A Computer-Aided Protocol Design by Production Systems Approach

CHUNG-MING HUANG, STUDENT MEMBER, IEEE, YE-IN CHANG, STUDENT MEMBER, IEEE, AND MING T. LIU, FELLOW, IEEE

Abstract—**A protocol design system is a collection of software tools for assisting protocol designers to specify, validate, and implement communication protocols. In this paper, we propose a computer-aided protocol design system based on the OPS5 production system approach. In reality, communication protocols are rule-based and data-driven without a fixed order in which the submodules can follow and computations in communication protocols are mainly symbolic with a few numerical computations. These characteristics fall into the applicable problem domain of OPS5. Using the OPS5 production system approach, communication rules (state transitions) are specified as triples of "object-attribute-value" and the modeling of state transitions are specified by production rules. For protocol validation, the modeling of global states, global state transitions, all logical errors and logical properties can be formally defined in terms of production rules. This paper also presents an incremental validation algorithm to facilitate protocol design. Based on a globally shared dataspace (working memory) in which different types and levels of information are all represented in a uniform structure (element), the OPS5 production system integrates both rule-based and procedure-based computations. Using this characteristic, the machine-dependent part can be abstractly specified through external procedure calls, the details of which can be coded in a procedure language until the implementation phase. Since computations in OPS5 are based on pattern matching, all of the attributes of elements that are in production rules or in external procedure calls can act as data templates for generic data types. This capability enhances the generic specification that allows different realization for various implementation environments. In this way, our computer-aided protocol design system can be used not only as a rapid prototyping tool for simulation but also as a real implementation tool for communication protocols.**

I. INTRODUCTION

A communication protocol is a set of rules that governs the interactions among the communication entities. In order to design a communication protocol that is free from logical errors, a complex and repeated cycle consisting of respecification and revalidation is executed until there is no logical error in the communication protocol. Next, a machine executable code is generated according to the validated specification for the protocol implementation.

Many of the popular methods used to formally specify communication protocols are based on the state transition model [1], such as communicating finite state machines (CFSM) [2], petri nets [3], and formal grammars [4]–[6]. For example, Fig. 1 depicts the X.25 packet level DTE/DCE interface communication protocol modeled in CFSM [7], where '−' represents a send transition, '+' represents a receive transition, and the circle represents a state of a communication entity. Protocol validation is a process to detect logical errors and logical properties in a communication protocol. These errors and properties include deadlock, unspecified receptions, channel overflow (when communication channels are finite), nonexecutable interactions, quiescent states, and ambiguous states [8]. Global state reachability analysis is one of the most straightforward ways to validate logical correctness of a communication protocol specified in the state transition model [1], [9], [10]. In this method, a reachable global state graph containing all possible transition sequences in the communication protocol is generated. All of the logical errors and logical properties mentioned above can be identified in the global state reachability analysis. After logical correctness is validated, a machine executable code is generated, according to the operational architecture and the host operating system, for protocol implementation.

A formal specification of a communication protocol is usually composed of two parts, namely, the machine-independent part and the machine-dependent part. The machine-independent part includes those rules that define the interactions of communication entities in response to incoming events and the interactions in response to changes in the local environment. This part can be specified in the specification phase and a high level of abstraction is needed to permit different realizations for various implementation environments. The machine-dependent part includes the procedures for invoking and detecting events, memory management, and encoding and decoding various protocol data units (PDU) for interlayer communication. This part cannot be completely specified until the implementation phase, due to the fact that its realization relies heavily on the operational architecture and the host operating system. Therefore, abstraction mechanisms are needed in the specification phase to describe communication protocols without specifying implementation details and without sacrificing generic realization. Since the machine-independent part is more descriptive and the machine-dependent part is more prescriptive, a rule-based representation formalism is suitable for the specification

Manuscript received September 12, 1989; revised Aug. 2, 1990. This paper was supported by U.S. Army CECOM, Ft. Monmouth, NJ Contract number DAAB07-88-K-A003.

The authors are with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

IEEE Log Number 9039277.

0733-8716/90/1200-1748$01.00 © 1990 IEEE

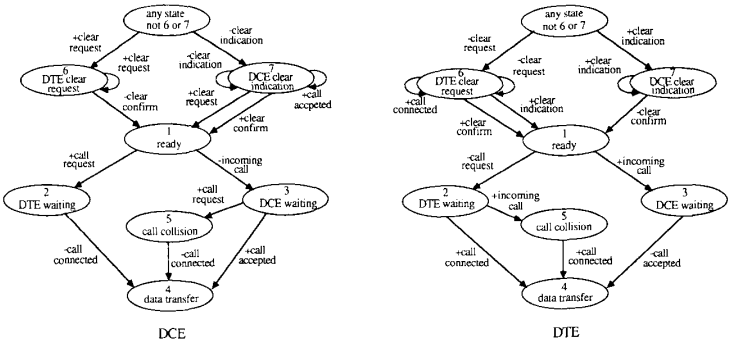

Fig. 1. X.25 communication protocol specified in CFSM model.

phase and a procedure-based representation formalism is suitable for the implementation phase.

In the traditional partitioning approach, a different representation formalism is used in each phase. Therefore, a translation is required between the specification phase and the validation phase (for simulation) [11], [12] and between the specification phase and the implementation phase [12]–[15]. A design method based on such a partitioning approach is proposed in [12]. In this method, three different representation formalisms, NESDEL, EYPA, and IDL, are used in the specification, validation, and implementation phases, respectively. Accordingly, three translators are needed to translate NESDEL to EYPA, EYPA to NESDEL, and NESDEL to IDL, respectively.

As has been reported in [11], [16] for LOTOS behavior expression interpreters, direct generation of the machine executable code from a formal specification becomes harder when a higher level of abstraction is used as the specification formalism. To translate a LOTOS specification into C code, therefore, an interactive compiler is used in which the protocol designer needs to guide and intervene the compilation procedure [16], [17]. On the contrary, overspecification may occur if the representation formalism is too prescriptive and contains implementation-specific features. As has been reported in [18]–[20], ESTELLE specifications suffer from this kind of drawback, since they include too many implementation-specific details. Therefore, a trade-off between these two endpoints could be attempted for a compromise.

In this paper, we propose an integrated approach to protocol design based on the OPS5 production system. In reality, communication protocols are rule-based and data-driven without a fixed order in which the submodules can follow and computations in communication protocols are mainly symbolic with a few numerical computations. These characteristics fall into the applicable problem domain of the OPS5 production system, because the OPS5 production system uses data-sensitive, unordered rules rather than sequential instructions as the basic unit of computation and because the design of the OPS5 production sytem is mainly focused on symbolic computations with some numerical computations [21]. Therefore, the OPS5 production system is appropriate for those problems, such as modeling communication protocols, whose knowledge to be programmed occurs naturally in rule forms [21]. Furthermore, based on a globally shared dataspace (working memory) in which different types and levels of information are all represented in a uniform structure (element), the OPS5 production system integrates both rule-based and procedure-based computations [21]. This integration characteristic is well matched with the requirement of both descriptive and prescriptive computations in implementing communication protocols.

Currently, several shared dataspace languages, such as PROLOG [22, 23] and L.0 (recently developed at Bellcore) [25], are used in rapid prototyping for simulation (validation or performance analysis) only, but are not used for real implementation. The reason is that both PROLOG and L.0 lack the integrated functionality for interprogram communication [24]. In contrast to these shared dataspace languages, OPS5 can be used for both simulation and real implementation. Using the OPS5 production system approach, communication rules (state transitions) are specified as triples of "object-attribute-value" and the modeling of state transitions are specified by production rules. The inference engine (IE) in the OPS5 production system acts as the dispatcher to trigger the applicable production rule according to the current configuration of the working memory (WM). For the protocol validation, global state reachability analysis can be formally modeled in production rules. Some of the production rules describe the transition beween reachable global states, and others define logical errors and logical properties. An incremental validation process [26] to facilitate protocol design can also be formally realized using this approach. Based on the integrated functionality in OPS5, the abstract machine-independent specification can be formally described in production rules by protocol designers and the machine-dependent part can be abstractly described through external procedure calls whose details are coded in a procedure language by protocol implementors. Since computations in the OPS5 production system are based on pattern matching, all of the attributes of elements that are in production rules or in external procedure calls can act as data templates for generic data types. This capability enhances the generic specification that allows different realizations

for various implementation environments. Therefore, in contrast to the traditional partitioning approach [12] which uses three representation formalisms and three translators in the three phases of the protocol design process, our integrated approach employs a single representation mechanism for the entire protocol design process.

Our proposed computer-aided protocol design system is called the Protocol Design Production System (PDPS), which is implemented in the Encore Multimax with 12 processors. In PDPS, a user-friendly interactive environment is supported with an incremental validation process and a knowledge base is provided to answer all possible questions. Through the combined use of parallel computing [27] and a fast pattern matching algorithm [28], PDPS can render good performance for reachability analysis.

The rest of the paper is organized as follows. Section II gives a brief introduction to the OPS5 production system. Based on the OPS5 production system, Section III describes how to model a communication protocol in the specification phase, Section IV describes how to validate a communication protocol in the validation phase and Section V describes how to implement a communication protocol in the implementation phase. Moreover, an algorithm for incremental validation based on this approach is also presented in Section IV. Finally, Section VI describes the features of PDPS, compares it with other approaches, and discusses possible extensions to PDPS.

