FORMAL METHODS FOR PROTOCOL CONVERSION*

Kenneth L. Calvert and Simon S. Lam

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-23 August 1989

ABSTRACT

We consider ways of overcoming a *protocol mismatch* using *protocol conversion.* Three different methods for finding a protocol converter are described. Two of these are "bottom-up" in nature, and involve relating the conversion system to existing protocols. The third approach, which is new, is "top-down:" the desired *global* properties of the conversion system are used in deriving the converter. An example is used to illustrate each method. We discuss more general forms of the abstract problem in the context of layered network architectures.

Keywords: internetworking, communication protocols, protocol mismatch, protocol converter, specification, verification, quotient problem.

* To appear in *IEEE Journal on Selected Areas in Communications.* Work supported by the National Science Foundation under grant number NCR-8613338.

1 Introduction

Computer communication networks today practically span the globe. Yet achieving useful communication between programs residing in different computer systems remains a nontrivial problem. Often this is because the systems are designed to communicate using different *protocols*: the form and meaning of the messages they send are governed by different sets of rules and procedures. In Figure 1, system P_0 is designed to communicate with system P_1 using protocol P, while Q_0 and Q_1 are designed to use protocol Q. When P_0 needs to interact with Q_1, a *protocol mismatch* exists.

The existence of different protocols to perform the same function is a fact of life that is unlikely to change. One reason for this is the large installed base of systems from various manufacturers, whose different protocol architectures were developed prior to the definition of adequate "open system" standards.[1] Another reason is that communication protocols evolve with technology. In other words, we are still learning how to build networks, and we will continue to learn. As new protocols replace old ones, several "generations" of architecture may coexist at any time, and upward compatibility may eventually be sacrificed for superior performance. Still another reason, noted in [12], is the desirability of having different protocols for the same general purpose, to serve the needs of different user communities. For example, a protocol optimized for transfer of bulk data over long-haul networks will differ from one intended for transfer of interactive terminal session data over the same networks [7]. For these and other reasons, convergence to a single protocol architecture is likely to take a long time, if indeed it ever occurs.

The most obvious solution to the problem of Figure 1 is to modify P_0 or Q_1, or both, to achieve compatibility. This may in fact be the best solution in some cases. However, in general it is tantamount to convergence to a single architecture, and therefore we seek other solutions. If we cannot modify P_0 or Q_1, some form of *translation* between protocols would seem to be the best alternative. Figure 2 shows an intermediary called a *protocol converter*, which translates messages sent by P_0 into messages of protocol Q, forwards them to Q_1, and performs a similar translation in the other direction. Protocol converters of this kind have been mentioned in the literature, where they are sometimes called "gateways" [10, 14, 35]. We use the term *protocol conversion* to refer to the general approach of using translation to solve protocol mismatch problems.

Green [13] considered the general problem of protocol conversion and thoroughly examined many of its practical aspects. He pointed out that no general solution methodology is known, and suggested that the formal methods used in specification and verification of protocols might form the basis for "a deeper and more systematic calculus of conversion." Since then, some approaches based on formal methods have been proposed [21, 31]. In this paper, we consider the application of formal methods to the problem of finding a protocol converter. After formalizing the problem, we discuss and compare three methods that might constitute part of a "calculus of conversion." The approaches of Lam [21] and Okumura [31] can be seen as "bottom-up," heuristic methods; the third, a new approach, is "top-down"

[1]Indeed, some might say that adequate standards do not yet exist.

and algorithmic, but computationally hard.

Note that it is *not* our intent to advocate protocol conversion as the preferred solution or only solution to protocol mismatch problems. Rather, it is our hope that a precise definition of the problem and experience with various solution methods will enable classification of protocol mismatch problems according to whether conversion is a reasonable solution.

The rest of this paper is organized as follows. In Section 2, we formalize the problem and introduce a simple example. In Sections 3, 4, and 5 we present three solution methods, and apply each to our example problem. In Section 6 we consider the problem as it may actually arise in the context of layered network interconnection. Section 7 contains some conclusions.

2 Formalizing the Problem

For the purposes of this paper, a *formal method* for specification and verification of a protocol system has three parts:

- A way of precisely describing the components of an *implementation* of the protocol and how they interact. In the context of a layered network architecture, the components of an implementation include the protocol "peer entities," and the lower-level services they use.
- A way of defining the *correct behavior* of the protocol system. For a protocol in a layered architecture, its desired correct behavior is often specified in the form of a *service* to be provided to the users of the protocol system.
- A definition of what it means for a specified implementation to *satisfy* a correctness specification, i.e., a *semantics*.

If there is to be any systematic and general approach to protocol conversion, we must abstract from details of the protocols and their function. Exploring the problem within a formal framework of this kind enables us — indeed, forces us — to take intuitive notions such as "achieving useful communication" and "protocol incompatibility" and make them precise and rigorous. Formal methods are also advocated as a way of managing the inherent complexity of concurrent systems; because they involve multiple protocols, conversion problems are likely to be even more complex, and formal methods may in fact be a necessity. Finally, because specifications are represented in a precise mathematical form, there may be classes of systems for which *automatic* generation of converter specifications is possible (we shall see that this is indeed the case).

It must be noted that few (if any) of the protocols and services in wide use today have been *formally* specified as described above. A systematic approach based on a particular formalism can be applied to *existing* implementations and services only after they have been specified in that formalism. This is likely to be a nontrivial task, and there is generally no way to prove that such a *post hoc* specification adequately captures the behavior of an implementation. Nevertheless, it is instructive to investigate what can be accomplished given the requisite formal specifications. By solving problems using formal models, we may obtain fundamental results, which can then be applied to real problems.

2.1 Conversion Problem

Referring again to Figure 1, suppose protocols P and Q provide services that are similar, but differ in certain details. The protocol P implementation consists of the peer entities P_0 and P_1, while Q consists of Q_0 and Q_1. (This is a simplified view of the problem; an implementation may, of course, have more than two components, including services implemented by lower-level protocols. We consider some of these cases in Section 6.) Now, suppose P_0 and Q_1 are able to exchange messages. We would like to use P_0 and Q_1 to provide a service similar to that provided by P and Q. We are given formal specifications of P_0 and Q_1, and a specification S_C of the desired service. We want to specify a converter, C, which will help P_0 and Q_1 to implement the service defined by S_C (Figure 3). There may be any number of converters satisfying these requirements. A general solution method for this problem would enable us to produce a correct converter, or an indication that no such converter exists, from specifications of P and Q, and S_C.

It is important to realize that the notion of *incompatibility* of protocols — which we equate with the nonexistence of a converter in the problem just defined — only makes sense relative to a given required service. Any converter will suffice if the conversion system only has to satisfy the trivial specification "true." The idea is that the service specification S_C defines the *minimal* properties required by the users of the conversion system. The notions of "hard mismatch" and "soft mismatch," discussed by Green in [13], can be interpreted in this context. When protocols are not compatible with respect to any useful service, a hard mismatch exists. In a soft mismatch, no converter can give the full functionality of the original protocols, but a converter exists for a less powerful, but still useful, service. Thus, S_C defines the boundary between a "hard" and "soft" mismatch.

Our assumption that a formal specification for the correctness of the conversion system is part of the problem input is a significant one. Even if specifications of the services implemented by P and Q are available, S_C will, in general, differ from both of them. (For example, the user interfaces of P_0 and Q_0 may differ.) Techniques for deriving S_C from existing service specifications are of interest as an alternative to constructing S_C "from scratch," i.e., by formalizing the functional requirements of the users. We consider the problem of obtaining S_C to be separate from the conversion problem, however.

2.2 Specifications

While the above formulation of the problem is independent of any particular formalism, a solution method may require that specifications be given in a particular notation. In this section, we briefly discuss the formalisms to be encountered in the methods described later. The reader is assumed to be familiar with the basics of protocol specification and verification.

For each of the methods we describe, protocol components are specified as interacting state machines. The Lam and Okumura approaches assume a message passing model. In this model, components interact and change state asynchronously, by sending and receiving named *messages* over unidirectional *channels*. Each event (sending or receiving a message) occurs under the control of exactly one component, and the channels form the interface between components. The approach presented in Section 5 is based on a "rendezvous" model, in which interaction between two components occurs synchronously, via named *actions*. Such an action can take place only if *both* components are "ready" for it, and the resulting state changes happen simultaneously in both components. In this model, communication channels are specified as separate components of the system.

A *state machine* is defined by a *state set*, including a distinguished *initial state*, and a set of *state transitions*. The state transitions tell how the state of the specified protocol component changes through interaction with its environment. (By "environment," we mean the users of the protocol, or other components of the protocol implementation.) If s and s'

are states, and there is a transition from s to s' associated with event e, we write $s \xrightarrow{e} s'$. (The term "event" refers to the kind of interaction appropriate to the context: sending or receiving messages in the message passing model, or synchronized actions in the rendezvous model.) When the state machine reaches state s in the course of its execution, we say the transition to s' and the associated event are *enabled.* The behavior of the component is modeled by the possible sequences of state transitions and associated events beginning in the initial state. If we regard a state machine as a directed graph, with states as nodes and transitions as directed edges, then the behaviors of a component correspond to the *paths* in this directed graph beginning in the initial state.

The behavior of a collection of interacting protocol components is modeled by a *global* state machine, formed from the state machines of the individual components. The state of this global system is a tuple comprising the state of each component (and each channel). Each transition of the global state machine comes from one of the component state machines; global state transitions occur instantaneously and indivisibly. A behavior of the global system is a linear sequence of states and transitions corresponding to a path through the global state space; concurrency is modeled by the possibility of events occurring in arbitrary order.

A correctness specification defines aspects of the desired global behavior that ensure that any implementation provides the desired service. It tells the users of a system what kind of behavior to expect, while leaving unspecified the details of how that behavior is achieved. *Safety* properties define the system's *allowed* behavior. If we think of a behavior as a sequence of interactions between the protocol system and its users, then a safety property can be viewed as prohibiting certain "bad" sequences or states. *Progress* properties, on the other hand, define *required* aspects of system behavior, e.g., that some response is generated for each input. Safety and progress properties of a global state machine can be defined using formulas of *temporal logic* [27, 25, 23, 3], or in terms of another global transition system having the desired properties [19, 22, 26].

We represent component specifications (state machines) pictorially in the form of directed graphs, as described above. The initial state is represented by node 0. The label "$-m$" on an edge indicates that message m is sent when that transition occurs; "$+m$" denotes receipt of the message m. Other labels denote other kinds of interactions besides sending and receiving of messages, e.g., timeouts. Where multiple labels appear on a single edge, a transition is associated with each of the indicated events.

In what follows, we shall not be too fussy about distinguishing between components and their specifications: the phrase "the component A" can be understood to mean "the specification of component A." Similarly, "finding a converter" can be read as "finding a specification for a converter."

2.3 An Example

In the next three sections, we describe and compare three methods for solving the problem defined above. As an aid to understanding and comparison of the methods, we pose a simple example problem and apply each method to it. The example involves a mismatch between the venerable Alternating-Bit (AB) protocol and a protocol that does not use any sequence numbers, called the non-sequenced (NS) protocol. Both provide delivery of data from a Sender to a Receiver in spite of possible message losses by the transmission medium. For the example problem, it is desired to transfer data from an AB Sender to an NS Receiver.

The protocol specifications are shown in Figures 4 and 5, respectively. The "acc" and "del" events model interaction with the user, denoting acceptance of a data unit from the user at the Sender end and delivery of a data unit to the user at the Receiver end, respectively. To distinguish between the messages of the two protocols, AB messages have all-lower-case names, while those of NS are capitalized.

The AB Sender (A_0) attaches a one-bit sequence number to each data unit transmitted; the data messages are thus represented as "d0" and "d1." The Receiver (A_1) uses this number to synchronize with the Sender and determine whether a received data message has already been delivered; this mechanism ensures that each data message is *delivered exactly once.* An acknowledgement message, containing the sequence number of the last-delivered data message ("a0" or "a1"), is returned for each data message received.

The NS protocol has no sequence numbers; a data message is represented by "D." The Receiver (N_1) simply delivers every received data message, and returns an acknowledgement message "A." The Sender (N_0) repeatedly transmits the data until an acknowledgement is received; if an acknowledgment is lost, the same message may be *delivered several times* by the Receiver of the NS protocol. The service implemented by the NS protocol is thus "weaker" than that of AB.

Both protocols use the standard technique for detecting losses, namely timeouts. Because our simple specifications do not include an explicit notion of time, we use other techniques to represent loss/timeout behavior. In the message-passing model, "virtual messages" model the causal relationship between lost messages and subsequent timeouts, as introduced in [5]; these messages and their associated transitions are not part of the protocol itself. We assume there are no premature timeouts. Whenever a timeout occurs (represented by receipt of a "tm" or "Tm" message), it is the result of losing a data or acknowledgement message (represented by sending "ls" instead of data, or "tm" instead of an acknowledgement). This way of modeling losses and timeouts is not always valid; it works here because of the stop-and-wait nature of the protocols. In Figures 4 and 5, the virtual messages are parenthesized because they are part of the protocol specifications in the message-passing model, but not the rendezvous model. In the rendezvous model, losses and timeouts are represented directly in the specification of the transmission media (section 5.5); the AB Sender has a "tm" action corresponding to each "+tm" in Figure 4, and similarly for NS. Again, a timeout occurs only after loss of a data message or an ack message.

