Deriving a Protocol Converter: a Top-Down Method*

Kenneth L. Calvert and Simon S. Lam
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

A *protocol converter* mediates the communication between implementations of different protocols, enabling them to achieve some form of useful interaction. The problem of deriving a protocol converter from specifications of the protocols and a desired service can be viewed as the problem of finding the "quotient" of two specifications. We define a class of finite-state specifications and present an algorithm for solving "quotient" problems for the class. The algorithm is applied to an example conversion problem. We also discuss its application in the context of layered network architectures.

1 Introduction

The interconnection of computer networks has made it possible to provide an information path between practically any two computer systems. Unfortunately, the ability to move bits between machines does not imply the capability for useful interaction: the connected systems must also have some common ground in terms of protocols. This situation is depicted schematically in Figure 1. System P_0 is designed to communicate with system P_1 according to protocol P, while Q_0 and Q_1 are designed to communicate according to protocol Q. If P_0 needs to interact with Q_1, a *protocol mismatch* exists.

The existence of different protocols for the same function is a fact of life that, for various reasons, is unlikely to change. Communication protocols, like other products, evolve with technology. As new protocols replace old ones, several "generations" of architecture must coexist, and upward compatibility will eventually be sacrificed for superior quality. Another factor is the desirability of having different protocols for the same general purpose, in order to serve the needs of different user communities. For example, a protocol optimized for transfer of bulk data over long-haul networks may differ from one optimized for transfer of interactive terminal-session data over the same networks [5].

The obvious solution to the protocol mismatch problem is to modify one or both of the installations so that they use the same protocol. Unfortunately, this can be expensive, especially when the mismatch involves several layers of the architecture. We call the problem of overcoming a protocol mismatch *without* an extensive modification of the existing systems the *protocol conversion* problem.

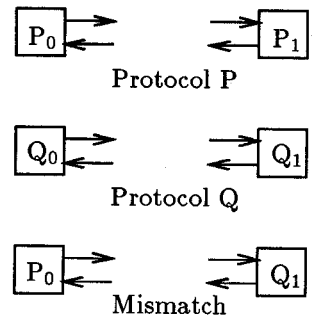


Figure 1: Protocol configurations

The approach we consider in this paper is to place an intermediary, or *protocol converter*, between P_0 and Q_1; the converter "translates" between the two protocols and allows P_0 and Q_1 to achieve some degree of useful interaction (Figure 2). However, it is not immediately clear what constitutes a correct "translation" of one protocol into another [13], or when such a trans-

Figure 2: Interposing a protocol converter.

*work supported by the National Science Foundation under grant number NCR-8613338

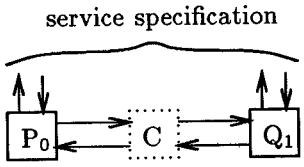


Figure 3: The abstract problem

lation is possible. For this reason, we turn to formal methods, which enable us to reason about the relationship between protocols and the services they are intended to implement.

In an earlier paper that considered many aspects of protocol conversion, Green [7] proposed the formulation of "a calculus of conversion" based on formal methods. In this paper, we present one part of such a calculus, a method for computing a converter specification from specifications of the mismatched protocol components and the service that they must provide to their users. The method differs from others that have been proposed, in that if it fails to produce a converter, no converter exists for the given inputs, i.e., the given protocol components cannot be used to provide the given service.

In the next section, we discuss the abstract problem and how our approach differs from others. In Section 3 we present a system of specifications for finite-state protocols and services, and define what it means for interacting protocol components to satisfy a service specification. In Section 4 we develop an algorithm for computing a solution to the problem, and give an example of its use in Section 5. In Section 6, we consider the problem in the context of layered network interconnection. In the final section, we sum up, and consider future work.

2 The Problem

Suppose a service (e.g., a data transport service) is to be provided to some users, using components of different protocols, say P_0 and Q_1. We would like to construct a component C to mediate the communication between P_0 and Q_1, and enable them to provide the desired service (Figure 3). However, we do not know whether any such component is possible. An algorithmic solution to this problem would take as input the specifications of the components P_0 and Q_1, and a specification of the desired service, and produce a correct specification of C, if and only if such a C exists.

Previous proposed approaches to the problem of finding a converter have been based on a somewhat different notion of correctness of the conversion system. In the method of Okumura [17], the problem is assumed to be as depicted in Figures 1 and 2. The specifications of the "missing" entities (in this case P_1 and Q_0) are part of the problem input. Also, instead of the service specification, a partial specification of the converter (called a *conversion seed*), is assumed to be part of the input. The conversion seed represents properties of the conversion system in terms of the converter itself, as opposed to the global specification of what is required by its users. Okumura gives an efficient algorithm for constructing a converter specification from the specifications of P_1, Q_0, and the conversion seed. If the algorithm fails to produce a converter, then one may conclude that none exists for the given inputs, but this does not mean that none is possible for the given global service specification.

Lam [14] has also shown how a converter can be heuristically derived in some cases by examining the existing protocol systems and finding a *projection* of each existing system onto a common image. (Note: this amounts to showing that each existing system satisfies the same abstract service specification.) When such a common image can be found, a simple, stateless converter is easily obtained. In this case, the common image protocol provides the definition of the service to be implemented by the conversion system. Calvert and Lam [3] showed how projection could also be used to reason about the correctness of more complex converters.

