Specification languages for communication protocols

Gregor v. Bochmann

Département d'informatique et de recherche opérationnelle,
Université de Montréal, CP 6128, Succ. A, Montréal, Québec, Canada

Abstract

Like software and hardware engineering, protocol engineering involves all the phases of the development of specifications and implementations of communication protocols. Like in the context of software and hardware, formal specification languages allow the automation of certain design, validation and implementation activities during these development phases. This paper gives an overview of protocol engineering and provides an short introduction to several languages that have been developed for the specification of communication protocols and services. Based on a simple example protocol which is specified in VHDL, Estelle, LOTOS and SDL, the characteristics of these different languages are discussed. The conclusions point to the similarities that exist among the languages and tools for system development related to hardware, software and protocols.

1. Introduction

The orderly introduction of new communication protocols, for proprietary systems or Open Systems Interconnection (OSI) [Larm 88], requires a careful analysis of the proposed protocols and services, and much effort for the development and testing of protocol implementations. Much research effort has gone into improving the working methods for these activities. In this context, the use of formal specification languages for the specification of communication protocols and services has received much attention, since such languages allow a more systematic approach for protocol validation, implementation and testing, as compared to the traditional use of protocol specifications given in natural language.

Similar concerns for the correctness of the proposed specifications and implementations occur in the areas of hardware and software development. Especially for hardware design, it is essential that the circuit layout of hardware components be designed correctly, since they cannot be changed after the manufacturing process. Different specification languages have been proposed for hardware design at the different levels of abstractions. The VHDL [VHDL 87] has been designed to cover several of these levels, and in particular the more abstract levels of hardware design which are in many respect similar in nature to software specifications.

The purpose of this paper is to give an overview of specification languages which are used during the protocol design and implementation process, and to show the similarities and differences that exist between these languages and VHDL. The paper also relates to the development process for communication protocols and the tools that can be used for the different activities of this process. The concerns of the development process of communication protocols are similar to those for the development of hardware and software.

The paper is structured as follows. Section 2 gives an overview of the protocol development process and presents several standardized formal description techniques and other languages that have been developed for the specification of communication protocols and services. In Section 3, a very simple protocol entity is presented as an example and described in several languages, including VHDL. Certain similarities and differences between these languages are discussed. Specific issues, such as inter-module communication and addressing, as well as support tools for validation, automatic implementation (synthesis), and testing are discussed in Section 5. The conclusions point to the similarities that exist among the languages and tools for system development related to hardware, software and protocols.

2. Specification techniques for communication protocols

2.1. Protocol engineering: An overview

Communication protocols are the rules that govern the communication between the different components within a distributed computer system. In order to organize the complexity of these rules, they are usually partitioned into a hierarchical structure of protocol layers, as exemplified by the seven layers of the standardized OSI Reference Model.

As they develop, protocols must be described for many purposes. Early descriptions provide a reference for cooperation among designers of different parts of a protocol system. The design must be checked for logical correctness. Then the protocol must be implemented, and if the protocol is in wide use, many different implementations may have to be checked for compliance with a standard. Although narrative descriptions and informal walk-throughs are invaluable elements of this process, painful experience has shown that by themselves they are inadequate.

The informal techniques traditionally used to design and implement communication protocols have been largely successful, but have also yielded a disturbing number of errors or unexpected and undesirable behavior in most protocols. The use of a specification written in natural language gives the illusion of being easily understood, but leads to lengthy and informal specifications which often contain ambiguities and are difficult to check for completeness and correctness. The arguments for the use of formal specification methods in the general context of software engineering [Somm 89] apply also to protocols.

The following activities can be identified within the protocol engineering process. They can be partially automated if a formal protocol specification is used [Boch 90g].

(a) Protocol design: The protocol specification is developed based on the communication service to be provided by the protocol. The protocol also depends on the underlying (existing) communication service; e.g. the protocol may have to recover from transmission errors or lost messages if the underlying service is unreliable. The design process is largely based on intuition.

(b) Protocol design validation: The protocol specification must be checked (1) for logical consistency, (2) to provide the requested communication service, and (3) to provide it with acceptable efficiency.

(c) Implementation development: The protocol implementation must satisfy the rules of the protocol specification; the implementation environment and the user requirements provide additional constraints to be satisfied by the implementation. The implementation may be realized in hardware or software.

(d) Conformance testing and implementation assessment: The purpose of conformance testing is to check that a protocol implementation conforms to the protocol specification, that is, that it satisfies all rules defined by the specification. This activity is especially important for interworking between independently developed implementations, as in the case of OSI standards. The testing of an implementation involves three sub-activities: (1) the selection of appropriate test cases, (2) the execution of the test cases on the implementation under test, and (3) the analysis of the results obtained during test execution. The sub-activities (1) and (3) use the protocol specification as a reference.

