . 2] of OR-gate;              ⎧ These elementary
    AND : array [1 . . 2] of AND-gate;            ⎪ processes have ports
    Ca, Cb : array [1 . . 2] of C-element;        ⎨ named "input" and
    ME : ME-element;                              ⎪ "output" of type
    OR3 : OR-gate;                                ⎩ circuit.
    internal connection
      for i in 1 . . 2 begin
              MEO[i] is ME . output[i] = AND[i] . input[1];
              MEI[i] is ME . input[i] = OR[i] . output;
              SR[i] is Cb[i] . output = OR3 . input[i]
      end;
    external connection
      for i in 1 . . 2 begin
              user[i] using four-cycle is
              (OR[i] . input[2] = AND[i] . input[2] = Cb[i] . input[2],     (1)
                Ca[i] . output = OR[i] . input[1]);                          (2)
              transfer[i] using four-cycle is
              (Cb[i] . input[1], AND[i] . output = Ca[i] . input[1]);        (3)
              resource using four-cycle is
              (Ca[1] . input[2] = Ca[2] . input[2], OR3 . output)            (4)
      end
  end arb-impl.
```

For clarity purposes, the internal connections are given a name which is also used in Fig. 6. As in the example of Section II-B, the external connections define the ports of the abstract process specification. In this case, however, these ports are implemented by a refinement. Therefore the specification of the external connections define the correspondence between the abstract ports and the ports of the implementation, as well as the implementation model used, i.e., the port implementation *four-cycle.*

As explained in Section III-A, a connection between two ports imposes a synchronized execution of the interactions by the two connected processes. The above example shows how such a synchronization can be extended to more than two ports. For instance, the three-port connection of line (1) between the ports *OR*[1] . *input*[2], *AND*[1] . *input*[2], and *Cb*[1] . *input*[2] imposes a synchrony between these three circuits, and defines in addition the correspondence with the subport *R* of the *four-cycle* implementation of the abstract *reservation* port *user*[1].

V. Verification of Specification Consistency

We discuss in this section the steps that are required to verify that the definition of a process implementation is consistent with the more abstract specification of the process. We first discuss in Section V-B the case of a process implementation with an internal structure of subprocesses, but no port refinement. Port refinements are considered separately in Section V-C. In the first subsection some comments on the underlying semantic model are made. No formal definition of the semantics of process specifications is given. Such a definition would depend on the scope and generality of the language adopted for the specification of process and port constraints. These considerations are beyond the scope of this paper. We think, however, that the following considerations are valid for most semantic models that could be adopted.

A. Comments on the Semantic Model

The following definitions and comments are made in order to clarify the meaning of the specifications of processes and their implementations, and to define the framework in which a semantic model for processes and their interactions can be given which would be the formal basis for proofs of consistency between specifications at different levels of detail.

1) *Interaction type:* The type of interaction defines the possible parameter values that are exchanged over a connection during the occurrence of an interaction of that type.

2) *Interaction occurrence:* An interaction occurrence is defined by the following informations.

- The connection (within the system) where the interaction occurs; this may be an external port of the process considered or an internal connection between the ports of (several) subprocesses of an implementation.
- The time period during which the interaction is executed, (beginning and ending point in real time).
- The interaction value that occurs, i.e., the parameter values exchanged. (We note that these values must adhere to the corresponding interaction type.)

3) *Execution histories:* An execution history is a set of interaction occurrences. (We note that this definition excludes the possibility of including in the same history two identical interaction occurrences, i.e., identical interactions (with same parameter values) during the same time period and on the same connection.) In the case of a model with atomic inter-

actions, the interaction occurrences of an execution history may be put into a (in part arbitrary [21]) order. One obtains a semantic model based on execution sequences.

4) *Safeness properties:* Safeness properties of a process specification or implementation are usually stated by defining conditions that are satisfied by all possible execution histories in which the process can be involved.