These approaches attempt to solve the problem by relating some part of the conversion system to the missing entities of the original protocols. Their starting point is the existing protocols, including components that would not be part of the intended conversion system. They are "bottom-up," in the sense that if a converter is found, the whole system must then be checked to see if it implements the required global service. If it does not, another converter must be sought.

In this paper, we consider a "top-down" method, in which the converter is derived from the relevant parts of the conversion system, without reference to the "missing" entities of the protocols. The input consists of the specifications of the conversion system components, P_0 and Q_1, and a service specification, S. Our method must determine whether there exists a C, such that C, P_0, and Q_1 interacting together provide the service defined by S, and if so, produce a specification. By analogy with number problems, we call this the *quotient* problem — in effect we want to "divide" S by the given specifications P_0 and Q_1.

Problems of this kind arise in many areas of computing. Merlin and Bochmann [15] considered the "submodule construction" problem, and presented a solution adequate for specifications of safety properties. More recently, a semi-algorithmic method based on CCS specifications has been described [19], and a somewhat similar problem called the "supervisor synthesis

problem" has been discussed in the control-theory literature [20]. Our method is the first that we know of to deal with progress properties.

Our view of the problem is a very high-level one. We have abstracted away all the particulars of the protocols — their roles in the architecture as well as details such as timeout values and window sizes. We regard P_0 and Q_1 as mathematical objects of some known form. Note that we have thrown away some information that is used in the bottom-up methods — the specifications of P_1 and Q_0, along with the services provided by P and Q. This results in a more general method, but as might be expected, there is a cost in terms of additional computational complexity. The existence of such an algorithmic solution method depends on the form of the given specifications, and the classes of systems they represent. If the systems specified can be completely general (i.e., Turing machines), then no general algorithmic method is possible (any such would entail solving the halting problem). However, by restricting the classes of systems and the forms of specifications, algorithmic methods can be found.

3 Specifications

In this section, we describe a theory of specifications for a class of concurrent finite-state systems. The restriction to finite systems makes an algorithmic solution to the quotient problem possible, albeit computationally hard. Systems are modeled as finite state machines interacting via named *events*. This form of interaction is similar to that of CSP [9] and LOTOS [12, 2]. Besides being easy to represent and manipulate, the finite state machine model is well understood and has a long history of use in specifying and analyzing protocols.

We are concerned with two main ideas: a notion of *composition* of systems, i.e., viewing a system of interacting components together as a composite whole; and a notion of what it means for one system to *satisfy*, or *implement*, another. These appear in the theory as a composition operator and a satisfaction relation on specifications. Together, they allow us to reason about whether a specified system of interacting protocol components correctly implements a specified service.

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

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

The set Σ of events of a component completely defines its interface with the environment (by "environment," we mean its users or other systems with which it is composed). The events of the interface are the *only* way components can interact; intuitively, they model an exchange of information or handshake across the interface, possibly involving a state change on both sides. In a protocol or service specification, events are abstractions of relatively complex occurrences such as submission of a message for transmission, or expiration of a timer.

The relations T and λ define the *transitions* of the system, i.e., how its state may change over time. Each state change in T has an associated interface event in Σ; these define how the system's state is affected by interaction with its environment. If (s, e, s') is in T, we say e is *enabled* in s, and write $s \xrightarrow{e} s'$. Whenever the system is in state s, and the event e is also enabled in the environment, e may occur, and change the state of the system to s'. It is important to realize that external events are not under the exclusive control of either side of the interface, but can only occur when enabled on *both* sides. If multiple events are enabled in a state s, then the system cannot prevent any of them from occurring whenever it is in state s.

The relation λ defines *internal* state transitions that may occur unobserved, *without* environmental interaction. When (s, s') is in λ, we write $s \ \lambda \ s'$. Internal transitions introduce *nondeterminism* into specifications, by allowing some state transitions to occur under the exclusive control of one side of the interface. Nondeterminism complicates the theory, but plays at least two important roles in specifications. In specifying a service, it can represent a choice among multiple acceptable behaviors. For example, in a transport service, if an incorrectly-formatted service data unit is submitted, the acceptable behaviors might include ignoring it, or responding with an error message. This kind of nondeterminism allows high-level specifications to avoid unnecessary overspecification. The choice among the acceptable behaviors only needs to be made once, when designing an implementation.

Nondeterminism is also useful as an abstraction mechanism in describing an implementation. Here, internal transitions model low-level behavior that may be random *or* deterministic, but which 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. The composition operator, in abstracting from the interactions among the components of a system, also introduces internal transitions. This kind of nondeterminism is often assumed to be *fair*, meaning that an internal transition that is repeatedly enabled will eventually

occur.

Instead of having explicit fairness requirements attached to each specification, we make certain assumptions about fairness. In defining "B satisfies A" for specifications B and A, we shall regard A as a service specification, and assume that all nondeterminism in A is unfair. We view B as the specification of an implementation, and assume that all nondeterminism in B is fair. These assumptions yield a considerable simplification of the theory at a relatively small cost in generality.

It is sometimes helpful to view a specification as a directed graph with labeled edges, and we represent them as such in the figures. Nodes of the graph denote states; edges of the graph denote transitions. The label of an edge is its corresponding event, if any; transitions in λ have no label. In what follows, we use upper case letters A, B, C, and D to name specifications. Components of different specifications are distinguished by subscripts. The states of a specification are represented by (primed) lower-case italic letters corresponding to the name of that specification: a and a' are members of S_{A}, etc. 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 $\mathrm{T_A}$, while the other is in T_B.

