Conformity Analysis for Communication Protocols*

Nien-Chen Liu and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210

ABSTRACT

In this paper we propose a methodology for conformity analysis of communication protocols. By conformity analysis it is meant to demonstrate that a protocol does indeed provide the service for which it is intended. We specify both a protocol and its service by a CSP (Communicating Sequential Processes) based language. To perform the conformity analysis we develop a transformation system to extract from a CSP process the communication sequences that may arise during its execution, and to express these sequences in terms of behavior expressions in CCS (Calculus of Communicating Systems). By performing algebraic manipulations and the equivalence proof on these expressions, we can show that the external behavior of a protocol conforms to its intended service. A version of the Alternating Bit Protocol is used to demonstrate the feasibility of the methodology.

1. Introduction

The correctness issue of communication protocols has become the subject of a great deal of research in recent years. While many techniques have been developed to detect various syntactical errors which may be introduced in the protocol design phase, much work remains to be done on the analysis of protocol functional behavior.

*Research reported herein was supported by U. S. Army CECOM, Ft. Monmouth, NJ, under Contract No. DAAB07-83-K-K542. The views, opinions and/or findings contained in this paper are those of the authors and should not be construed as an official Department of the Army position, policy or decision.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

In the Open System Interconnection (OSI) Reference Model [3], the concepts of **Service** and **Protocol** are precisely defined. As illustrated in Fig.1, for any layer N , the **(N)-service** defines the logical interface between the users, which normally are the $(N+1)$ -entities, and the **(N)-service provider**, which consists of the layer N and the layers below. While a service defines the constraints on the interactions taking place at the boundary of two adjacent layers, it provides no information on how protocol entities are functioning inside a layer to provide the service. In this aspect, as illustrated in Fig.2, the **(N)-protocol** defines the behavior of the (N) -entities which communicate with each other through the $(N-1)$ -service provider, for providing the (N) -service.

Figure 1: The Service of layer N

Figure 2: The Protocol of layer N

A protocol is considered to be correct if it satisfies two kinds of requirements, namely, syntactic properties and functional objective. The syntactic properties are those desired properties common to all protocols such as freedom from deadlock, the absence of unexpected reception, and progress. They form the set of implicit requirements which any protocol should fulfill to ensure that its logical structure is free from syntactical errors. The absence of syntactical errors, however, does not necessarily imply that the protocol would attain its intended functional objective. In the context the OSI Reference Model, the task of functional analysis is to demonstrate that the protocol entities within one layer together with the underlying service provider can indeed work cooperatively to provide the intended service.

In this paper we focus mainly on the functional analysis for communication protocols. To analyze a protocol against its service, the service as well as the protocol should be formally specified. Here we use a CSP (Communicating Sequential Processes) [7] based language for specifying both protocols and services. By comparing Fig.1 and Fig.2, we can see that if the (N)-entities and (N-1)-service provider in Fig.2 are specified as individual processes, our task is to show that, after hiding all the internal communications, the integrated behavior of these processes with respect to the users should conform to the behavior of the process representing the (N)-service provider in Fig.1.

To perform the conformity analysis, we develop a transformation system to extract from a CSP process the communication sequences which may arise during its execution, and to express these sequences in terms of behavior expressions in CCS (Calculus of Communicating Systems) [11]. Based on this system, we are able to transform a set of cooperating CSP processes into a set of CCS expressions, and then derive the integrated behavior with respect to the environment by using the CCS composition and restriction operators.

Also, the conformity of the (N)-protocol to its service can be shown by proving that the CCS expression, representing the integrated behavior of the (N)-entities and (N-1)-service provider with respect to the users at the next layer, is observation equivalent to the CCS expression which represents the behavior of (N)-service provider.

The overall steps in conformity analysis are outlined in Fig.3. In the next section, the motivations of our specification and analysis methods are presented. Section 3 is devoted to the transformation system. In Section 4, we apply this approach to a version of the Alternating Bit Protocol. The conclusions are presented in Section 5.

Figure 3: An Overview of Conformity Analysis

2. Specification and Analysis Methods

2.1 Formal Specification:

The service specifications and protocol specifications represent two levels of abstraction in the OSI Reference Model. At the higher level, a service specification of a layer describes the externally visible service in terms of the valid sequences of the interactions taking place at the upper boundary. At the lower level, a protocol specification describes the logical implementation of a service in terms of the behavior of the protocol entities inside a layer. Due to their inherently distinct characteristics, past experience has shown that **sequence-oriented** specification techniques are more suitable for service specifications, whereas **state-oriented** specification techniques are better for protocol specifications. A good survey on a spectrum of various specification methods is in [13].

Despite that different description methods have been commonly used for service specifications and protocol specifications, a single specification technique to support both of them is in order. First, by using one technique, both the service and protocol specifications can be interpreted and analyzed on a common semantic basis. Second, a major task in the conformity analysis involves the composition of the protocol specification at one layer and the service specification at the lower layer. A uniform specification technique will facilitate this composition step.

In [10], we have used a CSP-based language for both the service and protocol specifications. The basic idea is summarized as follows:

- **To specify a service**, one or more CSP processes are used to describe the behavior of a service provider. Furthermore, these processes can only communicate with the processes which represent the service users. In this way, a CSP specification can be viewed as a **communication sequences generator** in the sense that a service is defined in terms of all the possible communication sequences which may arise during its execution.

