A Discrete Event Systems Approach for Protocol Conversion *

Ratnesh Kumar
Sudhir Nelvagal
Department of Electrical Engineering
University of Kentucky
Lexington, KY 40506-0046

Steven I. Marcus
Department of Electrical Engineering and
Institute of Systems Research
University of Maryland at College Park
College Park, MD 20742

Abstract

A protocol mismatch occurs when heterogeneous networks try to communicate with each other. Such mismatches are inevitable due to the proliferation of a multitude of networking architectures, hardware, and software on one hand, and the need for global connectivity on the other hand. In order to circumvent this problem the solution of protocol conversion has been proposed. In this paper we present a systematic approach to protocol conversion using the theory of supervisory control of discrete event systems, which was partially first addressed by Inan. We study the problem of designing a converter for a given mismatched pair of protocols, using their specifications, and the specifications for the channel and the user services. We introduce the notion of *converter languages* and use it to obtain a necessary and sufficient condition for the existence of protocol converter and present an effective algorithm for computing it whenever it exists.

Keywords: Protocol conversion, discrete event systems, supervisory control, controllability, observability, normality, safety, progress, synchronous composition.

*This research was supported in part by the Center for Robotics and Manufacturing Systems, University of Kentucky, and in part by the National Science Foundation under the Grants NSF-ECS-9409712 and NSF-EEC-9402384.

1 Introduction

There is a growing need for global communication over networks of computers. However, the heterogeneity of existing networks does not allow direct and consistent communication and results in mismatch of protocols. Such mismatches are inevitable due to the proliferation of differing hardware, software and networking standards, and the desired urgency towards global connectivity. While a possible solution to such differing standards or protocol mismatches would be to standardize networking protocols, such a move may not be practical, and in any case will take years to be agreed upon worldwide, resulting in connectivity problems for the present. As a result, the alternative method of protocol conversion has been proposed. These alternative measures of developing protocol converters will be the only form of solution available until such time when everybody adheres to a global standard.

Green [21] argues that protocol conversion is a necessity that cannot be ignored citing reasons that it is a little too late to standardize architectures. An established base of DECnet, ARPAnet, IBM SNA, TCP/IP and X.25 users will find it difficult to switch to an open and global standard simply because of the sheer effort involved, as well as rendering obsolete many existing hardware and software solutions by numerous vendors. Different protocols and architectures also tend to serve different communities of users, with the need to maintain contact with the external world still being of great importance.

In Figure 1 protocol P consists of the sending end protocol P_0 and the receiving end protocol P_1. Similarly, the protocol Q is composed of Q_0 and Q_1. A protocol mismatch occurs when the sending end protocol P_0 of P tries to communicate with the receiving end protocol Q_1 of Q, and similarly also when Q_0 tries to communicate with P_1.

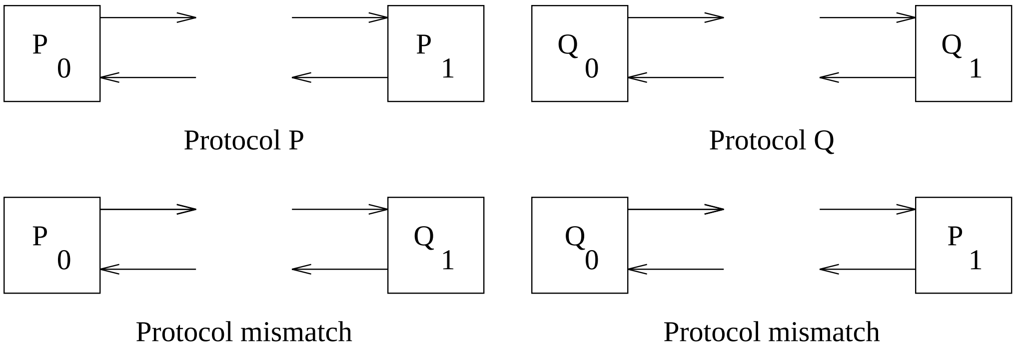

Figure 1: Protocol configuration

A practical solution to such a protocol mismatch is to interpose a translator or a converter between the two protocols so that it traps all messages being sent from one system to the other and translates the messages of the sender system in such a manner that the receiver system can understand them without loss in consistency. This is depicted in Figure 2, where P_0 denotes one of the protocols, Q_1 the other of the mismatched protocols, and C denotes the converter. The resulting protocol conversion system has to adhere to the user service specification, which defines the properties of the protocol system as needed by the users.

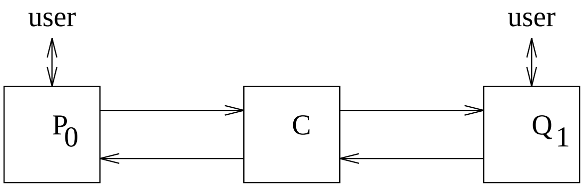

Figure 2: Interposing a protocol converter

Tanenbaum [28] provides a classification for such converters depending on the layer of the protocol stack at which they are used. *Repeaters* are used at the physical layer; *bridges* are used at the link layer; *routers* or *gateways* are used at the network layer; and finally *protocol converters* are used at the transport layer.

The events that occur at the user interface are called the *external events*; the remaining events are called the internal events. Let G denote the composition of P_0 and Q_1, and K denote the user service specification. Then G is a system that evolves over external as well as internal events, whereas K is a formal language defined only over the external events. The composition of G with a converter C, denoted $G||C$, *implements* the service specification K if certain safety and progress properties hold. Safety captures the notion that nothing illegal should happen, i.e., the event traces of $G||C$ should correspond to those allowed by K. Progress captures the notion that the composition should not block the occurrence of an external event whenever it is feasible in the specification K, i.e., whenever the specification does not "block" it.

The role of the converter C is to restrict the behavior of G so that $G||C$ implements K. However, it must do so under its *limited control and observation capabilities*. In other words, certain events are *controllable*—their enablement/disablement can be controlled by C (the remaining are *uncontrollable*); similarly, certain events are *observable*—their occurrence can be sensed by C (the remaining are *unobservable*). Thus the design of protocol converters can be studied in the framework of supervisory control of discrete event systems pioneered by Ramadge and Wonham [23] and subsequently extended by other researchers (refer to the survey articles [24, 29], and the book [12]). This is the motivation for the work presented here.

The problem of protocol conversion has been studied by some researchers and the paper by Calvert and Lam [2] provides a nice survey. One of the first approaches to protocol conversion is the *bottom-up* approach taken by Okumura [20] and by Lam [17]. The bottom-up approaches are heuristic in nature and may not be able to determine a converter even when one exists. The *top-down* approach of Calvert and Lam [2] is algorithmic in nature. However, the converter they design is a state machine that evolves over the set of internal events, which implies that the set of internal events is treated as the set of controllable as well as the set of observable events for the converter. This is unrealistic, since the converter can observe only those events that occur at its interface, i.e., its "input" and "output" events, and it can control only its output events. So in general these event sets are different. This is further illustrated in the example below. The top-down approach or the algorithmic approach is also known as the *quotient* approach since C can be viewed as the "quotient of G and K".

The protocol conversion problem was first treated by Inan in the supervisory control framework as an important application example in [8], in further depth in [9], and the theoretical foundation on which this application is based was given in [10]. However, these work of Inan *only* address the safety constraint of the protocol conversion problem which requires that the projected language of the supervised system should equal the given specification language. The additional progress constraint, which requires that the supervised system never block an external event that is not blocked by the specification itself, was *also* considered as part of the protocol conversion problem in the work of Calvert-Lam [2]. We treat both the safety and progress constraints in the work presented here. The existence of converter requires controllability and observability conditions together with extra safety and progress conditions implying that the work presented here also requires the generalization of the conventional supervisory control. Rudie-Wonham [25] have also applied the supervisory control techniques for deriving a missing portion (either sender or receiver portion) of the protocol assuming some knowledge of it. This problem is quite similar to that of protocol conversion but again considers safety specification only.