2.2. Formal description techniques (FDT's)

Many different formal description techniques have been proposed for the protocol engineering cycle, including finite state machines (FSM), Petri nets, formal grammars, high-level programming languages, process algebras, abstract data types, and temporal logic. The simpler models, such as FSM, Petri nets and formal grammars, were often extended by the addition of data parameters and attributes in order to naturally deal with certain properties of the protocols, such as sequence numbering and addressing [Boch 89g].

With the beginning work on the standardization for Open Systems Interconnection (OSI), special working groups on "Formal Description Techniques" (FDT) were established within ISO and CCITT in the early eighties with the purpose of studying the possibility of using formal specifications for the definition of the OSI protocols and services. Their work led to the proposal of three languages, Estelle, LOTOS and SDL, which are further discussed below (for a tutorial introduction and further references, see [Budk 87], [Bolo 87] and [Beli 89], respectively). These languages are called formal description techniques, since care has been taken to define not only a formal syntax for the language, but also a formal semantics which defines the meaning, in a formal manner, of any valid specification. This is in contrast to most programming languages which have a formally defined syntax (for instance in BNF), but an informally defined

semantics. The formal semantics is essential for the construction of tools which are helpful for the validation of specifications or the development of implementations.

While SDL had been developed within CCITT since the seventies for the description of switching systems, Estelle and LOTOS were developed within ISO for the specification of communication protocols and services. However, all these languages have potentially a much broader scope of applications. However, their effective use in the OSI area, so far, has been relatively slow. This may be partly explained by the competition between these three languages, which each have certain advantages, and by the difficulty many people have in learning a new language.

In Estelle, a specification module is modelled by an extended FSM. The extensions are related to interaction parameters and additional state variables, and involve type definitions, expressions and statements of the Pascal programming language. In addition, certain "Estelle statements" cover aspects related to the creation of the overall system structure consisting in general of a hierarchy of module instances. Communication between modules takes place through the interaction points of the modules which have been interconnected by the parent module. Communication is asynchronous, that is, an output message is stored in an input queue of the receving module before it is processed.

SDL, which has the longest history, is also based on an extended FSM model. For the data extensions, it uses the concept of abstract data types with the addition of a notation of program variables and data structures, similar to what is included in Estelle. However, the notation is not related to Pascal but to CHILL, the programming language recommended by CCITT. The communication is asynchronous and the destination process of an output message can be identified by various means, including process identifiers or the names of channels or routes. Recently, the language has been extended to include certain features for object-oriented specifications [SDL 92].

LOTOS is based on an algebraic calculus for communicating systems (CCS [Miln 80])which includes the concept of finite state machines plus parallel processes which communicate through a rendezvous mechanism which allows the specification of rendezvous between two or more processes. Asynchronous communication can be modelled by introducing queues explicitely as data types. The interactions are associated with gates which can be passed as parameters to other processes participating in the inteactions. These gates play a role similar to the interaction points in Estelle. The data aspects are covered by an algebraic notation for abstract data types, called ACT ONE [Ehri 85], which is quite powerful, but would benefit from the introduction of certain abbreviated notations (see for instance [Boch 90a]) for the description of common data structures.

In contrast to the other FDT's, SDL was developed, right from the beginning, with an orientation towards a graphical representation. The language includes graphical elements for the FSM aspects of a process and the overall structure of a specification. The data aspects are only represented in the usual linear, program-like form. In addition, a completely program-like form is also defined, called SDL-

PR, which is mainly used for the exchange of specifications between different SDL support systems. A graphical representation for LOTOS has also been defined.

2.3. Other specification languages

The above discussion is limited to the standardized formal description techniques developed within ISO and CCITT. There are a number of other important specification methods and associated tools that have been used for protocol engineering. Many tools are based on Petri nets and their extensions. Several kinds of extended FSM languages have been used for large scale applications [Schu 80], [Holz 91], [Agga 83b]. The state of the art in protocol engineering is yearly discussed in the IFIP conferences FORTE [Diaz 92] and PSTV [Linn 92]. The specific issues related to protocol testing and verification is also discussed in the conferences IWPTS [Boch 93] and CAV [Boch 93v], respectively.

In addition to the formal description techniques discussed above, the OSI standardization committees also use certain semi-formal languages (which have no formally defined semantics). In particular, a language called TTCN [Sari 92a] is used for describing conformance test cases, and the ASN.1 notation is used for describing the data structures of the protocol data units (messages) exchanged by the OSI application layer protocols [Neuf 92a]. This notation is associated with a coding scheme which defines the format in which these data units are exchanged over the communication medium. All the other languages mentioned above do not address this problem.

In the context of application layer protocols, in particular for Open Distributed Processing and distributed systems management, certain forms of object-oriented specifications are being used which are based on extensions of ASN.1. Also the formal specification language Z [Spiv 92] has been proposed, which is based on set theory and predicate calculus, and was originally developed for software specifications.

