AN ILLUSTRATION OF PROTOCOL PROJECTIONS

Simon S. Lam and A. Udaya Shankar

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

The method of protocol projections to facilitate the analysis of communication protocols is illustrated with two examples. In our model, protocol entities are connected by communication channels; messages sent by the protocol entities through the channels may be lost, duplicated and/or reordered. Protocols with several distinguishable functions are considered. Image protocols are obtained by aggregating protocol entity states, messages and events using equivalence relations. As a result, image protocols are typically much simpler than the original protocol and can be analyzed more easily using available techniques. We employ two protocol examples to illustrate the construction of entity states, messages and events of some image protocols. In the first example, the protocol entities are described by a finite state machine model. The second example is a half-duplex data transfer protocol and its entities are described by a programming language model. In both cases, the image protocols constructed satisfy a well-formed property. As a result, each image protocol constructed is faithful in the sense that its logical correctness properties are the same as the logical correctness properties of the original protocol with respect to the projected function.

1. INTRODUCTION

The method of protocol projections is intended to facilitate the analysis of nontrivial protocols that are too complex to be analyzed with one of the basic approaches [1-5]. We observe that real-life protocols typically have several distinguishable functions. For example, the HDLC protocol has at least three functions: connection management, and one-way data transfers in two directions. We would like to ask questions regarding the logical correctness behavior of the protocol concerning these functions. Instead of asking such questions all at the same time, we may ask them with respect to one function of the protocol at a time. The analysis approach is to construct from the given protocol an image protocol for each of the functions that are of interest to us. These functions will be referred to as the projected functions. The entity states, messages, and events of an image protocol are obtained from those of the original protocol using certain equivalence relations. Entity states, messages and events that are equivalent with respect to a projected function are aggregated in the image protocol. As a result, image protocols are simpler than the original protocol. Each image protocol can thus be more easily analyzed...
using available means. For example, an image protocol for the function of
one-way data transfers of HDLC will be of the same complexity as Stemming’s data
transfer protocol which has been successfully analyzed [3,4].

A communication protocol system consists of protocol entities connected by
communication channels (real or virtual) over which messages are exchanged. A
communication protocol system can be modeled abstractly as a transition system
specified by the pair (G,T) where G is the set of global states and T is a
binary relation in G called the set of transitions [6]. The global state of
such a model of interacting protocol entities is specified by a joint
description of the states of the protocol entities and communication channels.
State transitions correspond to the occurrence of various events: an entity
sending or receiving messages, errors in channels, an entity receiving signals
from its user and timers, etc.

Given an initial state s_0, T determines the reachability tree (space) R. R
is a directed graph with nodes being elements of G and arcs being elements of T.
R contains all available information on logical correctness properties of the
protocol. Let R_g denote the set of reachable states in G. R_g, which may be
obtained from R, determines the safety (partial correctness) properties of the
protocol. Liveness properties, however, require knowledge of the set of paths
in R.

The protocol projection idea can be illustrated by the following example.
Consider a protocol model with the state description (x, y, z) and the set R_g of
reachable states. Suppose that we are only interested in a safety assertion
that involves only the variables x and y. To determine whether the assertion is
true, it is sufficient for us to know the image of R_g on the (x, y) plane.
Obviously, if R_g is known, its image on the (x, y) plane is readily available.
However, the complexity of R (and thus R_g) is the basic source of difficulty in
protocol analysis. Thus, we would like to take the original protocol and
construct from it an image protocol whose set of reachable states duplicates the
image of R_g on the (x, y) plane. In fact, we want to construct image protocols
that are faithful with respect to both safety and liveness properties. In
general, an image protocol is said to be faithful if its logical correctness
properties are the same as the logical correctness properties of the projected
function in the original protocol.

The method of protocol projections was first described by these authors in
[7]. Subsequently, the theory has been extended to a protocol model with
communication channels that may lose, duplicate or reorder messages [8,9]. The
main result in [8,9] is that if an image protocol is constructed to satisfy a
well-formed property there
in a sufficient condition, we can have without any knowledge
(Not again that we cannot
a basic source of our difficulty.

Given a multi-functional
image protocols that are
well-structured if it has
its functions.

The application of
follows. Suppose that
specify the correct behav
the function). Suppose
validity of assertions
assertion(a) and the one
first construct a well-for
the original protocol).

verifier for evaluation.
protocol, a different im

In Section 2, we show
3 and 4, two protocol
well-formed image protocol
entities are described.
Section 4 is a full-duplex
language described by a programmi

For a complete expon
protocol projections are
[8,9,10]. In a companion
to our basic protocol
protocols and distribut

2. THE PROTOCOL MODEL

Our basic model is a
Let P_1 and P_2 be two
duplicates messages to
channel C_2. (See Fig
reordered, duplicated, an
well-formed property then it is faithful. Although the well-formed property is a sufficient condition, we have found that it is the weakest condition that one can have without any knowledge of the reachability tree $R$ of the protocol. (Note again that we cannot assume any knowledge of $R$ since its complexity is the basic source of our difficulties.)

Given a multi-function protocol, the successful construction of well-formed image protocols that are much smaller than the given protocol depends upon the protocol’s structure. One can think of a multi-function protocol as well-structured if it gives rise to “minimal” well-formed image protocols for its functions.

The application of protocol projections to the analysis of protocols is as follows. Suppose that we are given a protocol and one or more assertions that specify the correct behavior of some protocol function (service specification of the function). Suppose also that a verifier is available for checking the validity of assertions for a given protocol. Instead of feeding the assertion(s) and the original protocol into the verifier, our objective is to first construct a well-formed image protocol (which should be much simpler than the original protocol). The image protocol and assertions are then fed into the verifier for evaluation. If we are interested in several functions of the protocol, a different image protocol is generated for each function.

In Section 2, we shall first present our basic protocol model. In Sections 3 and 4, two protocol examples are employed to illustrate the construction of well-formed image protocols. In the first example (Section 3), the protocol entities are described by a finite state machine model. The second example in Section 4 is a full-duplex data transfer protocol whose protocol entities are described by a programming language model.