5) *Liveness properties:* For the consideration of liveness properties of a process specification or implementations, it seems natural to distinguish between the process that initiates an interaction occurrence over a given connection and its counterpart. This distinction is related to guarded commands [11], [15] that may be part of a process implementation and determine a choice among several possibilities for future behavior. Liveness properties may also be described by the concept of a *process state* which defines the possible future execution histories of the process [26], [29].

6) Sometimes it is necessary to make certain assumptions about the safeness and liveness properties of the environment in which the process evolves [27], [8].

B. Consistency of Subprocess Structures

We consider in this subsection a process specification and an implementation of that specification in terms of several subprocesses without port refinements, i.e., each (external) port of the process is a port of one of the subprocesses.

A process type specification defines the following properties of the execution histories in which a process of that type can be involved. Such a process may be part of a large system.

1) *Interaction type constraints:* The values of each interaction occurrence in a history correspond to the interaction type associated with the external connection where the interaction occurs.

2) *Port constraints (one for each port of the process):* The subhistory of those interaction occurrences (of a history) that occur at a given external connection of the process satisfies the constraints explicitly specified in the port type definition corresponding to that connection. (Such subhistories are special cases of "projections" as considered in [25], [20].)

3) *Global constraint:* A history satisfies the constraints explicitly defined in the process specification.

The definition of a process implementation, similarly, implies the following properties of the execution histories possible for this implementation:

4) *Subprocess constraints (one for each subprocess of the implementation):* The subhistory of those interaction occurrences of a history that occur over (external and/or internal) connections to which the subprocess is connected satisfies the properties 1)–3) of that process. (We note that a suitable port/connection name substitution must be made corresponding to the connection structure of the implementation before the property 3) of the subprocess can be applied to the execution history of the implementation.)

To show the consistency of a process implementation with the specification of the process, we have to prove that properties 1)–3) of the process specification can be implied from the subprocess constraints 4) of the implementation. Since there are no port refinements, and the type compatibility of connected ports is compile-time checked, the properties 1) and 2) follow directly from the corresponding properties of the subprocesses. The property 3) is in general more difficult to prove.

For the example of the *multiserver* process given in Section II-A, the consistency proof is outlined as follows. The global constraint 3) of the *multiserver* specification (to be proven) has the form:

$$\textit{for each (external) connection } \text{users}[u] \ \underline{\text{holds}}$$
$$\forall i \ (\text{users}[u]/\text{response})_i.\ Y =$$
$$\text{FUNCTION OF } ((\text{users}[u]/\text{request})_i.X).$$

This property can be implied from the global constraints of the subprocesses *server* and *multiplexer* implementing the *multiserver* (see Section II-B); these constraints are rewritten as follows with the appropriate renaming of the port/connection names used in the implementation:

$$(S.\ \text{user}/\text{response})_i.\ Y = \text{FUNCTION OF } ((S\ .\ \text{user}/\text{request})_i.\ X)$$
$$\textit{and for each connection } \text{users}[u] \ \underline{\text{holds}}$$
$$\forall i\ \exists i' \ ((\text{users}[u]/\text{request})_i = (S\ .\ \text{user}/\text{request})_{i'}$$
$$\underline{\text{and}} \ (\text{users}[u]/\text{response})_i = (S\ .\ \text{user}/\text{response})_{i'})$$

This example is particularly simple. In more typical cases, the proof involves induction and invariant properties that hold throughout the execution of the system. The proofs of the input/output properties of a procedure, based on its implementation in an algorithmic language, is the corresponding consistency problem for sequential programs. Related work can be found in [2], [27], [10]. A constructive approach for deriving the specification of a subprocess from the specification of the process and the other subprocesses is explored in [25].

C. Consistency and Port Refinements

We consider in this subsection in addition to the problem of Section V-B that the process implementation uses one or several port implementations, as explained in Section IV-B.

1) *Consistency of port implementations:* A port type specification defines the following properties of the port histories of the interactions occurring over the port (which are subhistories of global system execution histories with appropriate renaming).

1′) *Operation type constraint:* The values of each interaction occurrence in a history correspond to the interaction type specified.

2′) *Port constraint:* A history satisfies the explicitly specified constraints.