II. The OPS5 Production System

Being one of the powerful production-system languages, OPS5 has been implemented in BLISS, LISP [21], and C [27]. It is composed of three components: a set of rules (stored in the production memory), a database (working memory), and an interpreter (inference engine) [21].

A rule consists of a precondition-action pair. The left hand side (LHS) of a rule is the preconditions that determine the applicability of this rule. The right hand side (RHS) of a rule is the actions that will be executed if this rule is applied. A rule in OPS5 has the following general form.

(*p* ⟨ identifier ⟩
⟨ condition 1 ⟩ ⟨ condition 2 ⟩ ... ⟨ condition *n* ⟩
-->
⟨ action 1 ⟩ ⟨ action 2 ⟩ ... ⟨ action *m* ⟩).

Each condition is a triple of "object-attribute-value" and is called an *element*. An element can have zero or more attributes. Each attribute can be referenced by the attribute identifier or the field identifier. For example, (person ^sex M ^name Jack ^age 35) is an element that is called *person* (field 1) with three attributes: *sex* (field 2), *name* (field 3), and *age* (field 4); the values of these attributes are M, Jack, and 35, respectively. Conditional elements are simple templates to be matched against data items in the working memory. The values of attributes in conditional elements can be specified as constants, or can be specified by using the pattern expressions, including variables, predicates (>, <, =, etc), disjunction, and conjunction. For example, (candidate ^residence << Columbus Cleveland >> ^sex F ^age { > 18 < 25 } ^name ⟨ n ⟩) is a conditional element that has the following conditions: (1) the value of attribute *residence* can be "Columbus" or "Cleveland" (disjunction); (2) the value of attribute *sex* should be the constant "F"; (3) the value of attribute *age* should be greater than 18 and less than 25 (conjunction); (4) the value of attribute *name* is not restricted and is represented by a variable ⟨ *identifier* ⟩ (= ⟨ *n* ⟩). The relationships among conditional elements are represented similarly by using variables in related attributes.

A working memory (WM) is a collection of elements. There are three main actions that can alter the contents of the WM.

1. Make: add a new element.
2. Remove: delete an old element.
3. Modify: update a matched element.

An interpreter (inference engine) executes the recognize-act cycle between the execution of the applied rules. First, the interpreter executes the *pattern matching* phase to recognize those rules whose LHSs are all satisfied with the current contents of the WM and puts them into a *conflict set*. Then the interpreter executes the *conflict resolution* phase to select one of these rules from the conflict set according to a predefined control strategy. Finally, the interpreter executes the action part of the selected rule to update the contents of the WM.

The conflict resolution strategy of OPS5 is *refractoriness*, *recency*, and *specificity* [21]. Refractoriness means that a rule should not be allowed to apply more than once to the same elements in the WM. Recency means that, if more than one production rules are applicable, the production rule that matches with the most recently inserted elements has a higher priority to be selected than those with older ones. (It can be regarded as the first-in-last-out stack-like processing, or the depth-first processing.) Specificity means that, if more than one applicable production rules match with the same most recently inserted element, the production rule with more conditions to satisfy has a higher priority to be selected than those with fewer conditions, since the former is harder to satisfy than the latter. If there are still more than one production rules that satisfy the above criteria, the system nondeterministically selects any production rule from the applicable ones.

The OPS5 production system permits external procedure and function calls from the right hand side of any production rule [21]. This facility can enhance the computational capability of the OPS5 production system, especially when some nonsymbolic computations or when computations that will be awkwardly solved in rule-based forms are included. The format for an external function call is as follows:

(⟨ function-name ⟩ ⟨ argument-lists ⟩)

where ⟨ argument-lists ⟩ is a sequence of zero or more variables (attributes) or constants. A function call will re-

turn a value to the corresponding attribute. The format for an external procedure call is as follows:

(call ⟨ procedure-name ⟩ ⟨ interface element ⟩).

All communication between OPS5 and the external procedure is via an interface element, which contains the messages used in the external procedure. In order to manipulate the attributes in an interface element or to insert some elements into the WM from the procedure-based computation side [21], [27], OPS5 also provides commands for use in the called functions and procedures. Depending on the version of OPS5 available, these external functions and procedures can be coded in the implementation language directly (such as LISP, BLISS, or C) or other languages indirectly [21].

Pattern matching is usually the bottleneck in rule-based systems. The OPS5 production system improves its capability of pattern matching by using an efficient pattern match algorithm [28] and the *C*-based implementation [27]. To improve the pattern matching execution, this algorithm tests the common preconditions only once and focuses on the updated information to perform incremental matches. Performance of the OPS5 production system can be further improved if it is executed in a parallel machine, such as the Encore Multimax [27], [29].

III. Specification Phase

A formal specification technique should provide concise and precise descriptions for communication protocols. In this section, we use the OPS5 production rules to formally specify communication protocols and show that this approach is well matched with the requirements for protocol specification.

A. Protocol Specification Using Production Rules

There are two types of transitions in communication protocols: one is external events with other layers, such as the send and receive transitions; and the other is internal events with the local environment, such as the time-out event. In PDPS, transitions of external events can be described in one uniform element *rule*. For example, the following two elements describe the possible send and receive transitions for entities DTE and DCE, respectively.

```
(rule ^id 1 ^entity DTE ^type s ^cstate 1 ^message
   call.request ^nstate 2)
(rule ^id 2 ^entity DCE ^type r ^cstate 1 ^message
   call.request ^nstate 2)
```

The modeling of a state transition can be specified by a production rule. For example, a send transition is modeled by the following production rule:

```
(p send_transition
(cstate ^entity ⟨ e ⟩ ^state ⟨ cs ⟩)
(rule ^id ⟨ i ⟩ ^entity ⟨ e ⟩ ^type s ^cstate ⟨ cs ⟩
   ^message ⟨ m ⟩ ^nstate ⟨ ns ⟩)
-->
(call transmit DataRequest ^entity ⟨ e ⟩ ^message
   ⟨ m ⟩)
(modify 1 ^state ⟨ ns ⟩)).
```

This rule can be applied when entity ⟨ *e* ⟩ is in state ⟨ *cs* ⟩ and there is a send transition in which entity ⟨ *e* ⟩ sends a message ⟨ *m* ⟩ from current state ⟨ *cs* ⟩ to next state ⟨ *ns* ⟩. In the LHS, element *c*state records the current state of the entity. In the RHS, the first action calls external procedure transmit with element DataRequest. Element DataRequest represents the abstract service primitive for data transmission. External procedure transmit, in turn, invokes the corresponding system process to transmit the message. The second action updates the entity's state recorded in the first element *cstate*.

Similarly, a receive transition is modeled as follows:

```
(p receive_transition
(DataIndication ^entity ⟨ e ⟩ ^message ⟨ m ⟩)
(cstate ^entity ⟨ e ⟩ ^state ⟨ cs ⟩)
(rule ^id ⟨ i ⟩ ^entity ⟨ e ⟩ ^type r ^cstate ⟨ cs ⟩
   ^message ⟨ m ⟩ ^nstate ⟨ ns ⟩)
-->
(modify 1 ^state ⟨ ns ⟩)
(remove 2))
```

where element DataIndication represents the abstract service primitive for data indication. Element DataIndication contains the receive message and is inserted into the WM by the corresponding system process.

Internal events are represented by the insertion of elements or by external procedure calls with the corresponding interface elements. For example, a time-out event for the Transport Alternating Bit Protocol (ABP) [30] is modeled as follows:

```
(p Receive_Time_out
(timeout ^Aentity ⟨ A ⟩ ^N_Ipoint ⟨ N ⟩)
(Inform ^Aentity ⟨ A ⟩ ^Tseq ⟨ sn ⟩ ^Time ⟨ t ⟩)
{(Tdata ^N_Ipoint ⟨ N ⟩ ^data ⟨ m ⟩) ⟨ buf ⟩}
(cstate ^Aentity ⟨ A ⟩ ^N_Ipoint ⟨ N ⟩ ^U_Ipoint
   ⟨ U ⟩ ^state ⟨ s1 ⟩)
(commrule ^Ipoint server ^mess time_out ^cstate
   ⟨ s1 ⟩ ^nstate ⟨ s2 ⟩
-->
(call trandata DataRequest ^Aentity ⟨ A ⟩ ^N_Ipoint
   ⟨ N ⟩ ^time ⟨ t ⟩ ^seq ⟨ sn ⟩ ^data (substr
⟨ buf ⟩ data inf))
(modify 4 ^state ⟨ s2 ⟩)
(remove 1))
```

Attributes *A*entity, *N_I*point, and *U_I*point represent the identifiers of the local communication entity, the network interaction point and the user interaction point, respectively. Element timeout representing the abstract primitive of the time-out event is inserted by the system process that administers the timer for message transmission. Element Inform records variables: attribute *T*seq records the current send sequence number and attribute Time indicates the time-out period. Another attribute *R*seq (which indicates the currently expected receive sequence number) in element Inform is not expressed explicitly, because attribute *R*seq is not referenced in this production rule. Element *T*data is the send buffer that stores the currently transmitted data block to be acknowledged. Element

commrule records the state transition rule for the time-out event. Element DataRequest represents the abstract network data request service primitive, where function (substr ⟨ buf ⟩ data inf) extracts values from attribute data to the end field (denoted as inf) of element *T*data (designated by "⟨ buf ⟩"). This rule is applied when local entity ⟨ *A* ⟩ is in state ⟨ *s*1 ⟩ and receives a time-out event. In the RHS, the production rule calls external procedure trandata which, in turn, invokes the corresponding system process to re-issue the time-out transmission message and to initiate a timer to monitor this retransmission event.