- **To specify a protocol**, the entities are described by CSP processes which may communicate with the processes which represent the underlying service provider and the upper users. Typically, a control point in a process just before an input command reflects the major (or control) state of a protocol entity, whereas the variables are used to represent the “context variables” associated with a protocol entity, such as the messages. Therefore, a CSP specification can be viewed as a **state transition machine**. A detailed comparison between CSP and the extended finite state machine model [15] is presented in [10].

In short, our experience has shown that, by using CSP in a disciplined manner, one can make use of language constructs as mechanisms in generating a set of communication sequences corresponding to the allowable sequences of interactions of a service. On the other hand, one can also specify a protocol entity as a CSP process which resembles a state transition machine with the state space determined by the variables and a set of control points in the process. Therefore, even though CSP is a high level language, we can employ it as a unified method of sequence-oriented techniques and state-oriented techniques that can be used for both service and protocol specifications.

We assume that the reader is familiar with the CSP language [7]. Particularly, an **interaction** between two processes **P** and **Q** is the simultaneous execution of a **matching communication pair**:

$$\begin{array}{ccc} P?t(x) \text{ in process } Q & & P!t(e) \text{ in process } Q \\ Q!t(e) \text{ in process } P & \text{or} & Q?t(x) \text{ in process } P \end{array}$$

The effect of this interaction is to assign the value of expression e to variable x . The identifier t is called a constructor which serves as the mechanism to determine the matching communication pairs. Throughout this paper, we shall interpret a constructor as the **type** of an interaction. Furthermore, we impose that a constructor can appear in no more than two processes; thus, interactions of a certain type can only occur between a fixed pair of processes. Also, we use $t()$ to represent an t -type interaction in which the data passing is of no interest and has been omitted.

2.2 Conformity Analysis:

The purpose of the conformity analysis, as mentioned in the previous section, is to show that the composite behavior of the (N) -protocol specification and the $(N-1)$ -service specification with respect to the upper users conforms to the (N) -service specification. Consequently, any method for conformity analysis should be able to reason the communication behavior of a given specification, to integrate several specifications into an overall behavior, to hide the internal communications, and to demonstrate the equivalence of two communication behaviors.

For CSP, a number of proof systems have been proposed [1, 9, 14]. While each of them provides a different way to reason the CSP programs, they are all based on Hoare's axiomatic approach [6]. By these systems, one can prove that the behavior of a CSP program has some desired set of properties. However, for the purpose of conformity analysis, we decide not to take the axiomatic approach for the following reasons:

- Most of the axiomatic-based systems can deal with partial correctness of CSP programs only. In other words, they are used to prove that a certain properties would hold after the execution of a program, provided that it terminates. In contrast, we are interested in the communication sequence patterns presented by systems which usually involve infinite computations.
- In axiomatic-based systems, program behavior is described by a set of properties. It is hard to guarantee that the behavior of a program can be completely characterized via such properties; thus making it difficult to establish the equivalence of two program behaviors.
- The axiomatic-based systems aim at proving some desired properties of a CSP program, i.e. a set of processes communicating with each other. However, we are concerned with the external behavior of a set of processes which may interact with the environment.

Rather than taking the axiomatic approach, we develop a transformation system from CSP into CCS. Milner's CCS aims at describing the communication behaviors of concurrent systems in terms of a small set of operators. In addition, an associated mathematical theory is provided for reasoning about the communication behaviors. The reader is referred to the original CCS work [11], or an introductory paper in [5].

The main idea of our approach is that, by using the transformation system, we can extract the information on the communication behavior of a CSP process and express it in CCS behavior expressions. We choose CCS as the target language of the transformation system for the following reasons:

- CCS bears many similarities with CSP; thus making the transformation system quite straightforward. In particular, the concept of “interaction” is based on synchronous communication in both languages.
- Besides being an elegant notation in describing communication behavior, CCS provides a set of operators to manipulate communication behaviors. Especially, the composition operator can be used to derive the integrated behavior of a set of cooperating system components, while the restriction operator can be used for hiding internal communications.

- CCS is associated with a sound underlying theory in showing the equivalence of two communication behaviors – an essential activity in conformity analysis.

The transformation system is presented in the next section. In passing, we like to point out that the transformational approach [12] has been used for software development. That is, starting with a formal specification, a transformation process is performed for transforming the specification into an implementation. However, we use the transformational approach here for functional analysis. Through the transformation process, we can derive from the specifications the properties of communication behaviors in terms of algebraic expressions.

3. Transformation System

The purpose of the transformation system, as mentioned earlier, is to extract the information about the communicating behavior of processes. A set of transformation rules is used such that, for a given CSP process, the corresponding CCS behavior expression, which characterizes its communication sequences, is generated.

3.1 Perspective View

The basic ideas of the transformation system are as follows:

1) For individual processes,

- The variables $x_1, x_2, \dots, x_n$ (denoted as a vector $\tilde{x}$) of a CSP process are mapped into the parameters of the corresponding CCS expressions. In the rest of this paper, a state is referred to as either the values of variables of a CSP process or the values of the parameters of a CCS expression, depending on the context.

DEFINITION 1: for a process with variables $x_1, x_2, \dots, x_n$,

$$\tilde{x} = \langle x_1, x_2, \dots, x_n \rangle$$

i.e. $\tilde{x}$ is a state variable.