A port implementation defines, similarly as a process type specification, three properties 1′)–3′) for a port implementation history (i.e., an execution history of the interactions occurring over the subports of the implementation) where the port constraints [property 2′)] are those of the declared subports of the implementation, and the global constraint [property 3′)] is the constraint explicitly defined for the implementation.

In addition, a port implementation defines a *correspondence* which is, in the more general case, a mapping (possibly one to many) from occurrences of port interactions to (finite) port implementation subhistories. We assume that the inverse of this mapping is not ambiguous, by which we mean that for each port implementation subhistory, it can be determined whether it is the implementation (i.e., image of the correspondence mapping) of a port interaction occurrence or not; and that it is the implementation of at most one interaction occurrence.

The correspondence mapping can be naturally extended to a mapping from port histories (sets of port interaction occurrences) to port implementation subhistories, where the resulting history is the set union of the subhistories corresponding to the different operation occurrence in the port history.

A port implementation should satisfy the following consistency conditions:

i) *Inverse functionality of correspondence:* The extended correspondence mapping should have an inverse function over the set of those implementation histories that satisfy the conditions 1′)–3′) of the implementation.

ii) *Realization of port constraints:* The image, under the inverse correspondence mapping, of an implementation history satisfying 1′)–3′) should satisfy the port constraints (property 2′) of the port type specification.

As an example, we consider the *four-cycle* implementation of the *reservation* port type discussed in Section IV-B, the global port implementation constraint (property 3):

$$(R/\text{up})_i \dashrightarrow (/)_i \dashrightarrow (R/\text{down})_i \dashrightarrow (A/\text{down})_i \dashrightarrow (R/\text{up})_{i+1}$$

and the given correspondence rule imply that the ith *reservation* interaction occurrence of a port history corresponds to the ith sequence of (R/up) --⟩ (A/up) in the corresponding implementation history; and the ith *release* to the ith sequence of $(R/down)$ --⟩ $(A/down)$. Therefore the inverse functionality of the correspondence mapping is guaranteed. This also shows that

$$(/\text{reserve})_i \dashrightarrow (/\text{release})_i$$

holds; and applying the global constraint one more for $i + 1$ one obtains

$$(/\text{release})_i \dashrightarrow (/\text{reserve})_{i+1} \dashrightarrow (/\text{release})_{i+1}$$

which implies the constraints of the port type *reservation* which is implemented.

2) *Consistency of Process Implementations:* We now consider the consistency conditions that should be satisfied by a process implementation using port implementations. As in the case without port refinements discussed in Section V-B, the general consistency condition is that the properties 1)–3) of the process specification can be implied from the subprocess constraints (property 4) of the implementation. Considering the structure of the implementation, this consistency condition can be implied from the following conditions:

a) For each port not using any refinement: properties 1′) and 2′) (see Section V-C-1) are satisfied.

b) For each port using an implementation:

b-1) For each subport of the implementation: properties 1′) and 2′) (see Section V-C-1) are satisfied.

b-2) The global port implementation constraint (property 3′) can be implied from the subprocess constraints.

c) The global process constraint (property 3, see Section V-B) can be shown to be satisfied by all (abstracted) execution histories obtained by the application of the inverse correspondence functions of the port implementations, from the implementation execution histories satisfying the subprocess constraints.

We note that the conditions b-1) and b-2) for a given refined port together with the consistency conditions of the port implementation (see Section V-C-1) imply that the properties 1′) and 2′) are satisfied for the given port. Therefore the conditions a)–c), together with the consistency conditions 1) and 2) for the port implementations, imply the general consistency condition mentioned above.

The verification of the conditions a) and b-1) is made by the port type compatibility checking at compile-time. The verification of the conditions b-2) and c) is in general more difficult and must be compared with the verification of property 3) in the absence of port implementations (see Section V-B). Often the verification involves the establishment of invariant properties that are needed for deriving both conditions b-2) and c).