In a communication protocol, predicates are used as the preconditions to trigger a transition and variables are used to record some local or global status. By using OPS5, variables can be expressed by attributes in some elements, or by one specific element to record all of the variables. For predicates, they can be modeled by the pattern expressions in elements. The satisfaction of a predicate is embedded in the existence of the corresponding element(s) in the WM and/or in the matching with some local or global status recorded in other elements. OPS5 also supports the functionality of processing the message text and numerical computations for variables, such as substr and compute (see the following paragraphs). For example, to specify the reception of an abstract network data indication service primitive in the Transport Alternating Bit Protocol (ABP) [30], one can use the following production rule:

```
(p Receive_Data
{(DataIndication ^N_Ipoint ⟨ N ⟩ ^seq ⟨ sn ⟩ ^type
  trandata ^data ⟨ m ⟩) ⟨ buf ⟩}
(Inform ^Aentity ⟨ A ⟩ ^Rseq ⟨ sn ⟩)
(cstate ^Aentity ⟨ A ⟩ ^N_Ipoint ⟨ N ⟩ ^U_Ipoint ⟨ U ⟩
  ^state ⟨ s1 ⟩)
(commrule ^Ipoint server ^mess R_D ^cstate ⟨ s1 ⟩
  ^nstate ⟨ s2 ⟩)
-->
(call tranack DataRequest ^Aentity ⟨ A ⟩ ^N_Ipoint
  ⟨ N ⟩ ^seq ⟨ sn ⟩ ^data (substr ⟨ buf ⟩ data inf))
inf))
(modify 3 ^state ⟨ s2 ⟩)
(modify 2 ^seq (compute 1 − ⟨ sn ⟩))
(remove 1))
```

Element DataIndication representing the abstract network data indication service primitive is inserted by the system process that administers the receptions from the network layer. Element DataIndication records the network interaction point, the sequence number of the message, the message type in this primitive, and the data block. To express the predicate that the receive sequence number is equal to the currently expected one, the same identifier is used in attribute seq of element DataIndication and in attribute *R*seq of element Inform.

This rule is applied when local entity ⟨ *A* ⟩ is in state ⟨ *s*1 ⟩ and receives a data block with the expected sequence number from network interaction point ⟨ *N* ⟩. In the RHS, the production rule calls external procedure tranack which, in turn, invokes the corresponding system process to transmit an acknowledgement to the peer entity and the corresponding system process that administers the receive data blocks to indicate the readiness for upper users' reception. Then, the next expected receive sequence number in attribute seq of element Inform is calculated by using function compute. Five arithmetic operators (addition, subtraction, multiplication, division, and modulus) used in communication protocols are all supported.

B. Discussion

One of the major advantages of the OPS5 rule-based specification is its simple and flexible structure. It uses production rules for functional specifications and uses elements to store different types and levels of information. The pattern match computation allows each attribute to act as a data template for generic data types and allows both high-level and low-level specifications. Therefore, a protocol designer can specify a communication protocol very primitively or in detail. If the protocol designer wants to specify a protocol primitively or to specify a protocol for more generic realization without dealing with the details of Protocol Data Unit (PDU), then PDU can be shrunk into a single attribute. That is, for those fields in PDU which are not related to the predicates of state transitions, they can be represented by a single symbolic name (one attribute). On the other hand, if the protocol designer wants to specify a protocol in more detail, then PDU can be expanded to a number of attributes. That is, each field of PDU is represented by one attribute. For the abstract specification of interlayer interactions, the abstract primitives are represented by elements or external procedure calls with the corresponding elements.

The flexibility of OPS5 also allows the incorporation of new features. Since the semantics of each element or each attribute is decided by protocol designers, they can incorporate new features into the system easily without changing the underlined software. For example, the priority description in a communication protocol can be specified as an additional attribute in the corresponding element, i.e., the protocol designer can add attribute *priority* to element *rule* described in Section III-A. Furthermore, if the protocol designer wants to specify the control arbitration of the priority-based events, the following three production rules can be added to describe the control arbitration:

1.
```
(p Send_vs_Receive_Competition
(currentpriority ^priority ⟨ p ⟩ ^type ⟨ t ⟩)
(cstate ^entity ⟨ e ⟩ ^state ⟨ cs ⟩)
(rule ^entity ⟨ e ⟩ ^type s ^priority ⟨ p1 ⟩ ^cstate
  ⟨ cs ⟩)
(DataIndication ^entity ⟨ e ⟩ ^message ⟨ m ⟩)
(rule ^entity ⟨ e ⟩ ^type r ^priority {⟨ p2 ⟩
  ⟨ ⟨ p1 ⟩ } ^cstate ⟨ cs ⟩ ^message ⟨ m ⟩)
-->
(modify 1 ^priority ⟨ p1 ⟩ ^type s))
```

```
2. (p Receive_vs_Receive_Competition
   (currentpriority ^priority ⟨p⟩ ^type ⟨t⟩)
   (cstate ^entity ⟨e⟩ ^state ⟨cs⟩)
   (DataIndication ^entity ⟨e⟩ ^message ⟨m1⟩)
   (rule ^entity ⟨e⟩ ^type r ^priority ⟨p1⟩ ^cstate
     ⟨cs⟩ ^message ⟨m1⟩)
   (DataIndication ^entity ⟨e⟩ ^message ⟨m2⟩)
   (rule ^entity ⟨e⟩ ^type r ^priority {⟨p2⟩
     ⟨p1⟩} ^cstate ⟨cs⟩ ^message ⟨m2⟩)
   -->
   (modify 1 ^priority ⟨p2⟩ ^type r))
3. (p Multiple_Transition_Send_Win
   (currentpriority ^priority ⟨p⟩ ^type s)
   (cstate ^entity ⟨e⟩ ^state ⟨cs⟩)
   (rule ^id ⟨i⟩ ^entity ⟨e⟩ ^type s ^priority ⟨p⟩
     ^cstate ⟨cs⟩ ^message ⟨m⟩ ^nstate ⟨ns⟩)
   -->
   (call transmit DataRequest ^entity ⟨e⟩ ^message
     ⟨m⟩)
   (modify 2 ^state ⟨ns⟩)
   (modify 1 ^priority 0)).
```

Note that the production rule of Multiple_Transition_Receive_Win can be added in the same way as the third production rule, and the action (make currentpriority ^priority 0 ^type nil) is added to the end of the RHS of each production rule used for describing the modeling of send and receive transitions. According to the *recency* conflict resolution strategy, production rules *X*_Competition, where *X* is either Send_vs_Receive or Receive_vs_Receive, will be executed after the execution of an event in order to find the highest priority event among the applicable ones in the WM; then the winning event is executed by production rule Multiple_Transition_*Y*_Win, where *Y* is either send or receive. If there is only one event is in the WM, production rule *X*_Competition is not triggered and one of the original production rules for the modeling of send and receive transitions is executed.

IV. Validation Phase

In communication protocols, there are four types of logical errors, which are unspecified reception, deadlock, channel overflow, and nonexecutable interaction, and two types of logical properties, which are quiescent state and ambiguous state [8]. Since global state reachability analysis is more straightforward and widely used to validate logical correctness of a communication protocol, we describe in this section how global states, global state transitions, logical errors, and logical properties can be formally modeled using the OPS5 production system.

A. Modeling of Global States

A *globalstate* element is used to describe a global state. For example, the initial global state of the example in Fig. 1 can be described as follows:

```
(globalstate ^entity DTE ^id 1 ^size 0 ^state ready
  ^queue nil)
(globalstate ^entity DCE ^id 1 ^size 0 ^state ready
  ^queue nil).
```

Since the size of a queue is variable and since only one vector value (an attribute value with variable size) is allowed in OPS5, two elements are used to describe a global state and are connected together by the same ID. Note that the queue in the two connected elements represents a channel from the peer entity to the local entity.