3 Conversion via Projection

Lam showed how the techniques of *protocol projection* [22] can be used to reason about the correctness of conversion systems, and in some cases to derive a converter specification [21, 20]. In the context of protocol verification, projection is a way to focus on the aspects of each system that are relevant to the properties to be proved. This is achieved by projecting the system onto an *image protocol.* The method is based on the following result: let P be a protocol, and P′ its image protocol under a projection mapping. For any safety property of P′, there is a corresponding safety property that holds for P. If, in addition, transitions of the image protocol satisfy a *well-formedness* condition, then an analogous result holds for progress properties. (A statement and proof of the above results using temporal logic can be found in [23].) The projection method can be used for protocols specified with finite state machines, in a programming language notation, or in a relational notation [23, 24]. The message passing model is assumed, and communication channels may lose, duplicate and reorder messages in transit.

3.1 Image Protocols

We briefly describe the projection method. An image protocol is derived from a given protocol by partitioning the state set of each protocol component; states in the same block of the partition are considered to be indistinguishable in the image of that component. This defines a mapping from each component state to a state in the *image component.* The state space partition induces an equivalence relation on the set of messages sent and received in the protocol. Messages whose receptions cause the same image state transitions are considered equivalent, and are mapped to the same *image message.* Messages that cause no change in the image state of their receiver do not appear in the image protocol at all, and are said to have a *null image.*

A simple example of a protocol projection is shown in Figure 6; the original protocol is on top, with its image below. Primes indicate image quantities. States 0 and 1 of each original protocol component are indistinguishable in the image; thus P_0' and P_1' each have only two states. Because neither message "x" nor message "y" causes any image state change, each has a null image.

One type of safety property is an *invariant:* a predicate that is true at all states reachable by a path in the global state machine. An invariant of the image protocol in the figure is $(p_0' = 0') \Rightarrow (p_1' = 0')$, where p_0' and p_1' represent the state of the left and right image components, respectively. Each state $0'$ is the image of original states 0 and 1, so the corresponding invariant of the original protocol is $(p_0 = 0 \vee p_0 = 1) \Rightarrow (p_1 = 0 \vee p_1 = 1)$.

3.2 Common Image Protocol

Conversion can be considered as a solution to a protocol mismatch only when the protocols "provide similar services." Projection can be used to formalize this notion. Suppose protocols P and Q can each be projected onto the same image protocol, say R. Then R, the *common image*, embodies some functionality that is common to P and Q. Each protocol has properties corresponding to those of R; each has messages whose meanings correspond to those of R. On the other hand, messages that have a null image in the projection have no meaning with respect to the common functionality represented by R. The common image R defines a *semantic correspondence* between states of P and states of Q: states with the same image have the same meaning with respect to the service implemented by R.

If a common image with adequate functionality can be found, specification of a converter is straightforward. The projection mapping defines an equivalence between the messages of P and Q, just as it does for states: messages with the same image have the same "meaning." This static equivalence can easily be implemented by a *stateless converter*, as follows: whenever the converter receives a message, it immediately forwards a message of the other protocol that has the same image. Null-image messages are ignored. It can be shown that the common image protocol is an image of the resulting conversion system; thus the conversion system has safety properties corresponding to those of the common image. If the image is well-formed in each projection, then the correspondence holds for progress properties as well.

It is always possible to find a common image for any two protocols: the degenerate protocol, in which each component has only one state and no transition, is a well-formed image of every protocol. The problem is to find an image protocol that satisfies the conversion service specification S_C. This is a process that must be carried out heuristically, using intuitive understanding of the protocols. Unfortunately, a common image protocol satisfying S_C may not exist.

3.3 Example

When no common image with the desired characteristics can be found, a *finite-state* converter may be constructed based on intuition. Projection can also be useful in proving properties of a system with such a converter. This is illustrated in [1], and also by our example problem involving AB and NS, as shown in [21]. Refer to Figure 7, which shows a projection of A_0 that resembles N_0, but is not quite the same. The difference is the "+a0" and "+a1" transitions from image state (2/5) to image state (1/4), which are not present in N_0. After receiving an acknowledgement, the image Sender may still retransmit data. Thus, we cannot statically map "A" to either "a0" or "a1." We therefore propose a converter of more complex structure, one that emulates the AB Receiver and the NS Sender in an alternating manner (Figure 8).

We can derive properties of this conversion system by viewing as a *single component* the subsystem consisting of C, A_0, and the channels between them (Figure 9). As shown in [21], this composite system can in fact be projected onto N_0 by partitioning the subsystem states based on the states of C. With N_1 projected onto itself, the conversion system has NS as a well-formed image. Similarly, the system can be projected onto AB by aggregating C and N_1. It follows that the properties of AB hold for the communication between the Sender and the Converter, and those of NS hold between Converter and Receiver. Using these properties, and the structure of the converter, we can deduce properties of the global system. The AB protocol guarantees that each accepted message will be delivered exactly once. The NS protocol, however, guarantees that each accepted message will be delivered *at least* once. What can we say about the service of the conversion system?

Informally, we reason as follows. In the projection of C and N_1 onto A_1, the "+A" event maps to the "del" transition in the image. Thus we can infer (from the properties of AB — not given here explicitly) that "+A" occurs exactly once for each "acc" event. However, as we have already noted, "del" can occur at N_1 several times for each occurrence of "+A" at N_0 because of the possibility of losses. Thus, the service of the whole system corresponds to that of NS. If the desired service is that of AB, then converter C must be connected to the NS Receiver by reliable channels that do not lose messages.

3.4 Discussion

The projection method provides a *sufficient condition* for finding a useful converter to overcome a protocol mismatch. If a common image protocol with the desired properties can be found, a very simple converter can be obtained easily. However, such a common image need not exist, and must be sought using intuitive understanding of the protocols. Projection can also be used to reason formally about correctness of converters obtained by other methods.

4 Okumura's Approach

Another approach to the problem of Figure 3 has been presented by Okumura [31]. For this method, the protocols must be specified as finite-state machines (FSMs), which interact by passing messages over channels. The sets of messages sent and received in each protocol are assumed to be disjoint; this is easily achieved by renaming of messages, if necessary. The method employs an algorithm to construct a converter from components of the original protocols and a partial specification of the converter's behavior. For protocols P and Q, where communication between P_0 and Q_1 is desired, the input to the algorithm consists of P_1, Q_0, and an additional FSM called a *conversion seed.*

4.1 The Algorithm

Okumura's algorithm is based on a somewhat different idea of correctness for the conversion system than that of Figure 3. In particular, there is no explicit definition of a service to be provided to users; indeed there is no notion of "users" in the model at all. Instead, the conversion system is considered correct if it is free from deadlock and unspecified reception,[2] and if the converter, C, satisfies two requirements. The first is that C must be a *reduced-FSM* of P_1 in its communication with P_0, and a reduced-FSM of Q_0 in its communication with Q_1. This is defined as follows. For communicating FSMs A, B, and E, "A is a reduced-FSM of B in its communication with E," means that

(i) for every path in A, there is a corresponding path in B that has the same sequence of send and receive events (of messages to and from E); and

(ii) if a path in B that corresponds to some path in A can be extended by reception of a message from E, then the corresponding path in A can also be extended by the same reception event.

This property of the converter implies that any sequence of messages sent by C to P_0 (Q_1) is a sequence that *could* have been sent by P_1 (Q_0). If the original protocols are free from deadlock and unspecified reception, this is a sufficient condition for the conversion system to be free from those faults.

The conversion seed defines the other required property of C as follows. The seed — call it X — is a finite state machine whose message set is a subset of the union of the message sets of P_1 and Q_0. This message set contains the *significant* messages of the conversion, and X defines a constraint on the order in which these messages may be sent and received *by the converter.* In particular, each of the converter's possible sequences of sends and receives of significant messages must correspond to some sequence of sends and receives of X. Messages that are not in X's message set are unconstrained and may be sent or received at any time by the converter, as long as the reduced-FSM requirement is satisfied.

[2]An *unspecified reception* is a reachable global state in which a message is at the head of some channel, and the receiving component has no receive event specified for that message.

Figure 10 shows a simple example, adapted from [31]. We want to provide communication between P_0 and Q_1. The "data" message of P corresponds to the "msg" of Q, so we want each "msg" received to be forwarded as "data" by the converter. The seed X specifies that "+msg" and "–data" events must alternate. For inputs P_1, Q_0, and X, the algorithm produces the converter shown in Figure 11. Note that the "–data" for a given "+msg" need not occur immediately, but may be preceded by any number of "poll,end" exchanges between P_0 and C.

The algorithm constructs a converter from the input FSMs, P_1, Q_0, and X, in several steps, as follows. Let S_{P_1} and S_{Q_0} represent the state sets of P_1 and Q_0, respectively. Let e and f represent arbitrary send or receive events.

1. Construct a FSM Y with states $\langle p, q\rangle$, where $p \in S_{P_1}$ and $q \in S_{Q_0}$. For each q, and each transition $p \xrightarrow{e} p'$ of P_1, Y has a transition $\langle p, q\rangle \xrightarrow{e} \langle p', q\rangle$. Similarly, for each p, and each transition $q \xrightarrow{f} q'$ of Q_0, Y contains a transition $\langle p, q\rangle \xrightarrow{f} \langle p, q'\rangle$.

2. Remove transitions of Y that violate the constraints defined by the conversion seed, by combining Y and X to form FSM Z as follows. Z has states $\langle y, x\rangle$, where y is a state of Y and x is a state of X. For each transition $y \xrightarrow{e} y'$ of Y involving a message that is *not significant*, Z has a transition $\langle y, x\rangle \xrightarrow{e} \langle y', x\rangle$. If the message *is significant*, then there is a transition $\langle y, x\rangle \xrightarrow{e} \langle y', x'\rangle$ in Z if and only if there is also a transition $x \xrightarrow{e} x'$ in X.

3. Mark all states $\langle y, x\rangle$ of Z such that either $\langle y, x\rangle$ has no outgoing transitions, or there is a receive event $y \xrightarrow{+m} y'$ of Y for some m and y', with no corresponding receive event $\langle y, x\rangle \xrightarrow{+m} \langle y', x'\rangle$ in Z. Remove marked states from Z, together with all their incoming transitions, thereby possibly creating new marked states (by removal of a receive transition or the last outgoing transition of a state). Iterate as long as there are marked states.

Upon termination, the remaining states and transitions, if any, form the correct converter. The running time of the algorithm is polynomial in the product of the sizes of P_1 and Q_0.

A component FSM is said to be *effective* in a protocol if for every sequence of messages that can be sent and received by the component, there is a path in the protocol system's *global* state machine containing the same sequence of events. If the input FSMs P_1 and Q_0 are effective, and the state set of Z is empty when the algorithm terminates, it means that the reduced-FSM requirement conflicts with the requirements of the conversion seed. Thus, failure of the algorithm to produce a converter means that none exists for the given inputs, provided the inputs P_1 and Q_0 are effective in their original protocols.

Okumura considers an *unspecified reception* to occur when a message appears at the head of a channel and no receive event is specified for it *in that state* of the component. This is a strong condition — usually it is not considered an unspecified reception if the message can eventually be received after some sequence of sends and/or internal transitions. Under the Okumura definition, removal of a receive transition from an effective FSM *will* result in an unspecified reception. If the algorithm fails to produce a converter for effective protocols, the conclusion that no converter exists is based on this strong definition; a converter that would be considered correct under the usual definition of unspecified reception could exist.

4.2 Example

The input for our example includes the specifications of the AB Receiver (A_1) and NS Sender (N_0) from Figures 4 and 5. The algorithm permits only send and receive transitions, so the "acc" and "del" events and associated transitions of A_1 and N_0 are removed. For the required service, we want every message accepted by A_0 to be delivered by N_1 eventually. Clearly, "d0" and "d1" should be forwarded as "D" by the converter; our conversion seed should reflect this. However, the algorithm is sensitive to the way this functionality is represented in the seed. With the naive seed shown in Figure 12, the algorithm fails to produce a converter. (The input FSMs are not effective, so we cannot conclude that no converter exists for these inputs.) The seed of Figure 13, however, produces the converter shown in Figure 14. Analysis of a system including this converter shows that the service provided is similar to that of NS.

4.3 Discussion

A general method for solving the problem of Figure 3 using Okumura's algorithm is the following. From the service specification S_C, construct (heuristically) a conversion seed X, and run the algorithm on P_1, Q_0, and X. If a converter C is produced, analyze the system comprising P_0, C, and Q_1. If this system satisfies S_C, then C is the desired converter; otherwise, iterate with a different seed.

The algorithm allows efficient construction of the converter from the existing components P_1 and Q_0, provided a suitable seed can be found. However, desired *global* properties of the conversion system cannot be input directly; a service specification given in terms of P_0 and Q_1 must be changed into a conversion seed constraining the *converter's* behavior. As we have seen, this transformation may not be straightforward. Moreover, if the algorithm fails to produce a converter, it is difficult to conclude that a hard mismatch exists (even if the protocols are effective), because the problem may be in the way the conversion seed is specified, or due to the strong definition of unspecified reception.

5 The Quotient Approach

In the previous two methods, finding a converter involves relating the conversion system to the original protocols or to some other protocol. The global service specification enters the picture only after this relationship has been established. The advantage of this approach is that a converter can be efficiently constructed; the disadvantage is that no systematic way of finding the desired relationship — a satisfactory common image protocol in the projection method, or a proper conversion seed in the Okumura method — is available. It is also difficult to conclude with certainty that the required service cannot be provided with the given protocols. In contrast to these "bottom-up" approaches, a "top-down" approach would derive the converter directly from the given specifications, and give precise conditions for detection of a hard mismatch. In this section, we describe such an approach.