As an example we consider the arbiter specification and implementation discussed in Section IV-C. A detailed verification of the consistency conditions is given in [8]. We limit the discussion here to the major points of the verification. It is important to note that the verification of the port implementation constraint [condition b-2)] involves several subprocesses of the implementation. As Fig. 6 shows, the constraints that the *UA*1 circuit of the *user*[1] port, for example, is not set up before the *UR*1 circuit of the same port goes up, involves the propagation of the *UR*1 circuit signal through the subprocesses *OR*1, *ME*, *AND*1 and *Ca*1; and also relies on the assumption that the same port implementation constraint is satisfied by the processes in the environment of the arbiter connected to the ports *transfer*[1] and *resource.* As a consequence condition b-2) can only be verified by considering the interplay between the different parts of the process implementation and its environment, which is formalized in [8] by an invariant and a reachability analysis based on the properties of the subprocesses, their interconnection structure and assumptions on the environment. From this invariant and reachability analysis, it is then straightforward to derive the constraints of the port implementations. The same invariant is also the basis for deriving the global constraints (especially the mutual exclusion) of the arbiter specification [condition c)]. We note, however, that the global constraints of the arbiter specification in [8] are not given in terms of an abstracted *reservation* port specification, but in terms of the subport operations of the implementation. This simplifies the proof since it is not necessary to consider the inverse correspondence function to make the abstraction from the detailed interactions of the implementation subports to the more abstract interactions of the *reservation* port used in the arbiter specification of Section IV-C. With the correspondence discussed in Section V-C-1, it is however not difficult to derive the global constraints of the arbiter specification given in Section IV-C from the global properties of the arbiter specification given in [8].

VI. Conclusions

A method for specifying communicating systems is discussed and demonstrated by several examples. The method is based on the concepts of communicating processes and process interactions that are associated with communication ports. The step-wise refinement of a process specification is considered by the definition of an internal structure of the process, consisting of subprocesses and connections. Internal connections identify the ports of the subprocesses, and external connections define the ports of the abstract process specification in terms of some internal ports of the subprocesses.

The described specification method combines this process refinement approach with a method for the refinement of ports and their interactions, which is analogous to the refinement of sequential software modules, where an operation at the higher level of abstraction is implemented in general by several operations at the more detailed level. This port refinement method is important for considerations of subsystem interfaces, their abstract specification and implementation.

The stepwise refinement of process specifications as discussed in this paper is similar to some approaches described in the literature [10], [27], [18]. These approaches do not foresee the refinement of interactions and port specifications. Such refinement is considered in the literature on stepwise refinement of abstract datatype specifications [34], [14]. These references, however, consider the refinement of the operations defined for a given module in the context of sequential processes or shared resources where the operations of the module are called upon by the active agents of the system. The approach to port refinement, as discussed in this paper, may be considered a generalization of the refinement of abstract data types to the context of distributed systems.

The structured specification method described in this paper seems applicable to the areas of software as well as hardware design. It addresses in particular the question of system structure and the communication between the different system parts, which is an important aspect in the design of distributed systems. It is to be noted that the described method only considers static implementation structures. This seems to be suitable for description at the hardware (or physical subsystem) level, and for many software applications in systems for communication and realtime control.

While this paper concentrates on the *structure* of communicating systems, it is clear that appropriate specification techniques must be available for defining the *constraints* that determine the possible interactions and their order of execution for the specified system. For the examples of this paper we have used certain specification language elements which have been used on different occasions, such as temporal ordering of interactions, constraints on parameter values expressed by assertions, etc. Other specification languages, involving for instance the notion of process states [19], [12], could also be used. A complete specification method could therefore be based on the process and port structure discussed in this paper and a semantic model which provides the framework for formally defining the execution histories considered in Section V. This semantic model would also be the basis for the specification language by which process and port constraints can be expressed. However, such considerations go beyond the scope of this paper.

References

[1] S. Andler, "Predicate path expression," in *Proc. 6th Ass. Comput. Mach. Conf. POPL*, Jan. 1979, pp. 226-236.

[2] K. R. Apt, N. Francez, and W. DeRoever, "A proof system for communicating sequential processes," *ACM Toplas*, vol. 2-3, pp. 359-384, July 1980.