- The transformation rule, from a CSP command C (in a process with $\tilde{x}$ as its variables) to a CCS expression $\mathcal{E}$, is in the form of: $\llbracket C \rrbracket(\tilde{x}) = \mathcal{E}$

For simple commands such as assignment, input/output, or skip command, the general form of their corresponding CCS expressions is as follows :

GENERAL FORM:

$$\langle \text{action} \rangle . \bar{\alpha} \tilde{x} . \text{Nil}$$

That is, the effect of a simple statement is to perform the action part, followed by the output of the resultant state, and finally ends its life.

- For the sequential composition of two CSP commands $C_1; C_2$, we use a derived CCS operator “before” to link the CCS expressions derived from C_1 and C_2 .

DEFINITION 2: for two behavior expressions $\mathcal{E}_1$ and $\mathcal{E}_2$,

$$\mathcal{E}_1 \text{ before } \mathcal{E}_2 = (\mathcal{E}_1[\beta/\alpha] \mid \beta \tilde{x}'. \mathcal{E}_2) \setminus \beta$$

Based on this operator, the sequential composition rule can be expressed as follows:

Sequential Composition Rule :

$$\llbracket C_1 ; C_2 \rrbracket(\tilde{x}) = \llbracket C_1 \rrbracket(\tilde{x}) \text{ before } \llbracket C_2 \rrbracket(\tilde{x}')$$

Intuitively speaking, the “before” operator in the sequential composition rule is used for the following purposes:

- To create a binding for the state variable $\tilde{x}'$ by $\beta \tilde{x}'$ (recall the definition of before operator). As we will see below, the variable $\tilde{x}'$ is bound to the resultant state after performing the actions of $\llbracket C_1 \rrbracket(\tilde{x})$.
- To synchronize the actions of $\llbracket C_1 \rrbracket(\tilde{x})$ and $\llbracket C_2 \rrbracket(\tilde{x}')$, the action part of $\llbracket C_1 \rrbracket(\tilde{x})$ is first performed; then the resultant state is delivered to $\tilde{x}'$ via β communication. (recall that an $\bar{\alpha} \tilde{x}$ action is expected to be at the end of $\llbracket C_1 \rrbracket(\tilde{x})$, and its name has been changed to β by the relabeling operation $[\beta/\alpha]$); and finally $\llbracket C_2 \rrbracket(\tilde{x}')$ is performed.
- To hide the state-passing communication by the restriction operation $\setminus \beta$.

In short, the sequential composition rule states that, the behavior of $C_1; C_2$ at state $\tilde{x}$ is to first perform the action part of $\llbracket C_1 \rrbracket(\tilde{x})$, then pass the resultant state to $\tilde{x}'$ “secretly”, and behave as $\llbracket C_2 \rrbracket(\tilde{x}')$.

2) For a set of cooperating processes,

The integrated behavior of a set of processes is obtained by applying the parallel composition operation $\mid$ to a set of CCS expressions derived from individual processes, and at the same time by hiding all the internal communications between the processes. Consequently, the resultant expression would characterize all the sequences of communications with respect to the environment, i.e. externally observable behavior.

Formally, let $P_1, P_2, \dots, P_n$ be a set of processes with $\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_n$ as their state vectors, respectively. The externally observable behavior can be derived by the following rule:

Parallel Composition Rule :

$$\begin{aligned} & \llbracket P_1 \parallel P_2 \parallel \dots \parallel P_n \rrbracket(\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_n) \\ &= \bigcup_{i=1, \dots, n} \llbracket P_i \rrbracket(\tilde{x}_i) \setminus \text{internal} \end{aligned}$$

where *internal* is

$$\{t \mid t \text{ is a name in } \llbracket P_l \rrbracket \text{ and } \llbracket P_m \rrbracket, l, m \in 1, \dots, n\} \cup \{\alpha\}$$

3.2 Transformation Rules

In the following, the transformation rules for the CSP commands are presented, each of which is followed by a brief explanation.

Skip Command :

$$\llbracket \text{Skip} \rrbracket(\tilde{x}) = \bar{\alpha} \tilde{x} . \text{Nil}$$

The **skip** command is a null command. Consequently, there is no action part in its corresponding expression (recall the general form of simple commands). The only thing needs to do is to output the value of the current state and ends its life.

Assignment Command :

$$\llbracket x_i := e \rrbracket(\tilde{x}) = \bar{\alpha} \tilde{x} \{e/x_i\} . \text{Nil}$$

The assignment command assigns the value of expression e to the variable x_i . Here, the effect of state change is achieved by $\bar{\alpha} \tilde{x} \{e/x_i\}$, which denotes the output of current state with the exception that x_i is replaced by the value of e .

Input Command :

$$\llbracket P_l ? t(x_i) \rrbracket(\tilde{x}) = t x_i . \bar{\alpha} \tilde{x} . \text{Nil}$$

The command $P_l ? t(x_i)$ (say, in process P_m) expresses the intention to communicate with $P_m ! t(e)$ in process P_l . The CCS expression for input command denotes that the actions performed are the t -type input, followed by the state-passing output.

It should be noticed that, the variable x_i in $\bar{\alpha} \tilde{x}$ is bounded by $t x_i$; thus, its value sent by $\bar{\alpha} \tilde{x}$ is the one received by $t x_i$. Also, as mentioned in Section 2, we impose that the constructor t can occur in a fixed pair of processes only; this convention will ensure the proper communications during the parallel composition step even though we drop the process name in this rule.