Consider the problem depicted in Figure 15. Let A be a service specification, and let B specify one component of an implementation. B has two interfaces; one is "external," and is the same as the interface of A, while the other is "internal," comprising a set of possible interactions between B and another component. The goal is to specify another component C, which interacts with B via its internal interface, so that the behavior observed at B's external interface implements the service defined by A. Let the operator "$\|$" on specifications represent interaction between components, and let *satisfies* be a relation between specifications that means one implements the service defined by the other. We want to find C such that (B $\|$ C) satisfies A. By analogy with the problem of finding the multiplicative inverse of a number, we call this the *quotient* problem: in effect, we want to "divide" A by B. As with real numbers, a quotient does not exist for every A and B.

It should be clear that Figure 3 depicts a form of the quotient problem: P_0 and Q_1 correspond to B, while the service specification of the conversion system corresponds to A. The interface between P_0 and Q_1 and their users corresponds to B's external interface; B's internal interface corresponds to the actions by which P_0 interacts with P_1, and Q_1 interacts with Q_0. Thus, a solution method for the quotient problem can be used to solve the problem of Figure 3.

An algorithmic solution for the quotient problem is impossible for any sufficiently powerful specification formalism: if the specified systems can imitate Turing machines, an algorithm that decides whether a quotient exists could be used to solve the halting problem! By restricting the specification language such that only finite-state systems can be specified, an algorithm is possible, but the problem is computationally hard.[3] This is not surprising — the problem is extremely general, because only the abstract structure of the given components can be used to solve it. In other words, without resorting to the kind of intuitive understanding that must be used to find a common image, or a conversion seed, we are faced with an exhaustive search of possibilities.

In [28], Merlin and Bochmann discussed the similar problem of "construction of submodule specifications" using a simple trace-set semantics. They described a solution that

[3]For specifications represented as (nondeterministic) finite state machines, a method to decide whether a quotient exists can be used to decide whether two nondeterministic finite state machines accept different languages, a problem as hard as any that can be solved in polynomial space [11].

dealt with safety properties only. More recently, the "supervisor synthesis problem" for discrete-event systems has received some attention in the control theory literature, and solutions based on language-theoretic semantics have been proposed [34, 6, 4]. Parrow [33] has described an interactive system for solving "equations" in components, based on the bisimulation semantics of CCS [29].

In what follows, we describe briefly a theory of specifications of finite-state systems, and present an algorithm for solving quotient problems for a class of such systems. Our algorithm deals with both safety and progress properties, and produces a specification of a solution if and only if one exists.

5.1 Specifications

We model protocols as collections of finite state machines interacting via named *actions*. This form of interaction is used in many formalisms [16, 29, 26, 19], including LOTOS [18]. We are concerned with two main ideas: *composition*, i.e., viewing interacting components together as a composite whole; and what it means for one system to *satisfy*, or *implement*, another. These ideas appear in the theory as a composition operator and a satisfaction relation on specifications. Together, they allow us to reason about whether a collection of protocol components correctly implements a specified service.

A *specification* is a tuple $(\Sigma, S, \mathrm{T}, \mathrm{I}, s_0)$, where

Σ is a finite set of *actions*
S is a nonempty finite set of *states*
$\mathrm{T} \subseteq S \times \Sigma \times S$ is the *external transition relation*
$\mathrm{I} \subseteq S \times S$ is the *internal transition relation*
$s_0 \in S$ is the distinguished *initial state.*

The set Σ of actions completely defines a systems's interface with its environment. (Note: by "system" here, we mean the specified object. It may be an individual component, a composite formed from several interacting components, or a service — all are specified the same way.) The actions of the interface are the *only* way systems can interact; intuitively, actions model an exchange of information or handshake across the interface, possibly involving a state change on both sides. In a protocol or service specification, actions are abstractions of occurrences such as submission of a message for transmission, or expiration of a timer.

The relations T and I define the *transitions* of the system. Each transition in T has an associated interface action in Σ; these define how the local state is affected by interaction with the environment. If (s, e, s') is in T, we write $s \xrightarrow{e} s'$. Whenever the local state is s, and the environment is also ready for action e, e may occur, accompanied by a state change to s'. It is important to realize that external events are not under the exclusive control of either side of an interface, but can occur only when the associated action is enabled on *both* sides.

The relation I defines *internal* state transitions that may occur unobserved and without environmental interaction. When (s, s') is in I, we write simply $s \rightarrow s'$. Internal transitions allow some state changes to occur under the exclusive control of one side an the interface, and play several important roles in specifications. In a correctness or service specification, an internal transition can represent a choice among different acceptable behaviors, and help avoid unnecessary overspecification. For example, suppose the buffering capacity of a transport service is not specified in its correctness specification. After accepting the first data unit for transmission, the allowable behaviors are to accept another data unit, or to refuse to accept another until the previous one has been delivered. The choice among these behaviors is made once, by the designer of the implementation.

Internal transitions also serve as an abstraction mechanism in considering the service implemented by a collection of interacting components. The environment (user) is not concerned with interactions between the individual components, so these are hidden by making them internal transitions. Thus A || B can have internal transitions corresponding to the synchronized interactions between A and B, even if neither A nor B separately has internal transitions.

Finally, internal transitions can model low-level behavior that would add too much complexity if modeled explicitly. An example is the loss of a message in a communication channel. Modeling the actual causes of the loss would greatly complicate the channel specification. Instead, the chain of events constituting a loss is represented by a single internal transition, which may or may not occur. This kind of transition is often regarded as *fair*, meaning that if it is repeatedly enabled, it will eventually occur. On the other hand, such a fairness requirement would usually *not* be placed on internal transitions used to avoid unnecessary overspecification, as in the buffering capacity example above.

Instead of attaching explicit fairness requirements to each internal transition in our specifications, we make certain assumptions about fairness. In defining "B satisfies A," we regard A as a service specification, and B as the specification of an implementation. In this paper, service specifications are assumed to be *deterministic* in the following sense: they have no internal transitions, and no action is associated with more than one transition originating in the same state. (Our results hold, and our quotient algorithm works, for *nondeterministic* service specifications in a certain "normal form;" the restriction to deterministic service specifications simplifies the presentation, and is adequate for the examples in this paper. A fuller treatment appears in [2].) We also assume that implementation specifications satisfy the following fairness requirement: any transition that is repeatedly enabled will eventually occur.

In what follows, A, B, C, and D refer to distinct specifications. Parts of different specifications are distinguished by subscripts: Σ_B is the set of actions of B, S_A is the state set of A, etc. The states of a specification are represented by (primed) lower-case italic letters corresponding to the name of that specification; thus a and a' are members of S_A. The letter denoting a state makes it clear to which specification it belongs, so that when we write $a \xrightarrow{e} a' \wedge b \xrightarrow{e} b'$, it should be clear that one transition is defined in T_A, while the other is in T_B. Function and predicate application are denoted by a period, as in $f.c$.

5.2 Composition

When components interact, each becomes part of the other's environment; their interactions with each other are synchronized and hidden from the rest of the environment. The specification of the resulting composite system is determined by the specifications of its components, as denoted by the infix operator "||."

For any specifications A and B, (A || B) is a specification given by:

$$\begin{aligned}
\Sigma_{(\mathrm{A}\|\mathrm{B})} &= (\Sigma_\mathrm{A} \cup \Sigma_\mathrm{B}) - (\Sigma_\mathrm{A} \cap \Sigma_\mathrm{B}) \\
S_{(\mathrm{A}\|\mathrm{B})} &= S_\mathrm{A} \times S_\mathrm{B} \\
\mathrm{T}_{(\mathrm{A}\|\mathrm{B})} &= \{\, (\langle a,b\rangle, e, \langle a',b'\rangle) : e \in \Sigma_{(\mathrm{A}\|\mathrm{B})} \wedge ((a = a' \wedge b \xrightarrow{e} b') \vee (b = b' \wedge a \xrightarrow{e} a')) \,\} \\
\mathrm{I}_{(\mathrm{A}\|\mathrm{B})} &= \{\, (\langle a,b\rangle, \langle a',b'\rangle) : (b = b' \wedge a \rightarrow a') \vee (a = a' \wedge b \rightarrow b') \vee \\
&\qquad (\exists e : e \in \Sigma_\mathrm{A} \cap \Sigma_B \wedge a \xrightarrow{e} a' \wedge b \xrightarrow{e} b') \,\} \\
\langle a,b\rangle_0 &= \langle a_0, b_0\rangle
\end{aligned}$$

Each internal transition of the composite comes from one of two sources: an internal transition in one of the components, or a synchronized action in $\Sigma_\mathrm{A} \cap \Sigma_\mathrm{B}$ that becomes hidden in the composition.

5.3 Satisfaction

A *trace* is a sequence of actions, and represents a behavior of the system as it might be observed by its environment. In terms of the directed graph structure, a trace corresponds to the sequence of labels along a finite path in the graph. We associate a particular prefix-closed set of traces with each specification, namely those corresponding to all finite paths in the graph beginning at the initial state. This set describes all possible behaviors of the system, and thus captures all of its safety properties. Each trace represents a sequence of actions that the environment might observe over some finite time interval, and is not necessarily maximal or complete. The empty trace, denoted by ε, corresponds to the interval before anything happens, and is a possible behavior of every system. We denote traces by the letters t, r, etc. Individual actions are considered traces of length one, and concatenation is denoted by juxtaposition: te is a trace ending with action e. For any specification A, we write A.t to denote "t is a trace of A." The symbol $\xrightarrow{*}$ denotes the reflexive and transitive closure of λ; thus $s \xrightarrow{*} s'$ means s' is reachable from s via zero or more internal transitions. Also, for a set Σ of actions, Σ^* is the set of all finite sequences of members of Σ.

Every specification defines a relation "$\longrightarrow$," which is the least relation satisfying, for any states s, s', s'', trace t, and event e,

- $s \xrightarrow{\varepsilon} s$.
- $s \xrightarrow{t} s' \wedge s' \xrightarrow{*} s'' \Rightarrow s \xrightarrow{t} s''$.
- $s \xrightarrow{t} s' \wedge s' \xrightarrow{e} s'' \Rightarrow s \xrightarrow{te} s''$.

In other words, $s \xrightarrow{t} s'$ means there is a path from s to s' corresponding to trace t. Thus we have $\mathrm{A}.t \equiv (\exists a : a_0 \xrightarrow{t} a)$.

"B satisfies A with respect to safety" means that every possible behavior of B is a possible behavior of A. Using the trace set interpretation of specifications, this is easy to express: the set of traces of B is contained in the set of traces of A. Thus, B satisfies A with respect to safety if and only if $\forall t : \mathrm{B}.t \Rightarrow \mathrm{A}.t$.

Because both sides of the interface must "be ready" for an action to occur, the notion of *progress* in this model deals with the actions (or sets of actions) enabled in the system after any particular trace. With this information, the environment can ensure that there is always *some* action enabled on both sides of the interface, and thus prevent deadlock. Intuitively, "B satisfies A with respect to progress" means that if an environment cannot reach a deadlock with A, then it cannot reach a deadlock with B. This idea of progress is similar to the "refusals" of Hoare [16], or the "acceptance sets" of Hennessey [15]. To define this in terms of specifications, we must consider what it means for an action to be enabled after a trace.

For a deterministic specification, there is at most one path corresponding to any trace. Thus, the state of the system — which cannot be observed directly by its environment — after any trace is uniquely determined, and the environment can always "know" exactly what actions are enabled. If internal transitions are present, however, things are more complicated. The problem is that a transition associated with an action may be "pre-empted" by an internal transition, if the two are enabled in the same state. Thus we might consider an action to be "enabled" only in a state with no outgoing internal transitions. But a trace may lead to a cycle of internal transitions; if these internal transitions occur continuously, the system may *never* enter a state with no outgoing internal transition. However, our fairness assumption says that a repeatedly-enabled transition must eventually occur; under this assumption no cycle of internal transitions can pre-empt any transition infinitely many consecutive times. If there is a transition, internal or otherwise, leading out of the cycle, then enventually it or some other cycle-breaking transition will occur. As a consequence, we can regard a set of states connected by a cycle of internal transitions as a single state for the purposes of defining the set of enabled events. We call such a set of states a *sink set* if no internal transition (except those in the cycle) is enabled in any state of the set.

In the left-hand specification of Figure 16, the two unlabeled states constitute a sink set. Once either state is reached, the actions f and g cannot forever be preempted by internal transitions, and one of them will eventually occur. Thus we can view the sink set as a single state with two events enabled, as on the right-hand side. We write $sink.s$ to indicate that a state s is a member of a sink set.

We denote the set of actions associated with transitions originating in state s by $\tau.s$:

$$e \in \tau.s \;\equiv\; (\exists s' : s \xrightarrow{e} s')$$

We write $\tau^*.s$ for the set of all actions enabled in any state internally reachable from s:

$$e \in \tau^*.s \;\equiv\; (\exists s' : s \xrightarrow{*} s' \wedge e \in \tau.s')$$

The set $\tau^*.s$ contains all actions that *may* occur next if the current state of the system is s; if s is in a sink set, the set of actions considered enabled at s is defined to be $\tau^*.s$. Observe that for a deterministic specification, every state is a singleton sink set, and $\tau^*.s = \tau.s$.