B. Modeling of Global State Transitions

Production rules can be used to describe the transition between reachable global states. The following two production rules can describe all possible transitions from a global state:

```
1. (p e1_send_message_to_e2
   {(globalstate ^entity ⟨e1⟩ ^id ⟨i⟩ ^size ⟨s1⟩
     ^state ⟨x⟩) ⟨h1⟩}
   {(globalstate ^entity {⟨e2⟩ ⟨⟩ ⟨e1⟩} ^id ⟨i⟩
     ^size ⟨s2⟩ ^state ⟨y⟩) ⟨h2⟩}
   (rule ^id ⟨r⟩ ^entity ⟨e1⟩ ^type s ^cstate ⟨x⟩
     ^message ⟨m⟩ ^nstate ⟨n⟩)
   -->
   (bind ⟨j⟩ (genatom))
   (bind ⟨s3⟩ (compute ⟨s2⟩ + 1))
   (make globalstate ^entity ⟨e1⟩ ^id ⟨j⟩ ^size
     ⟨s1⟩ ^state ⟨n⟩ ^queue (substr ⟨h1⟩ 6 inf))
   (make globalstate ^entity ⟨e2⟩ ^id ⟨j⟩ ^size
     ⟨s3⟩ ^state ⟨y⟩ ^queue (substr ⟨h2⟩ 6 inf)
     ⟨m⟩)
   (make rule-state ^ruleid ⟨r⟩ ^stateid ⟨j⟩))
2. (p e1_receive_message_from_e2
   {(globalstate ^entity ⟨e1⟩ ^id ⟨i⟩ ^size ⟨s1⟩
     ^state ⟨x⟩ ^queue {⟨m⟩ ⟨⟩ nil}) ⟨h1⟩}
   {(globalstate ^entity {⟨e2⟩ ⟨⟩ ⟨e1⟩} ^id ⟨i⟩
     ^size ⟨s2⟩ ^state ⟨y⟩) ⟨h2⟩}
   (rule ^id ⟨r⟩ ^entity ⟨e1⟩ ^type r ^cstate ⟨x⟩
     ^message ⟨m⟩ ^nstate ⟨n⟩)
   -->
   (bind ⟨j⟩ (genatom))
   (bind ⟨s3⟩ (compute ⟨s1⟩ - 1))
   (make globalstate ^entity ⟨e1⟩ ^id ⟨j⟩ ^size
     ⟨s3⟩ ^state ⟨n⟩ ^queue (substr ⟨h1⟩ 7 inf))
   (make globalstate ^entity ⟨e2⟩ ^id ⟨j⟩ ^size
     ⟨s2⟩ ^state ⟨y⟩ ^queue (substr ⟨h2⟩ 6 inf))
   (make rule-state ^ruleid ⟨r⟩ ^stateid ⟨j⟩)
```

where bind is an operation to assign the value of the second parameter to the first parameter and genatom is a function to create an unique number.

The first production rule describes a send transition of entity ⟨*e*1⟩. This rule can be applied when entity ⟨*e*1⟩ is in state ⟨*x*⟩ and there is a send rule in which entity ⟨*e*1⟩ sends a message ⟨*m*⟩ to entity ⟨*e*2⟩ from current state ⟨*x*⟩ to next state ⟨*n*⟩. Then, a new global state is generated and inserted into the WM and a mark is set to

record that this rule has been used. Actions 3 and 4 copy the original contents of the queue to the new global state and append the message to the corresponding queue, respectively.

The second production rule describes a receive transition of entity ⟨ e1 ⟩. This rule can be applied when entity ⟨ e1 ⟩ is in state ⟨ x ⟩, a message ⟨ m ⟩ is in the head of the queue from entity ⟨ e2 ⟩ to entity ⟨ e1 ⟩ and there is a receive rule in which entity ⟨ e1 ⟩ receives a message ⟨ m ⟩ from state ⟨ x ⟩ to state ⟨ n ⟩. The remaining parts are similar to those of the send transition.

C. Modeling of Logical Errors and Logical Properties

In the same way, all logical errors and logical properties are defined in the form of production rules.

1.) Unspecified Reception: An unspecified reception error occurs when an entity is in a state such that there is no specified receive transition corresponding to the reception of a message which is in the head of the receive channel [8], [9]. The formal definition in a production rule can be described as follows:

```
(p reception_error
(globalstate ^entity ⟨e1⟩ ^id ⟨i⟩ ^size {⟨s1⟩ >
  0} ^state ⟨x⟩ ^queue {⟨m⟩ ⟨ ⟩ nil})
- (rule ^entity ⟨e1⟩ ^type r ^cstate ⟨x⟩ ^message
  ⟨m⟩ ^nstate ⟨n⟩)
-->
(make reception_error ^id ⟨i⟩)
(write RECEPTION ERROR IN ENTITY ⟨e1⟩!
  STATE ID = ⟨i⟩))
```

where the symbol "-" before the second condition element represents "not exist." The other more restricted definition of the unspecified reception error is limited to a receive state only (in which there is no send transition) [31]. In this case, one more condition element should be added to the precondition part: - (rule ^entity ⟨ e1 ⟩ ^type ⟨ s ⟩ ^cstate ⟨ x ⟩).

2.) Deadlock: A deadlock error occurs when all channels are empty and all entities are in the states in which no send transition exists [8], [9]. If the protocol is acyclic, these states should not be final states. In other words, if one of the entities is not in the final state, this global state is in a deadlock state. The formal definition in the form of a production rule is described as follows:

```
(a) (p deadlock1
    (globalstate ^entity 1 ^id ⟨i⟩ ^size 0 ^state ⟨x⟩)
    (globalstate ^entity 2 ^id ⟨i⟩ ^size 0 ^state ⟨y⟩)
    - (rule ^entity 1 ^type s ^cstate ⟨x⟩ ^message
      ⟨m1⟩ ^nstate ⟨n1⟩)
    - (rule ^entity 2 ^type s ^cstate ⟨y⟩ ^message
      ⟨m2⟩ ^nstate ⟨n2⟩)
    - (globalstate ^entity final-state1 ^state ⟨x⟩)
    -->
    (make deadlock_error ^id ⟨i⟩)
    (write DEADLOCK ERROR! state ID = ⟨i⟩))
```

```
(b) (p deadlock2
    (globalstate ^entity 1 ^id ⟨i⟩ ^size 0 ^state ⟨x⟩)
    (globalstate ^entity 2 ^id ⟨i⟩ ^size 0 ^state ⟨y⟩)
    - (rule ^entity 1 ^type s ^cstate ⟨x⟩ ^message
      ⟨m1⟩ ^nstate ⟨n1⟩)
    - (rule ^entity 2 type s ^cstate ⟨y⟩ ^message
      ⟨m2⟩ ^nstate ⟨n2⟩)
    - (globalstate ^entity final-state2 ^state ⟨y⟩)
    -->
    (make deadlock_error ^id ⟨i⟩)
    (write DEADLOCK ERROR ! state ID = ⟨i⟩)).
```

3.) Channel Overflow: For protocols whose communication channels are finite, a channel overflow error occurs when an entity attempts to send a message into a channel which has already reached its maxiumum capacity [8], [9]. The formal definition in the form of a production rule is described as follows:

```
(p channeloverflow
(channelbound ^number ⟨n⟩)
(globalstate ^entity ⟨e1⟩ ^id ⟨i⟩ ^size {⟨s1⟩ >
  ⟨n⟩})
-->
( make channel_overflow ^id ⟨i⟩)
(write CHANNEL OVERFLOW IN QUEUE TO
  ENTITY ⟨e1⟩ STATE ID = ⟨i⟩)).
```

4.) Nonexecutable Interaction: A nonexecutable interaction is a communication rule which has been specified but has never been executed [8], [9]. It can be found by the following production rule after all global states are generated:

```
(p redundant_rule
(rule ^id ⟨i⟩)
- (rule-state ^ruleid ⟨i⟩ ^stateid ⟨s⟩)
-->
(make redundant_rule ^id ⟨i⟩)
(write REDUNDANT RULE ID = ⟨i⟩).
```

5.) Quiescent State: A quiescent state is a state in which both communication channels are empty [8]. The formal definition in the form of a production rule is described as follows:

```
(p quiescent-state
(globalstate ^entity 1 ^id ⟨i⟩ ^size 0)
(globalstate ^entity 2 ^id ⟨i⟩ ^size 0)
-->
(make quiescent_state ^id ⟨i⟩)
(write IT IS A QUIESCENT STATE ID = ⟨i⟩)).
```

6.) Ambiguous State: An ambiguous state is a state in which an entity's state can coexist in more than one different quiescent state [8]. The formal definition in the form of a production rule is described as follows:

```
(p ambiguous-state
(globalstate ^entity ⟨e1⟩ ^id ⟨i⟩ ^size 0 ^state
  ⟨x⟩)
```

```
(globalstate ^entity { <e2> <> <e1> } ^id <i> ^size
  0 ^state <y>)
(globalstate ^entity <e1> ^id { <j> <> <i> } ^size 0
  ^state { <z> <> <x> })
(globalstate ^entity <e2> ^id <j> ^size 0 ^state
  <y>)
-->
(make ambiguous_state ^entity <e1> ^id1 <i> ^id2
  <j>)
(write IT IS AN AMBIGUOUS STATE ID = <i>
  IN ENTITY <e1> WITH STATE ID = <j>)).
```

D. Incremental Validation

In order to design a correct communication protocol, a complex and repeated cycle consisting of respecification and revalidation is executed. In the nonincremental validation case, any modification of communication protocols will invoke a revalidation process from the beginning. When there are a lot of modifications, including adding or deleting too many rules or modifying the data structures that result in many modifications in rules, this nonincremental approach is preferable. However, when only a few rules are modified, the nonincremental approach makes protocol design very time-consuming. The reason is that those global states that are error-free and are unrelated to the modification still have to be reexplored in the nonincremental validation approach. The incremental validation process shows its effectiveness and usefulness when modification is small. Therefore, an incremental validation process is developed for use in PDPS. In this subsection, an incremental validation algorithm and its formal representation in production rules are briefly described.