3. Comparison of specification languages

A comparison of the different specification languages shows that certain specification concepts can be found in most languages, although different syntactic constructs may be used for realizing them. We discuss in the following certain specification concepts and how they may be realized within the languages Estelle, LOTOS and SDL, as well as in VHDL. This comparison will be based on a simple example protocol module. Since the audience of this conference is most familiar with VHDL, the example will first be introduced informally and specified in VHDL. Then sketches of corresponding specifications in Estelle, SDL and LOTOS will be presented, and the representation of several general concepts in these different specifications will be discussed.

3.1. A simple protocol entity

Communication protocols are introduced to obtain a more sophisticated communication service over a given, more basic, so-called "underlying" communication service. This leads to a system structure as shown in Figure 3.1. A protocol entity communicates with another protocol entity through the underlying service. It has two interfaces: the one with the underlying service, which is often called the "lower" interface, and the so-called "upper" interface through which the more sophisticated communication service is provided to the user.

Figure 3.1: Communication system architectures with two protocol entities

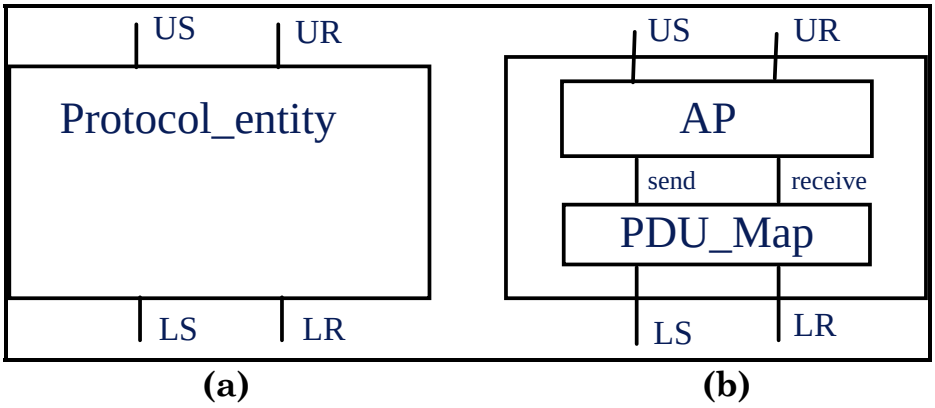

Figure 3.2: Two structures of a protocol entity, *simple_structure* (a) and *composed_structure* (b)

We consider in the following a very much simplified version of the OSI Transport protocol [Larm 88] (formal specifications of a more complete version of this protocol may be found in [Boch 90a]). This protocol has the three phases of connection establishment, data transfer and disconnection. The protocol specification defines the possible interactions at the upper and lower service interfaces, the allowed sequences of interaction and the corresponding parameter values which may occur during these interaction sequences. The interactions at the upper service interface allow the user to request a new connection (Creq), to accept the confirmation of a connection (Cconf), accept a disconnection indication (Xind) or to send or receive data (Dreq or Dind, respectively). The Creq interaction has a parameter which represents the destination address of the requested connection. Also the Dreq and Dind interactions have a parameter which represents the data being transmitted. Figure 3.2(a) shows a diagram representing a protocol entity and its upper and lower interfaces. The upper interface is partitioned into two parts, one leading interactions from the user to the entity (called "US", for Upper Send) and one leading interactions from the entity to the user (called "UR", for Upper Receive). Similarly, the lower interface has the two components LS and LR.

The interactions at the lower service interface are of similar nature. However, for our example, we assume that the lower service is always in the data transfer phase. The protocol messages, also called protocol data units (PDU's) that are exchanged between the two protocol entities are sent in the form of coded data packets through the underlying communication service. A PDU called Connect Request (CR) represents a request for a new connection, a Connect Confirm is returned in order to confirm a requested connection, while a Disconnect Request (XR) requests a disconnection. User data is sent in so-called Data PDU's. Often the coding and decoding of these PDU's is confined to a specific component, as shown in Figure 3.2(b) where this component is called PDU_Map.

Figure 3.3 shows a state diagram which defines the order of interaction of a protocol entity during a connection establishment phase for the case that the connection is initiated by its local user (dark transitions) and for the case that the connection is initiated at the other side (pointed transitions).