Now consider a deterministic service specification A, and an implementation specification B satisfying the fairness requirement. Let t be a trace of both systems, and suppose that $a_0 \xrightarrow{t} a$ in A and $b_0 \xrightarrow{t} b$ in B, where b is a sink set. In order for B to satisfy A, the set of actions enabled at b must contain all actions enabled at a; otherwise, some action e can be enabled at a but not b. An environment that has only e enabled after trace t can deadlock with B after t, because no action is enabled on both sides; however, it would not deadlock with A after t. Formally, B satisfies A with respect to progress if and only if

$$\forall t, a, b : (a_0 \xrightarrow{t} a \wedge b_0 \xrightarrow{t} b \wedge sink.b) \Rightarrow \tau.a \subseteq \tau^*.b$$

Using the fact that a sink set is reachable from every state, the above formula can be shown to be equivalent to

$$\forall t, a, b : (a_0 \xrightarrow{t} a \wedge b_0 \xrightarrow{t} b) \Rightarrow \tau.a \subseteq \tau^*.b$$

For deterministic A, and B satisfying the fairness requirement, B satisfies A if and only if both of the following conditions hold:

(Safety) $\forall t : \mathrm{B}.t \Rightarrow \mathrm{A}.t$

(Progress) $\forall t, a, b : (a_0 \xrightarrow{t} a \wedge b_0 \xrightarrow{t} b) \Rightarrow \tau.a \subseteq \tau^*.b$.

5.4 The Algorithm

The algorithm described here takes a (deterministic) service specification A and a specification B describing part of an implementation, and produces C such that B || C satisfies A, if such a C exists. If no such C exists, the algorithm produces a degenerate specification with an empty state set. In computing C, safety and progress are handled in sequential phases. In the first phase, the state set and transition relation of C are built up inductively, beginning with the initial state; the result is a specification with the *largest* trace set consistent with safety of B || C. In the second phase, states of C at which a progress violation by B || C cannot be prevented are iteratively removed. (Such progress problems can only be corrected by *removal* of transitions from C, because C already has the largest possible trace set; no transitions can be added without violating safety requirements.) When the second phase terminates, if C's state set is nonempty, then it is a quotient, and moreover it is a *maximal* quotient in the sense that, for any other quotient D, any trace of D is a trace of C.

Let the specifications A and B be given, the user interface of A consist of the set *Ext* of actions, and *Int* be the set of actions comprising the B-C interface (Figure 15). We have $\Sigma_\mathrm{A} = Ext$, $\Sigma_\mathrm{B} = Int \cup Ext$, $\Sigma_\mathrm{C} = Int$, and *Int* and *Ext* are disjoint. In terms of the conversion problem of Figure 3, the event set *Ext* is the interface between the user and the service, and *Int* represents the interactions (messages that may be sent and received) between the peers of protocols P and Q.

Now, the observable aspects of the behavior of B $\|$ C are determined by B: the trace set of B $\|$ C consists of members of B's trace set with actions in *Int* removed. Also, the behaviors that can occur at the B-C interface are traces of B with events in *Ext* removed. The functions i and o will be used to make these ideas precise. If t is a trace of B, $i.t$ and $o.t$ are subsequences of t containing only actions in *Int* and *Ext*, respectively.

$$\begin{aligned}
i.\varepsilon &= \varepsilon \\
i.te &= \begin{cases} (i.t)e & \text{if } e \in \mathit{Int} \\ i.t & \text{if } e \notin \mathit{Int} \end{cases} \\
o.\varepsilon &= \varepsilon \\
o.te &= \begin{cases} (o.t)e & \text{if } e \in \mathit{Ext} \\ o.t & \text{if } e \notin \mathit{Ext} \end{cases}
\end{aligned}$$

In what follows, the variable q denotes a member of Ext^* (e.g., a trace of A or B $\|$ C); t denotes a member of $(\mathit{Int} \cup \mathit{Ext})^*$ (a trace of B); and r is in Int^* (a trace of C).

For each trace q of B $\|$ C, there is a corresponding trace t of B such that $o.t = q$ and $i.t$ is a trace of C. In a similar way, a trace r of C corresponds to the *set* of traces t of B such that $i.t = r$ and $o.t$ is a trace of B $\|$ C. The following formula characterizes the relationship among traces and states of B, C, and B $\|$ C, for any q, b, b', c, and c' (recall that $\langle b, c\rangle$ is a state of B $\|$ C):

$$\langle b, c\rangle \xrightarrow{q} \langle b', c'\rangle \equiv (\exists t : o.t = q \wedge b \xrightarrow{t} b' \wedge c \xrightarrow{i.t} c')$$

We say a trace r in Int^* is *safe*, and write *safe*.r, if every trace of B that matches r is a trace of A when projected on *Ext*:

$$\mathit{safe}.r \equiv (\forall t : (i.t = r \wedge \mathrm{B}.t) \Rightarrow \mathrm{A}.(o.t))$$

Note that *safe*.re does not imply *safe*.r, and that r is trivially safe if no trace of B matches it. For any specification C, B $\|$ C satisfies A with respect to safety if and only if every trace of C is safe.

In the first phase of the algorithm, we construct a specification C satisfying the following:

(Safety) $\forall r\colon \mathrm{C}.r \Rightarrow \mathit{safe}.r$

(Maximality) $(\forall r\colon \mathrm{D}.r \Rightarrow \mathrm{C}.r)$, for any specification D such that B $\|$ D satisfies A with respect to safety.

The first requirement says that C is a solution with respect to safety; the second says that it has the largest possible trace set. To accomplish this, we must consider each trace over *Int* as a possible trace of C. Because trace sets are prefix-closed, the obvious way to do this is inductively, beginning with the empty trace.

In constructing C, we "tag" each state with information about the traces leading to it, the corresponding traces of B, and their projections on the *Ext* interface. This information enables us to ensure that every trace of C is safe, and also makes an inductive computation possible. We first introduce a mapping h from traces in Int^* to sets of pairs (a, b), where a is a state of A, and b is a state of B. The mapping is defined by

$$(a, b) \in h.r \equiv (\exists t : i.t = r \wedge b_0 \xrightarrow{t} b \wedge a_0 \xrightarrow{o.t} a).$$

Each pair in $h.r$ represents a possible state of B after some trace t matching r, and the state of A after the trace $o.t$. The idea is that from $h.r$, we determine what events might be enabled in B $\|$ C after C observes the behavior r at the B-C interface. We set up a bijection *tag* between such sets of pairs and states of C, and construct C so that $tag.c_0 = h.\varepsilon$, and for any trace r and state c such that $c_0 \xrightarrow{r} c$, $tag.c = h.r$. We know S_{A} and S_{B} are finite, hence the number of distinct sets of (a, b) pairs is finite. Since each state of C corresponds to a different set, S_{C} is finite.

We can check the safety of traces inductively using a predicate over these sets of pairs. For a set J of (a, b) pairs such that $J = h.r$ for some trace r, we define the predicate $ok.J$ by

$$ok.J \equiv (\forall a, b : (a, b) \in J : (\tau.b \cap Ext) \subseteq \tau.a)$$

Intuitively, $ok.J$ says that for every pair (a, b) in J, any event in Ext that is enabled in b is also enabled a. Note that $ok.J$ is easily checked by examination of J and the specifications A and B. The following properties are consequences of the definitions given above.

- $ok.(h.\varepsilon) \equiv safe.\varepsilon$
- For any $r \in Int^*$ and $e \in Int$: $safe.r \wedge ok.(h.re) \Rightarrow safe.re$

We begin the inductive construction of C by computing $h.\varepsilon$, and checking $ok.(h.\varepsilon)$, a necessary and sufficient condition for existence of a solution (with respect to safety). If $ok.(h.\varepsilon)$ holds, we create an initial state c_0 and set $tag.c_0 = h.\varepsilon$. Then we iterate, computing $h.re$ for each e from an already-computed (and safe) $h.r$, and adding a state with tag $h.re$, if $ok.(h.re)$ holds; this continues until closure is achieved.

To obtain $h.re$ from $h.r$, we define a function φ that maps a set J of (a, b) pairs and an action e to another set of pairs, such that if $J = h.r$, then $\varphi.(J, e) = h.re$. Such a function, easily computed from J, A, and B, is given by

$$(a, b) \in \varphi.(J, e) \equiv (\exists a', b', t : (a', b') \in J \wedge i.t = e \wedge b' \xrightarrow{t} b \wedge a' \xrightarrow{o.t} a)$$

Observe that $\varphi.(J, e)$ is empty if and only if the action e is not enabled in B in any of the possible states represented by set J. So if $tag.c = J$ and $\varphi(J, e) = \emptyset$, e will never occur at c. The quotient will in general have a "dead state," whose tag is the empty set, which can never be reached via any interaction with B. Figure 17 shows the safety phase of the algorithm, which implements this inductive construction. The internal transition relation $\mathrm{I_C}$ is defined to be empty. The state set S_{C} is empty upon termination of this phase if and only if B has a trace t such that $i.t = \varepsilon$, and $o.t$ is not a trace of A.

In the progress phase — which is executed only if the first phase produces a nonempty S_{C} — we identify states of B $\|$ C where a progress requirement of A is violated (because some required event is not enabled in B || C), and remove the corresponding states of C. We say a state c of C is *bad* if and only if

$$\exists a, b : (a, b) \in tag.c \wedge \tau.a \not\subseteq \tau^*.\langle b, c\rangle$$

From the properties of *tag.c* and the definition of satisfaction, it follows that B || C satisfies A with respect to progress if and only if C contains no bad states. Because the definition of a bad state depends on $\tau^*.\langle b, c\rangle$, which depends on T_C, if we remove any bad states and modify T_C we must then recalculate $\tau^*.\langle b, c\rangle$ for each b and c, and re-check for bad states. The process terminates when there are no more bad states to remove. Note that removing the initial state is equivalent to removing all remaining states, because it makes them all unreachable.

The progress phase of the algorithm is shown in Figure 18. This phase preserves the maximality of C: any trace removed from C's trace set cannot be a trace of any D such that B || D satisfies A. It follows that if the algorithm terminates with an empty state set, and therefore an empty trace set, no quotient exists.

5.5 Example

To apply the algorithm to our example AB-NS conversion problem, we have to model the transmission media as separate components; Figure 19 shows the protocol configuration. The converter interacts only with the channels, and not directly with A_0 and N_1. The specifications for channels ABchan and NSchan are shown in Figure 20. The unlabeled (internal) transitions in the specifications represent loss of a message; after each such loss, a timeout event occurs at the "Sender end" of the channel. In the case of NSchan, the "Sender end" is the converter end.

The specifications of A_0 and N_1 are as shown in Figures 4 and 5, without the "virtual message" transitions. The inputs to the quotient algorithm are A_0 || ABchan || NSchan || N_1 (this composite specification is not shown, but is straightforward to compute) and the service specification shown in Figure 21, which requires service similar to that of AB.

The output of the safety phase of the quotient algorithm for these inputs is shown in Figure 22. (For clarity, the "dead state" and transitions leading to it are omitted from the figures showing the output of the quotient algorithm.) This is a correct converter with respect to safety: all traces of the system A_0 || ABchan || C || NSchan || N_1 are prefixes of the sequence "acc, del, acc, del, ..." However, the converter is not correct with respect to progress. We have already seen the problem in previous sections: when a timeout occurs at the converter, there is no way to determine whether the loss occurred before or after the "del" action, and thus no more "D" messages can be forwarded. As soon as a loss occurs in NSchan, the system enters a set of states in which neither "del" nor "ack" can be safely enabled, while the service specification requires that one of these two be enabled at all times. In the progress part of the algorithm, states 3, 4, 6, 8, 12, 13, 15 and 17 are immediately marked "bad;" this leaves only states 0, 1, and 2 reachable, and they are removed in the second iteration. The algorithm terminates with a degenerate converter, and we conclude that the "exactly once" service of Figure 21 cannot be provided with the given protocol components.

The constructed quotient has the largest possible trace set, so it may contain states and transitions that are harmless, but contribute nothing to system progress; these useless parts of the converter are indicated in the figures by dotted boxes. The converter in Figure 22 can return the "wrong" acknowledgement to A_0 even after receiving the data message correctly. Removal of such superfluous sections simplifies the converter without affecting correctness, but is hard to accomplish automatically, and is best left to a human.

We can solve the problem of losses in NSchan by assuming that the converter is co-located with N_1. We then have the configuration of Figure 23. The inputs to the algorithm are $A_0 \parallel$ ABchan $\parallel N_1$, and the same service specification. The quotient algorithm yields a converter for this case; it is shown in Figure 24. Note that the "+D" action of the converter matches that of N_1, and denotes the passage of a data message from C to N_1; similarly, the "–A" action denotes passage of a message from N_1 to C.

An alternative way to solve the problem is to weaken the service specification to allow more than one "del" per "acc." To accomplish this with a deterministic service specification, we can model loss or correct transmission of a message with interface actions of NSchan (and of the service). Each time a message is submitted to NSchan, the environment chooses between "ls" and "xmit," representing the loss or correct transmission of the message. The "ls" and "xmit" actions are not part of the interface with the *user* of the service, but with another part of the environment that we regard as a random process, modeling the actual events that lead to message loss in channels. The service specification (Figure 25) indicates that "del" may occur until there are two "xmit" events in a row. In other words, only as many extra deliveries as necessary are allowed. The modified channel specification is shown in Figure 26. The output of the quotient algorithm for these inputs is shown in Figure 27. Removal of the "useless" states in the dotted boxes results in a converter similar to those obtained with the other two methods.