Composition

Composition of systems involves making each a part of the other's environment, and viewing them together as a composite whole; their interactions with each other are synchronized and hidden from the rest of the environment. This operation is modeled by a binary composition function || on specifications.

Definition. For any specifications A and B, (A||B) is given by:

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

Each internal transition of the composite comes from one of two sources: an internal transition in one of the components, or a synchronized event of the interface between them that becomes hidden in the composition. Note that interface events that are not enabled in both components do not appear in the composite.

Satisfaction

Our definition of satisfaction of one specification by another has two parts, one dealing with *safety* ("what *can* happen"), and one dealing with *progress* ("what *will* happen"). Thus, "B satisfies A" if and only if "B satisfies A with respect to both safety and progress." Satisfaction with respect to safety is a necessary condition for satisfaction with respect to progress, so we deal with it first.

A *trace* is a sequence of events in Σ, and represents a possible behavior of the system, as it might be observed by its environment. In terms of the directed graph structure, we say a trace *corresponds to* the sequence of labels along a finite directed path in the graph. We associate a particular set of traces with each specification, namely those corresponding to paths in the graph beginning at the initial state. This set includes all possible behaviors of the system, and thus captures all of its safety properties. Note that trace sets are prefix-closed, and hence the empty trace, denoted by ε, is a possible behavior of every system.

In what follows, the letters t, r, q, t', etc. denote arbitrary traces, while e denotes an arbitrary event. Events are treated as traces of length one, and concatenation is denoted by juxtaposition: te is a trace ending with event e. For specification A, we write A.t for "t is a trace of A." With this one exception, predicates will be represented by words or parts of words, while functions are denoted by single letters.

Definition. The relation λ^* is the reflexive and transitive closure of λ. Thus, $s\ \lambda^*\ s'$ means s' is reachable from s via zero or more internal transitions.

Definition. Every specification defines a relation "$\rightarrow$" which is the least relation satisfying

for any states s, s', s'', trace t, and event e:

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

Intuitively, $s \xrightarrow{t} s'$ says that there is a path from s to s' corresponding to trace t. Also, we write $\overset{t}{\mapsto} s$ as shorthand for $s_0 \xrightarrow{t} s$.

Definition. For any specification A, A.$t \equiv (\exists a : \overset{t}{\mapsto} a)$.

The trace set interpretation yields a simple definition of satisfaction with respect to safety: for specifications

A and B with the same interface, "B satisfies A with respect to safety" means that every possible behavior of B is a possible behavior of A.

Definition. For A and B with the same interface, 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 cooperate in order for something to happen, our notion of *progress* deals with what events are enabled in the system after any particular trace. Given this information, the environment can ensure that there is always *some* event enabled on both sides of the interface, and thus prevent deadlock. Intuitively, our definition of "B satisfies A with respect to progress" should say "any environment guaranteed not to deadlock with A is certain not to deadlock with B." This notion of progress is similar to the "refusals" of Hoare [9], or the "acceptance sets" of Hennessey [8].

If there is at most one path corresponding to any trace (i.e., if the specification is *deterministic*), then the environment can always determine the state of the system after any trace, and thus "know" what events are enabled. If nondeterminism is present, things are more complicated. For one thing, a transition associated with an external event is *not guaranteed to be accepted* in a state with an outgoing internal transition, because it may be "pre-empted" by occurrence of the internal transition. So we can only talk about sets of events being enabled in states with no outgoing internal transitions. However, a trace may lead to a cycle of internal transitions. If these internal transitions occur continuously, the system may never enter a state in which no internal transition is enabled. What can we say about which events the system is "guaranteed" to accept in this case?

At this point, recall our discussion about fairness. If nondeterminism is assumed to be fair, no internal transition can pre-empt an external event or another internal transition infinitely many consecutive times. If there are internal transitions that leave the cycle, then eventually one of them must occur. If no internal transitions leave the cycle, then eventually one of the external events enabled on the cycle will occur. This means that a cycle of internal transitions can be regarded as a single state for the purposes of defining the set of enabled events; the set of events enabled in a state on such a cycle is therefore the set of all events enabled in any state on the cycle.

We call a the states on a cycle of internal transitions with no internal transitions leaving the cycle a "sink set." Consider Figure 4. In the left-hand specification, the two unlabeled states constitute a sink set; once either of them is reached, the set of events enabled is f, g. Thus we can view the sink set as a single state as on the right-hand side.

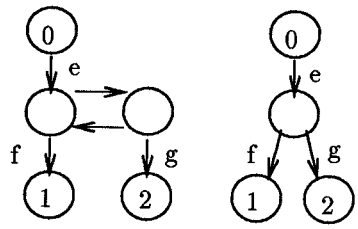


Figure 4: Collapsing internal cycles

Property. For any s, "s is in a sink set" if and only if $(\forall s' : s\ \lambda^*\ s' \Rightarrow s'\ \lambda^*\ s)$. We write *sink.s* to denote the latter predicate.

Recall that in defining satisfaction of A, we regard A as a service specification, and its nondeterminism as representing choices among different acceptable behaviors. Because a cycle of such choices does not make sense, we shall assume that A does not contain any cycle of internal edges. In fact, we require that A be in a certain "normal form."

Definition. A specification is in "normal form" iff:

(i) no state has both internal and external transitions leaving it.

(ii) For any s and s', $s\ \lambda^*\ s' \wedge s'\ \lambda^*\ s \Rightarrow s = s'$.