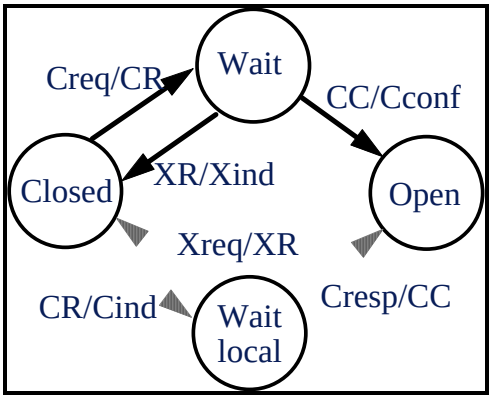

Figure 3.3: State diagram showing behavior of protocol entity

3.2. Protocol specification in VHDL and other languages

A specification in VHDL of the protocol entity described in Section 3.1 is shown in Figure 3.4(a). The figure contains a VHDL entity definition which corresponds to what is shown in Figure 3.2. The behavior of the protocol entity could be defined in different specification styles. If a decomposition into two components, as shown in Figure 3.2(b) is desired, the VHDL architecture definition called *composed_structure* could be adopted (note that the behaviors of the components *Map_behavior* and *AP_behavior* are not defined here). In the case that a monolitic specification, as shown in Figure 3.2(a) is desired, the VHDL architecture (and behavior) called *simple_structure* could be adopted.

A similar specification of the protocol entity in Estelle is shown in Figure 3.4(b). One main difference is the fact that the interactions between different modules in Estelle (and in SDL) is through message passing and unlimited input queues. An interface is modelled in Estelle by a so-called channel which allows for

```
type U_serv_pr is (Creq, Cconf, Xind, Dreq, Dind);
type options_type is (high, low);
...
U_interf is record  -- upper service interface
          primitive : U_serv_pr;
          opt : options_type;
          data : data_type
          ...  end record;
type L_interf is ...

entity protocol_entity  is  -- protocol entity
port  (    US: in U_interf;   LR: in L_interf;
                    UR: out U_interf;  LS: out
                    L_interf );
end protocol;


architecture simple_structure of protocol_entity  is
          -- behavior in terms of an FSM
begin
type State_type is (Closed, Waitx, Openx, ... );
signal State : State_type := Closed;
signal options : options_type;
process begin
     wait on LR'transaction, US'transaction;
case State is
  when Closed =>
     if US'active and US#primitive = Creq then
          State <= Waitx; options <= US#opt;
          LS <= (primitive => Dreq,
              data => encode_CR(US#dest, US#opt);
     else ... end if;
  when Waitx => ...
  end case;
end process;
end simple_structure;

architecture composed_structure of protocol_entity
    is      -- composition of two components
type PDU's is record ...
signal send, receive : PDUs;
component PDU_Map
          port  (    LS: in L_interf;
                     LR: out L_interf;
                     PDU_S: out PDUs;
                     PDU_R: in PDUs  );
          end component;
component AP ...

for all : PDU_Map use entity PDU_Map
                              (Map_behavior);
for all : AP use entity AP (AP_behavior);
begin
m: PDU_Map port map (LS, LR, send, receive);
ap: AP port map (US, UR, send, receive);
end composed_structure;
```

Figure 3.4 (a): Example in VHDL

```
type options_type is (high, low);
...
channel U_interf (user, provider) =
by user:  Creq (opt : options_type; dest : ...);
          Dreq (data: data_type);
           ...
by  provider: Cconf (opt : options_type);
              Dind;
               ...
end;

channel L_interf (user, provider) = ...



module protocol_entity systemactivity;
  ip      U: U_interf (provider);      L: L_interf
(user);
end;

body simple_structure for protocol_entity;
var       opt : options_type;
          State Closed, Wait, Open, ...;
trans  -- a transition
          from Closed        when US.Creq  -- input
          to Wait  -- new state
          begin
             options := opt;
             output LS.Dreq (encode_CR(dest, opt))
          end
trans ...
end;


body composed_structure for protocol_entity;
channel PDUs (A, B) =
   by A, B:    CR (opt: options_type; dest : ...);
               CC (...);
                ...
   end;
module PDU_Map systemactivity;
          ip         L_service: L_interf (user);
                     send_receive: PDUs (provider);
          end;
module AP systemactivity; ...
modvar  m: PDU_Map;
          ap: AP
initialize begin
          init m with Map_behavior;
          init ap with AP_behavior;
          attach m.L_service to L;
          attach ap.U_service to U;
```

```
        connect m.send_receive to
ap.send_receive;
end;
```

Figure 3.4 (b): Example in Estelle

Figure 3.4 (c): Example in SDL

```
        type options_type is
      * abbreviated notation *)
        EitherOf ligh, low endtype;
type Creq is  (* abbreviated notation *)
        Tuple make_Creq     comp
   opt : options_type,
   dest : ...    endtype
type Cconf is ...
process protocol_entity
        [US, LS, UR, LR]: noexit :=
   (* this is the simple_structure version *)
   Closed [US, LS, UR, LR ]   -- initial state
where
process Closed [US, LS, UR, LR ] : noexit :=
     US ? x: Creq;
     LS ! Dreq (encode_CR (dest(x), opt(x));
     Wait [US, LS, UR, LR ] (opt(x))
   []  ... (* error cases *)
endproc
```

```
process Wait
        [US, LS, UR, LR ] (o : options_type)
         ...
         endproc
process Open ...
         endproc

process protocol_entity
                [US, LS, UR, LR]: noexit :=
  (* this is the composed_structure version *)
hide PDU_S, PDU_R in
PDU_Map [LS, LR, PDU_S, PDU_R ]
     |[PDU_S, PDU_R]|
           AP [US, UR, PDU_S, PDU_R]
where
   process PDU_Map
      [LS, LR, PDU_S, PDU_R ] : noexit := ...
   process AP
     [US, UR, PDU_S, PDU_R]  : noexit := ...
endproc
```