For a complete exposition of our basic protocol model, the method of protocol projections and the well-formed property, the reader is referred to [8,9,10]. In a companion paper [11], time variables and time events are added to our basic protocol model for the analysis of time-dependent communication protocols and distributed systems.

2. THE PROTOCOL MODEL

Our basic model is an extension of the model of Brand and Zafiropulo [12]. Let $P_1$ and $P_2$ be two protocol entities that communicate with each other. $P_1$ sends messages to $P_2$ through channel $C_1$, and $P_2$ sends messages to $P_1$ through channel $C_2$. (See Fig. 1.) Messages transmitted through a channel may be resequenced, duplicated, and/or lost (due to corruption by noise).
The consideration of a protocol model with only two entities is strictly for notational simplicity. Our results in this paper and in [7-10] are applicable to any network of protocol entities interconnected by communication channels.

A channel from one entity to another consists of all buffers and communication media between the entities. At any time, the channel contains a (possibly empty) sequence of messages. We assume that a message can be sent into the channel without any constraint by the channel (i.e., unblocked sends). Note that most communication protocols have some measure of flow control. As a result, their buffer requirements for messages in transit between entities are bounded. Hence, the assumption of unblocked sends is equivalent to being able to satisfy those buffer requirements.

We also assume that the message flow in a channel cannot be blocked due to an entity refusing to accept messages indefinitely. The first message in a channel's sequence of messages will be deleted after some finite time duration (by the channel, the entity or some other agent).

The above two assumptions are both reasonable whether the communication channels are real or virtual.

For i = 1 and 2, let $S_i$ be the set of states of $P_i$, $O_i$ be the initial state of $P_i$ in $S_i$, and $M_i$ be the set of messages sent by $P_i$.

Let $M_1$ denote a sequence of messages representing the state of channel $C_1$. A message reception event removes the first message in the sequence. A message send event appends a new message to the end of the sequence. The set of all possible message sequences in $C_1$ is a subset of

$$N_1 = \left\{ \begin{array}{c} U_i \in M_1^k \cup \{O\} \end{array} \right\}$$

where $O$ denotes the null sequence and $M_1^k$ is the cartesian product of $M_1$ with itself $k$ times.

The state space of the global model of protocol interaction is

$$G = S_1 \times N_1 \times M_2 \times S_2.$$  

Each global state is a $4$-tuple $<s_1, m_1, m_2, s_2>$ where $s_1 \in S_1$ and $m_1 \in M_1$ for $i = 1$ and 2. The initial global state denoted by $s_0$ will be assumed to be $<O_1, O, O, s_2>$ in the rest of this paper (without any loss of generality).

The events in the protocol are either entity events or channel events. An event can occur only if certain conditions, denoted as its enabling condition, hold. When an enabled event occurs, it changes the state of one or more components of the global state.

There are three types of entity events. We describe these events for $P_1$.

1. $(s, r, m)$, where $m \in M_1$, denotes a message $m$ by $P_1$ into channel state $s$. After the event, the state of $P_1$ is appended to the end of the events in a subset of $S_1$; $s$. The set of such receive events is $S_2 \times S_2 \times M_2$.

2. $(s, r, m)$, where $m \in M_2$, denotes a message $m$ by $P_1$ from channel state $s$ and $m$ is the first message of $P_1$ in state $r$ and $m$. The set of such receive events is $S_2 \times S_2 \times M_2$.

3. $(s, r, e)$, where $e$ is an internal event, is $S_2 \times S_2 \times \{a\}$. Our model defines 3 types of duplication and reordering messages in a communication channel messages. Furthermore, the channel error axiom is assumed. If channel error occurs, then any message in any entity is repositioned.

3. PROTOCOL PROJECTION ILLUSTRATION

The first example consists in Figure 2. Protocol entity $P_1$ sends messages from $M_1 = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, events of entity $P_1$ are shown with a label $a_1$ specifies events $s_1$ for sending message $s_1$, events of entity $P_2$ are similarly shown.
An Illustration of Protocol Projections

1. \((s, r, m)\), where \(m \in M_1\), denotes an event of \(P_1\) due to the sending of a message \(m\) by \(P_1\) into channel \(C_1\). This send event is enabled when \(P_1\) is in state \(s\). After the event occurrence, \(P_1\) is in state \(r\) and \(m\) has been appended to the end of the message sequence in \(C_1\). The set of such send events is a subset of \(S_1 \times S_1 \times M_1\).

2. \((s, r, m)\), where \(m \in M_2\), denotes an event of \(P_1\) due to the reception of message \(m\) by \(P_1\) from channel \(C_2\). This receive event is enabled when \(P_1\) is in state \(s\) and \(m\) is the first message in \(C_2\). After the event occurrence, \(P_1\) is in state \(r\) and \(m\) has been deleted from the message sequence in \(C_2\). The set of such receive events is a subset of \(S_1 \times S_1 \times M_2\).

3. \((s, r, \alpha)\), where \(\alpha\) is a special symbol indicating the absence of a message, denotes an internal event of \(P_1\) that does not involve a message. This internal event is enabled when \(P_1\) is in state \(s\); after the event occurrence \(P_1\) is in state \(r\). Internal events model such events as timeouts and interactions between the entity and its local user. The set of internal events is a subset of \(S_1 \times S_1 \times \{\alpha\}\).

The three types of events for entity \(P_2\) are similarly defined, but with the send events in \(S_2 \times S_2 \times M_2\), receive events in \(S_2 \times S_2 \times M_1\), and internal events in \(S_2 \times S_2 \times \{\alpha\}\).

Our model defines 3 types of events in communication channels (loss, duplication and reordering events). These events transform the sequence of messages in a communication channel and are enabled whenever the channel has messages. Furthermore, the following axiom about the behavior of channels is assumed:

Channel error axiom: If channel \(C_i\) loses (duplicates, repositions) messages, then any message in any message sequence in \(C_i\) may be lost (duplicated, repositioned).