(iii) For any states s, s', s'', $\hat{s}$, and $\hat{s}'$:

$$s\ \lambda^*\ s' \wedge s\ \lambda^*\ s'' \wedge s' \xrightarrow{e} \hat{s} \wedge s'' \xrightarrow{e} \hat{s}' \Rightarrow \hat{s} = \hat{s}'$$

This restriction ensures that for each trace of A, every path corresponding to that trace passes through a particular state. In other words, if A is in normal form, then for each trace t such that A.t, there is a unique state a such that $(\forall a' : \overset{t}{\mapsto} a' \equiv a\ \lambda^*\ a')$. We denote this state by $\psi_{\mathrm{A}}.t$. Note that normal form allows nondeterminism, but "focuses" it so that after any trace, the sets of events that may be enabled after that trace are easily determined. Also, in a normal form specification, all sink sets are singletons.

We now present the rest of the definitions necessary to define "B satisfies A with respect to progress."

Definition. $\tau.s$ denotes the set of external events associated with transitions enabled in state s:

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

Definition. $\tau^*.s$ denotes the set of all external events enabled in any state internally reachable from s:

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

Intuition: $\tau^*.s$ contains all states that *may* occur next if the system's current state is s.

Definition. For states a and b (of specifications A and B, respectively), the predicate $prog.a.b$ means

$$(\exists a' : a \; \lambda^* \; a' \wedge sink.a' \wedge \tau^*.a' \subseteq \tau^*.b)$$

The intuition here is that if b is reachable via some trace t in B, and a is reachable via the same trace in A, then any environment that will not deadlock with A after t will not deadlock with B, because A may be in a sink set where every enabled event is also enabled in B. We can now define satisfaction with respect to progress.

Definition. For A and B such that

(i) A is in normal form

(ii) Nondeterminism in B is fair and nondeterminism in A is not fair

(iii) B satisfies A with respect to safety

B satisfies A with respect to progress if and only if

$$\forall t, b : (\stackrel{t}{\mapsto} b \wedge sink.b) \Rightarrow prog.(\psi_A.t).b$$

It turns out that because a sink set is reachable from every state, the above formula is equivalent to

$$\forall t, b : \stackrel{t}{\mapsto} b \Rightarrow prog.(\psi_A.t).b$$

Thus, we may ignore sink sets when actually using this definition.

4 Quotient Algorithm

Returning now to the quotient problem, we are given specifications A and B, such that $\Sigma_A = Ext$, $\Sigma_B = Int \cup Ext$, and Int and Ext are disjoint event sets. We must produce C such that $\Sigma_C = Int$, and B||C satisfies A, or show that no such specification exists. The events in Int constitute the interface between B and C. The events in Ext are B||C's interface to the environment as well as A's interface. In terms of the conversion problem of Figure 3, B is $P_0||Q_1$, and A is the service specification. 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.

In computing C, we deal with safety and progress in separate phases. In the first phase, we construct the state set and transition relation of C inductively, beginning with the initial state. The result is a C_0 with the largest trace set consistent with safety of B||C. In the second phase, we iteratively remove states of C_0 that do not have sufficient outgoing external transitions to prevent a possible progress violation. When this phase terminates, if C has a nonempty state space, then it is a solution, and moreover it is a *maximal* solution in the sense that, for any other solution D, we have $(\forall r : D.r \Rightarrow C.r)$.

The following presentation is necessarily brief; the interested reader is referred to [4] for further detail. We first define the projection functions i and o. If t is a trace of B (a sequence of events in $Int \cup Ext$), then $i.t$ is the projection of t onto its interface with C (Int), and $o.t$ is the projection onto its interface with the environment (Ext). The two functions are defined by the following axioms:

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

Definition. 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:

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

(Note that $safe.re$ does not imply $safe.r$, and that r is trivially safe if no trace of B matches r.)

Property. For any specification C:

$$(\forall q : (B||C).q \Rightarrow A.q) \equiv (\forall r : C.r \Rightarrow safe.r)$$

(recall that $(B||C).q$ means q is a trace of B||C).

Note that a quotient (with respect to safety) exists if and only if $safe.\varepsilon$, because $C.\varepsilon$ is true for any specification C.

The basis of our method is that we associate with each state of C information about the possible current state of B and the externally-observable portion of the trace that led to it. This information can be encoded in a set of (a, b) pairs, and we define a one-to-one correspondence f between states of C and such sets. Moreover, we define a mapping h from traces in Int^* to such sets of (a, b) pairs, and then ensure that, for each r and c,

$$\stackrel{r}{\mapsto} c \Rightarrow (f.c = h.r)$$

Definition.

$$(a,b) \in h.r \equiv (\exists t : i.t = r \wedge \stackrel{t}{\mapsto} b \wedge a = \psi_A.(o.t))$$

We want to construct C_0 to satisfy the following:

- (C1) $\forall r: C_0.r \Rightarrow safe.r$
- (C2) If D is a specification with interface *Int*, satisfying $(\forall r: D.r \Rightarrow safe.r)$, then $(\forall r: D.r \Rightarrow C_0.r)$.

The first requirement says that C_0 is a solution with respect to safety; the second says that it has the largest possible trace set. In order to accomplish this, we must consider each trace over *Int* as a possible trace of C_0.

We can use the sets $h.r$ for each r to check safety inductively, with the help of a predicate, *ok*, which we now define.

Definition. For a set J of (a,b) pairs such that $J = h.r$ for some trace r:

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

Intuitively, $ok.J$ says that for every pair (a,b) in J, any event in *Ext* that is enabled in b is also enabled in a state reachable from a. Note that $ok.J$ is easily checked by examination of J and the specifications A and B.