5.6 Discussion

The quotient algorithm can, in theory, be used to find a converter for any mismatch problem that can be represented by our finite-state specifications. However, the state set of the quotient is constructed so that each state corresponds to a set of pairs of states of A and B. There are $2^{|S_A| \times |S_B|}$ such sets of pairs, so the state set of the quotient can grow exponentially in the size of the inputs. In terms of running time, the problem of finding C such that B $\parallel$ C satisfies A with respect to *safety only* is hard enough that we cannot hope to do better than exponential time in the worst case. Interestingly, the progress phase does not add significantly to the overall worst-case running time of the algorithm: it takes time polynomial in the size (of the state set) of the output of the safety phase.

However, as our example showed, the algorithm does not always use exponential time and space. In some cases it can very quickly yield a correct converter or determine that we are dealing with a hard mismatch. In taking the quotient view of the conversion problem, we trade guaranteed efficiency for generality.

6 Architectural Issues

In the foregoing discussion, we considered a simplified form of the conversion problem in order to focus on solution methods. In practice, however, protocol mismatches may involve multiple layers in an architecture. In this section, we broaden our view to consider the problem of interconnection of layered networks with different architectures. Note that this is still an abstract view, in that we ignore issues such as addressing, routing, network management, etc.

Although protocol mismatches can occur at any layer, the problems of primary interest today occur at the network and transport layers. Figure 28 shows two "adjacent" networks, each with a different architecture. The network services are represented by a single box labeled NS in each network; the transport protocol peer entities are labeled TA_0 and TA_1, and TB_0 and TB_1, respectively, and the transport services are denoted by TS. The goal is to provide a transport service conforming to a service specification CST between the user on Network A and the user on Network B. (Note that these "users" may be peers of some higher-level protocol.)

We assume that the two networks are in some sense close to each other; that is, the two network services can be easily physically connected (perhaps, a host is connected to both networks). By connecting the different transport services to each other via a simple pass-through device (Figure 29), we provide a "concatenated" data transport service between the two users. However, any end-to-end synchronization capability of the existing services will not be preserved. In Figure 29, sychronization occurs only between user and converter; this is not sufficient for the transport level, which is supposed to provide end-to-end functionality. In particular, the connection management function is concerned only with synchronization. An example is the "orderly close" function, which guarantees that all user data have been delivered to the remote end if the connection closes normally. One user might successfully close the connection, and think that all data had been delivered to the far user, when it was actually only delivered as far as the converter.

One solution is to replace TA_1 and TB_0 with a converter, as shown in Figure 30. This is essentially the same configuration as in Figure 19, where the network service between the converter and the transport peer entities may be lossy. In terms of the simplified problem discussed earlier, we can consider TA_0 and NS_A as P_0, and TB_1 and NS_B as Q_1, and apply any of the three methods. A description of a conversion between two transport-level protocols, the DoD Standard Transmission Control Protocol [9] and the International Organization for Standardization's Transport Protocol Class 4 [17], can be found in [14].

Figure 31 shows a different approach, combining conversion with *augmentation*, the addition of a "sublayer" protocol in both architectures. This sublayer deals with routing and addressing, combining all the (intra-) network services into an (unreliable) internetwork service. An example of this approach is the Internet Protocol [8] used in the DARPA Internet, a collection of heterogeneous networks. In Figure 31, the internetwork service provides a transmission path between the transport peers TA_0 and TB_1. At that point, however, a protocol mismatch occurs. To handle the mismatch, a converter is co-located with the TB_1 implementation (it could also be placed at the TA_0 end). As in Figure 23, the configuration is asymmetric, because the path between converter and TA_0 is unreliable, while that between converter and TB_1 is (presumably) reliable. As we have seen from the AB-NS example, such a setup allows the converter to have better "knowledge" of the state of the local entity, and may allow a more powerful conversion system than would be possible in the symmetric configuration of Figure 30. With the internetwork service specified by IS, the required converter is the quotient of CST and $TA_0 \parallel IS \parallel TB_1$.

This configuration has other advantages. We have already noted that addressing issues are essentially confined to the network layer, at the boundary between networks. Another advantage is that, if both NS_A and NS_B provide alternate routing, and the two networks "intersect" at more than one place, then the conversion system can have the advantages of alternate routing. This is not possible when the converter is placed at the network boundary, and state information for each internetwork connection is maintained in the converter. (For a discussion of this and other issues related to transport-level "gateways," the reader is referred to [32].)

Although the problems of interest today are primarily at the transport level, it might be expected that in the future, solutions of one kind or another will be found, and an adequate, end-to-end, reliable transport service will be more or less universally available. At that point, the conversion problems of interest may be those at higher levels, as shown in Figure 32. AP_A and AP_B are application protocol peers that perform some similar function, and AS is the service to be provided by them. TS is a standard internetwork transport service, which both are designed to use.

As a simple example, AP_B might be a "yellow pages" server on Network B, and AP_A is a yellow pages client on Network A, designed to work with Network A's service. The converter serves as a "front man" for the B server, allowing Network A clients to access the service. At the same time, Network A clients can access the server directly. Interoperation of clients and servers using different protocols is discussed in [30]. The approach described there involves modification of the server entity to use a single protocol service, which can be implemented by placing a so-called "thin-veneer" on top of any of several different underlying protocols. This differs from protocol conversion in that the server must be modified.

7 Conclusions

We have formalized the problem of constructing a protocol converter as a way to overcome protocol mismatch, and discussed a range of approaches to solving the problem. The problem of finding a converter can be viewed as the problem of finding a "quotient" of specifications. For some classes of (finite state) protocols, general algorithmic methods for deriving a converter (solving a quotient problem) exist, but the problem is hard: the quotient algorithm has exponential worst-case running time.

Okumura's algorithm is efficient, but it can be applied only when a partial specification of the converter — the seed — is known. If it terminates without producing a converter, it is difficult to conclude that a hard mismatch exists, because a converter might exist for a slightly different seed.

Lam's projection approach provides a *sufficient condition* for finding a useful converter, based upon our intuitive understanding of the protocols; as such, it is a heuristic. It is useful for formalizing semantic equivalences between protocols, and can be used to reason formally about correctness of converters obtained by other methods.

Even if convergence to a "universal" network architecture is achieved, different implementations of the same protocol standard may not be compatible with each other. We are also witnessing a proliferation of variants of the same standard as time goes by. Thus, protocol conversion will remain a problem for the forseeable future.