3. Protocol Projection Illustrated by an Example

The first example consists of two interacting finite state machines shown in Figure 1. Protocol entity \(P_1\) has state space \(S_1 = \{0, 1, 2, 3, 4, 5, 6\}\) and sends messages from \(M_1 = \{a_1, a_2, a_3\}\). Protocol entity \(P_2\) has state space \(S_2 = \{0, 1, 2, 3, 4, 5, 6\}\) and sends messages from \(M_2 = \{b_1, b_2, b_3\}\). The events of entity \(P_1\) are shown in Figure 1 (a). An arc from node \(i\) to node \(j\) with a label \(m\) specifies event \((i,j,m)\), where \(m\) is an internal event, \(a\) is \(-a_j\) for sending message \(a_j\), and \(m\) is \(+b_i\) for receiving message \(b_i\). The events of entity \(P_2\) are similarly shown in Figure 1 (b).
Note that from state 4 in \( S_2 \), the reception of \( a_2 \) can cause a transition to either state 3 or 5. This nondeterministic behavior is allowed in our model, and is useful for representing certain features in real protocols and systems.

Projections are achieved using equivalence relations. The image of a protocol quantity is obtained by aggregating all those protocol quantities that are equivalent. Images of protocol entity states, messages, and events are next defined and should be obtained in the order shown below.

**Projection of entity states**

We start by examining the entity states in \( S_1 \) and \( S_2 \) that are equivalent with respect to the projected function. For \( i = 1 \) and 2, for any entity state \( s \in S_i \), the image of \( s \) is the set of entity states in \( S_i \) that are equivalent to \( s \). We denote the image of \( s \) by \( s' \). \( s' \) is also referred to as an image entity state. Let \( S_i' \) denote the set of images of states in \( S_i \). \( S_i' \) is the image state space of \( P_i \).

Consider the example in Figure 2. Suppose that for a particular function, states 0, 1, 2, 3 and 4 are equivalent, and states 5 and 6 are equivalent. This equivalence relation corresponds to the partition \( \{(0,1,2,3,4),(5,6)\} \) of \( S_1 \) indicated by Figure 3 (a). Let image state \( 0' \) denote the partition cell \( \{(0,1,2,3,4)\} \), and image state \( 5' \) denote the partition cell \( \{(5,6)\} \). \( 0' \) is the image of states 0, 1, 2, 3 and 4 in \( S_1 \). \( 5' \) is the image of states 5 and 6 in \( S_1 \). The image state space of \( P_1 \) is \( S_1' = \{0', 5'\} \).

For the same function, suppose that in \( S_2 \) states 0, 3 and 4 are equivalent, states 1 and 5 are equivalent, and states 2 and 6 are equivalent. This equivalence relation corresponds to the partition \( \{(0,3,4),(1,5),(2,6)\} \) of \( S_2 \) indicated in Figure 3 (b). Let image states \( 0', 1' \) and \( 2' \) respectively denote the partition cells \( \{(0,3,4)\}, \{(1,5)\} \) and \( \{(2,6)\} \). Then, the image state space of \( P_2 \) is \( S_2' = \{0', 1', 2'\} \).

**Projection of messages**

The above equivalence relation is next extended to messages in \( M_1 \) and \( M_2 \). Two messages \( m \) and \( n \) in \( M_i \) are said to be equivalent if and only if they cause identical changes in both image state spaces \( S_i' \) and \( S_i' \). For any message \( m \in M_i \), the image of \( m \) is the set of messages in \( M_i \) that are equivalent to \( m \). The image of \( m \), denoted by \( m' \), is also referred to as an image message. Messages that cause only changes internal to image entity states in both \( S_i' \) and \( S_i' \) are said to have the null image. The image message sets are defined by

\[
M_i' = \{m' : m' \text{ is not null, } m \in M_i\}, \text{ for } i = 1 \text{ and } 2.
\]

Consider the example protocol \( P \). Deleting any event, then we can observe the state changes in \( S_2 \).

We will first examine the message \( a_1 \) (0',0') in \( S_1 \). Both messages \( a_2 \) and \( a_3 \) cause the transition (0',1') in \( S_2 \). Here, \( a_2 \) causes state transition (0',0') in \( S_1 \).

This equivalence relation contains \( \{a_3\} \). The image of \( a_1 \) is null, partition cells \( \{a_2\} \) and \( \{a_3\} \) represent \( \{a_2', a_3\} \).

We will now examine the message \( b_1 \) (5,0') in \( S_1 \) and message \( b_2 \) (5,0') in \( S_2 \). The state changes caused by message \( b_1 \) in \( S_1 \). Hence \( b_2 \) has a null image in \( S_2 \). The partition cells \( \{b_1, b_3\}, \{b_2\} \) represent the partition cells \( \{b_1', b_2'\} \).

**Projection of entity events**

The above equivalence relation is a use of an event \( (s,r,m) \) is given \( \alpha \) for internal events. Recall the definition \( s\alpha = r \). Consequently, change the image global state of \( \alpha \) to have the null image. Similarly, \( (s',r',m') \).

\[
T_i' = \{(s',r',m') : (s\alpha = r' \alpha = m) \}.
\]

Consider the example protocol \( P \). From Figures 3 (a) and (b), \( \{(0,1,b_1), (1,2,a_1), (2,0,b_2), (0,3,a_2), (3,5,a_2)\} \) has the image \( (0',5') \). Events \( (5,0,b_1), (5',4,b_3), (6,4,b_1) \) have the image \( (5',0',b_1') \). Hence,
Consider the example protocol. In Figures 3 (a) and 3 (b), if we relabel each state by its image and collapse all identically labelled states, without deleting any events, then we obtain Figures 4 (a) and 4 (b). From these figures we can observe the state changes in \( S_1 \) and \( S_2 \) caused by the messages in \( M_1 \) and \( M_2 \).