There are two parts in the incremental validation process: adding or deleting rules. In both cases, the system will first check whether this rule has already existed or not. In the case of adding a rule, if this rule does exist, then it is a duplicated rule. In the case of deleting a rule, if this rule does not exist, then it is meaningless to delete a non-existing rule. The following two production rules are used to check these two cases:

1.
```
(p check_add
(new <i>)
(rule ^id <i> ^entity <e> ^type <t> ^cstate <c>
  ^message <m> ^nstate <n>)
(rule ^id { <j> <> <i> } ^entity <e> ^type <t>
  ^cstate <c> ^message <m> ^nstate <n>)
-->
(write THIS RULE EXISTS)
(remove 1 2))
```
2.
```
(p check_delete
(old <i>)
(rule ^id <i> ^entity <e> ^type <t> ^cstate <c>
  ^message <m> ^nstate <n>)
- (rule ^id { <j> <> <i> } ^entity <e> ^type <t>
  ^cstate <c> ^message <m> ^nstate <n>)
```

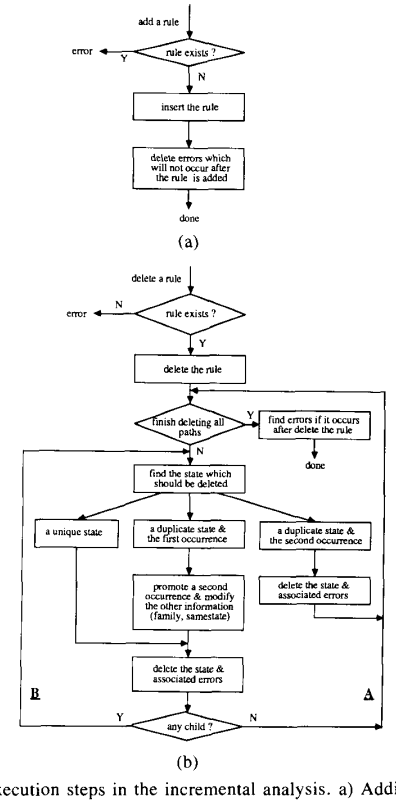

Fig. 2. Execution steps in the incremental analysis. a) Adding a rule; b) Deleting a rule.

```
-->
(write THIS RULE DOESNOT EXIST)
(remove 1 2))
```

where element new (old) is a flag to denote the rule that is to be added (deleted).

Fig. 2(a) shows the following execution steps in adding a rule. After the new rule is stored in the WM, the system deletes those associated errors that should disappear after adding this rule. For example, an unspecified reception in a communication entity should be deleted if the new rule is the missing receive transition; a deadlock should be deleted if the new rule is a send transition that can be executed in the deadlock state. The following production rule describes the deletion of a reception error.

```
(p Add_delete_reception_error
(reception_error ^id <s>)
(new <i>)
(rule ^id <i> ^entity <e> ^type r ^cstate <c> ^mes-
  sage <m> ^nstate <n>)
(globalstate ^entity <e> ^id <s> ^size { <z> > 0 }
  ^state <c> ^queue <m>)
-->
(remove 1)).
```

The deletion of deadlock errors can be described in a similar way. After these steps, by the data-driven property of the OPS5 production system, the new rule will be

automatically applied to the suitable global states by triggering the send and receive production rules (described in Section IV-B). Those production rules for checking logical errors and logical properties will also be triggered if this added rule would introduce new logical errors and logical properties.

Fig. 2(b) shows the following execution steps in deleting a rule. If this rule does exist, it can further divide into two conditions: 1) this rule has not generated any global state yet, i.e., protocol designers find it is useless by their insight before it has ever been applied; 2) this rule has been applied and has generated some global states. The following two production rules are used to distinguish these two conditions.

```
1. (p delete_rule_1
   (old ⟨i⟩)
   (rule ^id ⟨i⟩ ^entity ⟨e⟩ ^type ⟨t⟩ ^cstate ⟨c⟩
      ^message ⟨m⟩ ^nstate ⟨n⟩)
   (rule ^id {⟨j⟩ <> ⟨i⟩} ^entity ⟨e⟩ ^type ⟨t⟩
      ^cstate ⟨c⟩ ^message ⟨m⟩ ^nstate ⟨n⟩)
   - (rule-state ^ruleid ⟨j⟩)
   -->
   (remove 1 2 3))
2. (p delete_rule_2
   (old ⟨i⟩)
   (rule ^id ⟨i⟩ ^entity ⟨e⟩ ^type ⟨t⟩ ^cstate ⟨c⟩
      ^message ⟨m⟩ ^nstate ⟨n⟩)
   (rule ^id {⟨j⟩ <> ⟨i⟩} ^entity ⟨e⟩ ^type ⟨t⟩
      ^cstate ⟨c⟩ ^message ⟨m⟩ ^nstate ⟨n⟩)
   (rule-state ^ruleid ⟨j⟩)
   -->
   (make oldrule ⟨j⟩)
   (make task delete_state)
   (remove 1 2 3)).
```

In the first condition, there is no side effect after deleting this rule. In the second condition, after deleting this rule, all of the global states generated by this rule, all descendants of these global states and the associated information should be deleted. To support these actions, the following information is recorded in the WM:

1.) Those states generated by applying a specific rule: This information is stored in element rule-state: (rule-state ^ruleid ⟨i⟩ ^stateid ⟨s⟩). This element denotes that state ⟨s⟩ is produced after rule ⟨i⟩ is triggered.

2.) The hierarchy among states: This information is stored in element *family*: (family ^mother ⟨m⟩ ^child ⟨c⟩). This element denotes that state ⟨m⟩ is the mother of state ⟨c⟩ in the reachable global state graph. That is, state ⟨c⟩ is produced from state ⟨m⟩ by triggering a transition.

3.) The duplicated states: This information is stored in element *samestate*: (samestate ^first ⟨f⟩ ^second ⟨s⟩). This element denotes that ⟨f⟩ is the first occurrence of this state and ⟨s⟩ is the duplicated one.

There are three cases in deleting the associated states:

1.) The corresponding generated state is a unique state in the reachable global state graph: In this case, the system deletes this state and its descendants by referring to element family iteratively. The following production rule is used for this case:

```
(p delete_state_unique
(task delete_state)
(oldrule ⟨i⟩)
(rule-state ^ruleid ⟨i⟩ ^stateid ⟨s⟩)
(globalstate ^entity 1 ^id ⟨s⟩)
(globalstate ^entity 2 ^id ⟨s⟩)
- (samestate ^first ⟨s⟩)
- (samestate ^second ⟨s⟩)
-->
(make ^oldstate ⟨s⟩)
(make task delete_more)
(remove 3 4 5)).
```

2.) The corresponding generated state is a duplicated state: In this case, it means that this state is a second occurrence of an existing state. Since this state will not have any descendant in the reachable global state graph, the system deletes this duplicated state only. The following production rule is used for this case:

```
(p delete_state_not_first
(task delete_state)
(oldrule ⟨i⟩)
(globalstate ^entity 1 ^id ⟨s⟩)
(globalstate ^entity 2 ^id ⟨s⟩)
(rule-state ^ruleid ⟨i⟩ ^stateid ⟨s⟩)
(samestate ^first ⟨f⟩ ^second ⟨s⟩)
-->
(make ^oldstate ⟨s⟩)
(make task delete_more)
(remove 3 4 5 6)).
```

3.) The corresponding generated state is the first occurrence and there are duplicated states in the reachable global state graph: In this case, the system deletes this state and continues to delete its descendants if they exist. The following production rule is used for deleting the first state and making elements for the following tasks:

```
(p delete_state_first
(task delete_state)
(oldrule ⟨i⟩)
(rule-state ^ruleid ⟨i⟩ ^stateid ⟨s⟩)
(globalstate ^entity 1 ^id ⟨s⟩)
(globalstate ^entity 2 ^id ⟨s⟩)
(samestate ^first ⟨s⟩ ^second ⟨n⟩)
-->
(make ^oldstate ⟨s⟩)
(modify 4 ^id ⟨n⟩)
(modify 5 ^id ⟨n⟩)
(make changesamestate ^first ⟨s⟩ ^second ⟨n⟩)
(make task change_same_state)
(make task delete_more)
(remove 3 6))
```

Since this state can also be produced by triggering other rules, the system promotes one of the duplicated states from element samestate to be the first occurrence and updates the related information: replacing the value of attribute first in the elements of samestate with the ID of the promoted one. By using elements family and samestate to describe the relationship of global states, the promotion process can be described in the similar way as the above production rules.

When the system deletes a state, all of the associated errors and information should also be deleted. For simplicity, we use the production rule that deletes deadlock errors as illustration.

```
(p Del_delete_deadlock
(task delete_more)
(oldstate ⟨i⟩)
(deadlock_error ^id ⟨i⟩)
-->
(remove 3)).
```

After deleting the first state, the system continues to delete the descendant states [path B in Fig. 2-(b)]. After deleting one path traced from a state generated by applying this rule, the system will continue to delete the other paths traced from those states that were also generated by applying this rule [path A in Fig. 2-(b)]. Finally, by triggering those production rules that describe logical errors and logical properties, the system will check whether the absence of this rule would lead to any new logical error or logical property.