Figure 3.4 (d): Example in LOTOS

message transmission in both directions; the type of messages transmitted in each direction are declared in an Estelle channel definition. For instance, a single interaction point, called *U*, with an associated channel type *U_interf* represents the upper service interface of the protocol entity, and this entity plays the role of the *provider*, as indicated in the declaration of the interaction point.

A corresponding specification in SDL is shown in Figure 3.4(c). It is very similar to the one in Estelle, although a graphic representation provided by SDL has a very different "look". We note that the textual content of the graphical symbols in SDL are often written in an informal language, such as english. This is the case for the text "Dreq with CR" in the output symbol in the transition shown in the protocol behavior. The other textual information in the example have a formal meaning defined by the language.

Finally, Figure 3.4(d) shows a corresponding specification written in LOTOS. In the case of LOTOS, there is no syntactic separation between the definition of the interface of a process and its behavior, as in Estelle and VHDL. Therefore two separate specifications of the protocol entity are given, one with a behavior corresponding to the simple structure, and one with a decomposition into the sub-processes PDU_Map and AP, as shown in Figure 3.2(b).

3.3. Modules and step-wise refinement

The modular structure of specifications and the possibilities for multiple versions and step-wise refinement are similar in all these languages. The

following table shows the names of corresponding concepts in these different languages.

VHDL	Estelle	SDL	LOTOS
entity	module	block (or process)	process (ignoring behavior)
architecture	body	process	process with behavior
port	interac. point	--	gate
signal/port map	channel	channel / route	gate parameter passing

In each case a system consists of a certain number of "modules" which interact with one another and have a certain externally visible behavior which is realized through some internal processing. In the case of SDL, there are two notions: a *block* and its subblocks are used to statically decompose the system into parts; the behavior of a non-decomposed block is defined in terms of one or several *processes* that are included in the block and which have a behavior defined as an extended FSM.

It is important to note that the *modules* in Estelle and the *processes* in SDL and LOTOS can be dynamically created during the execution of the system. Therefore these languages can be used to describe systems with an evolving structure. Dynamic structures are not possible within VHDL.

The interconnection between the different "modules" of a system is specified by using of a "port" (*interaction point* in Estelle, *gate* in LOTOS) which is declared within the connected "module". In VHDL and LOTOS, the interconnection of ports is specified by the identification of the "port" names with the effective parameters (*signals* or *ports* in VHDL, *gates* in LOTOS) that are provided during module instantiation. In Estelle and SDL, two "ports" of different "modules" are explicitly interconnected by an instance of a *channel* which is established between them (SDL distinguishes between *channels* interconnecting *blocks* ,and *routes* interconnecting *processes*); only two-way connections are possible.

It is noted that similar concepts are also supported by many high-level programming languages, as for instance ADA.

3.4. State variables and transitions

In specification languages based on the extended FSM model, such as Estelle and SDL, each module that is defined by an explicit behavior contains usually a STATE variable indicating the present major state of the module and certain additional state variables that store other state information (for instance, the *opt* variable in the example). This is similar to the internal signals declared within VHDL architectures. The situation is different for LOTOS which is more functional in nature. State information is usually carried by process parameters, which are initialized when the process is instantiated and cannot be changed. An example is the parameter o, representing the options, of the process Wait.

We note that a process in LOTOS is not a "process" in the usual sense. It may represent a module, as in the case of the *protocol_entity* process, or a state of a

process, as in the case of the *Closed* or *Wait* processes which are declared within the *protocol_entity*.

The notion of a "state transition" is an important concept in protocol specifications. It represents an amount of processing which is either initiated by the reception of an input or internally depending on the state of the module (so-called "spontaneous transition"). In state transition diagrams, such as Figure 3.3, a transition is represented by an arrow labelled by the input and any produced output. In Estelle and SDL, each module executes one transition at a time, and the transitions are syntactically identified. In LOTOS, each rendezvous is a transition in the sense of the semantics of the language. We note that the specification style used in our example associates a process with each arrow in the transition diagram of Figure 3.3.

3.5. Data structures and types

The facilities for data type definitions are similar in nature for VHDL, Estelle, and SDL. The are a number of predefined, primitive data types, such as integers, a facility for user-defined enumeration types, and a number of type constructors, such as arrays, records. These concepts are quite standard in most programming languages. The notation ASN.1 mentioned in Section 2.3 is also of similar nature.