[3] R. M. Balzer, "PORTS: A method for dynamic interprogram communication and job control," *AFIPS Conf.*, vol. 38, SJCC, pp. 485-489, 1971.

[4] W. Bartussek and D. L. Parnas, "Using traces to write abstract specifications for software module," in *Proc. ECI Conf.*, Venice, Italy, 1978.

[5] A. Bernstein, "Output guards and non-determinacy in CSP," *ACM Toplas*, vol. 2, pp. 234-238, Apr. 1980.

[6] G. V. Bochmann and C. A. Sunshine, "Formal methods in communication protocol design," *IEEE Trans. Commun.*, vol. COM-28, pp. 624-631, Apr. 1980.

[7] G. V. Bochmann, "High-level modular hardware design and interfaces," Département d'I.R.O., Université de Montréal, P.Q., Canada, Pub. 393, Dec. 1980.

[8] G. V. Bochmann, "Hardware specification with temporal logic: an example," *IEEE Trans. Comput.*, vol. C-31, pp. 223-231, Mar. 1982.

[9] N. Buckle, "Restricted datatypes: specification and enforcement of invariant properties of variables," in *Proc. Ass. Comput. Mach., Conf. Languages, Reliable Software, Sigplan Notices*, vol. 12, Mar. 1977, pp. 68-76.

[10] Z. C. Chen and C. A. R. Hoare, "Partial correctness of communicating sequential processes," in *Proc. 2nd Int. Conf. Distribut. Comput. Syst.*, Paris, France, Apr. 1981, pp. 1-12.

[11] E. W. Dijkstra, "Guarded commands, non-determinacy and formal derivation of programs," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 354-357, Aug. 1975.

[12] ISO TC97/SC16/WG1 ad hoc group on FDT, "A formal description technique based on an extended state transition model," 1982.

[13] I. Greif, "A language for formal problem specification," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 931-935, Dec. 1977.

[14] J. Guttag, "Abstract data types and the development of data structures," *Commun. Ass. Comput. Mach.*, vol. 28, pp. 396-404, June 1977.

[15] C. A. R. Hoare, "Communicating sequential processes," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 666-677, Aug. 1978.

[16] J. D. Ichbiah *et al.*, "Rationale for the design of the ADA programming language," *Sigplan Notices*, vol. 14, June 1979.

[17] A. K. Jones and B. Liskov, "A language extension for controlling access to shared data," *IEEE Trans. Software Eng.*, vol. SE-2, Dec. 1976.

[18] X. Jorrand, "Bases for the specification of communicating processes," in *Proc. 1981 ICS Conf. Syst. Arch., ACM Europe*, London, England, pp. 124-133.

[19] R. M. Keller, "Formal Verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 371-384, July 1976.

[20] S. S. Lam and A. V. Shankar, "Protocol projections: A method of analyzing communication protocols," in *Proc. Nat. Telecommun. Conf.*,

New Orleans, LA, Dec. 1981, pp. E3.2.1.-E3.2.8.

[21] L. Lamport, "Time, clocks and the ordering of events in a distributed systems," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558-564, July 1978.

[22] L. Lamport, "A new approach to proving the correctness of multiprocess programs," *ACM Toplas*, vol. 1, pp. 84-97, July 1979.

[23] P. Le Guernic and M. Raynal, "Eléments d'un langage adapté à la communication entre processus," *Actes du Congrès AFCET*, Nancy, France, Nov. 1980, pp. 667-676.

[24] B. Liskov and S. Zilles, "Programming with abstract data types," in *Proc. Ass. Comput. Mach. SIGPLAN Conf. VHLL, Sigplan Notices*, vol. 9, 1974, pp. 52-59.

[25] P. Merlin and G. V. Bochmann, "On the construction of submodule specifications and communication protocols," *ACM Toplas*, vol. 5, Jan. 1983.

[26] G. Milne and R. Milner, "Concurrent processes and their syntax," *J. Ass. Comput. Mach.*, vol. 26, pp. 302-321, Apr. 1979.