We will first examine the message set \( M_1 \). The state transitions caused by message \( a_1 \) are \((0',0')\) in \( S_1 \), and \((1',1')\) in \( S_2 \). Hence \( a_1 \) has a null image. Both messages \( a_2 \) and \( a_3 \) cause the state transition \((0',5')\) in \( S_1 \) and the state transition \((0',0')\) in \( S_2 \). However, \( a_2 \) is not equivalent to \( a_3 \) because \( a_2 \) causes state transition \((0',0')\) in \( S_2 \) while \( a_3 \) does not.

This equivalence relation corresponds to partitioning \( M_1 \) as \( \{a_1, a_2, a_3\} \). The image of \( a_1 \) is null. Let the image messages \( a_2^* \) and \( a_3^* \) denote the partition cells \( \{a_2\} \) and \( \{a_3\} \) respectively. The image message set \( M_1^* \) is \( \{a_2^*, a_3^*\} \).

We will now examine the message set \( M_2 \). The state changes caused by message \( b_1 \) are \((5',0')\) in \( S_1 \) and \((2',0')\) in \( S_2 \). The state changes caused by message \( b_3 \) are \((5',0')\) in \( S_1 \) and \((2',0')\) in \( S_2 \). Hence \( b_1 \) is equivalent to \( b_3 \). The state changes caused by message \( b_2 \) are \((0',0')\), \((5',5')\) in \( S_1 \), and \((0',0')\) in \( S_2 \). Hence \( b_2 \) has a null image. This equivalence relation corresponds to the partition \( \{\{b_1, b_3\}, \{b_2\}\} \) of \( M_2 \). The image of \( b_2 \) is null. Let image message \( b_2^* \) denote the partition cell \( \{b_1, b_3\} \). The image message set \( M_2^* \) is \( \{b_2^*\} \).

**Projection of entity events**

The above equivalence relations are next extended to entity events. The image of an event \((s,r,m)\) is given by \((s',r',m')\), where the image of \( a \) is still \( a \) for internal events. Recall that if the image message \( m' \) is null, then by definition \( s' = r' \). Consequently, if \( m' \) is null, the event \((s,r,m)\) does not change the image global state of the protocol model. Such an event is said to have a null image. Similarly, an internal event \((s,r,a)\) with \( s' = r' \) has a null image. The set of image events at \( P_1 \) for \( i = 1 \) and \( i = 2 \) is defined as

\[
T_i = \{(s',r',m') : (s',r',m') \text{ is not null, } (s,r,m) \in T_i\}.
\]

Consider the example protocol. Recall that messages \( a_1 \) and \( b_2 \) have null images. From Figures 3 (a) and 4 (a), the events in \( P_1 \) having null images are \((0,1,b_2), (1,2,0), (2,0, a_1), (3,4, a_1), (4,3, a_1) \), and \((3,6, b_2)\). Event \((3,5, a_2)\) has the image \((0',5', a_2')\). Event \((4,5, a_3)\) has the image \((0',5', a_3')\). Events \((5,0,b_1), (5,4,b_3), (6,4,b_1) \), and \((6,2,b_3)\) are equivalent and have the image \((5',0', b_1')\). Hence,
\[ T_1 = \{ (0^*, 5^*, a_2^*), (0^*, 3^*, a_3^*), (5^*, 0^*, b_1^*), \} \]

From Figures 3 (b) and 4 (b), the events in \( T_1 \) have null images are 
\( (0,3,a), \ (3,4,b_2), \) and \( (1,5,a_1) \). The image of \( (4,3,a_2) \) is \((0^*, 0^*, a_2^*)\). Note that the image of \( (4,3,a_2) \) is not null because \( a_2^* \) is not null. Events 
\( (0,1,a_2), \ (3,1,a_2), \) and \( (4,5,a_2) \) are equivalent and have the image \((0^*, 1^*, a_2^*)\).

Event \((4,5,a_3)\) has the image \((0^*, 1^*, a_2^*)\). Events \((1,2,a)\) and \((3,6,a)\) are equivalent and have the image \((1^*, 2^*, a)\). Events \((2,0,b_1)\) and \((6,4,b_3)\) are equivalent and have the image \((2^*, 0^*, b_1^*)\). Therefore,

\[ T_2 = \{ (0^*, 0^*, a_2^*), (0^*, 1^*, a_2^*), (0^*, 1^*, a_2^*), (1^*, 2^*, a), (2^*, 0^*, b_1^*) \} \]

**Image protocol**

Our objective is to define a new protocol that is faithful to the projected function, i.e., such that its logical correctness properties are the same as the logical correctness properties of the projected function in the original protocol. We call this protocol an image protocol. A natural candidate is a protocol between entities \( P_1^* \) and \( P_2 \), using channels \( C_1 \) and \( C_2 \), obtained as follows. For \( i = 1 \) and \( 2 \), let \( S_i^* \) be the entity state space of \( P_i^* \); let \( M_i^* \) be the set of messages sent by \( P_i^* \), and let \( T_i^* \) be the set of entity events for \( P_i^* \). Given the channel error axiom, the events of channel \( C_i \) remain the same as before.

In general, the image protocol system may not be faithful to the projected function in the original protocol system. We present next sufficient conditions for the image protocol to be faithful to the projected function \([8,9]\). The objective in the construction of an image protocol is therefore to find the most coarse partitions of entity states (in other words, the smallest image protocol) which satisfy the sufficient conditions.

**Definition.** For \( i = 1 \) and \( 2 \), for \( a \) and \( b \) in \( S_i^* \), \( b \) is internally reachable from \( a \) if \( a^* - b^* \) (they have the same image), and there is a sequence of internal events in \( T_i \) with state changes inside \( a^* \) that will take \( P_i^* \) from \( a \) to \( b \).

Note that in our current model, all send events are unblocked. As a result, send events involving messages of null images can be regarded as internal events for the above definition.