It is noted, however, that the possibility of declaring types of interactions associated with channels in Estelle and SDL leads to a different structure of the declarations related to the module interfaces, as compared with the structure of VHDL which is based on shared interface variables (ports) with a fixed record structure. In the example above, for instance, the U_interf structure has a element for each of the parameters which may be associated with any kind of service primitive exchanged over the interface. In Estelle, on the other hand, these elements are associated directly with the different kinds of primitives, which provides more information. LOTOS does not use static type checking for the parameters of rendezvous interactions. Type checking is performed at execution time. For instance, the first interaction "US ? x: Creq; " of the process *Closed* implies that the interaction has a single parameter, called x, which must be of type *Creq*.

The data type facilities of LOTOS are quite different. An algebraic specification language for abstract data types (ACT ONE) is used for this purpose. It is quite powerful, but lacks convenient notations for specifying such data structures as enumeration types, arrays and records. In the example above, an abbreviated notation [Boch 90a] has been used for this purpose. Extensions of the language for this purpose are under discussion.

4. Specific issues

4.1. Inter-module communications

The inter-module communication primitives provided by a specification language have a strong impact on the nature and style of specifications that may be defined with the language. VHDL uses the notion of values which are carried by signals and ports. This is the natural communication primitive of hardware circuits. However, in the software area, other communication primitives are often used such as message passing, communication through shared variables and some synchronization primitives, (remote) procedure calls, or rendezvous communication, such as in LOTOS. We note that VHDL signals and ports have some similarity with shared variables, with a well-defined discipline of read and write access enforced through the declaration of *in* and *out* parameters.

For the specification of reactive systems, there are basically two complementary paradigms for defining inter-module communication: events and shared variables. A translation from one paradigm to the other is possible by noting that an event implies certain changes of variable values, or that the change of a variable value represents an event. VHDL uses the variable paradigm, although the notion of "Signal'transaction", as used in the example of Section 3, allows the event paradigm. The FDT's discussed in Section 2.2 use the event paradigm.

In the case of LOTOS, a rendezvous is an event in which several processes participate, and each process may restrict the range of the values of the interaction parameters that may occur, and which are visible to all participating processes. This leads to the possibility of defining a single rendezvous during which the value of one parameter is determined by one process and read by the other, and the value of another parameter is determined by the latter and read by the former. The constraints imposed on a third parameter may leave several possible values, one of which will be chosen (non-deterministically) during the execution of the rendezvous. Each execution of a rendezvous represents a state transition of the system.

In the case of Estelle and SDL, the communication is by asynchronous message passing where messages are stored in an input queue before being processed by the destination module. A state transition of the system corresponds to the processing of an input by a given module, including possibly changes to its local variables and the production of output messages. We note, however, that different extended FSM models may be defined which support synchronous communication [Agga 83b], [Merl 83].

The synchronous nature of the communication in VHDL and LOTOS makes these languages more suitable for describing module interactions at a high-level of abstraction, as compared with Estelle and SDL. For the description of a service interface, such as represented by the pair of ports US and UR shown in Figure 3.2 for example, the use of asynchronous message passing may lead to the occurrence of cross-over of messages in the queues associated with the interface. This introduces complications that are not relevant at a high level of abstraction

(for more details, see for instance [Boch 90a]). These complications do not occur with synchronous communication. However, they are a typical implementation issue, and may have to be dealt with if the implementation of the interface is based on an underlying message passing communication service.

From another point of view, the event-oriented paradigm used in the VHDL specification of Section 3 has many similarities with the message passing paradigm used with Estelle and SDL. In the case that an asynchronous hardware design is used, there will be at most one "unprocessed" event at any given interface at any time. This situation may be directly translated into an Estelle or SDL specification which has, at any given time, at most one message in the two queues (one in each direction) associated with a given interface.

Different specification styles have been discussed for LOTOS specifications [Viss 88]. One of these styles, the so-called constraint-oriented style, is based on the rendezvous communication of LOTOS and allows the separate description of different constraints on the temporal ordering of events and their parameter values. In this context, it is important to note that specifications may be non-deterministic in respect to the output produced for a given input and in respect to the state reached after a given sequence of interactions.

It is interesting to note that different communication paradigms may be used with object-oriented specifications. While certain object-oriented languages assume that objects communicate by asynchronous message passing, most object-oriented programming languages use procedure calls (with return parameters) as the basic communication primitives. A more general rendezvous communication primitive is used, for instance, in the experimental language Mondel [Boch 90l]. A further generalization of rendezvous communication, in the context of object-oriented entity-relationship modelling has been discussed in [Boch 93b].