In a recent work, Kumar-Fabian [11] show that the progress constraint may be expressed using a type of nonblocking property with respect to a family of marked languages—one for each external event. This nonblocking property differs from the conventional one used in supervisory control [3] in two ways: first, nonblocking with respect to a family of markings (as opposed to a single marking) is needed, and second, each type of marking depends, in a very specific manner, on the entire closed-loop behavior (as opposed to being just those traces of the closed-loop behavior that are marked in the open-loop behavior).

In this paper we derive a converter using the formal techniques from supervisory control of discrete event systems. The supervisory control framework provides necessary and sufficient conditions for the existence of supervisors (in the present setting converters) for a given plant (in the present setting composition of the two mismatched protocols) so that the closed-loop system (in the present setting composition of converter and the mismatched protocols) meets a desired specification (in the present setting the user service specification). However, since the user service specification is a *partial specification*, i.e., it is defined only on the subset consisting of the external events (as opposed to the entire event set which is customary in the supervisory control framework), the supervisory control theory results cannot be applied directly, and appropriate extensions have been obtained in the paper.

We introduce the notion of *converter languages* and show that the existence of a converter is equivalent to the existence of a converter language contained in the language of G. The set of converter languages is closed with respect to union. So a supremal converter language exists, and we provide an effective algorithm for computing it. The test for the existence of a converter reduces to the test of non-emptiness of the supremal converter language. Moreover, the generator for the supremal converter language serves as a choice for the converter. No converter exists if the supremal converter language is empty, in which case the given service specification may be minimally altered so that a converter exists. This problem has recently been addressed by Takai-Takae-Kodama [27] and Kumar-Fabian [11].

We illustrate our work by the example of two incompatible protocols namely the Alternating Bit protocol and the Nonsequenced protocol [28]. These two are obviously incompatible because the alternating bit protocol labels all data packets with sequence numbers 0 or 1, while the nonsequenced protocol does not work with labeled data packets. Abridged versions of the work presented here first appeared in [14, 15].

2 Motivating Example

For the example we assume that the converter is *collocated* with the receiving end. So, as shown in Figure 3, the sending end P_0 is the composition of the sender protocol P_s and the sender's channel P_c, whereas the receiving end Q_1 consists of only the receiver protocol Q_r. In

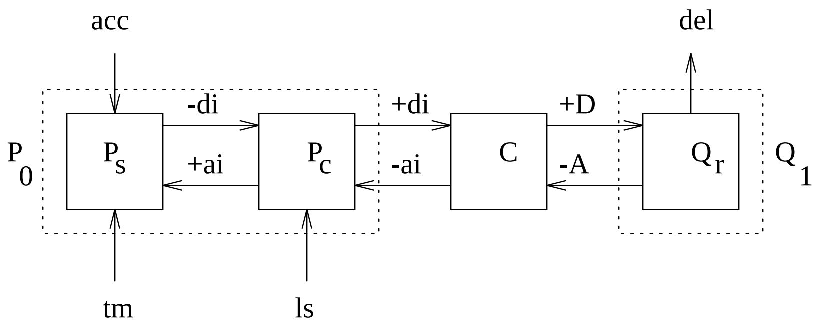

Figure 3: A typical protocol conversion system

general Q_1 is also a composition of Q_r and the receiver's channel. The mismatched protocol components for the example are the Alternating Bit protocol sender (P_s), Alternating Bit channel (P_c), and the Nonsequenced protocol receiver (Q_r). The state machines for each of these along with that for the service specification are presented in this section. Thus in this example $G = P_0 \| Q_1 = P_s \| P_c \| Q_r$.

The events occurring at various interfaces for the present example are indicated in Figure 3. The external event set consists of the accept event (`acc`) and the deliver event (`del`). Lower (respectively, upper) case letters are used for internal events occurring at the sender (respectively, receiver) end. An event label having a negative (respectively, positive) sign as its prefix represents a transmit (respectively, receipt) event of a protocol. However, the sign convention for the channel is the opposite since a receipt event of the channel is a transmit event of the adjoining protocol and vice-versa. So the receipt events of the channel are prefixed with negative signs, whereas the transmit events of the channel are prefixed with positive sign. Since the converter is interposed between the channel and the receiver protocol, this fixes the events that occur at the converter interface. Thus, for instance, `-di` represents a transmit event of a data packet with label `i` (where `i = 0,1`) at the sender protocol (and a receipt event of the same data packet at the channel), whereas `-A` represents a transmit event of an acknowledgment at the receiver protocol (and a receipt event of the same acknowledgment at the converter). Other events include the timeout event (`tm`) and the channel loss event (`ls`).

Thus in the above example the event set Σ consists of the following:

$$\Sigma = \{\texttt{acc}, \texttt{del}, +\texttt{di}, -\texttt{di}, +\texttt{ai}, -\texttt{ai}, +\texttt{D}, -\texttt{A}, \texttt{tm}, \texttt{ls}\}.$$

A subset of Σ consisting of the events

$$\{+\texttt{di}, -\texttt{ai}, +\texttt{D}, -\texttt{A}\}$$

occur at the converter interface. These constitute the set of observable events, whereas all the remaining events are unobservable. Part of the observable events are the output events for the converter, and their occurrence can be controlled. So the converter output events constitute the set of controllable events, which are:

$$\{-\texttt{ai}, +\texttt{D}\}.$$

All the other events are uncontrollable to the converter. Note that the set of controllable events for the converter is contained in the set of its observable events. We exploit this property when we design the converter. Also, note that the set of controllable events and the set of observable events are both different from the set of internal events. This distinction is not noted in the work of Calvert and Lam [2].

The Alternating Bit sender depicted in Figure 4 has six states. The initial state is the state 0, where it is in a position to accept data from the user. The data is transmitted with label 0. The next data is accepted from the user after an acknowledgment with the correct label 0 is received. Otherwise, when either the wrong acknowledgment with label 1 is received or the sender times out (due to loss of data or acknowledgment), the data is retransmitted with label 0. This procedure is repeated after each accept event, except that the label used alternates between 0 and 1.

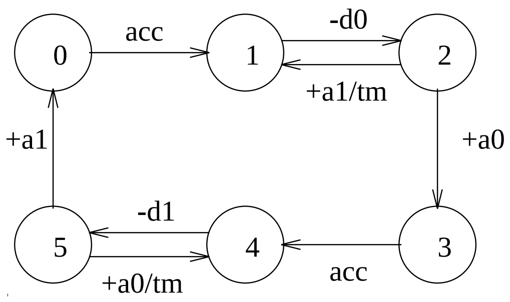

Figure 4: Alternating Bit sender

The Alternating Bit channel shown in Figure 5 has six states. The channel initially receives a data packet (events `-di`), which it may either lose (event `ls`) sending it back to its initial state, or may successfully transmit (events `+di`) sending it to the state where

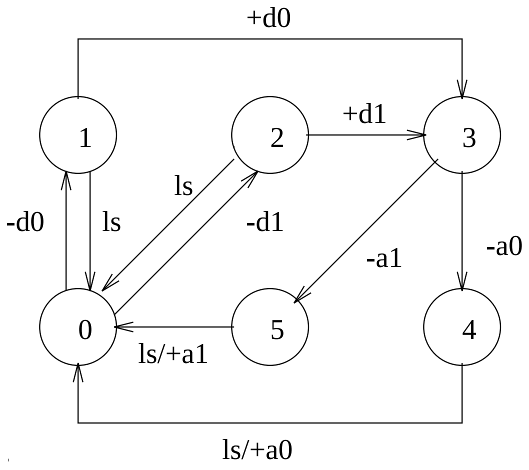

Figure 5: Alternating Bit channel

it receives an acknowledgment packet (events `-ai`). Acknowledgments may either get lost (event `ls`) or it may get successfully transmitted (events `+ai`) sending the channel back to its initial state in either case.

The Nonsequenced receiver is shown in Figure 6. This is a very simple state machine, which on receiving a data delivers it to the user and sends an acknowledgment to the sender. Since no labels are present data packets labeled by the Alternating Bit sender cannot be interpreted consistently by the Nonsequenced receiver resulting in a protocol mismatch.