Output Command :

$$\llbracket P_l ! t(e) \rrbracket(\tilde{x}) = \bar{t} e . \bar{\alpha} \tilde{x} . \text{Nil}$$

For an output command, the action is to output the value of expression e through a t -type communication, and then to output the unchanged state. Again, the communication pairs can be determined solely by the type names later in the parallel composition step, so that we can drop the process name here.

Abort :

$$\llbracket \text{abort} \rrbracket(\tilde{x}) = \omega \quad \text{where } \omega \Leftarrow \tau . \omega$$

The **abort** is not a CSP command, but a behavior to model the failure situation of the alternative command. It is specified as a recursive expression ω which does nothing but performing meaningless τ -actions an infinite number of times.

Alternative Command :

$$\begin{aligned} & \llbracket [G_1 \rightarrow C_1 \parallel G_2 \rightarrow C_2 \parallel \dots \parallel G_n \rightarrow C_n] \rrbracket(\tilde{x}) = \\ & (\sum_i \text{if } b_i(\tilde{x}) \text{ then } \tau . \llbracket C_i \rrbracket(\tilde{x}) \text{ else Nil} \\ & \quad \text{where } G_i = b_i \\ & + \\ & \sum_j \text{if } b_j(\tilde{x}) \text{ then } \llbracket P_l ? t(x_i); C_j \rrbracket(\tilde{x}) \text{ else Nil} \\ & \quad \text{where } G_j = b_j; P_l ? t(x_i) \\ & + \\ & \sum_k \text{if } b_k(\tilde{x}) \text{ then } \llbracket P_l ! t(e); C_k \rrbracket(\tilde{x}) \text{ else Nil} \\ & \quad \text{where } G_k = b_k; P_l ! t(e) \\ & - \\ & \text{if } \neg(\bigvee_i b_i) \text{ then } \llbracket \text{abort} \rrbracket(\tilde{x}) \text{ else Nil} \\ & \quad \text{where } \bigvee_i b_i = b_1 \vee \dots \vee b_n \\ &) \end{aligned}$$

For an alternative command, there is an if-then-else term for each guarded command, representing one of the many possible behaviors. By the definition of the if-then-else construct in CCS, the value of the boolean condition determines the actual behavior. That is, if the boolean condition is true, the whole expression can be textually replaced by the then-part, otherwise the else-part.

Thus, the rule for an alternative command can be explained as follows:

- 1) The boolean parts of all the guarded commands are evaluated first.
  - (a) For any guarded command with the boolean part being false, its corresponding CCS term is Nil, which means it contributes nothing to the actual behavior (for any CCS expression $\mathcal{E}$, $\mathcal{E} + \text{Nil} = \mathcal{E}$).
  - (b) For any guarded command with the boolean part being true, and its guard is:
    - (i) Boolean guard:
Its corresponding CCS term is a τ -action, which models the potential commitment of executing this guarded command, followed by the expression derived for its command list.
    - (ii) I/O guard:
Its corresponding CCS term is the expression derived from the sequential composition of input/output command in its guard and its command list.
  - (c) If all the boolean parts are evaluated to be false, the alternative command fails. As shown in the transformation rule, the failure situation is modeled as the behavior of **abort**.
- 2) The CCS expression for the alternative command is the sum of the terms derived in step 1). Each of these terms represents the potentially actual behavior of the alternative command.

Repetitive Command :

$$\begin{aligned}
 & \llbracket * [G_1 \rightarrow C_1 \square G_2 \rightarrow C_2 \square \dots \square G_n \rightarrow C_n] \rrbracket(\tilde{x}) = \omega(\tilde{x}) \\
 & \omega(\tilde{x}) \Leftarrow \\
 & (\sum_i \text{if } b_i(\tilde{x}) \text{ then } \tau_i \llbracket C_i \rrbracket(\tilde{x}) \text{ before } \omega(\tilde{x}') \text{ else Nil} \\
 & \quad \text{where } G_i = b_i \\
 & + \\
 & \sum_j \text{if } b_j(\tilde{x}) \text{ then } \llbracket P_i?t(x_i); C_j \rrbracket(\tilde{x}) \text{ before } \omega(\tilde{x}') \text{ else Nil} \\
 & \quad \text{where } G_j = b_j; P_i?t(x_i) \\
 & + \\
 & \sum_k \text{if } b_k(\tilde{x}) \text{ then } \llbracket P_i!t(e); C_k \rrbracket(\tilde{x}) \text{ before } \omega(\tilde{x}') \text{ else Nil} \\
 & \quad \text{where } G_k = b_k; P_i!t(e) \\
 & + \\
 & \text{if } \neg(\bigvee b_i) \text{ then } \llbracket \text{skip} \rrbracket(\tilde{x}) \text{ else Nil} \\
 & \quad \text{where } \bigvee b_i = b_1 \vee \dots \vee b_n \\
 &)
 \end{aligned}$$

For a repetitive command, its behavior is presented as a recursive expression $\omega(\tilde{x})$, which is similar to the expression for an alternative command except that:

- 1) After one of its terms is performed, the resultant state will be passed to $\tilde{x}'$, and then it behaves like $\omega(\tilde{x}')$.
- 2) At the time when all guards fail, the behavior is the same as a skip command.

In this rule, we exclude the distributed termination convention in the original CSP. In other words, a repetitive command can terminate only when all the boolean parts of its guarded commands become false. Another version of the rule which considers the distributed termination has been established, but it is irrelevant in this paper.

3.3 Examples

In the following, two simple examples are given for showing how a transformation process is conducted.