**Definition.** For \( i = 1 \) and \( 2 \), an image send or internal event \((s^*, r^*, m^*) \) is well-formed if \((s^*, r^*, m^*) \) is well-formed, for every \( a \) whose image is \( s^* \), the following condition holds: for some \( n \) whose image is \( m^* \) there is some \( b \in S_i^* \) such that \( b \) is internally reachable from \( a \), and \((b, c, n) \in T_i^* \) for some \( c \in r^* \).

**Definition.** For \( i = 1 \) and \( 2 \), an image send or internal event \((s^*, r^*, m^*) \) is well-formed if, for every \( a \) whose image is \( s^* \), the following condition holds: for some \( n \) whose image is \( m^* \) there is some \( b \in S_i^* \) such that \( b \) is internally reachable from \( a \), and \((b, c, n) \in T_i^* \) for some \( c \in r^* \).
Definition. For \( i = 1 \) and 2, an image receive event \((s',r',n') \in T_1\) for \( n' \in N'_j\) (if1) is well-formed if, for every \( a \) whose image is \( s'\), the following condition holds: for every \( n \) whose image is \( n'\) there is some \( b \in S_1 \) such that \( b \) is internally reachable from \( a \), and \((b,c,n) \in T_1\) for some \( c \in r'\).

If in either of the above definitions of well-formed events, the length of the internal path is zero (i.e., \( b = a \)), then we say that the image event is strongly well-formed.

Note from the construction of \( T_1 \) and \( T_2 \), that for every \((s',r',n') \in T_1\), there is an \((a,b,n) \in T_1\) such that \( a = s'\), \( b = r'\), and \( n = n'\).

Definition. The image entity state spaces \( S_1^2 \) and \( S_2^2 \) are said to be well-formed (strongly well-formed), if every image event \((s',r',n') \in T_1 \cup T_2\) is well-formed (strongly well-formed).

It has been shown that if the image entity state spaces \( S_1^2 \) and \( S_2^2 \) are well-formed, then the image protocol is faithful to the projected function in the original protocol [8,9].

Consider the example protocol introduced earlier. We have already obtained the image state spaces \( S_1^2 = \{0',5'\} \) and \( S_2^2 = \{0',1',2'\} \), the image message sets \( M_1^2 = \{a_2, a_3\} \) and \( M_2^2 = \{b_2\} \), and the image event sets \( T_2^2 = \{(0',5',a_2), (0',5',a_3), (5',0',b_2)\} \) and \( T_2^2 = \{(0',0',a_2), (0',1',a_2), (0',1',a_3), (1',2',0), (2',0',b_2)\} \). Using these quantities, we construct the image protocol system shown in Figures 3 (a) and 5 (b).

We will now show that the image events in \( T_1^2 \) are well-formed. First, consider event \((0',5',a_2) \in T_1^2\). Because of the paths 1-2-3-4-5-3, state 1 is internally reachable from states 1, 2, 0 and 4. From state 3, the event \((3,5,a_2)\) can be executed. Hence \((0',5',a_2)\) is well-formed.

Because of event \((3,4,a_1)\), state 4 is internally reachable from state 3, and hence from states 0, 1 and 2 also. From 4, event \((4,5,a_3) \in T_1^2\) can be executed. Thus, event \((0',5',a_3) \in T_1^2\) is well-formed.

Because of the events \((5,0,b_1), (5,4,b_3), (6,4,b_1), \) and \((6,2,b_3)\) in \( T_1^2\), image event \((5',0',b_2) \in T_2^2\) is strongly well-formed.

Next, we will show that the image events in \( T_2^2\) are well-formed. Consider image event \((0',0',a_2) \in T_2^2\). Because of events \((0,3,a)\) and \((3,4,b_2)\) in \( T_2\), state 4 is internally reachable from 0 and 3. This, and the event \((4,3,a_2) \in T_2\) makes image event \((0',0',a_2) \in T_2\) well-formed.
Image event \((0',1',a_2)\) in \(T_2\) is strongly well-formed because of events \((0,1,a_2), (3,1,a_2),\) and \((4,5,a_2)\) in \(T_2\).

Image event \((0',1',a_5)\) in \(T_2\) is well-formed because of event \((4,5,a_3)\) in \(T_2\), and because \(4\) is internally reachable from \(0\) and \(3\).

Image event \((1',2',a\_3)\) in \(T_2\) is strongly well-formed because of events \((1,2,a)\) and \((5,6,a)\) in \(T_2\).

Image event \((2',0',b_1)\) in \(T_2\) is strongly well-formed because of events \((2,0,b_1)\) and \((6,4,b_3)\) in \(T_2\).

Thus, we have shown that the image protocol in Figure 5 is well-formed.

A safety assertion

Given the initial state \((0',\_O,\_O,0')\) for this image protocol system, and assuming channels \(C_1\) and \(C_2\) are error-free, it is easy to verify that the following assertion holds for the image protocol:

\(P_1\) is in state \(5'\) \(\Rightarrow\) Exactly one of the following holds:

(a) message \(a_2'\) or \(a_5'\) is in \(C_1\), or
(b) \(P_2\) is in state \(1'\) or \(2'\), or
(c) message \(b_1'\) is in \(C_2\), or
(d) \(P_2\) is in state \(0'\) and channels \(C_1\) and \(C_2\) are empty (deadlock situation).

Note that \(P_1\) in state \(5'\) means that \(P_1\) is in any state whose image is \(5'\). Similarly, message \(b_1'\) is in \(C_2\) means that any message whose image is \(b_1'\) is in \(C_2\).

We can verify whether this assertion is valid by examining the reachability tree of the original protocol in Figure 2. However, since the image protocol has been shown to be well-formed, we know that the above assertion is valid for the original protocol by virtue of the theorems presented in [8,9].

4. A FULL-DUPLEX DATA TRANSFER PROTOCOL

We illustrate in this section the application of protocol projection to a full-duplex data transfer protocol whose entities are described by a programming language model. Since the protocol is full-duplex, it has two basic distinguishable functions: on \(P_1\).