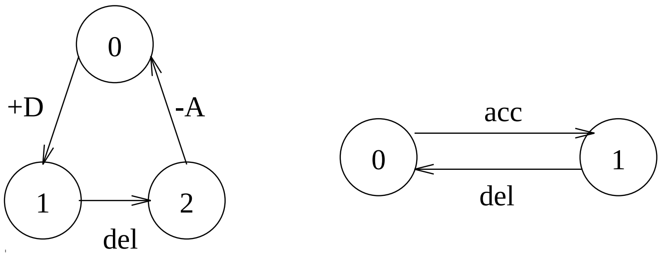

Figure 6: Nonsequenced receiver and Service specification

Finally, the protocol system should provide the service of loss free transmission over the lossy channel which is accomplished by requiring that the accept and deliver events should alternate. This service specification is depicted in Figure 6. Weaker service specifications of the type "the order of accepted and delivered message sequences be identical" can also be considered. However, more complex protocols will be needed to offer such a service.

3 Notation and Preliminaries

We use Σ to denote the universe of events. Σ^* denotes the set of all finite length event sequences, called *traces*, including the zero length trace, denoted ϵ. A subset of Σ^*, i.e., a collection of traces, is called a *language*. For a language H, the notation $pr(H)$, called the *prefix closure* of H, is the set of all prefixes of traces in H. H is said to be *prefix closed* if $H = pr(H)$. Given a trace $s \in \Sigma^*$ and a subset of events $\hat{\Sigma} \subseteq \Sigma$, the *projection* of s on $\hat{\Sigma}$, denoted $s\uparrow\hat{\Sigma}$, is the trace obtained by erasing the events not belonging to $\hat{\Sigma}$ from s.

State machines [7] are used for representing untimed behavior of discrete event systems (such as protocol and channel systems) as well as for representing qualitative or logical specifications (such as user service specifications). Formally, a state machine P is a quadruple $P := (X, \Sigma, \alpha, x_0)$, where X denotes the set of states, Σ denotes the finite set of events, $\alpha : X \times (\Sigma \cup \{\epsilon\}) \to 2^X$ is the partial transition function, and $x_0 \in X$ is the initial state. A triple $(x, \sigma, x') \in X \times (\Sigma \cup \{\epsilon\}) \times X$ is called a *transition* of P if $x' \in \alpha(x, \sigma)$; it is said to be an *epsilon-transition* if $\sigma = \epsilon$. For $x \in X$ and $\hat{\Sigma} \subseteq \Sigma$, the notation $\Re_{\hat{\Sigma}}(P, x) \subseteq X$ denotes the set of states reachable by execution of zero or more events in $\hat{\Sigma}$ from state x in P; and the notation $\hat{\Sigma}(P, x) \subseteq \Sigma$ denotes the set of events in $\hat{\Sigma}$ that are executable at x in P. The transition function is extended from events to traces $\alpha : X \times \Sigma^* \to X$ in an obvious way. The *generated language* of P is the set of all traces that it can execute starting from its initial state, i.e.,

$$L(P) := \{s \in \Sigma^* \mid \alpha(x_0, s) \neq \emptyset\}.$$

P is called a *deterministic* state machine if the transition function is a partial function of the form: $\alpha : X \times \Sigma \to X$. The *completion* of a deterministic state machine P, denoted $\overline{P} := (X \cup \{x_d\}, \Sigma, \overline{\alpha}, x_0)$, is the state machine obtained by "completing" the transition function of P by adding a new state, a "dump" state, x_d, and adding transitions from each state x of P to the dump state on those events that are not defined at x in P. Formally,

$$\forall x \in X \cup \{x_d\}, \sigma \in \Sigma : \overline{\alpha}(x, \sigma) := \begin{cases} \alpha(x, \sigma) & \text{if } \alpha(x, \sigma) \text{ defined} \\ x_d & \text{otherwise} \end{cases}$$

Note that $L(\overline{P}) = \Sigma^*$.

Synchronous composition [6] of state machines is used to represent concurrent operation of component systems. Given two deterministic state machines $P := (X_P, \Sigma_P, \alpha_P, x_{0,P})$ and $Q := (X_Q, \Sigma_Q, \alpha_Q, x_{0,Q})$, composition of P and Q denoted $P \| Q := (X, \Sigma, \alpha, x_0)$, is defined as: $X := X_P \times X_Q$, $\Sigma := \Sigma_P \cup \Sigma_Q$, $x_0 := (x_{0,P}, x_{0,Q})$, and for each $x = (x_P, x_Q) \in X$ and $\sigma \in \Sigma$:

$$\alpha(x, \sigma) := \begin{cases} (\alpha_P(x_P, \sigma), \alpha_Q(x_Q, \sigma)) & \text{if } \alpha_P(x_P, \sigma), \alpha_Q(x_Q, \sigma) \text{ defined}, \sigma \in \Sigma_P \cap \Sigma_Q \\ (\alpha_P(x_P, \sigma), x_Q) & \text{if } \alpha_P(x_P, \sigma) \text{ defined}, \sigma \in \Sigma_P - \Sigma_Q \\ (x_P, \alpha_Q(x_Q, \sigma)) & \text{if } \alpha_Q(x_Q, \sigma) \text{ defined}, \sigma \in \Sigma_Q - \Sigma_P \\ \text{undefined} & \text{otherwise} \end{cases}$$

Thus when P and Q are composed, their common events occur synchronously, while the other events occur asynchronously. The generated language of the composition is given by:

$$L(P \| Q) = \{s \in \Sigma^* \mid s\uparrow\Sigma_P \in L(P), s\uparrow\Sigma_Q \in L(Q)\}.$$

Note that when $\Sigma_P = \Sigma_Q = \Sigma$, then $L(P \| Q) = L(P) \cap L(Q)$ since all events must occur synchronously. Also note that although the state set for $P \| Q$ is $X_P \times X_Q$, many of the states remain unreachable in $P\|Q$. We adopt the convention that by writing $P\|Q$ we mean its *reachable* or *trim* component [7].

In supervisory control of discrete events systems, synchronous composition of an uncontrolled plant, modeled as a state machine G, and a supervisor, modeled as a state machine S having an identical event set as the plant, is used as a control mechanism. A certain sublanguage $H \subseteq L(G)$ represents the desired behavior of the plant, and the control objective is to design a supervisor S such that the controlled plant behavior $L(G\|S)$ equals H. The supervisor to be designed has limited control and observation capabilities in the sense that (i) it cannot prevent the occurrence of certain *uncontrollable* events, and (ii) it can only observe the occurrence of certain *observable* events. Letting $\Sigma_u \subseteq \Sigma$ denote the set of uncontrollable events and $\Sigma_o \subseteq \Sigma$ denote the set of observable events, the following result from supervisory control theory states a necessary and sufficient condition for the existence of such a supervisor:

Theorem 1 [23, 18] Given a plant G, a desired behavior $H \subseteq L(G)$, the set of uncontrollable events Σ_u, and the set of observable events Σ_o, there exists a supervisor S (compatible with control and observation capabilities) such that $L(G\|S) = H$ if and only if

Prefix closure and Non-emptiness: $H = pr(H) \neq \emptyset$.

Controllability: $pr(H)\Sigma_u \cap L(G) \subseteq pr(H)$.

Observability: $\forall s, t \in pr(H), \sigma \in \Sigma : s{\uparrow}\Sigma_o = t{\uparrow}\Sigma_o, s\sigma \in pr(H), t\sigma \in L(G) \Rightarrow t\sigma \in pr(H)$.

The controllability condition requires that extension of any prefix of H by an uncontrollable event that is feasible in the plant should also be a prefix of H. This is because the occurrence of uncontrollable events cannot be prevented. A pair of traces is called *indistinguishable* if each trace in the pair has identical projection on the set of observable events. The observability condition requires that extensions of a pair of indistinguishable prefixes of H by a common feasible event should either both or neither be prefixes of H. This is because identical control action must be taken following an indistinguishable pair of traces.