Properties. (These follow from the definitions given so far.) For any $r \in Int^*$ and $e \in Int$:

(P1) $ok.(h.\varepsilon) \Rightarrow safe.\varepsilon$

(P2) $safe.r \wedge ok.(h.re) \Rightarrow safe.re$

(P3) $safe.r \Rightarrow ok.r$

These properties suggest an inductive computation. We can begin by computing $h.\varepsilon$, and checking $ok.(h.\varepsilon)$, because it is a necessary and sufficient condition for existence of a solution with respect to safety. If $ok(h.\varepsilon)$ holds, we can create an initial state c_0 and set $f.c_0 = h.\varepsilon$. Given a way of computing $h.re$ from $h.r$, we can iterate, adding states and transitions until closure is achieved. Termination of this process is guaranteed by the finiteness of the range of h: the number of distinct sets of (a,b) pairs is finite.

We need a function φ that maps a set J of (a,b) pairs and an event e to another set of pairs, and satisfies $h.r = J \Rightarrow h.re = \varphi.(h.r,e)$. Such a function, easily computed from J, B, and A, is given by

$$\begin{aligned}(a,b) \in \varphi.(J,e) \equiv \\ (\exists a',b',t : (a',b') \in J \wedge \\ i.t = e \wedge b' \stackrel{t}{\rightarrow} b \wedge a' \stackrel{e}{\rightarrow\triangleright} a)\end{aligned}$$

```
S_C0 := ∅; new := ∅;
f.c0 := h.ε;
if ok.(f.c0) then new := {c0};
while new is not empty
    select c in new;
    for each e in Int:
        J := φ(f.c, e);
        if ok.J then
            if f^-1.J ∉ (S_C0 ∪ new)
                then create c';
                     f.c' := J;
                     add c' to new;
                else c' := f^-1.J
            add c --e--> c' to T_C0;
    move c from new to S_C0;
```

Figure 5: Algorithm – safety

where the relation $a' \stackrel{e}{\rightarrow\triangleright} a$ is similar to $\rightarrow$, but also implies that $a = \psi.qe$ if $a' = \psi.q$ for A in normal form.

We define $\Sigma_{C_0} = Int$, and $\lambda_{C_0} = \emptyset$. Thus C_0 has no internal transitions. The state space and external transition relation are computed by the algorithm in Figure 5.

Upon termination of the safety phase, we have for any $r \in Int^*$, $e \in Int$, $a \in S_A$, $b \in S_B$, and $c \in S_{C_0}$:

$$C_0.\varepsilon \equiv ok.(h.\varepsilon) \tag{1}$$

$$ok.(f.c) \tag{2}$$

$$\stackrel{r}{\mapsto} c \Rightarrow f.c = h.r \tag{3}$$

$$C_0.r \wedge ok.(h.re) \Rightarrow C_0.re \tag{4}$$

$$\begin{aligned}(a,b) \in f.c \equiv \\ (\exists q : q \in Ext^* \wedge \stackrel{q}{\mapsto} \langle b,c\rangle \wedge a = \psi_A.q)\end{aligned} \tag{5}$$

Theorem 1.

(i) $(\forall r: C_0.r \Rightarrow safe.r)$

(ii) If D satisfies $(\forall r: D.r \Rightarrow safe.r)$, then $(\forall r: D.r \Rightarrow C_0.r)$.