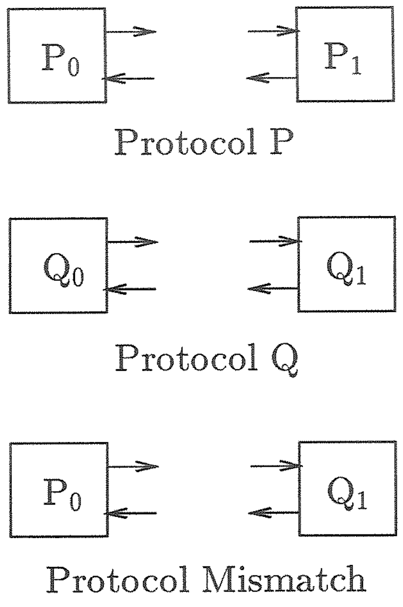

Figure 1: Protocol configurations

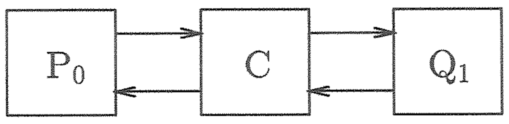

Figure 2: Interposing a protocol converter

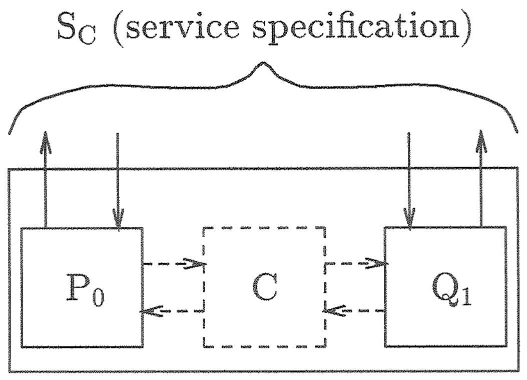

Figure 3: The abstract problem

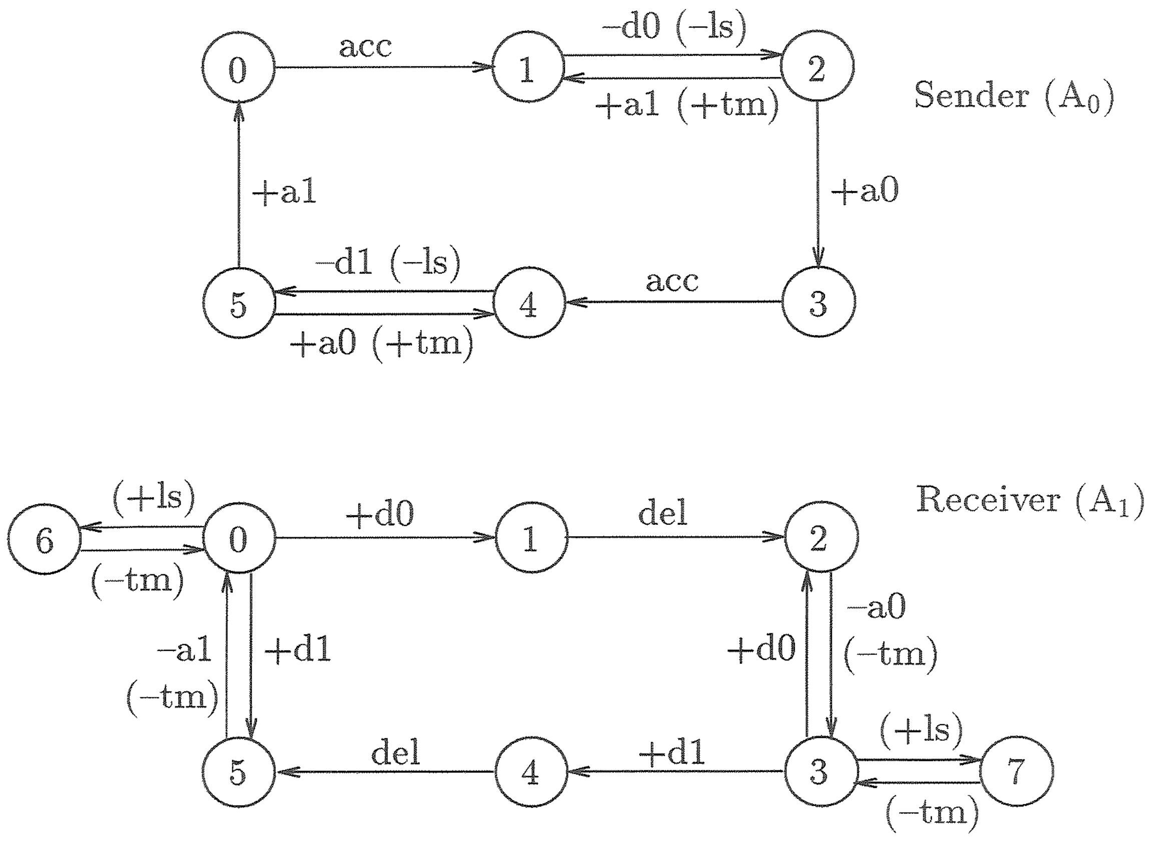

Figure 4: Alternating Bit Protocol

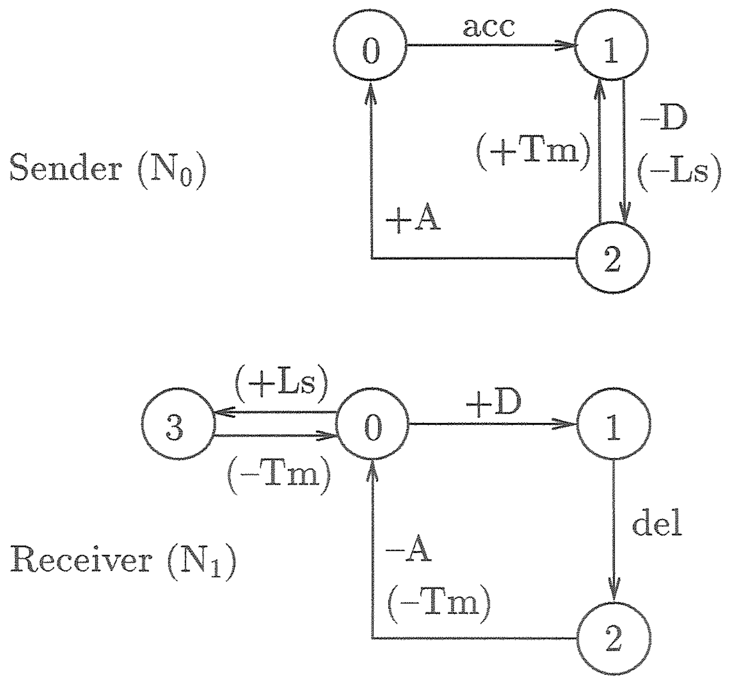

Figure 5: Non-sequenced Protocol

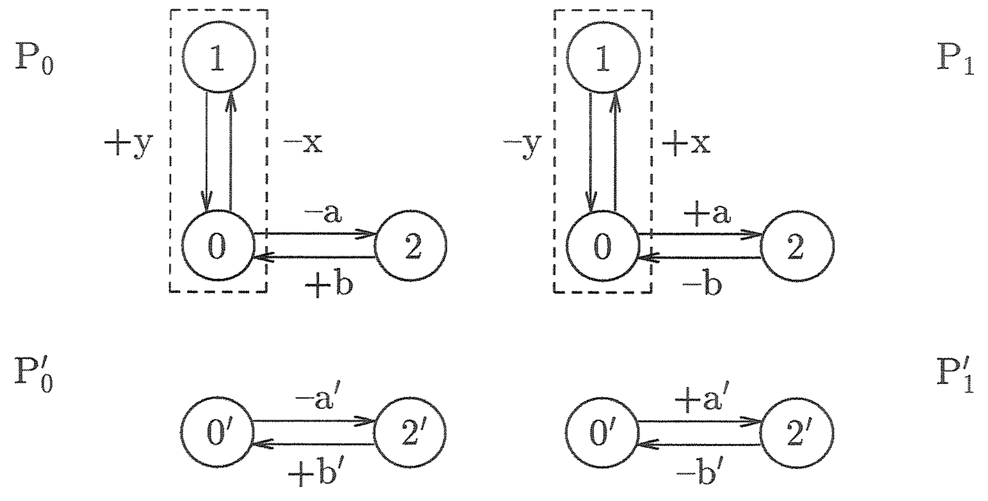

Figure 6: Simple projection example

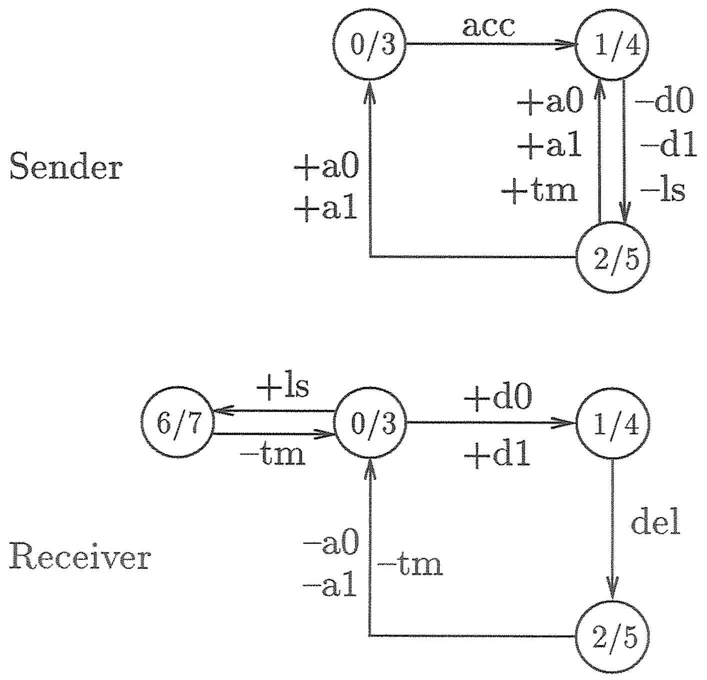

Figure 7: Projection of AB

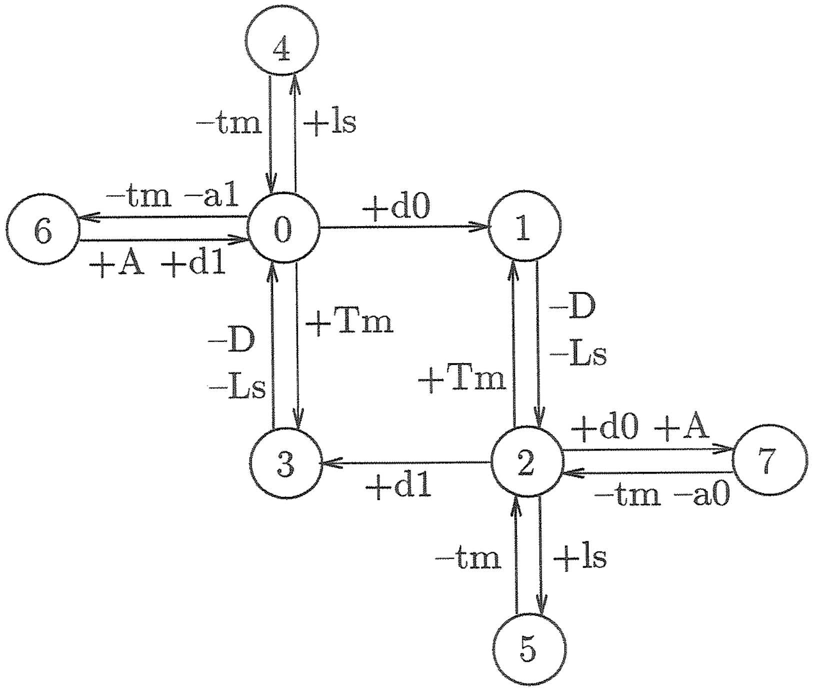

Figure 8: Converter for AB sender and NS receiver

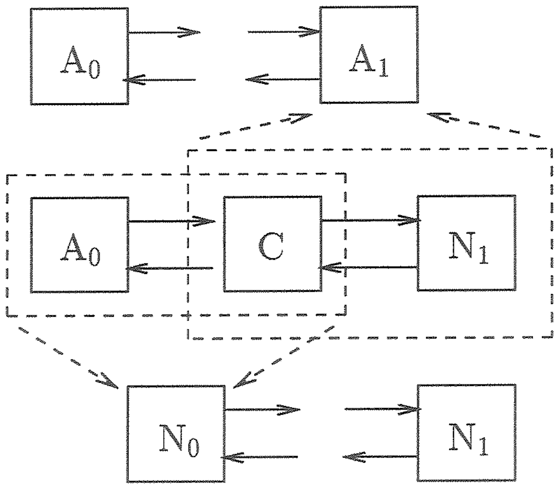

Figure 9: Projection of converter system

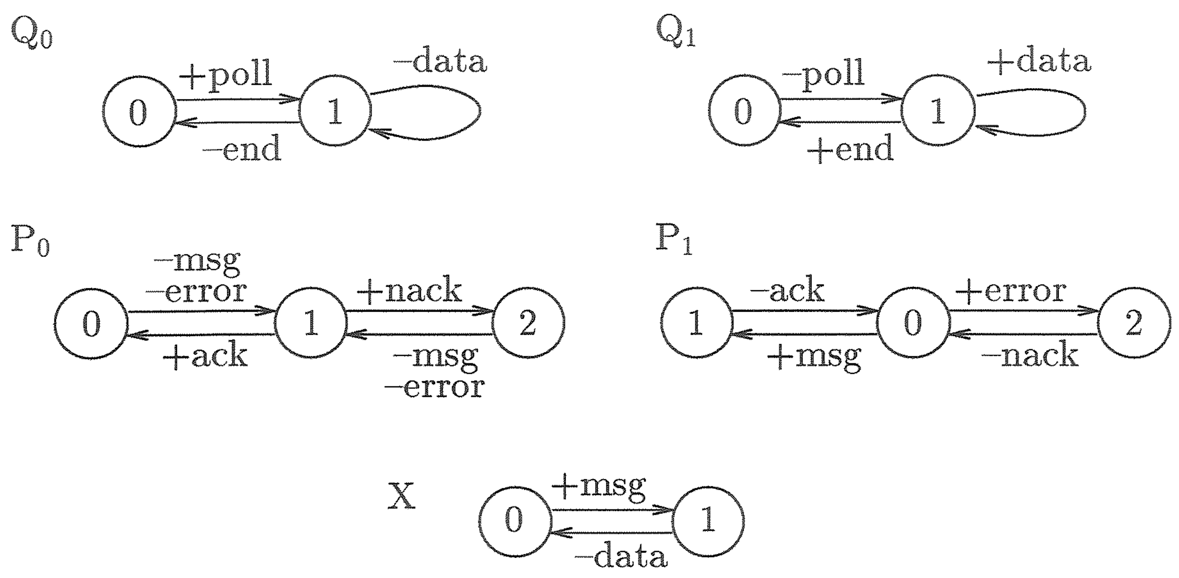

Figure 10: Example conversion problem with seed (X)

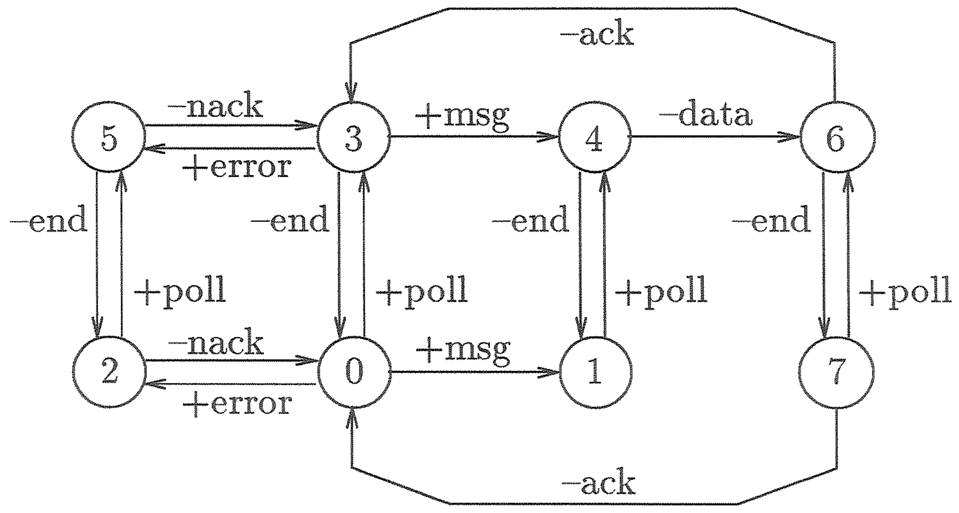

Figure 11: Converter for P_0 and Q_1

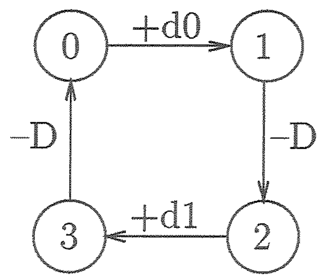

Figure 12: Improper seed for AB-NS example

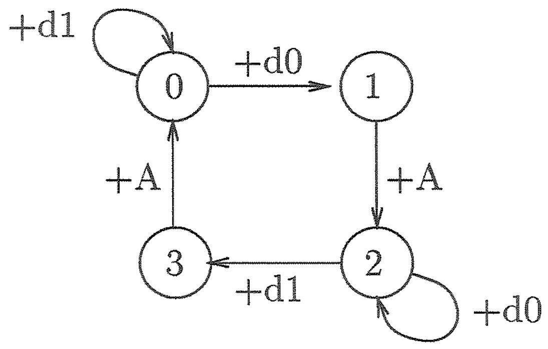

Figure 13: Correct seed for AB-NS example

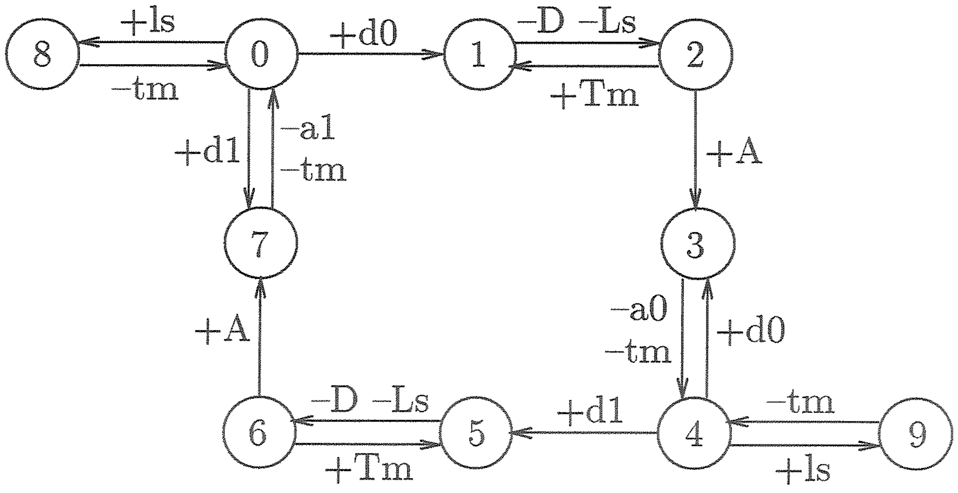

Figure 14: Resulting converter

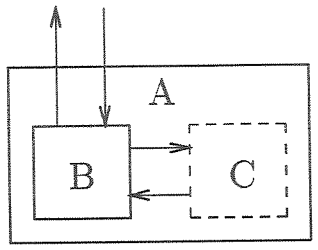

Figure 15: Quotient problem

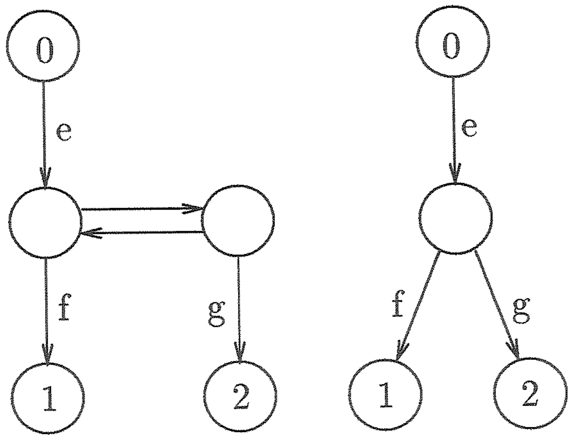

Figure 16: Collapsing internal cycles

```
$S_C := \emptyset$; $new := \emptyset$;
$tag.c_0 := h.\varepsilon$;
if $ok.(tag.c_0)$ then $new := \{c_0\}$;
while $new$ is not empty
      select $c$ in $new$;
      for each $e$ in $Int$:
            $J := \varphi(tag.c, e)$;
            if $ok.J$ then
                  if $tag^{-1}.J \notin (S_C \cup new)$
                        then create $c'$;
                             $tag.c' := J$;
                             add $c'$ to $new$;
                        else $c' := tag^{-1}.J$
                  add $c \xrightarrow{e} c'$ to $T_C$;
      move $c$ from $new$ to $S_C$;
```