Let \(\text{DATASET}\) denote the set of protocol blocks. \(\text{SOURCE}[i]\) for \(i=0,1,2,\ldots\), \(\text{SINK}[i]\) for \(i=0,1,2,\ldots\), to initially empty. Additionally, VR which are nonnegative integer variables. BS points to the destination the position in \(\text{SINK}\) to be used for each block has been sent but not yet received data block has an externally operated switch; all else is true.

The state of \(P_1\), at any \(C_1, C_2, D_{\text{OUT}}, \text{SOURCE}, VR, \text{ACK}_{\text{DU}}\) space of \(P_1\). Entity \(P_2\) has a queue of incoming and outgoing requests (1 or more) as it is clear whether we can represent \(P_1\) as in the form \(\text{DATASET}\) or empty, an algebraic entity. (In both entities, \(S_{\text{OUT}}\) and \(S_{\text{IN}}\) are taken into consideration.)

Each message in the protocol has a single, unique identifier usually in the form of an index (character string) values \(\text{DATASET}\) of messages. A DATA message is a 1-tuple \((d, \text{DU})\) from \(\text{DATASET}\). There are as many DATA message as there are received data block. Unlike \(\text{ACK}\) \(\text{ACK}_{\text{DU}}\) message is a 2-tuple \((d, \text{DU})\) from \(\text{DATASET}\).

The set of messages that \(M_1\) is the union of \(\text{ACK}_{\text{DU}}\) message from \(\text{DATASET}\).

The message set \(M_2\) for \(P_2\) is:

The set of entity events that correspond to the sending
Let $\text{DATASET}$ denote the set of data blocks that can be sent in this protocol. Consider protocol entity $P_1$. $P_1$ has an infinite array of data blocks, $\text{SOURCE}[i]$ for $i=0,1,2,\ldots$, destined for $P_2$, and an infinite array, $\text{SINK}[i]$ for $i=0,1,2,\ldots$, to store data blocks received from $P_2$. $\text{SINK}$ is initially empty. Additionally, the following variables are used in $P_1$: $\text{VS}$ and $\text{VR}$ which are nonnegative integers, and $\text{D\_OUT}$, $\text{ACK\_DUE}$ and $\text{BUSY}$ which are Boolean variables. $\text{VS}$ points to the data block in $\text{SOURCE}$ to be sent next. $\text{VR}$ points to the position in $\text{SINK}$ to be next filled. $\text{D\_OUT}$ is true if (and only if) a data block has been sent but not yet acknowledged. $\text{ACK\_DUE}$ is true if (and only if) a received data block has to be acknowledged. $\text{BUSY}$ can be viewed as an externally operated switch; all events of $P_1$ are inhibited if (and only if) $\text{BUSY}$ is true.

The state of $P_1$, at any time, is given by the value of the 7-tuple $<\text{VS}, \text{D\_OUT}, \text{SOURCE}, \text{VR}, \text{ACK\_DUE}, \text{SINK}, \text{BUSY}>$ of $P_1$. Let $S_1$ denote the state space of $P_1$. Entity $P_2$ has a similar set of variables. For convenience, we have omitted qualifiers (1 or 2) for these variables and we shall omit them as long as it is clear whether we are referring to $P_1$ or $P_2$. In both entities, the initial state is given by $\text{VS}$ and $\text{VR}$ equal to 0, $\text{D\_OUT}$ and $\text{ACK\_DUE}$ equal to false, $\text{SINK}$ equal to empty, and $\text{SOURCE}$ equal to some infinite array of data blocks. (In both entities, $\text{SOURCE}$ does not change its value during the protocol interaction.)

Each message in the protocol is a tuple with one or more components. The first component is used to identify the kind of message, and it can take the (character string) values DATA, ACK and DATA\_ACK, corresponding to three kinds of messages. A DATA message is a 2-tuple $(\text{DATA}, d)$, where $d$ is a data block from $\text{DATASET}$. There are as many DATA messages as data blocks in $\text{DATASET}$. An ACK message is the 1-tuple $(\text{ACK})$, signifying a positive acknowledgement for a received data block. Unlike DATA messages, there is only one ACK message. A DATA\_ACK message is a 2-tuple $(\text{DATA\_ACK}, d)$ where $d$ is a data block from $\text{DATASET}$.

The set of messages that can be sent by $P_1$ is given by

$$M_1 = \{\text{ACK}\} \cup \{\text{DATA}, d : d \in \text{DATASET}\} \cup \{\text{DATA\_ACK}, d : d \in \text{DATASET}\}.$$  

The message set $M_2$ for $P_2$ is the same as $M_1$.

The set of entity events for $P_1$ is presented in Table 1. Events 1-5 correspond to the sending of a message by $P_1$. Events 6-8 correspond to th
reception of a message by \( P_1 \). Events 4 and 5 correspond to internal events caused by an agent locally connected to \( P_1 \) (e.g., user, channel controller). The enabling condition of an event defines the entity states and channel states at which the event may take place. The action of each event causes the entity to enter a new state.

\( SDATA \) and \( RDATA \) are variables taking values from \( DATASET \). \( SDATA \) and \( RDATA \) can be thought of as temporary buffers for transmission and reception (respectively) of data blocks. In Table 1, the operation \( \text{put}(CHANNEL1, (DATA, SDATA)) \) sends a data message with the value of \( SDATA \) as its data block into \( CHANNEL1 \) (i.e., appends the data message to the end of the sequence of messages in \( CHANNEL1 \)). The operations \( \text{put}(CHANNEL1, (DATA\&ACK, SDATA)) \) sends into \( CHANNEL1 \) a \( DATA\&ACK \) message with the value of \( SDATA \) as its data block. The operation \( \text{put}(CHANNEL1, (ACK)) \) sends an \( ACK \) message into \( CHANNEL1 \).