Proof Sketch. Both parts are proved by induction on r. For (i), the base case follows from (1). For the induction step, $C_0.re$ implies $C_0.r$ by prefix-closure of trace sets, and by the inductive hypothesis we have $safe.r$. Also, $C_0.re$ means $\stackrel{re}{\mapsto} c$ for some c, hence by (3) we have $f.c = h.re$ for some c. This implies $ok.(h.re)$, by (2). From $safe.r$ and $ok.(h.re)$, we conclude $safe.re$ by P2 above. For the base case of (ii), $D.\varepsilon$ implies $safe.\varepsilon$, which by (P3) implies $ok.(h.\varepsilon)$, which implies

```
S_C := S_C0;
repeat
    save := S_C;
    compute τ*.⟨b, c⟩ for each b, c pair;
    foreach c ∈ S_C:
        foreach (a, b) ∈ f.c:
            if ¬prog.a.⟨b, c⟩ 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 6: Algorithm – progress

$C_0.\varepsilon$ by (1). For the inductive step, D.re implies D.r, which implies $C_0.r$ by inductive hypothesis. D.re also implies $safe.re$, which implies $ok.(h.re)$. By (4), $C_0.r$ and $ok.(h.re)$ imply $C_0.re$.

Turning to progress, the next phase of the algorithm identifies states of C_0 in which it is possible that a progress violation of A can occur.

Definition. A state c of C is *bad* if and only if:

$$\exists a, b : (a, b) \in c \wedge \neg prog.a.\langle b, c\rangle$$

By the definition of satisfaction with respect to progress and (5) above, this definition implies 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, we must iteratively identify and remove bad states, and recalculate $\tau^*.\langle b, c\rangle$ for each b and c. 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 algorithm is shown in Figure 6.

Theorem 2. If D is such that B||D satisfies A, and c is marked bad at some point in the progress phase of the algorithm, then $\forall r : \stackrel{r}{\mapsto} c \Rightarrow \neg$D.$r$.

We omit the proof of this result, which depends upon the maximality part of Theorem 1, and implies that the second phase of the algorithm maintains that property. As a consequence, if the algorithm terminates with an empty state set, we can conclude that no quotient exists.

5 An Example

We now show the application of the algorithm to a simple example. The protocols in the examples are the

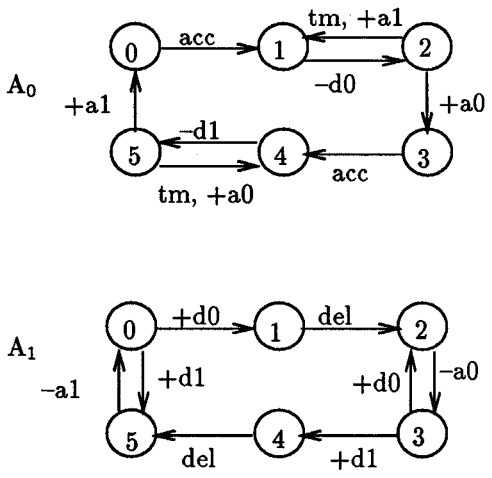


Figure 7: Alternating Bit Protocol

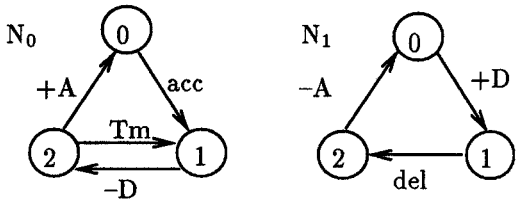


Figure 8: Non-sequenced Protocol

venerable alternating-bit (AB) protocol [1], and a non-sequenced (NS) protocol, which does not use sequence numbers. Each protocol provides delivery of data messages from a Sender entity to a Receiver, in spite of possible message losses by the transmission medium. The specifications of the AB protocol are depicted in Figure 7. The "acc" and "del" events (abbreviating "accept" a message from the Sender and "deliver" a message to the Receiver, respectively) constitute the interface with the user, while other events are interactions with the channel. Events beginning with "–" represent passing a message interface into the channel, while "+" indicates removal of a message from the channel. The Sender (A_0) attaches a one-bit sequence number to each data message transmitted. Data messages are denoted by "d0" and "d1." The Receiver (A_1) uses this sequence bit to synchronize with the Sender and determine whether a received data message has already been delivered; each data message is delivered exactly once (for each time it is accepted for transmission). An acknowledgement message, denoted "a0" or "a1," is returned for each data message; it contains the sequence number of the last-delivered data message.

The NS protocol, shown in Figure 8, has no sequence numbers; the Receiver (N_1) delivers every received data message. A data message is represented by "D." The

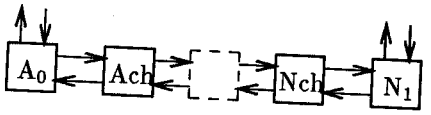
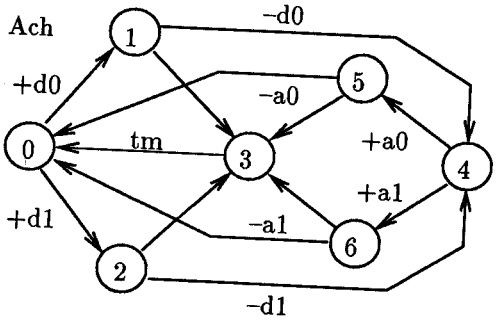


Figure 9: Problem configuration



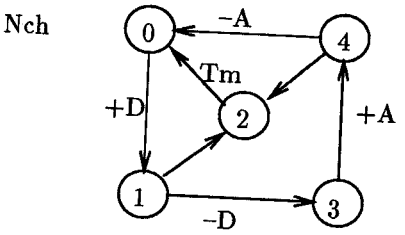


Figure 10: Channel specifications

Sender (N_0) repeatedly transmits the data until an acknowledgement "A" is received. While both protocols guarantee that a message will be delivered at least once, the NS protocol may deliver the same message multiple times, while AB delivers each exactly once. The service of NS is thus somewhat "weaker" than that of AB.

The (duplex) channels between Sender and Receiver are modeled as separate components of the system, as shown in Figure 9. The channels are assumed to be lossy; both protocols use timeouts to detect and recover from lost messages. The channel specifications appear in Figure 10. Unlabeled transitions represent the possible loss of a message; after such a loss, a timeout event occurs at the Sender. Note that these timeouts never occur prematurely. Again, "–" indicates passage of a message into the channel, and "+" indicates a message leaving the channel, so these events match their counterparts in the protocol specifications.