[Example 1]: A binary semaphore, shared by two users User1 and User2, can be specified as the following CSP process:

$$\begin{aligned}
 \text{Sema}:: & * [\text{User1?P}_1() \rightarrow \text{User1?V}_1() \\
 & \square \\
 & \text{User2?P}_2() \rightarrow \text{User2?V}_2()]
 \end{aligned}$$

To transform it into a CCS behavior expression, we have:

$$\begin{aligned}
 \llbracket \text{Sema} \rrbracket &= \text{SEMA} \\
 \text{SEMA} &\Leftarrow \llbracket \text{User1?P}_1(); \text{User1?V}_1() \rrbracket \text{ before SEMA} \\
 &+ \\
 &\llbracket \text{User2?P}_2(); \text{User2?V}_2() \rrbracket \text{ before SEMA} \\
 &= (\llbracket \text{User1?P}_1() \rrbracket \text{ before } \llbracket \text{User1?V}_1() \rrbracket) \text{ before SEMA} \\
 &+ \\
 &(\llbracket \text{User2?P}_2() \rrbracket \text{ before } \llbracket \text{User2?V}_2() \rrbracket) \text{ before SEMA} \\
 &= (P_1.\bar{\alpha}.\text{Nil}[\beta/\alpha] \mid \beta.V_1.\bar{\alpha}.\text{Nil}) \setminus \beta \text{ before SEMA} \\
 &+ \\
 & (P_2.\bar{\alpha}.\text{Nil}[\beta/\alpha] \mid \beta.V_2.\bar{\alpha}.\text{Nil}) \setminus \beta \text{ before SEMA} \\
 &= P_1.\tau.V_1.\bar{\alpha}.\text{Nil} \text{ before SEMA} \\
 &+ \\
 &P_2.\tau.V_2.\bar{\alpha}.\text{Nil} \text{ before SEMA} \\
 &= (P_1.\tau.V_1.\bar{\alpha}.\text{Nil}[\beta/\alpha] \mid \beta.\text{SEMA}) \setminus \beta \\
 &+ \\
 &(P_2.\tau.V_2.\bar{\alpha}.\text{Nil}[\beta/\alpha] \mid \beta.\text{SEMA}) \setminus \beta \\
 &= P_1.\tau.V_1.\tau.\text{SEMA} + P_2.\tau.V_2.\tau.\text{SEMA} \\
 &\approx P_1.V_1.\text{SEMA} + P_2.V_2.\text{SEMA}
 \end{aligned}$$

This expression describes the binary semaphore as a communicating agent which engages in either a pair of P_1, V_1 interactions or a pair of P_2, V_2 interactions, on each cycle. Notice that in this example $\bar{\alpha}$ is solely for the synchronization purpose, since there are no variables in the process.

[Example 2: A process **T**(eacher), which communicates with process **S**(tudent), is specified below. The label **L** is used to stand for the inner repetitive command.

```

T::
*{ S?ask() → done := false;
  L: *{¬done; S!answer() →
    { S?ok() → done:=true
    }
    S?Nok() → skip
  }
}

```

In this process, the state is the value of variable “done”. The transformation procedure for process **T** is presented as follows (we skip the intermediate steps when they are obvious) :

Let $\llbracket \mathbf{T} \rrbracket(\text{done}) = \tau(\text{done})$

$$\begin{aligned} \tau(\text{done}) &\Leftarrow \llbracket \mathbf{S}?ask(); \text{done}:=\text{false}; \mathbf{L} \rrbracket(\text{done}) \text{ before } \tau(\text{done}') \\ &= \text{ask.} \tau. \tau. \llbracket \mathbf{L} \rrbracket(\text{false}) \text{ before } \tau(\text{done}') \quad (1) \end{aligned}$$

It should be clear that $\tau(\text{true})$ is same as $\tau(\text{false})$, since the behavior of process **T** doesn't depend on the initial value of “done” which can be true, false, or (in fact) undefined. Later we will use τ to represent the behavior of process **T** in the final step of the transformation procedure.

Let $\llbracket \mathbf{L} \rrbracket(\text{done}) = \mathcal{L}(\text{done})$

By the transformation rule of the repetitive command, we get:

$$\mathcal{L}(\text{true}) = \llbracket \text{skip} \rrbracket(\text{true})$$

$$\begin{aligned} \mathcal{L}(\text{false}) &\Leftarrow \overline{\text{answer}}. \tau. (\llbracket \mathbf{S}?ok(); \text{done}:=\text{true} \rrbracket(\text{false}) \\ &\quad + \llbracket \mathbf{S}?Nok(); \text{skip} \rrbracket(\text{false})) \text{ before } \mathcal{L}(\text{done}') \\ &= \overline{\text{answer}}. \tau. (\text{ok.} \tau. \bar{\alpha} \text{true. Nil} \\ &\quad + \text{Nok.} \tau. \bar{\alpha} \text{false. Nil}) \text{ before } \mathcal{L}(\text{done}') \\ &= \overline{\text{answer}}. \tau. (\text{ok.} \tau. \tau. \mathcal{L}(\text{true}) \\ &\quad + \text{Nok.} \tau. \tau. \mathcal{L}(\text{false})) \\ &= \overline{\text{answer}}. \tau. (\text{ok.} \tau. \tau. \llbracket \text{skip} \rrbracket(\text{true}) \\ &\quad + \text{Nok.} \tau. \tau. \mathcal{L}(\text{false})) \end{aligned}$$