4.2. Module addressing

In VHDL, where the interconnection structure of modules is static, addressing is not a big issue since it is determined by the static interconnection structure of the system specification. In the case of dynamic system structures, as supported by most software specification languages, including the FDT's discussed in Section 2.2, module addressing is not trivial. The basic interconnection structures of Estelle, SDL and LOTOS are similar in nature as those of VHDL, except that they may be created dynamically.

In object-oriented languages, a given object usually addresses another object by its (unique) identifier. It is therefore necessary that the former "knows" the latter, which is sometimes called an "acquaintance", before it initiates any communication. This kind of addressing is supported by SDL.

In order to allow for the possibility of communication with objects that are not "known", certain languages provide for an "if-exists ... such that ..." construct which returns the identity of an object with the specified properties, if it exists. This kind of addressing is supported by Estelle, Mondel, and to some extent by LOTOS.

4.3. Tools and methods for protocol eningeering

The principal activities during the protocol engineering process were identified in Section 2.1 as: protocol design, design validation, implementation development, and conformance testing and implementation assessment. Research during the last 15 years has resulted in many methods and tools that can be helpful for performing these activities. An overview of these activities and methods can be found for instance in [Boch 90g], [Boch 89g]. Surveys on the associated tools can be found in [Boch 87c], [Lour 92].

Among these methods and tools, certain seem to be of direct interest to the hardware design community, such as the work on test suite development based on FSM specifications (see for instance [Fuji 91a]) or extended FSM models, and the work on equivalent transformations of high-level specifications which have been done in the context of implementation development from LOTOS specifications.

4.4. Critical comparison of languages

A comparative evaluation of the three FDT's Estelle, LOTOS, SDL is difficult to do. The following subjective statements address some of the issues: It seems that Estelle and SDL have the advantage of using well-known concepts of FSM and programming languages which make the initial understanding of the languages easier. The graphics aspects of SDL are also hepful in this respect. On the other hand, LOTOS has relatively few, but powerful language constructs which makes the learning of the complete language easier.

Implementation in software or hardware are usually obtained though a process of step-wise refinement which leads from specifications of requirements, possibly through several stages of design or implementation specifications, to the final product. An important attribute of a specification language is its ability to express abstract specifications which can be used as requirement or design specification without implying any design or implementation choices which would be left open at that stage.

The following properties of LOTOS make it particularly suitable for writing abstract specifications: multi-way synchronous communication, the possibility to write constraint-oriented specifications, and a process structure without an implementation model. On the other hand, these same properties also make it more difficult to generate implementations from LOTOS specifications, as compared to specifications written in Estelle or SDL.

5. Conclusions

The activities in the protocol engineering process are basically similar to those of the software engineering process or the process of hardware design. At each level of the step-wise refinement process which leads from the requirement specification to the implementation, the process includes the generation of a more detailed description, and its validation against the more abstract reference specification and against internal consistency conditions. Formal specifications

are useful for facilitating the partial automation of these activities, either through the automatic generation of the detailed specification, or by providing tools which help for the validation of a manually generated detailed specification either through logical verification or through systematic testing.

In the area of protocol engineering many different languages have been used for formally describing protocols. More recently, three languages, called Formal Description Techniques, have been standardized by ISO and CCITT for the description of communication protocols and services. While there are certain important differences between these languages, as well as with VHDL or high-level programming languages, there are many similarities between these different languages, as demonstrated by the example given in Section 3. One of the main differences relates to the primitives supported by the languages for communication between different system modules.

Communication protocols are important at very different levels of system designs. They play an important role at hardware interfaces, possibly on a single chip between different logical modules; they also play a key role in the construction of communication networks, and they are essential for the correct operation of distributed computer applications. While in the first case, their implementation is necessarily in hardware, the implementation of distributed applications is usually in software. The characteristics of protocols may be related to the presence of parallel processes within the system structure. This makes protocol engineering different from software engineering, which has traditionally been mainly concerned with sequential programing applications. In the same line of thought, there seems to be also much similarity between communication protocols and distributed real-time applications and embedded systems, such as system management or process control.

Given this context, we believe that it is important to note the similarities of hardware, software and protocol engineering. In fact, such a joint approach seems to be the only reasonable one for applications where it is not clear from the beginning which part of the system will be implemented in hardware and which part in software. A typical area of application seems to be protocols for high-speed networking. The specification language used in such a context should be naturally related to the modular structures of hardware and software and allow for behavior descriptions that are naturally translated into hardware or software implementations. Clearly, it would also be desirable to have an integrated tools environment which provide for automation of most of the development activities.

Maybe a closer collaboration between the experts in hardware, software and protocol engineering will lead in the future to such an integrated system development framework.

Acknowledgements: The author would like to thank M. Aboulhamid for helping him understand the concepts of VHDL and for discussing the example specifications included here, and also A. Ghedamsi for proofreading the manuscript. This work is supported by the IDACOM-NSERC-CWARC Industrial Research Chair in Communications Protocols at the University of Montreal.