The specification of the desired service is shown in Figure 11. The output of the safety phase of the quotient algorithm is shown in Figure 12. This is a correct converter with respect to safety: All possible sequences

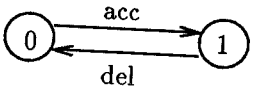


Figure 11: Desired Service Specification

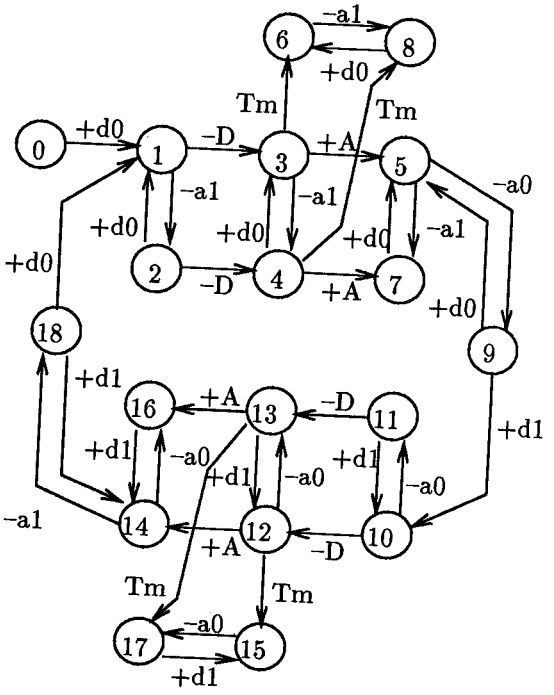


Figure 12: Output of Quotient Algorithm

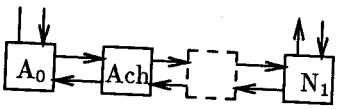


Figure 13: Revised configuration

of "acc" and "del" for the system $A_0||Ach||C||Nch||N_1$, are prefixes of the sequence "accept, deliver, accept, deliver, ..." However, some of these traces cannot be extended, i.e., this converter cannot satisfy the progress requirement of the service specification.

The problem is that if a message is lost between C and N_1, C cannot tell if it was data or acknowledgement. If it was data, progress will not occur unless C retransmits. However, if the acknowledgement was lost, retransmission will result in two consecutive "del" events, violating the safety requirement. In this example, if a loss ever occurs in Nch, the user sees no further progress, while C and A_0 exchange useless data and acknowledgement messages forever (states 6 and 8, and 15 and 17 in the figure). It is this kind of conflict between safety and progress that prevents existence of a converter.

It is possible to weaken the service specification to allow delivery of duplicates, and thereby obtain a converter. Another approach, illustrated here, is to require that the converter be co-located with the NS Receiver, so they may exchange messages directly, as shown in Figure 13. This eliminates the possibility of losses between C and N_1. The input to the algorithm is $(A_0||Ach||N_1)$, plus the same service specification. The

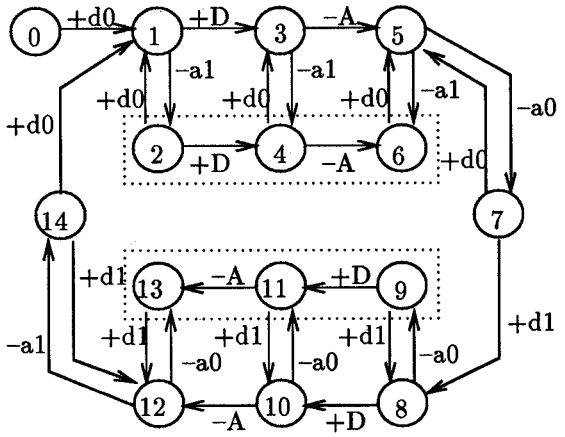


Figure 14: Quotient for second example.

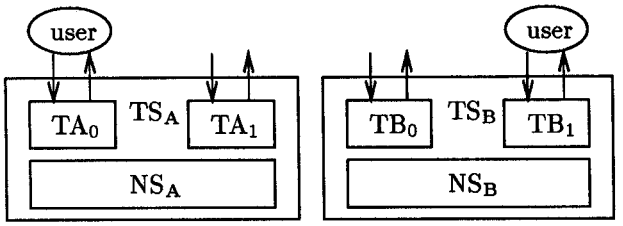


Figure 15: Heterogeneous networks

resulting converter is shown in Figure 14. Note that the "+D" and "−A" events match the same events in N_1, and denote passage of a data message from C to N_1, and acknowledgement message from N_1 to C, respectively.

The dotted boxes in Figure 14 indicate a superfluous portion of the converter; the cycles passing through these states are harmless, but do nothing for overall system progress. This is a consequence of deriving the maximal converter. Removing such "useless" portions of the converter is computationally expensive and is best done by hand.

6 Architectural Issues

In the foregoing discussion, we considered a simplified form of the problem, in order to focus on possible solutions. In practice, protocol mismatches may involve multiple layers in an architecture. In this section, we broaden our view somewhat, and consider the problem of deriving a converter for the interconnection of heterogeneous layered networks. We are still dealing with an abstraction of the problem in that many important issues are ignored, including addressing, routing and network management.

Although protocol mismatches can occur at any layer, the problems of primary interest today occur

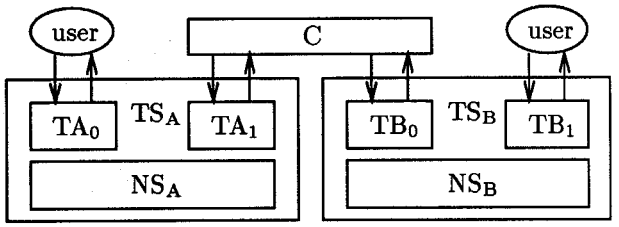


Figure 16: "Going up a level"

at the network and transport layers. Figure 15 shows schematically two "adjacent" networks, each having a different architecture. The network services are represented by a box labeled NS in each network; the transport protocol peers are TA_0 and TA_1, and TB_0 and TB_1, and the transport services are denoted by TS. Our objective is to provide a transport service conforming to specification CST (not indicated in the figure) between the user on Network A and the user on Network B. Note that these "users" may in turn be peer protocol entities.

We assume that the existing network and transport services can be physically connected to each other (the most likely situation is that both are located in the same host). By connecting TA_1 and TB_0 to each other via a simple pass-through entity as in Figure 16, we provide a "concatenated" data transfer service between the two users. However, any end-to-end synchronization capability of the existing services will not be preserved. In Figure 16, any sychronization happens 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 with end-to-end synchronization. An example is the "orderly close" function, which guarantees that all user data have been delivered to the remote end by the time the connection closes. One user might successfully close the connection, and think that all data had been delivered to the other end, 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 17. If all of the specifications are finite-state, and the service specification CST can be placed in normal form, the problem of finding C for this configuration can (in theory) be solved using the quotient algorithm. The inputs are

$$B = TA_0 || NS_A || NS_B || TB_1$$

and A = CST.

Figure 18 shows a different architectural approach, which combines 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 internetwork

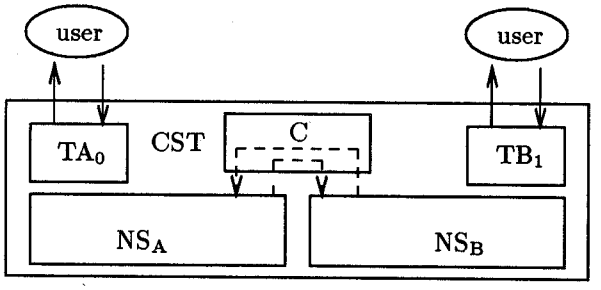