The function \( \text{first}(CHANNEL2) \) indicates the type of the message that is at the head of \( CHANNEL2 \) and has arrived at \( P_1 \). When a \( DATA \) message is at the head of \( CHANNEL2 \), the operation \( \text{get}(CHANNEL2, (DATA, RDATA)) \) removes the message from \( CHANNEL2 \) and assigns the data block in the message to \( RDATA \). When a \( DATA\&ACK \) message is at the head of \( CHANNEL2 \), the operation \( \text{get}(CHANNEL2, (DATA\&ACK, RDATA)) \) removes the message from \( CHANNEL2 \), and assigns the data block in the message to \( RDATA \). When an \( ACK \) message is at the head of \( CHANNEL2 \), the operation \( \text{get}(CHANNEL2, (ACK)) \) removes the message from \( CHANNEL2 \).

This example protocol has two functions corresponding to data transfers in the two directions. The protocol is extremely simple but it embodies two types of dependencies that one encounters when one attempts to decompose a protocol into functional components. First, the variable \( BUSY \) is shared by both functions of the protocol. Second, the messages of type \( DATA\&ACK \) are also shared. Such dependencies present major obstacles for protocol analysis using a decomposition approach. The method of protocol projections will be used to obtain a faithful image protocol for each function.

Image protocol

Consider the function of one-way data transfer from \( P_1 \) to \( P_2 \). Observe that variables \( VR \), \( ACK\_DUE \) and \( SINK \) in \( P_1 \), and \( VS \), \( D\_OUT \) and \( SOURCE \) in \( P_2 \) are not needed for the \( P_1 \) to \( P_2 \) data transfer. Thus, the variables of the projected function (referred to as function variables) are \( VS \), \( D\_OUT \), \( SOURCE \) and \( BUSY \) in \( P_1 \), and \( VR \), \( ACK\_DUE \), \( SINK \) and \( BUSY \) in \( P_2 \).

At any time, the image is given by the value of the state space of \( P_1 \). Thus, two states \( s \) and \( r \) in \( S_2 \) are equivalent only in the values of \( VR, ACK\_DUE \).

At any time, the image \( \{VR, ACK\_DUE, SINK, BUSY\} \) of two states \( s \) and \( r \) in \( S_2 \) are \( D\_OUT \) and \( SOURCE \).

By examining the state and receive events, the message \( (ACK) \) has a null image equivalent; let their \( W_1 = \{(DATA', d) : d \in DATASET\} \).

Similarly, the following messages have the null image \( ACK \) message; denote their image

The messages and events are shown in Tables 2 and 3.

For this image protocol, the following assertions state the projected one-way data transfer:

2. \( VS \geq VR \geq VS-1 \).
3. \( (DATA', d) \) in \( CHANNEL \) and \( d = \) and (exact) and (not) and (no A)
4. \( ACK\_DUE \to (D\_OUT) \) and (no D)
5. \( ACK \) in \( CHANNEL \) and (no D)
6. \( D\_OUT \to VS \)
At any time, the image state (i.e., state of the projected function) of $P_1$ is given by the value of $\langle \text{VS}, \text{D\_OUT}, \text{SOURCE}, \text{BUSY} \rangle$. Let $S_1$ denote the image state space of $P_1$. Thus, two states $s$ and $r$ in $S_1$ are equivalent if they differ only in the values of $\text{VS}$, $\text{ACK\_DUE}$ and $\text{SINK}$.

At any time, the image state of $P_2$ is given by the value of $\langle \text{VR}, \text{ACK\_DUE}, \text{SINK}, \text{BUSY} \rangle$. Let $S_2$ denote the image state space of $P_2$. Thus, two states $s$ and $r$ in $S_2$ are equivalent if they differ only in the values of $\text{VS}$, $\text{D\_OUT}$ and $\text{SOURCE}$.

By examining the state changes in the image spaces $S_1$ and $S_2$ due to send and receive events, the following can be shown about messages in $M_1$. The message (ACK) has a null image. The messages (DATA,d) and (DATA\_ACK,d) are equivalent; let their image be denoted by (DATA',d). Thus $M'_1 = \{(\text{DATA}',d) : d \in \text{DATASET}\}$.

Similarly, the following can be shown about messages in $M_2$. All (DATA,d) messages have the null image. All (DATA\_ACK,d) messages are equivalent to the ACK message; denote their image by (ACK'). Thus $M'_2 = \{(\text{ACK}')\}$.

The messages and events of the image protocol for the projected function are shown in Tables 2 and 3.

For this image protocol, assuming error-free channels $C_1$ and $C_2$, the following assertions stating some logical correctness properties of the projected one-way data transfer function have been found (by inspection):

1. $\text{SINK}[i] = \text{SOURCE}[i]$ for $0 < i < \text{VR}$.

2. $\text{VS} > \text{VR} \geq \text{VS}-1$.

3. (DATA',d) in $\text{CHANNEL1} \Rightarrow (\text{D\_OUT})$
   and ($d = \text{SOURCE}[\text{VS}-1]$)
   and (exactly one DATA' message in $\text{CHANNEL1}$)
   and (not $\text{ACK\_DUE}$) and ($\text{VS} = \text{VR} + 1$)
   and (no ACK' message in $\text{CHANNEL2}$).

4. $\text{ACK\_DUE} \Rightarrow (\text{D\_OUT})$
   and (no DATA' message in $\text{CHANNEL1}$)
   and ($\text{VS} = \text{VR}$) and (no ACK' message in $\text{CHANNEL2}$)

5. ACK' in $\text{CHANNEL2} \Rightarrow (\text{D\_OUT})$
   and (no DATA' message in $\text{CHANNEL1}$) and ($\text{VS} = \text{VR}$)
   and (not $\text{ACK\_DUE}$)
   and (exactly one ACK' message in $\text{CHANNEL2}$)

6. not $\text{D\_OUT} \Rightarrow \text{VS} = \text{VR}$
The image protocol can be shown to be well-formed. As a result, the above assertions are valid for the original full-duplex data transfer protocol by virtue of the theorems in [8,9].

Error-free channels have been assumed to make the protocol example small. As shown in [8,9], the method of protocol projections is applicable to protocol models in which a channel may lose, duplicate, and reorder messages. An example involving channels with errors is described in the companion paper [11].