[27] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 417-427, July 1981.

[28] S. S. Owicki, "Specification and verification of a network mail system," in *Program Construction*, F. L. Bauer and M. Broy, Eds. New York: Springer-Verlag, 1979.

[29] S. S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," submitted for publication.

[30] M. Raynal, "Contribution à l'étude de la coopération entre processus dans les langages et les systémes informatiques," Université de Rennes, France, Thèse d'Etat, Apr. 1981.

[31] C. L. Seitz, "Ideas about arbiters," *LAMBDA 1st Quarter 1980*, pp. 10-14.

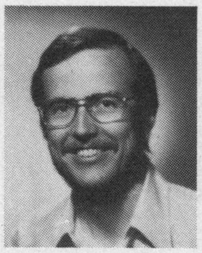
[32] A. C. Shaw, "Software descriptions with flow expressions," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 242-284, May 1978.

[33] A. Silberschatz, "Port directed communication," *Computer J.*, vol. 24, pp. 78-83, Feb. 1981.

[34] W. A. Wulf, M. Shaw, and R. L. London, "An introduction of the construction and verification of alphard programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 253-264, Dec. 1976.

Michel Raynal was born on January 16, 1949, in Cahors, France. He received the Doctorat es Sciences degree in computer science from Rennes University, France, in 1981.

From 1973 to 1975, he worked with the CNRS at the Irisa Laboratories in Rennes, on a project concerned with operating systems. From 1975 to 1981, he worked at IRISA-INRIA, where his research covered languages design to express concepts traditionally associated with operating systems. After expression and implementation of the concept of abstract data-type, he dealt with the design and writing of programs composed of cooperating processes. His doctoral thesis was obtained in the latter topic. At present, he is Head of the Computer Science Department at a Telecommunications Engineering Faculty, the ENST Br., Brest, France. His research interests include programming languages, operating systems, parallel processing and distributed applications.



Gregor V. Bochmann received the Diploma in physics from the University of Munich, Munich, Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

He has worked in the areas of programming languages and compiler design, communication protocols, and software engineering. Currently, he is Professor in the Departement d'Informatique et de Recherche Operationnelle, Université de Montreal Canada. His present work is aimed at design methods for communication protocols and distributed systems. From 1977 to 1978 he was a Visiting Professor at the Ecole Polytechnique Fédérale, Lausanne, Switzerland. From 1979 to 1980 he was a Visiting Professor in the Computer Systems Laboratory, Stanford University, Stanford, CA.

A Distributed Channel-Access Protocol for Fully-Connected Networks with Mobile Nodes

YARON I. GOLD, MEMBER, IEEE, WILLIAM R. FRANTA, MEMBER, IEEE, AND SHLOMO MORAN

***Abstract*—We present a theoretical description and analysis of a collision-free channel-access protocol for a shared channel with "mobile" nodes that are all within range and in line-of-sight of each other, in arbitrarily changing spatial (one-, two- or three-dimensional) configurations.**

The protocol provides distributed access-control under Fixed priority, Fair Round-Robin and Prioritized Round-Robin priority disciplines. The protocol employs information on the changing node configuration to dynamically tune its scheduling-function. This information is efficiently obtained by repeated, self-induced updates that are executed in a distributed manner by all nodes. The protocol's theoretic performance characteristics (throughput and delay) are at least as good as, and most often better than those associated with protocols that employ less information.

***Index Terms*—Adaptive, broadcast, collision-free, communication, distributed, local networks, mobile networks, multiple access, performance analysis, protocols.**

Manuscript received December 29, 1981; revised June 25, 1982.

Y. I. Gold is with the Department of Electrical Engineering and Computer Science, University of Connecticut, Storrs, CT 06268.

W. R. Franta is with the Department of Computer Science, University of Minnesota, Minneapolis, MN, 55455.

S. Moran is with the Department of Computer Science, Technion, Israel Institute of Technology, Haifa, Israel.

I. Introduction

MULTIACCESS protocols are schemes that multiplex node access to a shared communication channel.