Note that H is controllable (respectively, observable) if and only if $pr(H)$ is controllable (respectively, observable). Tests for controllability and observability conditions are known when G has finitely many states and H is a regular language so that it admits a finite state machine representation. In fact if G has m states and H has a state machine representation with n states, then controllability can be tested in $O(mn)$ time, whereas the observability can be tested in $O(mn^2)$ time [30, 16]. In case the desired behavior H fails to satisfy any of the required conditions, a *maximally permissive* supervisor is designed that achieves a maximal sublanguage of H satisfying the required conditions. It is known that controllability is preserved under union so that a unique maximal controllable sublanguage, called *supremal controllable sublanguage* of a given language exists [22]; however, observability is not preserved under union, so maximal observable sublanguages are not unique [18]. Hence sometimes *normal* sublanguages instead of observable sublanguages are considered [18]:

Normality: $\forall s, t \in \Sigma^* : s{\uparrow}\Sigma_o = t{\uparrow}\Sigma_o, s \in pr(H), t \in L(G) \Rightarrow t \in pr(H)$.

Normality requires that traces of G that are indistinguishable from a prefix of H must themselves be prefixes of H. Normality is preserved under union so that the *supremal normal sublanguage* of a given language exists [18, 1, 13] and it can be tested in $O(mn^2)$ time [12, p. 103]. Moreover, normality implies observability, and the converese holds in the presence of controllability when the controllable events are also observable:

Theorem 2 [9, Proposition 4.2] Given G and $H \subseteq L(G)$, if $\Sigma - \Sigma_u \subseteq \Sigma_o$, then H is controllable and nornaml if and only if it is controllable and observable.

Similar results appeared in [12, Theorem 4.3], and in the timed setting in [19, Proposition 4].

4 Existence and Computation of Converter

In this paper we are interested in solving a slightly different supervisory control problem, where the objective is to obtain a supervisor, which we refer to as a *converter* in this context, so that the closed-loop system implements a given service specification defined on the subset of external events.

Definition 1 Given $P := (X, \Sigma, \alpha, x_0)$, a set of external events $\Sigma_e \subseteq \Sigma$ and a service specification $K \subseteq \Sigma_e^*$, P *implements* K if the following hold:

Safety: $L(P){\uparrow}\Sigma_e = pr(K)$.

Progress: $\forall s \in L(P), \sigma \in \Sigma_e : (s{\uparrow}\Sigma_e)\sigma \in pr(K) \Rightarrow \exists t \in (\Sigma - \Sigma_e)^*$ s.t. $st\sigma \in L(P)$.

Safety requires that each generated trace of P should correspond to a prefix of the specification, i.e., no "illegal" traces should occur in P. Since K is a partial specification (defined only on the external event set), there may exist more than one trace of P that correspond to the same prefix of K. Progress requires that if an external event is possible after such a prefix of K, then it should also be possible "eventually", i.e., after occurrence of zero or more internal events following each corresponding trace of P. Note that safety only guarantees that such an external event is eventually possible following at least one (and not all) of the corresponding traces of P.

Remark 1 The definition of "implements" given above is equivalent to that given in [2] but is stated differently for simplicity. For example the definition of safety given in [2] uses containment instead of equality. However, containment can be replaced by equality since the reverse containment follows from progress. Similarly, the definition of progress given in [2] uses a "state characterization" instead of a "language characterization". This is because in [2] P is represented as a nondeterministic state machine over only external events by replacing each transition on an internal event by an epsilon-transition. So a language based characterization of progress is not possible and a state based characterization is used. In our case P is a state machine over both external and internal events.

In the following theorem we provide a necessary and sufficient condition for the existence of a converter for a given pair of mismatched protocols and a given service specification. We first introduce the notion of converter languages. As described above the notation G is used to denote the composition of the mismatched protocols, and K is used to denote the service specification.

Definition 2 Given a pair of mismatched protocols G, a service specification $K \subseteq \Sigma_e^*$, a set of uncontrollable events Σ_u, and a set of observable events Σ_o, a language $H \subseteq L(G)$ is called a *converter language* if the following hold:

Controllability: $pr(H)\Sigma_u \cap L(G) \subseteq pr(H)$.

Observability: $\forall s, t \in pr(H), \sigma \in \Sigma : s\uparrow\Sigma_o = t\uparrow\Sigma_o, s\sigma \in pr(H), t\sigma \in L(G) \Rightarrow t\sigma \in pr(H)$.

Safety: $pr(H)\uparrow\Sigma_e = pr(K)$.

Progress: $\forall s \in pr(H), \sigma \in \Sigma_e : (s\uparrow\Sigma_e)\sigma \in pr(K) \Rightarrow \exists t \in (\Sigma - \Sigma_e)^*$ s.t. $st\sigma \in pr(H)$.

Note that $H \subseteq L(G)$ is a converter language if and only if its prefix closure is also a converter language. Using the result of Theorem 1 we next show that a necessary and sufficient condition for the existence of a converter is the existence of a nonempty converter language.

Theorem 3 Given a pair of mismatched protocols G, a service specification $K \subseteq \Sigma_e^*$, a set of uncontrollable events Σ_u, and a set of observable events Σ_o, there exists a converter C (compatible with control and observation capabilities) such that $G\|C$ implements K if and only if there exists a nonempty converter language.

Proof: We first prove the necessity. Suppose there exists a converter C such that $G\|C$ implements K. We claim that $H := L(G\|C)$ is the the required converter language. Since C is control and observation compatible, from the necessity part of Theorem 1 it follows that H is nonempty, prefix closed, controllable and observable. Furthermore since $G\|C$ implements K, it follows from Definition 1 that $H = L(G\|C)$ also satisfies the safety and progress properties of Definition 2. Thus H is a nonempty converter language.

In order to see the sufficiency, suppose there exists a nonempty converter language $H \subseteq L(G)$. Then $pr(H)$ is nonempty, controllable and observable. So from the sufficiency of Theorem 1 it follows that there exists C which is control and observation compatible such that $L(G\|C) = pr(H)$. Furthermore, since H is a converter language, it follows from Definition 2 that it satisfies the safety and progress properties, which implies $G\|C$ implements K as desired. ■

Since the set of controllable events for a converter is contained in the set of its observable events, i.e., $\Sigma - \Sigma_u \subseteq \Sigma_o$, it follows from Theorems 2 and 3 that a language is a converter language if and only if it satisfies the properties of controllability, normality, safety, and progress. This fact can be used to obtain the following corollary which states that the set of converter languages is closed under union so that a supremal one exists. Define the following set of converter sublanguages of $L(G)$ which implement the specification K:

$$Conv(G,K) := \{H \subseteq L(G) \mid H \text{ is a converter language}\}.$$

Corollary 1 Given G, sets $\Sigma_e, \Sigma_u, \Sigma_o$, and $K \subseteq \Sigma_e^*$, the supremal converter language $sup\, Conv(G,K)$ exists.

Proof: First note that since $\emptyset$ is a converter sublanguage, $Conv(G,K) \neq \emptyset$. Let Λ be an indexing set such that for each $\lambda \in \Lambda$, $H_\lambda \subseteq L(G)$ is a converter language. We claim that $\bigcup_{\lambda\in\Lambda} H_\lambda$ is also a converter language, i.e., it satisfies the conditions of controllability, normality, safety, and progress. The first two properties follow from the fact that controllability and normality are preserved under union. Safety follows from the following series of equalities:

$$[pr(\bigcup_\lambda H_\lambda)]\uparrow\Sigma_e = [\bigcup_\lambda pr(H_\lambda)]\uparrow\Sigma_e = \bigcup_\lambda [pr(H_\lambda)\uparrow\Sigma_e] = pr(K),$$

where we have used the fact that prefix closure and projection operations commute with arbitrary union, and each H_λ satisfies safety. Finally, to see progress, pick $s \in pr(\bigcup_\lambda H_\lambda) = \bigcup_\lambda pr(H_\lambda)$ and $\sigma \in \Sigma$ such that $(s\uparrow\Sigma_e)\sigma \in pr(K)$. Then there exists $\bar\lambda \in \Lambda$ such that $s \in pr(H_{\bar\lambda})$. Also, since $H_{\bar\lambda}$ is a converter language, it satisfies progress. So there exists $t \in (\Sigma - \Sigma_e)^*$ such that $st\sigma \in pr(H_{\bar\lambda}) \subseteq \bigcup_\lambda pr(H_\lambda) = pr(\bigcup_\lambda H_\lambda)$. ■

The following theorem provides a concrete condition for the existence of a converter and forms a basis for the test developed in this paper. It also specifies a choice for a prototype converter.