Figure 17: Transport-level conversion

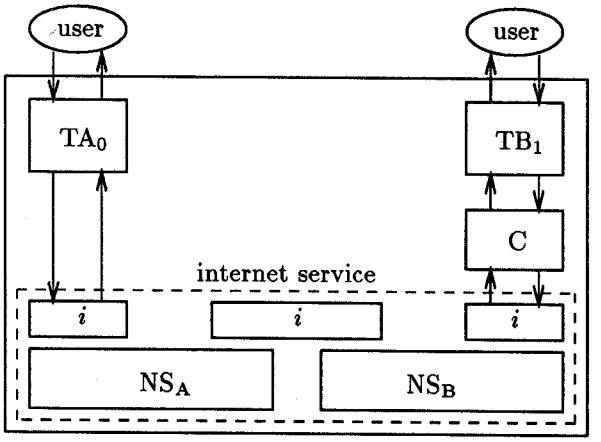


Figure 18: Asymmetric configuration

service that provides unreliable transfer. The canonical example of this is the DARPA-standard Internet Protocol [6]. In Figure 18, the internetwork service provides a data 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 TB_1 (it could also be placed at the TA_0 end). The transmission path between converter and TA_0 is unreliable, while that between converter and TB_1 is reliable. As we have already seen, this setup allows the converter to have better "knowledge" of the state of the local entity (TB_1), and may allow a more useful conversion service than would be possible in the symmetric configuration of Figure 17.

This configuration has other advantages. For one, addressing issues are confined to the network layer, at the boundary between networks. Another 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 service can also enjoy the benefits 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," see [18].)

Although the problems of interest today are mostly at the transport level, one might expect that in the future some form of end-to-end, reliable transport service will be more or less universally available. When that occurs, the conversion problems of interest will be those at higher levels. Figure 18 also depicts that problem, if the dashed box is considered to provide a reliable transport service, rather than a lower-level one, and TA_0 and TB_1 are viewed as applications entities.

As a simple example, TB_1 might be a yellow pages server, and TA_0 a client on a different network that is designed to work with a slightly different service. The converter serves as a "front man" for the B server, allowing Network A clients on the remote network to access the service. At the same time, "normal" clients of TB_1 can access the server directly. Other methods of achieving interoperation of clients and servers using different protocols are discussed in [16].

7 Conclusions

The work presented here shows that "quotient" problems can be solved algorithmically for a certain class of finite state specifications. The problem of finding a protocol converter is an example of such problems. By solving for the converter in a top-down fashion, we can detect when no converter exists for a particular protocol mismatch and desired service. This represents an extension of earlier work on deriving protocol converters. It is also an extension of earlier work on other forms of the quotient problem, in that we handle progress properties.

The quotient problem is computationally hard (in fact, it is PSPACE-hard, even if only safety is considered). The algorithm presented here has exponential time and space complexity in the worst case. However, the progress phase of the algorithm does not add significantly to its complexity (it is polynomial in the size of the quotient produced by the safety phase), which is somewhat surprising. A converter produced by this algorithm has a maximal set of traces, and thus may contain spurious sequences of events that do not affect correctness, but decrease efficiency. These are expensive to remove, and such optimization is best done by hand.

Areas of interest for future work include characterization of classes of instances for which the complexity of the converter is small relative to that of the input; these are the cases where interposing a converter will be practical. Characterization of general problems for which no converter exists would also be useful.

References

[1] K. A. Bartlett, R. A. Scantlebury, and P. T.

Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5), May 1969.

[2] E. Brinksma and G. Karjoth. A specification of the OSI transport service in LOTOS. In *Protocol Specification, Verification, and Testing IV*, 1984.

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

[4] K. L. Calvert and Simon S. Lam. Finding quotients of specifications. Technical report, University of Texas at Austin, Department of Computer Sciences, in preparation.

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

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

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

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

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

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

[11] ISO. Transport service definition. ISO 8072-CCITT X.214, June 1986.

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

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

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

[15] 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.

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

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

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

[19] Joachim Parrow. Submodule construction as equation solving in CCS. In *LNCS 287.* Springer-Verlag, 1988

[20] 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.