Figure 17: Algorithm – safety phase

```
repeat
    $save := S_C$;
    compute $\tau^*.\langle b, c\rangle$ for each $b, c$ pair;
    foreach $c \in S_C$:
        foreach $(a, b) \in tag.c$:
            if $\tau.a \not\subseteq \tau^*.\langle b, c\rangle$ then
                mark $c$ bad;
    remove bad states and their
        associated transitions from $S_C$ and $T_C$;
until $c_0$ is removed or $save = S_C$
```

Figure 18: Algorithm – progress phase

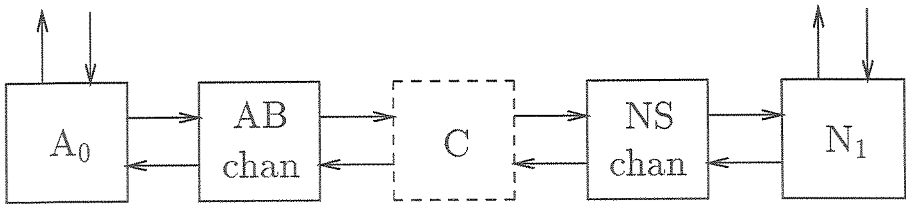

Figure 19: Example configuration

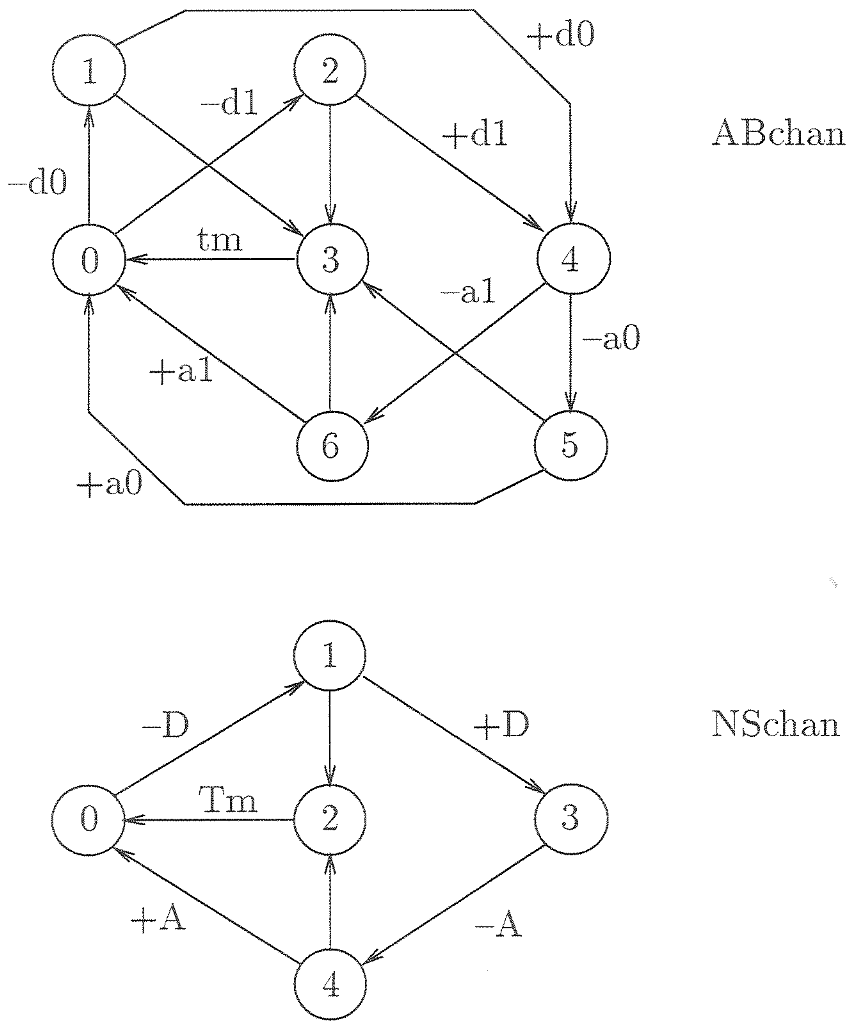

Figure 20: Channel specifications

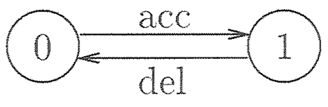

Figure 21: Service specification

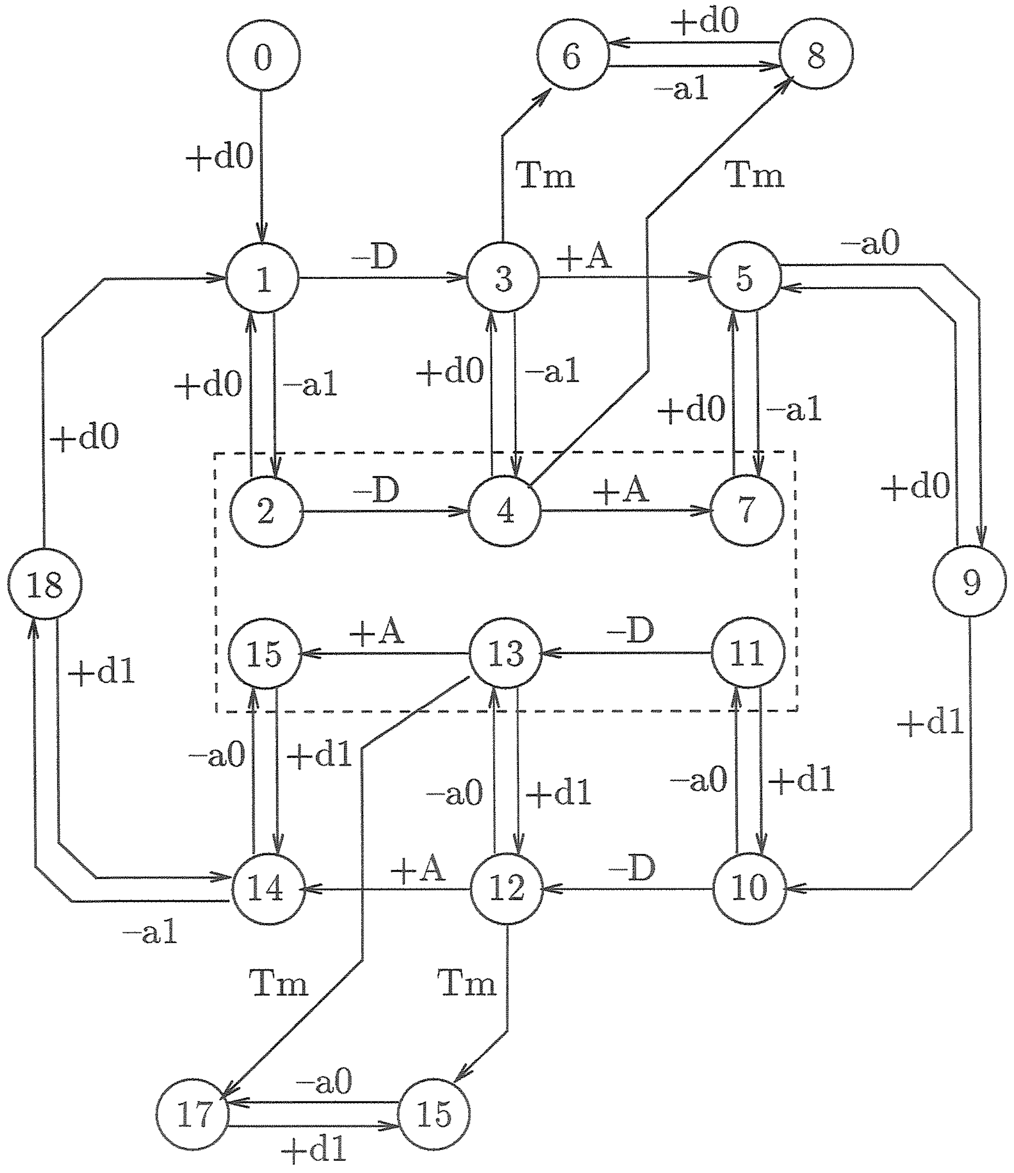

Figure 22: Output of safety phase of algorithm

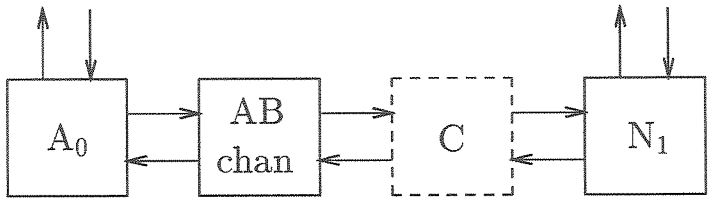

Figure 23: Converter co-located with N_1

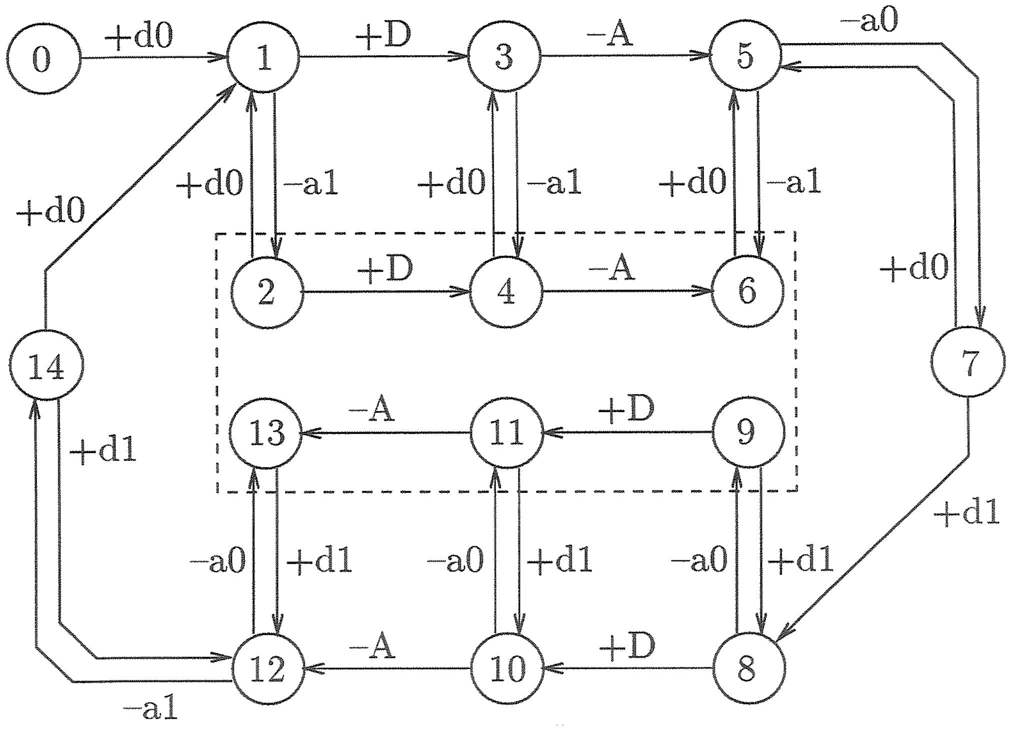

Figure 24: Resulting converter

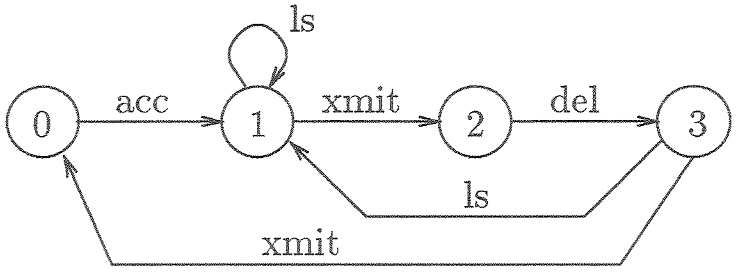

Figure 25: Weaker service specification

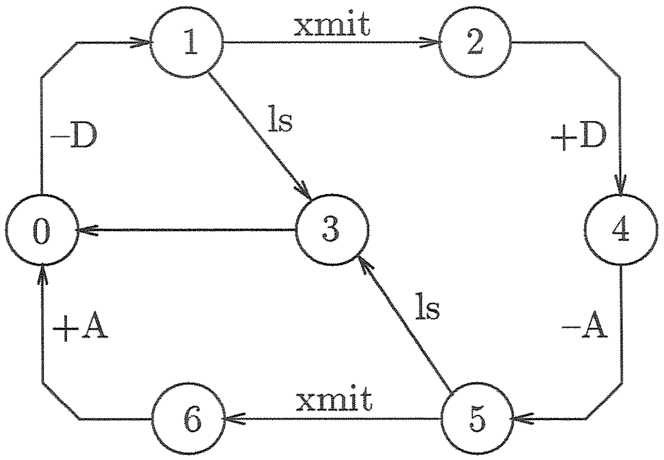

Figure 26: Modified NSchan specification

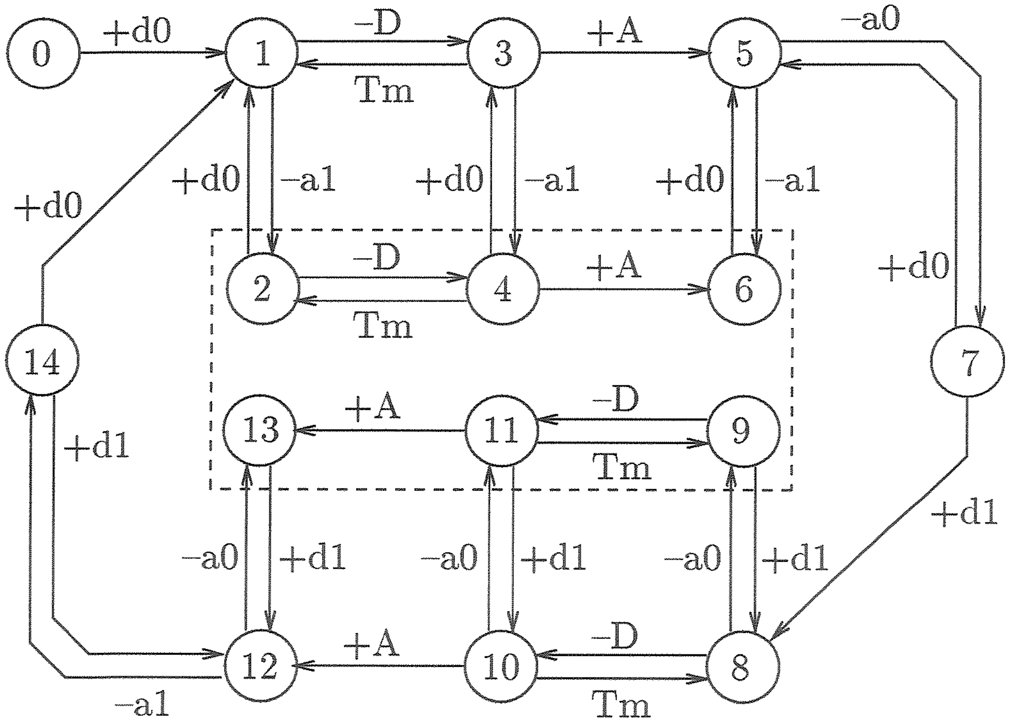

Figure 27: Quotient for weaker service

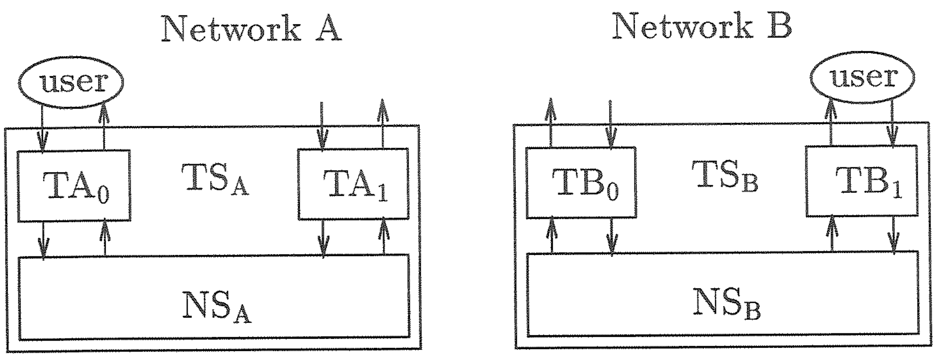

Figure 28: Heterogeneous networks

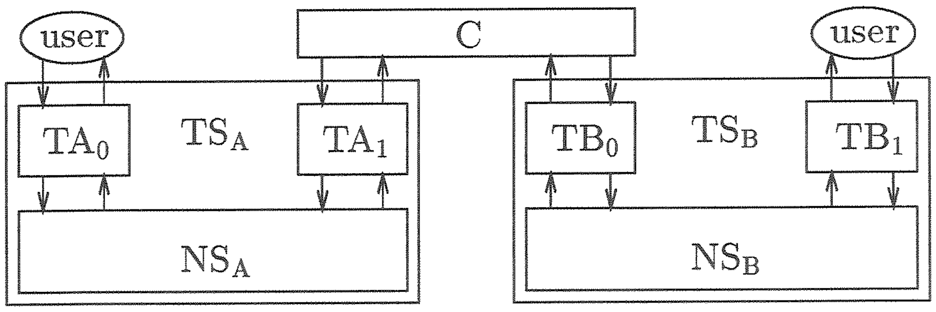

Figure 29: "Going up a level"

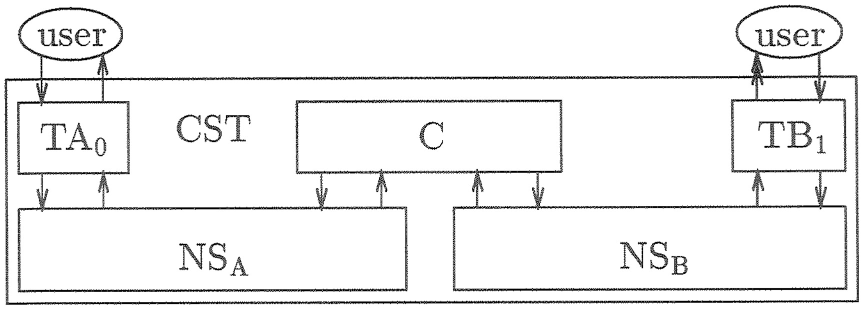

Figure 30: Transport-level conversion

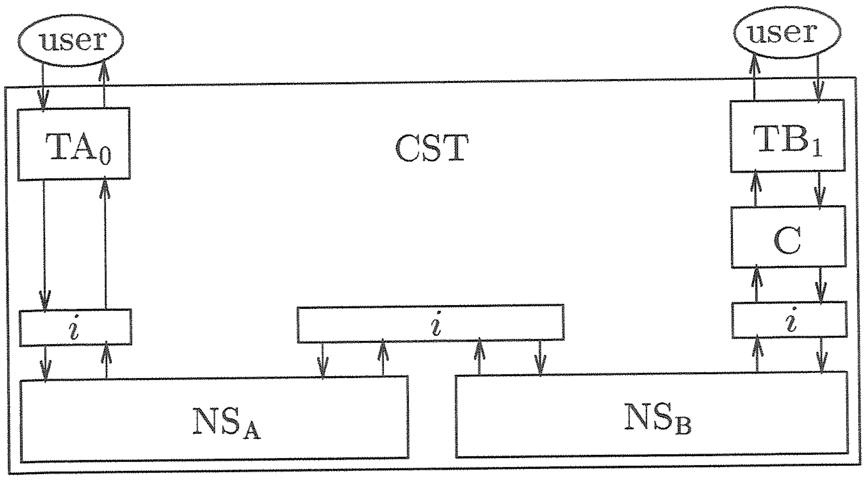

Figure 31: Asymmetric configuration

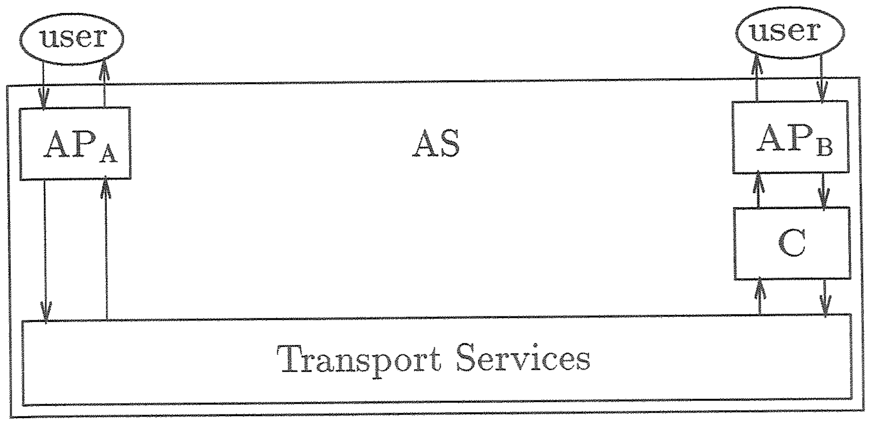

Figure 32: Application-level conversion

References

[1] K. L. Calvert and Simon S. Lam. An exercise in deriving a protocol conversion. In *Proceedings of SIGCOMM '87 Workshop, Stowe, VT*, 1987.

[2] K. L. Calvert and Simon S. Lam. Deriving a protocol converter: a top-down method. In *Proceedings of SIGCOMM '89 Workshop, Austin, TX*, 1989.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[4] H. Cho and S. I. Marcus. On maximal sublanguages arising in supervisor synthesis problems with partial observations. In *Proceedings of 22nd Annual Conference on Information Sciences and Systems, Princeton, NJ*, 1988.

[5] C-H. Chow, M. G. Gouda, and S. S. Lam. A discipline for constructing multiphase computer communication protocols. *ACM Transactions on Computer Systems*, 3(4):315–343, November 1985.

[6] R. Cieslak, C. Desclaux, A. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, March 1988.

[7] David D. Clark, Mark L. Lambert, and Lixia Zhang. NETBLT: A high throughput transport protocol. In *Proceedings ACM SIGCOMM '87 Workshop, Stowe, VT*, 1987.

[8] J. Postel (ed.). Internet protocol specification. DARPA Internet Request for Comments 791, September 1981.

[9] J. Postel (ed.). Transmission control protocol specification. DARPA Internet Request for Comments 793, September 1981.

[10] D. Einert and G. Glas. The SNATCH gateway: Translation of high-level protocols. *Journal of Telecommunications Networks*, 1983:83–102, 1983.

[11] M. Garey and D. Johnson. *Computers and Intractability.* Addison-Wesley, 1979.

[12] B. C. Goldstein and J. M. Jaffe. Data communications: the implications of communications systems for protocol design. *IBM Systems Journal*, 26(1), 1987.