Theorem 4 Let $G, K, \Sigma_e, \Sigma_u, \Sigma_o$ be as in Theorem 3. Then there exists a converter C such that $G\|C$ implements K if and only if $sup\, Conv(G,K)$ is nonempty, in which case C can be chosen to be a generator of $sup\, Conv(G,K)$.

Proof: In order to see the necessity suppose there exists a converter. Then from the necessity part of Theorem 3 there exists a nonempty converter language $H \in Conv(G,K)$, which implies $sup\, Conv(G,K)$ is nonempty. Sufficiency follows from the sufficiency part of Theorem 3 since $sup\, Conv(G,K)$ is a converter language and it is given to be nonempty. Finally, let C be any generator of $sup\, Conv(G,K)$, i.e., $L(C) = sup\, Conv(G,K)$, then

$$L(G\|C) = L(G) \cap L(C) = L(G) \cap sup\, Conv(G,K) = sup\, Conv(G,K),$$

where the last equality follows from the fact that $sup\, Conv(G,K) \subseteq L(G)$. Consequently, $L(G\|C)$ satisfies the safety and progress properties of Definition 2, which implies $G\|C$ implements K. ■

From Theorem 4 the task of checking the existence of a converter as well as that of designing one when it exists reduces to the task of computing the supremal converter language and verifying its non-emptiness. We next present an algorithm for computing $sup\, Conv(G,K)$ assuming that G has finitely many states, say m, and K is a regular language so that it admits a finite state machine representation say $S := (Y, \Sigma_e, \beta, y_o)$ with say n states. For the motivating example, the state machine representation for K consists of only two states as shown in Figure 6.

As with the computation of the supremal controllable sublanguage given in [13], the algorithm for the computation of $sup\,Conv(G, K)$ has two steps: In the first step it constructs a sufficiently "refined" version of the state machine G; [1] and in the next step, it removes certain "bad" states from this refined state machine.

Initially certain states which correspond to traces that violate safety are marked "bad", i.e., these are the states that are reachable by execution of traces whose projection on external event set are not prefixes of K. If there are no such bad states, then $sup\,Conv(G, K)$ equals $L(G)$. Otherwise, a converter must be designed to restrict the behavior of G so that only those "good" states remain reachable which correspond to traces that also satisfy progress, controllability, and normality conditions.

Progress requires that the set of external events that can be executed following the execution of zero or more internal events from a certain good state should contain the set of external events executable at the corresponding point in K. If a good state fails to satisfy the progress, it is marked bad. Controllability requires that no bad state should be reachable from a good state on an uncontrollable event, since execution of an uncontrollable event cannot be prevented and the system can uncontrollably reach a bad state from a good state. So in order to compute $sup\,Conv(G, K)$ if there exists an uncontrollable transition from a good state to a bad one, then that good state is marked bad. Finally, normality requires that the states corresponding to a set of indistinguishable traces should either be all good or all bad. So if a good state and a bad state can be reached by the execution of pair of indistinguishable traces, then that good state is marked bad. The algorithm terminates when there are no more additional bad states to be marked. Then $sup\,Conv(G, K)$ consists of traces corresponding to the remaining good states, and it is nonempty if and only if the set of good states is nonempty.

It is clear that the state machine representation of G needs to be sufficiently refined so that the states corresponding to the traces violating either of the conditions can be unambiguously identified. First in order to deal with safety, progress and controllability conditions we refine the machine G by composing it with the state machine $\overline{S}$ obtained by completing the transition function of S. For the motivating example, the state machine $\overline{S}$ is shown in Figure 7, in which the dump state is explicitly depicted. Note that $L(\overline{S}) = \Sigma_e^*$ as expected, and given a trace $s \in \Sigma_e^*$, $s \in pr(K) = L(S)$ if and only if its execution does not result in the dump state in $\overline{S}$. Let $G_1 := \overline{S} \| G$, then since $\Sigma_e \subseteq \Sigma$, we have

$$L(G_1) = \{s \in L(G) \mid s\uparrow\Sigma_e \in L(\overline{S})\} = L(G),$$

where the last equality follows from the fact that $L(\overline{S}) = \Sigma_e^*$. For notational simplicity let $G_1 := (Z, \Sigma, \gamma, z_0)$. $Z := (Y \cup \{y_d\}) \times X$, where y_d is the dump state of $\overline{S}$, denotes the state

[1] Given two state machines $G_i := (X_i, \Sigma, \alpha_i, x_{0,i})$, $(i = 1, 2)$, G_1 is said to be a refined version of G_2 if $L(G_1) = L(G_2)$ and there exists a function $h : X_1 \to X_2$ such that $h(\alpha_1(x, \sigma)) = \alpha_2(h(x), \sigma)$ for each $x \in X_1$ and $\sigma \in \Sigma$.

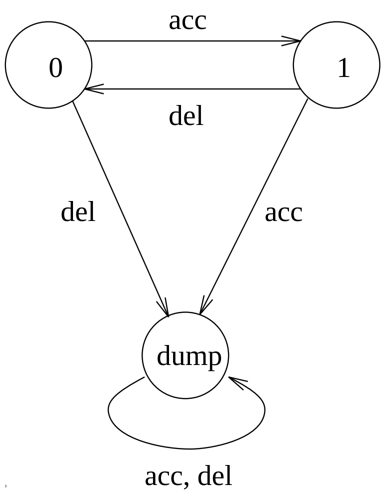

Figure 7: Completion of service specification

set of G_1, $\gamma : Z \times \Sigma \to Z$ is the transition function, and $z_0 = (x_0, y_0)$ is the initial state. Note that given a trace $s \in L(G_1) = L(G)$, its execution leads to a state $z = (y_d, x)$ in G_1 if and only if $s{\uparrow}\Sigma_e \notin pr(K)$.

Next in order to deal with the normality condition we further refine G_1. First we obtain a nondeterministic state machine G_2 that generates all traces that are indistinguishable from the traces of G_1. Since a trace remains indistinguishable when unobservable events are either inserted or erased, the following construction yields the desired G_2:

Algorithm 1 Given $G_1 := \overline{S}||G$, add transitions in G_1 to obtain G_2 as follows:

1. For each $z \in Z$ and $\sigma \in \Sigma - \Sigma_o$ add a self-loop transition (z, σ, z).

2. For each transition (z, σ, z') of G_1 such that $z \neq z'$ and $\sigma \in \Sigma - \Sigma_o$ add an epsilon-transition (z, ϵ, z').

Step 1 (respectively, 2) in the algorithm has the effect of inserting (respectively, erasing) unobservable events.

Remark 2 Note that if z is reachable by execution of a trace s in G_1, then z is also reachable by execution of all traces that are indistinguishable from s in G_2. So if $s' \in L(G_1)$ is a trace indistinguishable from s and if z' is reachable by execution of s' in G_1, then z' is also reachable by execution of s in G_2 (since s is indistinguishable from s'). In fact the set of states reachable by execution of s in G_2 is the set of states that are reachable in G_1 by execution of those traces in G_1 that are indistinguishable from s.