E. Discussion

Our PDPS is currently developed in the Encore Multimax with 12 processors by using the C-based OPS5 production system [27]. In PDPS, a user-friendly interactive design environment is supported and a knowledge base is provided to answer all possible questions. Many options are available to protocol designers in the beginning. These options includes: 1) specifications of the channel bound, initial states and final states for communication entities; 2) input from either an external file or the designer's termimal; 3) print all error messages immediately? 4) print global states step by step? 5) stop when an error is found or after all errors are found? 6) print the error path (from the initial state to the error state) immediately after an error is found? 7) which definition of the reception error (mentioned in Section IV, C.) is used?

PDPS works as follows. After the input and the designer's choices are made, PDPS will start to find all possible legal transitions and print out information. A duplicated state is allowed to be generated, but it will be detected and deleted later. When some logical errors or logical properties in a certain state are detected, the system can identify these errors or properties and then print out the related messages to the designer. Furthermore, during the intermediate stage of validation, many options are provided, including listing communication rules, currently generated states, and adding/deleting rules dynamically.

TABLE I
THE TIME-PROCESS TABLE OF VALIDATING X.25 PROTOCOL IN PDPS

PROCESS	1	2	3	4	5	6	7	8	9	10	11	12
TASK QUEUE	1	2	4	4	4	8	8	16	16	16	16	16
TIME (second)	479	468	253	188	152	135	122	118	112	107	106	107
SPEED-UP	1	1.024	1.893	2.548	3.151	3.548	3.926	4.059	4.277	4.477	4.519	4.477

Since incremental validation is supported in PDPS, only the associated states will be generated or deleted instead of regenerating all states from the beginning. Finally, after generating all possible states either from the beginning or from the point of incremental validation, the system can answer questions, including listing states, communication rules, duplicates states, errors and properties, the states generated by a specified communication rule and the paths from the initial state to a specified state. Meanwhile, the designer can add/delete rules at this point, then the system will continue to do the incremental validation until the designer is satisfied.

By using the *C*-based OPS5 executed in the Encore Multimax with 12 processors, we have observed that PDPS needs 106 seconds to validate the X.25 protocol depicted in Fig. 1 [7]. This time includes the system time and the user time. Table I shows the time-process relationship for validating the X.25 protocol in PDPS. In addition to specifying the number of processes, the protocol designer can also specify the number of task queues. A task queue holds a list of tasks that are waiting for processing. A task is an independently schedulable unit of pattern match work that can be executed in parallel with other tasks. In this way, each process searches for a task by scanning these task queues. When a small number of task queues are used, it will result in processes' contention for the task queues. On the other hand, when a large number of task queues are used, it will result in a situation that most of the task queues will be empty and processes will waste time scanning several empty task queues before finding one with a task. This table also lists the number of task queues for each number of processes that will produce optimal performance.

Since there are 12 processors and 1 processor is used for processes management, the speed-up reaches its peak when there are 11 processes. When the number of processes is more than 11 (12, 13, 14 ...) the performance becomes worse. The reason is that the overhead for process allocation overshadows the speed-up derived from multiple processes. A higher degree of parallelism can be explored, because the speed-up obtained from the parallel execution of production rules will multiply with the speed-up obtained from the parallel pattern matching [32]. A key issue to allow a high degree of parallel execution of production rules is that the action part of one production rule will not update the condition part of the other production rules. (Action MAKE makes nonexisting conditions false; actions MODIFY and REMOVE make existing conditions false.) The mutual exclusion issue in parallel protocol validation is *many-READ-one-WRITE*: after a global state is produced, this state will be used for (sharable) READ only. Hence, the RHSs (action parts) of

production rules for logical errors and logical properties, and for the modeling of transitions are independent of one another's LHS. Therefore, a much better performance can be achieved when the parallelism is extended to the parallel execution of production rules.

The main advantage of reachability analysis is that it is straightforward, well-suited for validating protocols in the state transition model and easy to automate [10]. While reachability analysis has been used for validation of protocols of low to middle complexity, the practical use of reachability analysis for more complex protocols has been hindered by the problem of state space explosion [33]. Therefore, even based on multiprocessor computing systems, it is impractical to use the straightforward reachability analysis to validate every kind of protocols. The reason is that a finite size of storage space and a finite number of processors cannot accommodate and handle an astronomical or infinite amount of information. However, by using multiprocessor computing systems (such as the Hypercube with 128 processors and 128*64 Mb memory space in [34] or the Encore Multimax with 12 processors and 64 Mb memory space in PDPS) that have more powerful computing capability and larger storage space, we may be able to use reachability analysis for validating more complex protocols. For very complex protocols with an astronomical or infinite amount of global states that are not suitable for reachability analysis, program proof techniques for rule-based languages can be used [35], [36]. However, program proof techniques are not so straightforward to apply and not so easy to automate [37]. The difficulty is that the formulation of assertions and proofs often requires human beings' insight [1], [10].

V. Implementation Phase

An abstract specification of a communication protocol describes the interactions of communication entities and is machine-independent. However, the real implementation of a communication protocol includes the interactions with the execution environment and is machine-dependent. Most often, a generic description of the communication protocol is specified in the specification phase by the protocol designer. In this generic description, the machine-dependent part is usually specified abstractly or left unspecified. The realization of machine-dependent part, pertaining to the details of the operational architecture and the host operating system, is deferred until the implementation phase. During the implementation phase, the machine-dependent part is coded by the protocol implementor.

In this section, we briefly describe the protocol implementation using the OPS5 production system approach.

A. General Implementation Model in OPS5

Fig. 3 shows the general implementation model for the OPS5 production system approach. Production rules for the modeling of transitions are stored in the production memory. Elements in the WM can be classified into three sets: 1) the specification of communication rules, 2) the representation of abstract primitives, and 3) other elements, such as the elements used for states, variables and predicates. These productions and elements are specified by the protocol designer.

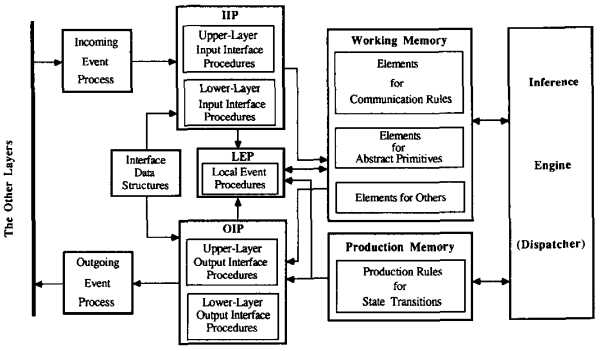

Fig. 3. Implementation model.

The inference engine (IE) acts as the dispatcher to select one of the applicable transitions for execution. The Interface Data Structures (IDS) store the real data formats that are used in communication with other layers. The external procedures (and functions) can be classified into the following three groups:

1.) Input Interface Procedures (IIP): IIP includes the upper-layer input interface procedures and the lower-layer input interface procedures. In the first step, IIP decodes the input events received by the incoming event process according to IDS. Then, IIP encodes the incoming events in the elements that represent the corresponding abstract primitives. By using the commands provided in OPS5, these interface elements can be inserted into the WM. For example, the following commands inserts element DataIndication used for data indication in X.25 into the WM:

```
dollar_value(dollar_intern(''DataIndication''));
dollar_tab(dollar_intern(''entity''));
dollar_value(dollar_cvna(EID));
dollar_tab(dollar_intern (''message''));
dollar_value(dollar_intern(userbuffer[EID]));
dollar_assert();
```

where variables EID and userbuffer[EID] represent the identifiers of the receive entity and the pointer pointing to the storage of the receive data block, respectively. Since every data item in OPS5 is regarded as an atom, function dollar_intern is used to translate a string into a string atom and function dollar_cvna is used to translate a number into a number atom. After the translation, function dollar_value puts the assigned value into the current field indicated by a pointer implicitly. In order to refer to a specific attribute, function dollar_tab is provided to move the pointer to the attribute expressed in the argument. Finally, when all values are assigned, function dollar_assert inserts element DataIndication into the WM.

2.) Output Interface Procedures (OIP): OIP includes the upper-layer output interface procedures and the

lower-layer output interface procedures. In the first step, OIP decodes the abstract primitives that are represented in some interface elements. Then, OIP encodes the corresponding outgoing events according to IDS. By using the commands provided in OPS5, OIP decodes the abstract primitives by extracting the values from the interface elements. For example, the following commands extracts the values of attributes from element DataRequest used for data transmission in X.25:

```
EID = dollar_cvan(dollar_parameter(dollar_litbind-
  (dollar_intern(''entity''))));
buffer[EID] = dollar_cvas(dollar_parameter-
  (dollar_litbind(dollar_intern(''message''))));
```

where variable buffer[EID] records the pointer pointing to the storage of the messages to be transmitted. In order to obtain the value of a specific attribute, the index of the attribute should be known first. Function dollar_litbind is provided to return the index of the field of the attribute indicated by the argument. Then, function dollar_parameter returns the atom of the field indicated by the argument. Thereafter, to obtain the value of the atom with the right data type, function dollar_cvan is used to translate a number atom into a number; function dollar_cvas is used to translate a string atom into a string.