Let $\llbracket \mathbf{L} \rrbracket(\text{false}) \text{ before } \tau(\text{done}') = \mathcal{W}$ (2)

$$\begin{aligned} \mathcal{W} &\Leftarrow \overline{\text{answer}}. \tau. (\text{ok.} \tau. \tau. \llbracket \text{skip} \rrbracket(\text{true}) \\ &\quad + \text{Nok.} \tau. \tau. \mathcal{L}(\text{false})) \text{ before } \tau(\text{done}') \end{aligned}$$

$$\begin{aligned} &= \overline{\text{answer}}. \tau. (\text{ok.} \tau. \tau. \bar{\alpha} \text{true} \text{ before } \tau(\text{done}') \\ &\quad + \text{Nok.} \tau. \tau. \mathcal{L}(\text{false}) \text{ before } \tau(\text{done}')) \\ &= \overline{\text{answer}}. \tau. (\text{ok.} \tau. \tau. \tau. \tau(\text{true}) + \text{Nok.} \tau. \tau. \mathcal{W}) \\ &\approx \overline{\text{answer}}. (\text{ok.} \tau(\text{true}) + \text{Nok.} \mathcal{W}) \quad (3) \end{aligned}$$

From (1), (2), (3), we can finally get:

$$\llbracket \mathbf{T} \rrbracket(\text{done}) = \tau$$

$$\tau \approx \text{ask.} \mathcal{W}$$

$$\mathcal{W} \Leftarrow \overline{\text{answer}}. (\text{ok.} \tau + \text{Nok.} \mathcal{W})$$

The expressions above can characterize the communication behavior of process **T**. In words, **T** first receives an **ask** signal and responds with an **answer** signal. It then waits for an input signal. If the signal is **ok**, a cycle is completed and **T** will be ready for another **ask** signal. On the other hand, if the signal is **Nok**, **T** will retransmit the **answer** signal and wait for another incoming signal.

4. Alternating Bit Protocol

In this section, a version of the alternating bit protocol [2] is used as an example to illustrate how we can perform the conformity analysis based on the transformation system discussed in the previous section.

As shown in Fig.4, two protocol entities **S** and **R**, together with a transmission medium **M** (which is the underlying service provider), form a communication system for **User1** and **User2**. In this version [2], it is assumed that due to occasional malfunction of transmission medium, data may be corrupted but cannot be lost during transmission*. The protocol entities **S** and **R**, using a control bit as the error recovery mechanism, work together to guarantee the reliable data transfer from **User1** to **User2**.

In Section 4.1 the intended service as well as the protocol entities and transmission medium are formally specified. In Section 4.2 we demonstrate the conformity analysis based on the transformation system.

Figure 4: ABP Communication System

*The reader is referred to [10] for the version considering message loss.

4.1 Service and Protocol Specifications

1) Service Specification :

Service :: *[User1?GET() → User2!OUT()]

From the users' point of view, ABP provides a reliable one-way data transfer service from **User1** to **User2**. The whole communication system can be described as a process **Service**. It engages in the communication GET() each time when **User1** offers a new message, followed by the communication OUT() when the message is delivered to **User2**. Here, we are only interested in the types of communications; the contents of the messages are irrelevant and have been omitted.

2) Protocol Entity S :

```
S ::
bit := 0;
*[ User1?GET() →
  done := false;
  | bit = 0 →
    *[-done; M!D0() →
      { M?a0() → done:=true;
        bit:=1
        | M?a1() → skip
        | M?ae() → skip
        }
    |
  | bit = 1 →
    *[-done; M!D1() →
      { M?a1() → done:=true;
        bit:=0
        | M?a0() → skip
        | M?ae() → skip
        }
    ]
  ]
]
```

S is the protocol entity which interacts with **User1** through the communication GET(). After getting a new message, **S** sends it one or more times through the transmission medium until an appropriate acknowledgment is received. Depending on the value of the control bit, a message may be sent through a communication of type D0 or D1, each of which represents that the associated sequence number of the message is 0 or 1, respectively. The input communications a0, a1, and ae represent the reception of an acknowledgment with sequence number 0, an acknowledgment with sequence number 1, and an erroneous acknowledgment, respectively. Again, we shall ignore the contents of the messages throughout this section. The control information such as the sequence number and error condition is encoded into the type of a communication.

3) Transmission Medium M :

M ::

```
*[ S?D0() → {true → R!d0() | true → R!de() |
  |
  S?D1() → {true → R!d1() | true → R!de() |
  |
  R?A0() → {true → S!a0() | true → S!ae() |
  |
  R?A1() → {true → S!a1() | true → S!ae() |
  ]
]
```

As mentioned earlier, the transmission medium may occasionally corrupt the message under transmission. Accordingly, when **M** gets a message with sequence number 0 or 1 from **S** through D0() or D1(), it may deliver the message to **R** through d0() or d1(), or it may inform **R** through de() that a transmission error has occurred. Similar behavior is presented for the transmission of acknowledgments from **R** to **S**.

4) Protocol Entity R :

```
R ::
exp := 0;
*[ exp = 0 → | M?do() → User2!OUT();
  | M!A0();
  | exp := 1
  | M?d1() → M!A1()
  | M?de() → M!A1()
  |
  | exp = 1 → | M?d1() → User2!OUT();
  | M!A1();
  | exp := 0
  | M?d0() → M!A0()
  | M?de() → M!A0()
  ]
]
```

The protocol entity **R** is responsible for receiving the messages from the transmission medium. For each reception, **R** delivers a message to **User2** if it has the expected sequence number; and then replies **S** with an appropriate acknowledgment. Otherwise, it only sends back an acknowledgment with the opposite value of the expected sequence number.