5. CONCLUSIONS

We have employed two protocol examples, a pair of interacting finite state machines and a full-duplex data transfer protocol described by a programming language model, to illustrate the method of protocol projections. We have shown how to construct the entity states, messages, and events of an image protocol from the corresponding quantities of a given protocol. Since image protocols of a well-structured multi-function protocol are much simpler than the original protocol, they can be analyzed more easily using available techniques. In our examples, the image protocols constructed are simple enough that various assertions describing the logical correctness properties of the image protocols have been obtained by inspection. The image protocols can be shown to be well-formed. As a result, the assertions obtained are also valid for the original protocol examples.

The consideration of a protocol model with only two entities is strictly for notational simplicity. Our main results (Theorems 1 and 2 in [7,8]) are applicable to any network of protocol entities interconnected by communication channels. The global state of the general protocol model is again specified by a joint description of the states of the protocol entities and communication channels. The definition of message equivalence, event equivalence, and well-formed image events are simply restated to cover all protocol entities in the model [10].

REFERENCES


ACKNOWLEDGEMENT

The research in this paper was supported by the National Science Foundation under Grant No. ECS78-01803.


ACKNOWLEDGEMENT

The research in this paper was supported by National Science Foundation Grant No. ECS78-01803.

---

Fig. 1. Components of the protocol model.
Fig. 2. Two communicating finite state machines.

Fig. 3. Classes of equivalent entity states.

Fig. 4. Transitions

Fig. 5. An image of a finite state...
An Illustration of Protocol Projections

Fig. 4. Transitions in image state spaces.

Fig. 5. An image protocol of the communicating finite state machines.
### Table 1. Events of entity F1 in the full-duplex data transfer protocol.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Enabling Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SEND_DATA</td>
<td>not BUSY and not D_OUT</td>
<td>DATA := SOURCE[VS]; put(CHANNEL1, DATA, SOURCEx); VS := VS + 1; D_OUT := true</td>
</tr>
<tr>
<td>2. SEND_DATAACK</td>
<td>not BUSY and not D_OUT and ACK_DUP</td>
<td>DATA := SOURCE[VS]; put(CHANNEL1, DATA, SOURCEx); VS := VS + 1; D_OUT := true; ACK_DUP := false</td>
</tr>
<tr>
<td>3. SEND_ACK</td>
<td>not BUSY and ACK_Ack</td>
<td>put(CHANNEL1, ACKx); ACK_DUP := false</td>
</tr>
<tr>
<td>4. START_BUSY</td>
<td>not BUSY</td>
<td>BUSY := true</td>
</tr>
<tr>
<td>5. STOP_BUSY</td>
<td>BUSY</td>
<td>BUSY := false</td>
</tr>
<tr>
<td>6. REC_DATA</td>
<td>first(CHANNEL2) = DATA</td>
<td>get(CHANNEL2, DATA, BACEx); SIM[VR] := BACEx; VS := VS + 1; D_OUT := true</td>
</tr>
<tr>
<td>7. REC_DATAACK</td>
<td>first(CHANNEL2) = DATA</td>
<td>get(CHANNEL2, DATA, BACEx); SIM[VR] := BACEx; VS := VS + 1; D_OUT := false</td>
</tr>
<tr>
<td>8. REC_ACK</td>
<td>first(CHANNEL2) = ACK</td>
<td>get(CHannel2, ACKx); D_OUT := false</td>
</tr>
</tbody>
</table>

### Table 2. Events of F1 in the image protocol for one-way data transfer.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Enabling Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SEND_DATA'</td>
<td>not BUSY and not D_OUT</td>
<td>DATA := SOURCE[VS]; put(CHANNEL1, DATA, SOURCEx); VS := VS + 1; D_OUT := true</td>
</tr>
<tr>
<td>2. START_BUSY'</td>
<td>not BUSY</td>
<td>BUSY := true</td>
</tr>
<tr>
<td>3. STOP_BUSY'</td>
<td>BUSY</td>
<td>BUSY := false</td>
</tr>
<tr>
<td>4. REC_ACK'</td>
<td>first(CHANNEL2) = ACK'</td>
<td>get(CHANNEL2, ACK'x); D_OUT := false</td>
</tr>
</tbody>
</table>

### Table 3. Events of F1 in the image protocol for one-way data transfer.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Enabling Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. REC_DATA'</td>
<td>first(CHANNEL2) = DATA'</td>
<td>get(CHANNEL1, DATA', REXEx); SIM[VR] := REXEx; VS := VS + 1; ACK_DUP := true</td>
</tr>
<tr>
<td>2. START_BUSY'</td>
<td>not BUSY</td>
<td>BUSY := true</td>
</tr>
<tr>
<td>3. STOP_BUSY'</td>
<td>BUSY</td>
<td>BUSY := false</td>
</tr>
<tr>
<td>4. SEND_ACK'</td>
<td>not BUSY and ACK_DUP</td>
<td>put(CHannel1, ACK'x); ACK_DUP := false</td>
</tr>
</tbody>
</table>

### INTRODUCTION

The general direction of the research of the Zurich Research Laboratory has been to use tools for Communicating Systems Network Architecture to validate protocols, with a strong emphasis on the self-consistency of a design regardless of its implementation. This emphasis requires a careful design that is consistent with the requirements of the validation process. The validation process involves the design of a model that represents the system being modeled. This model is then used to verify that it meets the specified requirements. The model is typically a mathematical model, such as a finite-state machine or a Petri net. The model is then analyzed to determine whether it satisfies the specified requirements. If the model does not satisfy the requirements, then the model is modified and re-analyzed until it satisfies the requirements. The process of verifying that the model satisfies the requirements is called model checking. Model checking is a powerful technique that can be used to verify that a system satisfies its requirements. Model checking is a form of formal verification, which is the process of using mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Formal verification is a form of software verification that uses mathematical techniques to verify that a system satisfies its requirements. Forma...