Next using the power-set construction [7] we obtain a deterministic state machine G_3 with the same language as $L(G_2)$. Finally we construct the machine $G_4 := G_1||G_3$. Since $L(G_3) = L(G_2) \supseteq L(G_1) = L(G)$, $L(G_4) = L(G_1) \cap L(G_3) = L(G)$. We show below that G_4 is a sufficiently refined version of G. We first outline the construction of G_4 in the following algorithm:

Algorithm 2 Given $G := (X, \Sigma, \alpha, x_0)$ and a deterministic generator $S := (Y, \Sigma_e, \beta, y_0)$ of $pr(K)$, obtain G_4 as follows:

1. Obtain $\overline{S}$ by adding a dump state and completing the transition function of S. (Then $L(\overline{S}) = \Sigma_e^*$.)

2. Obtain $G_1 := \overline{S} \| G$. (Then $L(G_1) = L(G)$, and the state set of G_1 is denoted Z.)

3. Obtain the nondeterministic state machine G_2 by adding transitions in G_1 as described in Algorithm 1. (Then $L(G_2) \supseteq L(G_1)$, and the state set of G_2 is Z.)

4. Obtain G_3 by "determinizing" G_1 using the power set construction. (Then $L(G_3) = L(G_2)$, and the state set of G_3 is 2^Z.)

5. Obtain G_4 as $G_1 \| G_3$. (Then $L(G_4) = L(G)$, and the state set of G_4 is $Z \times 2^Z$.)

Clearly, G_4 is a refined version of G. For notational simplicity, let $G_4 := (R, \Sigma, \delta, r_0)$, where $R = Z \times 2^Z$ is the state set for G_4. Note that each state r in G_4 is of the form $r = (z, \hat{Z})$, where $z \in Z$ and $\hat{Z} \subseteq Z$. $\delta : R \times \Sigma \to R$ denotes the transition function of G_4. The initial state of G_4 is $r_0 = (z_0, \{z_0\})$, where $z_0 := (y_0, x_0)$. In the following lemma we list some of the properties of G_4 which demonstrate that it is a sufficiently refined version of G. We first define the concept of a matching pair of states.

Definition 3 A pair of states $r_1 = (z_1, \hat{Z}_1), r_2 = (z_2, \hat{Z}_2) \in R$ of G_4 are said to be a *matching pair* if $\hat{Z}_1 = \hat{Z}_2$.

Lemma 1 The following hold for G_4 which is the state machine constructed in Algorithm 2:

1. $L(G_4) = L(G)$.

2. Consider $s \in L(G_4)$; then $s{\uparrow}\Sigma_e \in pr(K)$ if and only if $r = ((y, x), \hat{Z}) := \delta(r_0, s)$ is such that $y \neq y_d$.

3. If $r = (z, \hat{Z}) \in R$ is a state of G_4, then $z \in \hat{Z}$.

4. Consider a matching pair of states $r_1, r_2 \in R$. Then for each $s_1 \in L(G)$ such that $\delta(r_0, s_1) = r_1$, there exists $s_2 \in L(G)$ such that $s_2{\uparrow}\Sigma_o = s_1{\uparrow}\Sigma_o$ and $\delta(r_0, s_2) = r_2$.

Proof: The first part follows from the construction and is proved above. In order to see the second part, consider $s \in L(G_4)$ and let $r = ((y, x), \hat{Z}) := \delta(r_0, s)$. Then execution of $s{\uparrow}\Sigma_e$ in $\overline{S}$ results in the state y of $\overline{S}$. Since $L(S) = pr(K)$, the assertion follows from the fact that $s{\uparrow}\Sigma_e \in L(S) = pr(K)$ if and only if $y \neq y_d$.

In order to see the third part, let $s \in L(G_4)$ be such that $\delta(r_0, s) = r = (z, \hat{Z})$. Then execution of s results in the state z in G_1 and state $\hat{Z}$ in G_3. Since G_3 is obtained by "determinizing" the state machine G_2, this implies that execution of s results in each state $\hat{z} \in \hat{Z}$ in G_2. (Recall that G_2 is nondeterministic.) Since G_2 is obtained by adding transitions in G_1, z is one such state. Hence $z \in \hat{Z}$.

Finally in order to see the fourth part, consider a matching pair of states $r_i = (z_i, \hat{Z})$, $i = 1, 2$. From the third part we have that $z_1, z_2 \in \hat{Z}$. Consider $s_1 \in L(G_4)$ such that $\delta(r_0, s_1) = r_1$. Then execution of s_1 results in state z_1 in G_1, and each state $\hat{z} \in \hat{Z}$ in G_2 (including the states z_1 and z_2). Then from Remark 2 concerning G_2, there exists a trace $s_2 \in L(G)$ indistinguishable from s_1, i.e., $s_2 \uparrow \Sigma_o = s_1 \uparrow \Sigma_o$, such that its execution results in the state z_2 in G_1. (If no such trace exists, then z_2 cannot be reached by execution of s_1 in G_2.) Finally, since s_2 is indistinguishable from s_1, its execution in G_2 also results in the set of states $\hat{Z}$. So the execution of s_2 results in the state $r_2 = (z_2, \hat{Z})$ in G_4 as desired. ■

We are now ready to present the algorithm that iteratively marks bad states in G_4, and upon termination yields a generator for $sup\,Conv(G, K)$. The notation $R_b^k \subseteq R$ is used to denote the set of bad states at the kth iteration.

Algorithm 3 Consider $G_4 := (R, \Sigma, \delta, r_0)$ obtained in Algorithm 2.

1. Initialization step:

$$R_b^0 := \left\{ r = ((y, x), \hat{Z}) \in R \mid y = y_d \right\}; \quad k := 0.$$

2. Iteration step:

$$\begin{aligned} R_b^{k+1} := \quad & R_b^k \\ \cup \quad & \left\{ r \in R - R_b^k \mid \exists \sigma \in \Sigma_u \text{ s.t. } \delta(r, \sigma) \in R_b^k \right\} \\ \cup \quad & \left\{ r = (z, \hat{Z}) \in R - R_b^k \mid \exists r' = (z', \hat{Z}) \in R_b^k \right\} \\ \cup \quad & \left\{ r = ((y, x), \hat{Z}) \in R - R_b^k \mid \Sigma_e(S, y) \not\subseteq \cup_{r' \in [\Re_{\Sigma - \Sigma_e}(G_4, r)] \cap [R - R_b^k]} \Sigma_e(G_4, r') \right\} \end{aligned}$$

3. Termination step:
   If $R_b^{k+1} = R_b^k$, then stop; else $k := k + 1$, and go to step 2.

The algorithm initially sets the iteration counter $k = 0$ and marks a state $r = ((y, x), \hat{Z}) \in R$ to be a bad state if its first coordinate is the dump state, i.e., if $y = y_d$. This is because if $s \in L(G_4) = L(G)$ is any trace such that $\delta(r_0, s) = r$, then from the second part of Lemma 1, $s \uparrow \Sigma_e \notin pr(K)$, i.e., s violates the safety condition. The set of bad states at the kth iteration is denoted by R_b^k.

In the kth iteration step, a good state $r = ((y, x), \hat{Z}) \in R - R_b^k$ is marked bad if any of the following hold: (i) If there exists an uncontrollable event from r to a bad state. This is because if $s \in L(G_4) = L(G)$ is any trace such that $\delta(r_0, s) = r$, then s violates the controllability condition. (ii) If there exists a matching bad state $r' \in R_b^k$. This is because if $s \in L(G_4) = L(G)$ is any trace such that $\delta(r_0, s) = r$, then from the fourth part of Lemma 1, there exists a trace s' indistinguishable from s such that $\delta(r_0, s') = r'$, i.e., s violates the normality condition. (iii) If the set of external events that are executable at state y in S is not contained in the set of external events that are executable from those good states that are reached by the execution of zero or more internal events from r in G_4. This is because if $s \in L(G_4) = L(G)$ is any trace such that $\delta(r_0, s) = r$, then s violates the progress condition.