4.2 Conformity Analysis

To perform the conformity analysis, we first derive the communication behaviors of the service, the protocol entities, and the transmission medium from their specifications. Due to the space limitation, we skip all the transformation steps and present the final results only.

1) Service :

$$\llbracket \text{Service} \rrbracket = \text{SERVICE}$$

$$\text{SERVICE} \Leftarrow \text{GET}. \overline{\text{OUT}}. \text{SERVICE}$$

$$(S|M|R) \setminus \{D0, D1, d0, d1, de, A0, A1, a0, a1, ae, \alpha\} = ABP$$

$$ABP = \text{GET}. \mathcal{W} \quad (1)$$

$$\mathcal{W} \Leftarrow \tau. \overline{\text{OUT}}. (\tau. ABP + \tau. \text{TRAP}) + \tau. \mathcal{W} \quad (2)$$

$$\text{TRAP} \Leftarrow \tau. ABP + \tau. \text{TRAP} \quad (3)$$

2) Protocol Entity S :

$$\llbracket S \rrbracket(\text{bit}, \text{done}) = S$$

$$S = \tau. S_0$$

$$S_0 = \text{GET}. \tau_0$$

$$\tau_0 \Leftarrow \overline{D0}. (a0. S_1 + a1. \tau_0 + ae. \tau_0)$$

$$S_1 = \text{GET}. \tau_1$$

$$\tau_1 \Leftarrow \overline{D1}. (a1. S_0 + a0. \tau_1 + ae. \tau_1)$$

From the definition of observation equivalence, we can prove by induction that (the proof is skipped):

$$\text{SERVICE} \approx ABP$$

At this point we can conclude that the integrated external behavior, presented by the sending entity S , the transmission medium M , and the receiving entity R , is observation equivalent to its intended service.

It should be noted, however, the behavior expression ABP may perform an infinite number of τ -actions due to the terms $\tau. \mathcal{W}$ in (2) and $\tau. \text{TRAP}$ in (3). By examining the steps of parallel composition, it can be readily found that the presence of $\tau. \mathcal{W}$ is due to the occurrence of de interaction, i.e. a transmission error has occurred while a message is in transit. Similarly, the presence of $\tau. \text{TRAP}$ is due to the occurrence of ae interaction, i.e. a transmission error has occurred while an acknowledgment is in transit. Indeed, when looking at the specification of the transmission medium, one can see that a transmission error can occur nondeterministically, and nothing prevents it from being malfunctioning persistently.

3) Transmission Medium M :

$$\llbracket M \rrbracket = M$$

$$\begin{aligned} M \Leftarrow & D0. (\tau. \overline{d0}. M + \tau. \overline{de}. M) \\ & + D1. (\tau. \overline{d1}. M + \tau. \overline{de}. M) \\ & + A0. (\tau. \overline{a0}. M + \tau. \overline{ae}. M) \\ & + A1. (\tau. \overline{a1}. M + \tau. \overline{ae}. M) \end{aligned}$$

4) Protocol Entity R :

$$\llbracket R \rrbracket(\text{exp}) = R$$

$$R = \tau. R_0$$

$$\begin{aligned} R_0 \Leftarrow & d0. \overline{\text{OUT}}. \overline{A0}. R_1 \\ & + d1. \overline{A1}. R_0 \\ & + de. \overline{A1}. R_0 \end{aligned}$$

$$\begin{aligned} R_1 \Leftarrow & d1. \overline{\text{OUT}}. \overline{A1}. R_0 \\ & + d0. \overline{A0}. R_1 \\ & + de. \overline{A0}. R_1 \end{aligned}$$

To prove that the external behavior of the alternating bit protocol conforms to its intended service, we apply the parallel composition rule on the CCS expressions derived from S , M , and R .

In fact, it is easier to see the situation when we draw the transition graphs for both SERVICE (Fig.5) and ABP (Fig.6). Both graphs show that the externally observable behavior is an indefinite repetition of a pair of $\text{GET}; \overline{\text{OUT}}$ communications. However, the transition graph of ABP has a τ -loop at $\mathcal{W}$, corresponding to the $\tau. \mathcal{W}$ term in (2); and a τ -loop at TRAP , corresponding to the $\tau. \text{TRAP}$ term in (3). The former τ -loop reflects the potential error situation during the transmission from sending entity to receiving entity, while the latter reflects the transmission error in the opposite direction.

As stated in [11], to make the CCS observation equivalent relation sensitive to infinite τ -actions, one has to deal with the issue of fairness. Here, we leave the responsibility of reasoning the infinite τ -actions to the prover. In the case of alternating bit protocol, it should be clear that, under the assumption that the medium cannot permanently corrupt messages or acknowledgments, infinite τ -actions can never occur.

Figure 5: *SERVICE* Graph

Figure 6: *ABP* Graph

5. Concluding Remarks

In this paper a technique has been presented for the analysis of communication protocols against their services. CSP is used as a common specification language for both services and protocols. Based upon a transformation system, the communication behaviors presented by the service and protocol specifications can be derived and expressed in CCS behavior expressions, on which the functional analysis is subsequently conducted.