3.) Local Event Procedures (LEP): LEP includes procedures and functions that deal with the local system events, such as the memory management and the time-out monitor.

The control flow in this model (Fig. 3) is described as follows: When an input event arrives from the other layers, the corresponding incoming event process is awakened and the corresponding input interface procedure is called. Next, this input interface procedure interprets the input event according to the interface data structures and generates the corresponding elements. Then, the generated elements are inserted into the WM. These elements, in turn, trigger the corresponding production rules by the IE and invoke the related actions: changing a local entity's state, modifying the configuration of the WM and/or calling external procedures. After that, these called external procedures may invoke some outgoing events by extracting the values from the interface elements. According to these extracted values and the interface data structures, the output interface procedure generates the outgoing message and/or some local environment events. For example, a timer is set up to monitor a transmission event. Finally, the outgoing event process sends the outgoing message to the other layers. For local event processing, such as the initiation and the expiration of a timer, the corresponding execution is similar to that for an outgoing event process and for an incoming event process, respectively.

B. Discussion

In the OPS5 production system approach, the machine-independent part is specified in rule-based computations and the machine-dependent part is abstractly specified through external procedure calls by the protocol designer in the specification phase. In the implementation phase, the machine-dependent part is coded according to the operational architecture and the host operating system by the protocol implementor. The flexibility of OPS5 also allows the implementor to code the machine-dependent part using production rules, depending on whether the computation is more production-oriented or procedure-oriented. For example, the control arbitration for priority-based events can be expressed in production rules (discussed in Section III. *B.*)

For the code process paradigm, the formal specification in OPS5 production rule-based code is translated into assembly code by the ParaOPS5 compiler, then the assembler translates assembly code into object code (this object code can be executed in Encore or VAX); for the external procedures or functions that are written in a procedure language, such as C or PASCAL, the corresponding compiler compiles them into object code. Next, by using the linker and the loader, the complete object code for the communication protocol is produced. In order to allow the OPS5 code to be directly executed on a general UNIX system, instead of translating to assembly code and then object code, a CParaOPS5 compiler can translate the OPS5 production rule-based code into C code (which can be executed in either uniprocessor systems or multiprocessor systems). In this way, the OPS5 specification is allowed for direct execution on a general UNIX system.

VI. Summary and Conclusion

Fig. 4 shows an overview of the protocol design process in PDPS. Elements for representing communication rules, abstract primitives and others, such as states, predicates and variables, and production rules for the state transitions are formally specified in the specification phase. The machine-dependent part, such as input/output interface procedures, local events procedures, and interface data structures, are supplemented in the implementation phase. In the validation phase, some of the production rules generate all of the possible global states in a communication protocol, and others formally describe logical errors and logical properties in the communication protocol; some additional elements are used to describe the global states, logical errors and logical properties, and other frequently changed information.

For both simulation and implementation, there is a trade-off between the abstract specification level and the complexity in deriving the machine executable code. As has been pointed out in [16], [17], LOTOS specifications are too abstract so that they cannot be directly translated into the machine executable code; therefore, they need a series of translation steps, such as LOTOS specifications -> CCS specifications -> PROLOG specifications [11]. On the other hand, if a formal description technique becomes an overspecification in order to be directly machine executable, it is no longer qualified as an abstraction. As has been pointed out in [18]–[20], ESTELLE specifications suffer from this kind of drawback, since they include

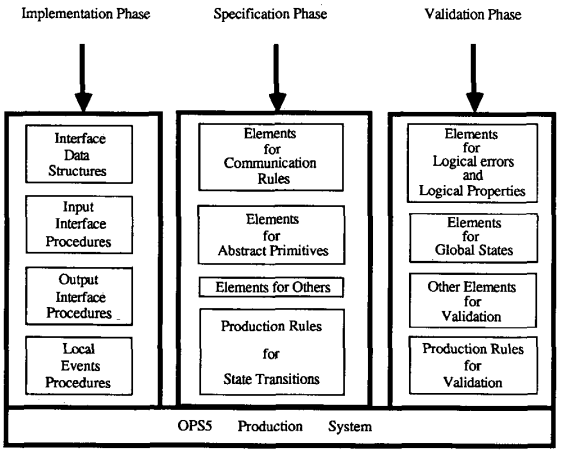

Fig. 4. Overview of the protocol design process in PDPS.

too many implementation-oriented details. Therefore, one of our motivations in using the OPS5 production system approach is trying to compromise these two endpoints: OPS5 provides both rule-based and procedure-based computations to avoid overspecification, and OPS5 provides generic data templates by using pattern matching for generic realizations.

Formal protocol description techniques can be classified into three categories [1]: the state-transition model, the abstract language model [38], and the hybrid model [39], [40]. The state transition model is abstract and is very easy to automate; however, reachability analysis is hindered by the problem of state explosion. The abstract language model is good for theoretical proof of functional correctness; however, efforts to prove the correctness of a program far exceed those required for developing the program, and the correctness proof of a program usually depends heavily on human ingenuity and is hard to automate. The hybrid model tries to combine the features of both the state-transition and the abstract language models; all of the standardized formal description techniques, such as ESTELLE [39] and LOTOS [40], belong to this model. In the hybrid model, the state-transition part of the model captures the control aspects of a protocol while variables and data are easily handled by the program part of the model. Depending on how high a level of abstraction is used, formal description techniques based on the hybrid model require different degrees of efforts in the machine executable phase, either for simulation or real implementation. The OPS5 production system approach is also based on the hybrid model; OPS5 uses production rules to specify the state-transition part of a protocol and uses attributes in elements as generic data templates to specify variables and data of the protocol. Both reachability analysis and rule-based program proof techniques [35], [36] for protocol validation are also applicable to the OPS5 production system approach, depending on the complexity of the protocol and the execution environment.

In the past, many people have tried to reduce the size of global-state space that must be explored [41], but few have tried to use parallel execution for reachability analysis [34]. We believe that both the parallel execution and the global state reduction strategy should be applied to obtain the best performance. Moreover, it is not wise to apply one global state reduction strategy to the reachability analysis of all kinds of protocols, due to the fact that each kind of protocols has its own characteristics and each global state reduction strategy has its own applicable domain of protocols. Nevertheless, when a combination of parallel execution and global state reduction strategy is used, the global state reduction strategy should keep the *one-WRITE-many-READ* characteristic in protocol validation (discussed in Section IV. *E.*). If the additional information used for the reduction strategy results in *many-WRITE-many-READ*, then the reduction strategy is not suitable for parallel execution. We are now studying the feasibility of using an intelligent interface to classify protocols and to apply the suitable global state reduction strategy to each kind of protocols. In this way, through an intelligent protocol validation system that is executed in a multiprocessor environment, reachability analysis of more complicated protocols can be carried out. In addition, communication protocol performance analysis [42] can be realized by adding probability and time constraints to each communication rule.

From our experience in constructing a protocol design system based on the OPS5 production sytem, we have observed two disadvantages. First, although the combined use of the *C*-based implementation and an efficient pattern matching algorithm in OPS5 can resolve the pattern matching bottleneck of low to middle complex computations [27], pattern matching is still the bottleneck for high complex computations in the OPS5 production system. Therefore, for those computations with high complexity, such as global state reachability analysis, special hardware (such as the Encore Multimax with 12 processors) is required. Second, OPS5 is a general-purpose specification language that is not designed for formal protocol specification only. Therefore, some parts of the formal specification cannot be described compactly. For example, $n^2 - n$ elements are required to describe a global state in an n-entity protocol.

Most of the existing approaches to protocol design are based on the translation of heterogeneous representations used in the specification, validation, implementation, and testing phases of a communication protocol. However, these approaches not only make the protocol design very time-consuming but also require a lot of work in developing different kinds of software tools to handle these phases. Our OPS5 production system approach integrates both rule-based and procedure-based computations by using a global working memory in which different types and levels of information are all represented in a uniform structure. Therefore, to realize phase B from phase A (specification -> validation, specification -> implementation, specification and implementation -> testing), we

can combine those related elements in phase A and some additional elements used in B to generate the corresponding computations, either rule-based or procedure-based. Currently, we are investigating the feasibility of applying the OPS5 production system approach to protocol testing. This extension, together with our current development, will result in a general-purpose computer-aided protocol design system.

ACKNOWLEDGMENT

The authors would like to thank G. Jones and G.-T. Hsu for their assistance in setting up the execution environment in the Encore Multimax. The authors would also like to thank the anonymous referees for providing valuable comments to improve this paper.

REFERENCES

[1] M. T. Liu, "Protocol engineering," in *Advances in Computers, vol. 29*, M.C. Yorits, Ed. New York: Academic, 1989, pp. 79–195.

[2] D. Brand and P. Zafiropulo, "On communication finite-state machines," *J. ACM*, vol. 30, no. 2, pp. 323–342, Apr. 1983.

[3] M. Diaz, "Modeling and analysis of communication and cooperation protocols using Petri net based models," *Computer Networks*, vol. 8, no. 6, pp. 419–441, 1982.

[4] D. P. Anderson and L. H. Landweber, "A grammar-based methodology for protocol specification and implementation," in *Proc. ACM 9th Data Commun. Symp.*, 1984, pp. 63–70.