[13] Paul E. Green, Jr. Protocol conversion. *IEEE Transactions on Communications*, COM-34(3), March 1986.

[14] Inge Groenbak. Conversion between the TCP and ISO transport protocols as a means of achieving interoperability between data communications systems. *IEEE Journal on Selected Areas in Communications*, SAC-4(2), March 1986.

[15] Matthew Hennessey. *Algebraic Theory of Processes.* MIT Press, 1988.

[16] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1986.

[17] ISO. Connection oriented transport protocol specification. ISO 8073-CCITT X.224, July 1986.

[18] ISO/TC97/SC21/WG16-1. LOTOS — a formal description technique based on the temporal ordering of observational behavior, March 1985.

[19] Bengt Jonsson. Modular verification of asynchronous networks. In *Proceedings ACM Symposium on Principles of Distributed Computing, Vancouver, BC.*, 1987.

[20] Simon S. Lam. Protocol conversion — correctness problems. In *Proceedings ACM SIGCOMM '86, Stowe, VT*, 1986.

[21] Simon S. Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, March 1988.

[22] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[23] Simon S. Lam and A. Udaya Shankar. A relational notation for state transition systems. In *Proceedings of the Eighth International Symposium on Protocol Specification, Verification, and Testing.* North-Holland, 1988. Full version available as Technical Report TR-88-21, Dept. of Computer Sciences, Univ. of Texas at Austin, May 1988 (revised August 1989).

[24] Simon S. Lam and A. Udaya Shankar. Specifying two implementations to satisfy a serializable database interface. Technical Report TR88-30, University of Texas at Austin, Department of Computer Sciences, August 1988. (revised July 1989).

[25] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[26] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings ACM Symposium on Principles of Distributed Computing, Vancouver, BC.*, 1987.

[27] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, December 1984.

[28] Philip M. Merlin and Gregor v. Bochmann. On the construction of submodule specifications and communications protocols. *ACM Transactions on Programming Languages and Systems*, 5(1), Jan 1983.

[29] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[30] D. Notkin, A. Black, E. Lazowska, H. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogenous computer systems. *Communications of the ACM*, 31(3), 1988.

[31] K. Okumura. A formal protocol conversion method. In *Proceedings ACM SIGCOMM '86, Stowe, VT*, 1986.

[32] M. A. Padlipsky. Gateways, architectures, and heffalumps. DARPA Internet Request for Comments 875, September 1983.

[33] Joachim Parrow. Submodule construction as equation solving in CCS. In *LNCS 287: Proceedings of Seventh Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 1987.

[34] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control and Optimization*, 25(1):206–230, January 1987.

[35] K. Sy, O. Shiobara, M. Yamaguchi, Y. Kobayashi, S. Shukuya, and T. Tomatsu. OSI-SNA interconnections. *IBM Systems Journal*, 26(2), 1987.

List of Figures