It can be argued that CCS or its derived languages (such as LOTOS [8]) have been used as specification languages; then why do we need to use CSP as the specification language, and later transform the specifications into CCS expressions? In our experience, while a CCS-like language can express many communication behaviors elegantly, the complexity of the specifications grows quickly when systems become large. In particular, due to its lack of state component (which is realized by process variables in CSP), and its lack of iteration construct (which is realized by the repetitive command in CSP), the specifications are usually presented as many instantiations of parameterized behaviors which are defined in a mutually recursive fashion. This may greatly reduce the comprehensibility of the specifications to the implementors. Furthermore, while a CCS-like language can specify services reasonable well, it may be awkward to specify protocols (in contrast to using extended finite state machines or CSP), as some of the information can fit naturally into the state components or variables. The observations above led us to feel that CCS-like languages should be used more as an analysis tool rather than a specification language.

In fact, in our approach, we can view the specifications based on CSP as **operational specifications**, and the derived CCS expressions as more **abstract specifications** showing only the observable behaviors. We believe that they can complement each other in the protocol design phase — operational specifications are easier to **construct** and to **understand**; whereas abstract specifications are more **amenable to analysis**.

The transformation system from CSP to CCS is quite straightforward and syntax-directed. For a given CSP process, the system allows us to suppress the local computations and deal with its communication behavior only. Especially, for a process which performs cyclic operations, the derived CCS expression can serve as an “invariant” property on its communication behavior. This gives some advantage over axiomatic proof systems, since by using these systems, it is the prover who has the responsibility to elaborate the invariant properties for cyclic computations — a heavy burden.

In [4], Hennessy, Li, and Plotkin proposed a technique for translating CSP into CCS. Their approach is different from ours in many ways. Most importantly, they aimed at translating a CSP program into a CCS expression with the same States $\times$ States relation, i.e. given an initial state, both the CSP program and the CCS expression would reach the same final state when they terminate. In contrast, we are more concerned with the communication behavior exhibited during the execution of CSP processes, rather than the States $\times$ States relation. In fact, as shown in this paper, we aim at studying the communication behaviors of the systems which may run forever (thus there is no final state), and may communicate with not only the other system components but the environment (thus they are not “programs”). Moreover, in their approach guards and commands are translated into “CCS contexts” — CCS terms with holes in them. The CCS term for two consecutive commands is obtained by filling the term of the second command into the hole in the term of the first command. We transform each CSP command into a CCS expression; and the expression for two consecutive commands is derived by linking two expressions via a special β communication. Finally, each variable in a CSP process is modeled by an individual CCS expression in their approach; on the other hand, we treat the variables of CSP processes as the parameters of the corresponding CCS expressions.

While we mainly focus on communication systems in this paper, the approach presented here can, we believe, be applied to perform functional analysis of other distributed systems. More work is needed in this direction.

ACKNOWLEDGMENT

The authors are grateful to Neelam Soundararajan for his many valuable discussions and comments while this work was in progress.

REFERENCES

- [1]. Apt, K. R., Francez, N., and De Roever, W. P. "A Proof System for Communicating Sequential Processes". *ACM TOPLAS* 2(3): 359-385, July 1980.
- [2]. Bochmann, G. V. "Communication Protocols and Error Recovery Procedures". *ACM SIGOPS Oper. Syst. Rev.* 9: 45-50, March 1975.
- [3]. Day, J. D., and Zimmermann H. "The OSI Reference Model". *Proceedings of the IEEE* 71(12): 1334-1340, June 1983.
- [4]. Hennessy, M., Li, W., and Plotkin, G. "A First Attempt at Translating CSP into CCS". *Proc. 2nd International Conference on Distributed Computing Systems*, pp. 105-115, April 1981.
- [5]. Hennessy, M. "An Introduction to a Calculus of Communicating Systems". In *New Advances in Distributed Computer Systems*, K.G. Beauchamp (ed.), D. Reidel Publishing Company, 1982, pp. 245-262.
- [6]. Hoare, C.A.R. "An Axiomatic Basis for Computer Programming". *Communications ACM* 12: 576-583, 1969.
- [7]. Hoare, C.A.R. "Communicating Sequential Processes". *Communications ACM* 21(8): 666-677, August 1978.
- [8]. ISO. Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behavior, ISO/DP 8807, March 1985.
- [9]. Levin, G. M., and Gries, D. "A Proof Technique for Communicating Sequential Processes". *Acta Inf* 15: 281-302, 1981.
- [10]. Liu, N. C. and Liu, M. T. "CSP-based Specifications for Network Protocols and Services". *IEEE Computer Networking Symposium*: 95-102, December 1984.
- [11]. Milner, R.. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer-Verlag, 1980.
- [12]. Partsch H. and Steinbruggen R. "Program Transformation Systems". *ACM Computing Surveys* 15(3): 199-236, September 1983.
- [13]. Schwartz, R. L. and Melliar-Smith, P. M. "From State Machines to Temporal Logic: Specification Methods for Protocol Standards". *IEEE Trans. on Communications COM-30(12)*: 2486-2496, December 1982.
- [14]. Soundararajan, N. "Axiomatic Semantics of Communicating Sequential Processes". *ACM Trans. Programm. Lang. Syst.* 6(4): 647-662, October 1984.
- [15]. Tenney, R. L. "One Formal Description Technique For ISO OSI". *Proceedings of the International Conference on Communications*, pp.1296-1300, June 1983.