[5] A. Y. Teng and M. T. Liu, "A formal approach to the design and implementation of network communication protocol," in *Proc. IEEE COMPSAC*, Nov. 1978, pp. 722–727.

[6] L. D. Umbaugh, M. T. Liu, and C. J. Graff, "Specification and validation of the transmission control protocol using transmission grammar," in *Proc. IEEE COMPSAC*, 1983, pp. 207–216.

[7] N. C. Liu, "A methodology for specifying and analyzing communication protocols and services," Ph.D. dissertation, Ohio State Univ., Columbus, OH, 1986.

[8] C. S. Lu, "Automated validation of communication protocols," Ph.D. dissertation, Ohio State Univ., Columbus, OH, 1986.

[9] C. H.West, "Generalized technique for communication protocol validation," *IBM J. Res. & Develop.*, vol. 22, no. 4, pp. 393–404, July 1978.

[10] D. P. Sidhu, "Experience with formal methods in protocol development strategy," in *Formal Description Techniques II*, S. T. Vuong, Ed. New York: North-Holland, 1989, pp. 437–453.

[11] L. Logrippo, A. Obaid, J. P. Briand, and M. C. Fehri, "An interpreter for LOTOS, a specification language for distributed systems," *J. Software—Practice and Exper.*, vol. 18, no. 4, pp. 365–385, Apr. 1988.

[12] N. Shiratori, K. Takahashi, and S. Noguchi, "A software design method and its application to protocol and communication software development," *Comput. Net. and ISDN Syst.*, vol. 15, pp. 245–267, 1988.

[13] G. V. Bochmann, G. W. Gerber, and J. M. Serre, "Semiautomatic implementation of communication protocols," *IEEE Trans. Software Engin.*, vol. SE-13, no. 9, pp. 989–1000, 1987.

[14] T. P. Blumer and D. P. Sidhu, "Mechanical verification and automatic implementation of communication protocols," *IEEE Trans. Software Engin.*, vol. SE-12, no. 8, pp. 827–843, 1986.

[15] S. T. Vuong, A. C. Lau, and R. I. Chan, "Semiautomatic implementation of protocols using an Estelle-C compiler," *IEEE Trans. Software Engin.*, vol. SE-14, no. 3, pp. 383–393, 1988.

[16] K. J. Turner, "A LOTOS-based development strategy," in *Formal Description Techniques II*, S. T. Vuong, Ed. New York: North-Holland, pp. 117–132, 1989.

[17] J. A. Manas, T. D. Mignel, and H. V. Thieuen, "The implementation of a specification language for OSI systems," in *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project*, P. H. J. Van Eijk, C. A. Vissers, and M. Diaz, Eds. New York: North-Holland, pp. 409–422, 1989.

[18] C. A. Vissers and G. Scollo, "Formal specification in OSI," in *Networking in Open Systems, Lecture Notes in Computer Science, 248*, G. Muller and R. P. Blanc, Eds. New York: Springer-Verlag, 1987, pp. 338–372.

[19] J. P. Courtiat, "How could Estelle become a better FDT?," *Protocol Specification, Testing and Verification, VII*, H. Rudin and C. H. West, Eds. New York: North-Holland, 1987, pp. 43–60.

[20] L. Svobodova, "Implementing OSI systems," *IEEE J. Select. Areas Commun.*, vol. 7, no. 7, pp. 1115–1130, Sept. 1989.

[21] B. Lee, F. Robert, K. Elaine, and M. Nancy, *Programming Expert Systems in OPS5*. Reading, MA: Addison-Wesley, 1985.

[22] D. P. Sidhu, "Protocol verification via executable logic specification," in *Protocol Specification, Testing and Verification, III*, H. Rudin and C. H. West, Eds. New York: North-Holland, 1983, pp. 237–248.

[23] D. P. Sidhu and C. S. Crall, "Executable logic specifications for protocol service interfaces," *IEEE Trans. Software Engin.*, vol. 14, no. 1, pp. 98–121, Jan. 1988.

[24] M. L. Brodie and M. Jarke, "On integrating logic programming and database," in *Proc. 1st Int. Workshop Expert Database Syst.*, 1986, pp. 191–207.

[25] D. M. Cohen, T. M. Guinther, and L. A. Ness, "Rapid prototyping of communication protocols using a new parallel language," in *Proc. IEEE 1st Int. Conf. Syst. Integration*, 1990, pp. 196–204.

[26] I. E. Liao and M. T. Liu, "Incremental protocol verification using deductive database system," in *Proc. 5th Int. Conf. Data Engin.*, Feb. 1989, pp. 216–223.

[27] D. Kalp, M. Tambe, A. Gupta, C. Forgy, A. Newell, A. Acharya, B. Milnes, and K. Swedlow, *Parallel OPS5 User's Manual*. Pittsburgh, PA: Carnegie Mellon U., Nov. 1988.

[28] C. L. Forgy, "RETE: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intell.*, vol. 19, no. 1, pp. 17–37, Sept. 1982.

[29] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell, "Parallel implementation of OPS5 on the encore multiprocessor: Results and analysis," *Int. J. Parallel Program.*, vol. 17, no. 2, 1989.

[30] R. J. Linn, "The features and facilities of ESTELLE," Natl. Inst. Standards Techn. Tech. Rep., Nov. 1988.

[31] M. G. Gouda and Y. T. Yu, "Synthesis of communicating finite-state machines with guaranteed process," *IEEE Trans. Commun.*, vol. COM-32, no. 7, pp. 779–788, July 1984.

[32] W. Harvey, D. Kalp, M. Tambe, A. Acharya, D. Mckeown, and A. Newell, "Measuring the effectiveness of task-level parallelism for high-level vision," in *Proc. DARPA Image Understanding Workshop*, 1989.

[33] F. J. Lin, P. M. Chu, and M. T. Liu, "Protocol verification using reachability analysis: The state explosion problem and relief strategies," in *Proc. ACM SIGCOMM Workshop*, 1987, pp. 126–135.

[34] O. Frieder and G. E. Herman, "Protocol verification using database technology," *IEEE J. Select. Areas Commun.*, vol. 7, no. 3, pp. 324–334, Apr. 1989.

[35] H. C. Cunningham and G. C. Roman, "A UNITY-style programming logic for shared dataspace programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 365–376, 1990.

[36] H. C. Cunningham and G. C. Roman, "Toward formal verification of rule-based systems: A shared dataspace perspective," Dept. of CIS, Washington Univ., St. Louis, MO, Tech. Rep. WUCS-89-28, 1989.

[37] H. A. Lin and M. T. Liu, "Verification of CSP programs based on communication assertions," in *Proc. IEEE Phoenix Conf. Comp. Commun.*, 1986, pp. 412–417.

[38] N. C. Liu and M. T. Liu, "CSP-based specification for network protocols and services," in *Proc. IEEE Comput. Net. Symp.*, 1984, pp. 95–102.

[39] ISO - Information Processing Systems - Open Systems Interconnection, "ESTELLE—A formal description technique based on extended state transition model," DIS. 9074, 1987.

[40] ISO - Information Processing Systems - Open Systems Interconnection, "LOTOS—A formal description technique based on the temporal ordering of observational behavior," DIS 8807, 1987.

[41] P. M. Chu and M. T. Liu, "Global state graph reduction techniques for protocol validation in the EFSM model," in *Proc. IEEE Phoenix Conf. Comput. Commun.*, 1989, pp. 371–377.

[42] F. J. Lin and M. T. Liu, "An integrated approach to verification and performance analysis of communication protocols," in *Protocol Specification, Testing and Verification VIII*, S. Aggarwal and K. Sabnani, Eds. New York: North-Holland, 1988, pp. 125–140.

Chung-ming Huang (S'87), photograph and biography not available at the time of publication.

Ye-In Chang (S'87), photograph and biography not available at the time of publication.

Ming T. (Mike) Liu (M'65-SM'82-F'83) received the B.S.E.E. degree from the National Cheng Kung University, Tainan, Taiwan, in 1957, and the M.S.E.E. and Ph.D. degrees from the Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, in 1961 and 1964, respectively.

Since 1969, he has been with The Ohio State University, Columbus, where he is currently Professor of Computer and Information Science. Since 1974, he has been actively involved in the research and development of computer networking and distributed computing, and has published over 100 technical papers in this and related areas.

Dr. Liu has received several awards for his technical contributions and for his dedicated services to the IEEE Computer Society. He was Program Co-Chair of the 1981 International Conference on Parallel Processing; Distinguished Visitor of the IEEE Computer Society from 1981 to 1984; Chairman of the Technical Committee on Distributed Processing from 1982 to 1984; Program and General Chairman of the Fifth and Sixth International Conference on Distributed Computing Systems, respectively; Chairman of the ACM/IEEE Eckert-Mauchly Award Committee for 1984–1985; Computer Society's Vice President for Membership and Information in 1986; and a member of the IEEE Fellow Committee from 1986 to 1988, among others. Since 1982, he has served as Guest Editor, Editor, and Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTERS. Three times he was elected a member of the Computer Society's Governing Board (1984–1990). Currently, he also serves as Chairman of the Steering Committee for the International Conference on Distributed Computing Systems, and Program Chairman for the Sixth International Conference on Data Engineering. He is also a member of the Association for Computing Machinery and Sigma Xi.