The algorithm terminates if no additional bad states are marked in the kth iteration; else, the iteration counter is incremented by one and the iteration step is repeated. Note that since G_4 has finitely many states, the algorithm is guaranteed to terminate in a finite number of iterations. Also, since the algorithm marks a state to be a bad state if and only if all traces that lead to it violate either safety, progress, controllability, or normality, the state machine obtained by the removal of the terminal set of bad states (and all transitions leading towards or away from them) from G_4 generates the language $sup\,Conv(G, K)$. Hence we obtain the following result stating the correctness of the algorithm.

Theorem 5 Given a finite state machine G and a regular language $K \subseteq \Sigma_e^*$, Algorithm 3 terminates in a finite number of steps, and the state machine obtained by removal of the states $R_b^* \subseteq R$ from G_4 generates $sup\,Conv(G, K)$, where R_b^* denotes the set of bad states at the termination of the algorithm.

Remark 3 Let m be the number of states in G and n be the number of states in the state machine representation of K. Then the computational complexity of Algorithm 3 is $O(m^2n^22^{2mn})$ since the number of states in G_4 is $O(mn2^{mn})$, which implies that there are $O(mn2^{mn})$ number of iterations with equally many computations in each iteration. Also, unless P = NP, no algorithm of polynomial complexity exists. This follows from the fact that in the special case when $\Sigma_e = \Sigma$, the converter design problem reduces to the standard supervisory control problem under partial observation with the desired behavior constraint specified as a range of languages, which is known to be an NP-complete problem [30].

5 Implementation Issues and Example Converters

Since the computation of $sup\,Conv(G, K)$ is intractable, a computationally tractable heuristic approach to converter design is desirable. We propose two heuristics and utilize each to design a converter for the motivating example.

Since the computational intractability arises due to the presence of partial observation, one possibility is to first compute the supremal sublanguage of $L(G)$ satisfying safety, progress, and controllability, and next verify whether it is also normal, each of which can be done in polynomial time. In case the language is also normal, then it equals the desired supremal converter language and we are successful in computing it in polynomial time.

We next outline a polynomial time computation of the supremal sublanguage of $L(G)$ satisfying safety, progress, and controllability. Clearly, this language equals $sup\,Conv(G, K)$ when all events are observable (so that normality trivially holds). Note that when all events are observable, G_2 equals G_1, i.e., no transitions are added when Algorithm 1 is invoked, so G_4 also equals G_1. Hence the following modification of Algorithm 3 computes the desired supremal sublanguage of $L(G)$ satisfying safety, progress, and controllability.

Algorithm 4 Given $G := (X, \Sigma, \alpha, x_0)$ and a deterministic generator $S := (Y, \Sigma_e, \beta, y_0)$ of $pr(K)$, let $G_1 = (Z, \Sigma, \gamma, z_0) := \overline{S} \| G$.

1. Initialization step:

$$Z_b^0 := \{z = (y, x) \in Z \mid y = y_d\}; \quad k := 0.$$

2. Iteration step:

$$\begin{aligned} Z_b^{k+1} := \quad & Z_b^k \\ \cup \quad & \left\{z \in Z - Z_b^k \mid \exists \sigma \in \Sigma_u \text{ s.t. } \gamma(z, \sigma) \in Z_b^k\right\} \\ \cup \quad & \left\{z = (y, x) \in Z - Z_b^k \mid \Sigma_e(S, y) \not\subseteq \cup_{z' \in [\Re_{\Sigma - \Sigma_e}(G_1, z)] \cap [Z - Z_b^k]} \Sigma_e(G_1, z')\right\} \end{aligned}$$

3. Termination step:
   If $Z_b^{k+1} = Z_b^k$, then stop; else $k := k + 1$, and go to step 2.

Using the fact that G_1 has $O(mn)$ states, the computational complexity of Algorithm 4 can be determined to be $O(m^2n^2)$. The algorithm computes the supremal sublanguage of $L(G)$ that satisfies safety, progress, and controllability (but may violate normality). The test whether this language is also normal (with respect to the given set of observable events) can be performed in $O(m(mn)^2) = O(m^3n^2)$ time [12, p. 103]. In case the test for normality fails, then as in the work of Cho-Marcus on iterative computation of supremal controllable and normal sublanguage [4], we can iterate between the supremal normal sublanguage computation and the computation of the supremal language that meets safety, progress, and controllability until a fixed point is reached, which however will result in an exponential computational complexity. This is outlined in the following algorithm:

Algorithm 5 Given G and $K \subseteq \Sigma_e^*$, compute the $sup\,Conv(G, K)$ as follows:

1. $H_0 := L(G)$; $k := 0$.

2. Compute the supremal sublanguage $\hat{H}_k$ of H_k satisfying safety, progress, and controllability using Algorithm 4. If $\hat{H}_k$ is normal, then $sup\,Conv(G, K) = \hat{H}_k$, and stop; else go to step 3.

3. Compute the supremal normal sublanguage H_{k+1} of $\hat{H}_k$. If H_{k+1} satisfies safety and progress (Algorithm 6 given below provides a test for safety and progress), then $sup\,Conv(G, K) = H_{k+1}$, and stop; else replace G by the generator of H_{k+1}, set $k := k + 1$, and go to step 2.

Remark 4 Note that in the above algorithm we do not need to check the controllability of H_{k+1} in step 3 since it is known that the supremal normal computation preserves controllability [5, Proposition 3.9]. Also, in step 3 if H_{k+1} does not satisfy safety and progress, then we need to compute its supremal sublanguage satisfying safety and progress. We can use Algorithm 4 for doing this (although in this case the controllability trivially holds). However, we need to replace G by the generator of H_{k+1} since Algorithm 4 only computes the supremal *sublanguage of* $L(G)$ satisfying safety, progress, and controllability.

We have written a C-program for Algorithm 5 (and the associated Algorithm 4) that utilizes a finite state machine library originally developed for supervisory control [26]. Using the program for Algorithm 4 we first computed the supremal sublanguage of $L(G)$ satisfying controllability, safety, and progress for the example. The state machine G, which is the composition of Alternating Bit sender, Alternating Bit channel, and Nonsequenced receiver contains 66 states. The composition of G with $\overline{S}$ (which has three states) contains 198 states. The initial iteration of the safety check disqualified 66 states. The next iteration which performs controllability and progress checks disqualified additional 34 states. No additional states were disqualified in the next iteration of controllability and progress checks. This resulted in a test converter with $198-(66+34)=98$ states. The test converter also passed the normality test; thus qualifying it as a valid converter.

A second possible heuristic is to guess a test converter C and verify its correctness by checking whether $L(G\|C)$ is a converter language, which as we show below can be done in polynomial time. First note that it is not difficult to guess a converter; a simple possibility is to design a system that emulates the missing portions of the mismatched protocols, i.e., the receiver protocol P_r of P and the sending protocol Q_s of Q. For the motivating example the "guess converter" we consider below emulates the functions of the Alternating Bit receiver and the Nonsequenced sender.

In order to test whether $L(G\|C)$ is a converter language, we must check whether it satisfies controllability, normality, safety, and progress. Polynomial time tests for controllability and normality can be found in [12, pp. 75, 103]. Here we present a polynomial time test for safety and progress. For notational simplicity let $G_5 := G\|C$ and let its state set be Φ. We first refine G_5 by composing it with $\overline{S}$. For notational simplicity, let $G_6 := \overline{S}\|G_5$ and let its state set be $\Theta := (Y \cup \{y_d\}) \times \Phi$. Note that since $\Sigma_e \subseteq \Sigma$,

$$L(G_6) = \{s \in L(G_5) \mid s\uparrow\Sigma_e \in L(\overline{S})\} = L(G_5),$$

where the last equality follows from the fact that $L(\overline{S}) = \Sigma_e^*$. Also, given a trace $s \in L(G_6) = L(G_5)$, its execution leads to a state $\theta = (y_d, \phi)$ if and only if $s\uparrow\Sigma_e \notin pr(K)$. So for $L(G_5)$ to satisfy safety, no state θ of G_6 should be of the form (y_d, ϕ). Also for $L(G_5)$ to satisfy progress, the set of external events that can be executed following the execution of zero or more internal events from each state $\theta = (y, \phi)$ in G_6 should contain the set of external events executable at the corresponding state y in S. So we have the following algorithm for checking safety and progress:

Algorithm 6 Consider the composition of mismatched protocols G, a test converter C, and a deterministic generator S of $pr(K)$.