References

[Agga 83b] S. Aggarwal and R. P. Kurshan, *A language for the specification and analysis of protocols*, Proc. IFIP Workshop on Protocol Specifications, Testing and Verification, North Holland Publ., 1983.

[Beli 89] F. Belina and D. Hogrefe, *The CCITT-Specification and Description Language SDL*, Computer Networks and ISDN Systems, Vol. 16, pp.311-341, 1989.

[Boch 87c] G. v. Bochmann, *Usage of protocol development tools: the results of a survey*, (invited paper), Protocol Specification, Testing and Verification VII, H. Rudin and C. West (eds.), North Holland Publ. (1987), pp.139-161.

[Boch 90a] G. v. Bochmann, *Specifications of a simplified Transport protocol using different formal description techniques*, Computer Networks and ISDN Systems, Vol. 18, no.5, June 1990, pp. 335-377.

[Boch 90g] G. v. Bochmann, *Protocol specification for OSI*, Computer Networks and ISDN Systems 18 (April 1990), pp.167-184.

[Boch 93b] G. v. Bochmann, *Abstract dynamic modelling of complex systems*, Technical Report 1993.

[Boch 93v] G. v. Bochmann and D. K. Probst (eds.), *Computer Aided Verification*, Proceedings of Fourth Int. Workshop on CAV (1992), Springer Verlag, LNCS 663 (1993).

[Boch 90l] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval and N. Williams, *Mondel: An object-oriented specification language*, publication départementale no. 748, Dépt. IRO, Université de Montréal, 1990.

[Boch 93] G. v. Bochmann, R. Dssouli and A. Das (Eds.), *Protocol Test Systems (IWPTS'92)*, North-Holland Publ., 1993.

[Boch 89g] G. v. Bochmann and C. Sunshine, *Formal methods for protocol specification and validation*, chapt. 17 in Computer Network Architectures and Protocols, 2-nd edition (C.Sunshine, ed.), Plenum Press, 1989.

[Bolo 87] T. Bolognesi and E. Brinksma, *Introduction to the ISO Specification Language Lotos*, Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59, 1987.

[Budk 87] S. Budkowski and P. Dembinski, *An introduction to Estelle: a specification language for distributed systems*, Computer Networks and ISDN Systems, vol. 14, no. 1, pp.3-23, 1987.

[SDL 92] CCITT, *Recommendation Z.100, "Specification and Description Language SDL" (Study Group X), 1992.*,

[Diaz 92] M. Diaz and R. Groz (Eds.), *Formal Description Techniques (FORTE'92)*, North-Holland Publ., 1992.

[Ehri 85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1*, Springer Verlag, 1985.

[Fuji 91a] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, *Test selection based on finite state models*, IEEE Transactions on Software Engineering, Vol.17, no.6, June 1991, pp. 591-603.

[Holz 91] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[VHDL 87] IEEE, *Standard VHDL Language Reference Manual*, IEEE Standard 1076-1987.

[Larm 88] K. G. Knightson, T. Knowles and J. Larmouth, *Standards for Open Systems Interconnection*, McGraw-Hill, 1988.

[Linn 92] J. Linn and M. U. Uyar (Eds.), *Protocol Specification, Testing and Verification*, North-Holland Publ. , 1992.

[Lour 92] A. A. F. Loureiro, S. T. Chanson and S. T. Vuong, *FDT tools for protocol development*, Proc. Fifth International Conference on Formal Description Techniques (FORTE'92, Tutorials), Lannion, France, October 1992, pp.38-78.

[Merl 83] P. Merlin and G. v. Bochmann, *On the Construction of Submodule Specifications and Communication Protocols*, ACM Trans. on Programming Languages and Systems, Vol. 5, No. 1 (Jan. 1983), pp. 1-25.

[Miln 80] R. Milner, *A calculus of communicating systems*, Lecture Notes in Computer Science, No. 92, Springer Verlag, 1980.

[Neuf 92a] G. W. Neufeld, *A tutorial on ASN.1*, Computer Networks and ISDN Systems.

[Sari 92a] B. Sarikaya and A. Wiles, *Standard conformance test specification language TTCN*, Computer Standards & Interfaces, Vol.14, No.2, 1992, North-Holland, pp.117-144.

[Schu 80] G. D. Schultz, D. B. Rose and C. H. West, *Executable description and validation of SNA*, IEEE Trans. COM-28, no.4 (April 1980), pp.661-677.

[Spiv 92] J. M. Spivey, *The Z Notation, A Reference Manual*, Prentice Hall, 1992.

[Viss 88] C. Vissers, G. Scollo and M. v. Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems*, Proc. IFIP Symposium on Prot. Spec., Verif. and Testing, Atlantic City, 1988.