1. Construct $G_5 := G\|C$, and denote its state set by Φ.

2. Construct $G_6 := \overline{S}\|G_5$, and denote its state set by Θ.

3. Then $L(G\|C)$ satisfies safety and progress if and only if

$$\forall \theta = (y, \phi) \in \Theta : [y \neq y_d] \wedge \left[\Sigma_e(S, y) \subseteq \cup_{\theta' \in \Re_{\Sigma-\Sigma_e}(G_6, \theta)} \Sigma_e(G_6, \theta')\right].$$

If the number of states in the converter is p, then the computational complexity of the algorithm is $O(m^2n^2p^2)$ since G_6 has $O(mnp)$ states and equally many reachability computations need to be performed.

A test converter for the motivating example is shown in Figure 8. The converter adopts

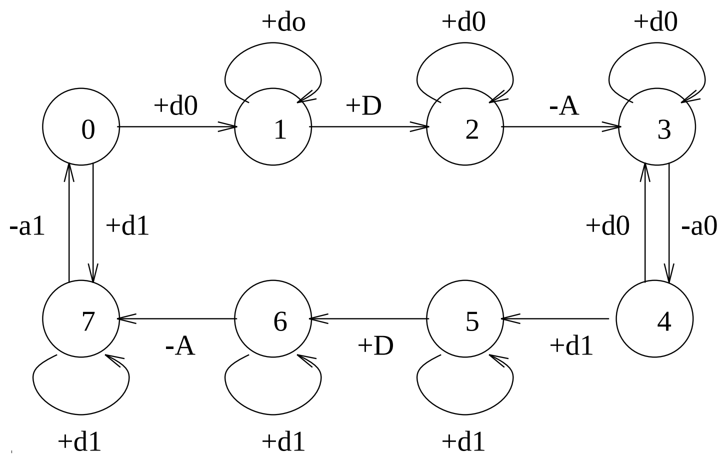

Figure 8: A test converter

the following simple conversion strategy. Initially when one or more data packets with label 0 arrive (event `+d0`), it removes the label and forwards a single data packet to the receiver (event `+D`). No action is taken at this point if more copies of the same data packet arrive (due to a sender timeout). When the receiver transmits an acknowledgment (event `-A`), the converter attaches the label 0 and puts it onto the sender's channel (event `-a0`). The procedure is repeated when data packets with a different label arrive (except for the difference in the label used). However, if another data packet with the same label arrives (due to a sender timeout or a loss of acknowledgment in the channel), then the same acknowledgment is retransmitted.

We have verified that $L(G\|C)$ (where $G = P_s\|P_c\|Q_r$) satisfies safety, progress, controllability, and normality, i.e., it is a converter language. Since $L(G\|C)$ is obviously nonempty, it follows from Theorem 3 that C is indeed a valid converter.

6 Conclusion

In this paper we have studied the problem of designing protocol converters that need to be interposed between a pair of mismatched protocols. Our approach is systematic and is based on the recent theory of supervisory control of discrete event systems. The work presented here provides a new framework for protocol converter designers on one hand, and serves as an application for the supervisory control theorists on the other hand. The basic concepts of controllability, observability, normality, and computation of supremal languages from supervisory control, and safety, and progress from protocol design play important role in the protocol conversion problem. The converter that we derive is *maximally permissive* in the sense that any other converter will further restrict the behavior of the entire system. However, the maximally permissive converter may not have a minimal number of states. Design of such minimal converters is an interesting problem for future research.

References

[1] R. D. Brandt, V. K. Garg, R. Kumar, F. Lin, S. I. Marcus, and W. M. Wonham. Formulas for calculating supremal controllable and normal sublanguages. *Systems and Control Letters*, 15(8):111–117, 1990.

[2] K. Calvert and S. S. Lam. Formal methods of protocol conversion. *IEEE Journal on Selected Areas in Communication*, 8(1):127–142, January 1990.

[3] E. Chen and S. Lafortune. Dealing with blocking in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 36(6):724–735, 1991.

[4] H. Cho and S. I. Marcus. On supremal languages of class of sublanguages that arise in supervisor synthesis problems with partial observations. *Mathematics of Control Signals and Systems*, 2:47–69, 1989.

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

[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1985.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[8] K. Inan. Supervisory control and formal methods for distributed systems. In *Discrete Event Systems: Modeling and Control (Proceedings of WODES 1992)*, pages 29–41. Birkhauser-Verlag-Basel, 1993.

[9] K. Inan. Supervisory control: Theory and application to the gateway synthesis problem. In *Belgian-French-Netherlands Summer School on Discrete Event Systems*, page 25 pages. Spa, Belgium, 1993.

[10] K. Inan. Nondeterministic supervision under partial observations. In Guy Cohen and Jean-Pierre Quadrat, editors, *Lecture Notes in Control and Information Sciences 199*, pages 39–48. Springer-Verlag, New York, 1994.

[11] R. kumar and M. Fabian. On supervisory control of partial specification arising in protocol conversion. In *1997 IASTED Control Conference*, Cancun, Mexico, May 1997. Submitted.

[12] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems.* Kluwer Academic Publishers, Boston, MA, 1995.

[13] R. Kumar, V. K. Garg, and S. I. Marcus. On controllability and normality of discrete event dynamical systems. *Systems and Control Letters*, 17(3):157–168, 1991.

[14] R. Kumar, S. Nelvagal, and S. I. Marcus. Design of protocol converters: A discrete event systems approach. In *Proccedinds of 1996 International Workshop on Discrete Event Systems*, pages 7–12, Edinburgh, UK, August 1996.

[15] R. Kumar, S. Nelvagal, and S. I. Marcus. Protocol conversion using supervisory control techniques. In *Proceedings of 1996 IEEE CCA/ISIC/CACSD*, pages 32–37, Dearborn, MI, September 1996.

[16] R. Kumar and M. A. Shayman. Automata-theortic tests for observability and co-observability. In *Proceedings of 1995 IEEE Conference on Decision and Control*, pages 919–920, New Orleans, LA, 1995.

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

[18] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

[19] F. Lin and W. M. Wonham. Supervisory control of timed discrete event systems under partial observation. *IEEE Transactions on Automatic Control*, 40(3):558–562, 1995.

[20] Kaoru Okumura. A formal protocol conversion method. *Proceedings ACM SIGCOMM*, pages 30–37, 1986.

[21] JR P. E. Green. Protocol conversion. *IEEE Transactions on Communications*, COM-34(3):257–268, March 1986.

[22] P. J. Ramadge and W. M. Wonham. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, 1987.

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

[24] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of IEEE: Special Issue on Discrete Event Systems*, 77:81–98, 1989.

[25] K. Rudie and W. M. Wonham. Supervisory control of communicating processes. In L. Logrippo, R. L. Robert, and H. Ural, editors, *Protocol Specification, Testing and Verification*, pages 243–257. Elsevier Science Publishers, North-Holland, 1990.

[26] Himanshu A. Sanghavi. A software library for discrete event systems and other finite state machine based applications. Master's thesis, University of Texas, Austin, 1991.

[27] S. Takai, A. Takae, and S. Kodama. The extremal languages arising in supervisory control for service specifications. In *Proceedings of 1996 IEEE Conference on Decision and Control*, Kobe, Japan, December 1996.

[28] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1990.

[29] J. G. Thistle. Logical aspects of control of discrete event systems: a survey of tools and techniques. In Guy Cohen and Jean-Pierre Quadrat, editors, *Lecture Notes in Control and Information Sciences 199*, pages 3–15. Springer-Verlag, New York, 1994.

[30] J. N. Tsitsiklis. On the control of discrete event dynamical systems. *Mathematics of Control Signals and Systems*, 2(2):95–107, 1989.