Verifying Safety and Deadlock Properties of Networks of Asynchronously Communicating Processes

by
Fredrik Orava
VERIFYING SAFETY AND DEADLOCK PROPERTIES OF NETWORKS OF ASYNCHRONOUSLY COMMUNICATING PROCESSES

Fredrik Orava
Swedish Institute of Computer Science
P.O. Box 1263
S-164 28 KISTA
SWEDEN
email (uucp): fredrik@sis.se

ABSTRACT
We present a method for specifying and verifying networks of non-deterministic processes that communicate by asynchronous message-passing. The method handles safety and deadlock properties. Networks are specified in an operational manner by transition systems. We say that the specification A implements another specification B if every safety and deadlock property true of A also is true of B. We establish a proof rule for verifying that A implements B in this sense. The proof rule is based on simulation between the states of A and B, and is shown to be complete under the assumption that B is deterministic. We illustrate the method by applying it to the alternating bit protocol.

1. INTRODUCTION
The importance of formal specification and verification in the design of distributed systems is now commonly recognized. A formal specification gives an unambiguous description of the intended behavior of a system, and can be used to verify that the design is correct before it is implemented.

A specification method suitable for distributed systems should allow abstraction from irrelevant details, and should allow modular composition of a specification of a large system from specifications of its components. For example, with OSI-terminology, given the (N-1)-service and the (N)-protocol, it should be possible to verify that their composition fulfills the requirements of the (N)-service.

A class of specification and verification methods that achieve this are based on traces, sequences of interactions between systems ([CH81], [MC81], [BM82], [MCS82], [NDGO86]). Many of these approaches specify a system by its finite traces. However, the set of traces of a system does not capture some aspects of its behavior, such as properties related to deadlock and termination. To illustrate this point, consider two processes: The first process outputs an a and then deadlocks. The second process is non-deterministic and may initially be deadlocked or it may output an a and then
deadlock. These two processes have the same set of traces, namely \(\{\}, \langle a \rangle\) although the second process has an potential deadlock (here \(\{}\) denote the empty sequence).

Extensions of the trace model for synchronously communicating networks usually includes information on potential deadlocks after a sequence of communication events. An example of such a model is the failure model ([BHR84]). A process is deadlocked if it can not accept more input or provide more output that matches the requirements of the environment. The difference compared with our model in this paper is that we consider asynchronously communicating processes, thus our network can not cause deadlock by refusing input.

Our method for specification and verification is based on finite traces. In addition, we specify deadlock properties by designating certain traces as quiescent. After performing a quiescent trace, the execution of a system is suspended until more input arrives ([Mi84], [Jo85], [Jo87]). In a formal sense, when the execution of a system is suspended this is either an intended behavior (i.e. a type of termination) or an unintended behavior (which we call deadlock). By reasoning about quiescent traces, it is possible to verify deadlock and termination properties. For example, consider again the two processes mentioned above. The first process has the only quiescent trace \(\langle a \rangle\), whereas the second process has the two quiescent traces \(\{}\) and \(\langle a \rangle\).

We use transition systems ([PI81],[MP81]) for specifying the set of traces and quiescent traces of networks in an operational style. Related uses of transition systems appear in [La83]. We define an implementation ordering between specifications. Intuitively, a specification \(A\) implements another specification \(B\) if everything \(A\) can do is allowed by \(B\). Formally, \(A\) implements \(B\) if the set of traces and quiescent traces of \(A\) are subsets of the set of traces and quiescent traces of \(B\). We present a rule on transition systems for establishing this implementation relation. The rule is based on a simulation between the states of the two transition systems. This method is related to the bisimulation technique, originally suggested in [Pa81] and later elaborated in the context of SCCS ([Mi83]).

The treatment of transition systems and simulation relations in this paper is adopted from [Jo87]and [LT87]. In the works by Back and Mannila ([BaMa82]), by Chen and Hoare ([ChHo81]), and by Misra and Chandy ([MiCh81]), networks are represented by traces. As we pointed out above, such models can not describe deadlock properties of networks. For networks that always can accept input, Misra and Chandy ([MiCh84]) used the idea of quiescent traces to model liveness properties. Jonson ([Jo85],[Jo87]) formalized this idea and included infinite traces and properties related to fairness. In an earlier paper Misra, Candy and Smith ([MCS82]) proposed an extension to the trace model by adding conditions under which a trace must be extended. Their model is closely related to ours; it does not contain infinite traces and as such can not represent fairness properties. Their specification method is different from ours, and a proof involves reasoning about sequences of actions, whereas a proof with our method only
involves reasoning about individual actions.
In summary, the contributions of this paper are:
(i) a simple model of asynchronously communicating processes, and
(ii) a method to decide whether an implementation conforms to a specification. The
correctness criteria involves safety properties including absence of deadlock.

In the next section we define transition systems as a means to specify networks. In
section 3 we define composition of transition systems. This operation corresponds to
parallel execution of a set of networks. In section 4 we present a rule for proving that
a transition system specifying a network implements another transition system. The
proof rule is shown to be sound. In section 5 we investigate under which conditions the
proof rule is complete. We apply our methods to the alternating bit protocol in section
6. Finally, in section 7 some concluding remarks are drawn.

2. SPECIFYING NETWORKS BY TRANSITION SYSTEMS
We will use transition systems to formulate specifications of networks. A transition sys-
tem has a set of states and a set of transitions between the states. A state consists of
values of internal variables, contents of message buffers, etc. To specify deadlock pro-
certies of networks, we designate some states of the transition systems as resting. The
intuition is that a transition system in a resting state is blocked until it receives more
input. The resting states of a transition system thus describes the allowed deadlocks
and terminations.

Communication events can occur simultaneously with transitions. There are two
types of communication events: input events and output events. We also consider silent
events which label transitions which merely update the state.

We represent a transition system as a set of labeled guarded commands of the form:

\[ g \xrightarrow{e} [\tilde{y} := \tilde{v}] \]

where the guard, \( g \), is a boolean expression, \( \tilde{y} \) is a subset of the local variables of the
network, \( \tilde{v} \) is a set of value expressions, and \( e \) is an event. The intuition of such a
labeled guarded command is that when the guard \( g \) is true, then the transition system
may perform a computation step, which assigns the values of \( \tilde{v} \) to the variables \( \tilde{y} \). The
computation step is labeled with an event \( e \). Events can be of two types:
(i) Internal events denoted by \( \tau \).
(ii) Communication events denoted by \( ch(m) \), where \( ch \) is an input channel or an output
channel and \( m \) is a message possibly dependent on state variables. The intended
meaning is that the network receives or transmits the message \( m \) via the channel
\( ch \).

Definition. A transition system is a tuple \((I, O, \bar{x}, \Phi, G)\) where
- \( I \) is a set of input events, not containing the silent event \( \tau \).
- \( O \) is a set of output events, not containing the silent event \( \tau \).
- \( \tilde{x} \) is a set of state variables \( \{x_1, \ldots, x_n\} \).
- \( \Phi \) is an initial predicate over the state variables.
- \( G \) is a set of labeled guarded commands.

**Definition.** Let \( N \) be the transition system \( \langle I, O, \tilde{x}, \Phi, G \rangle \).

- A state, \( \sigma \), of \( N \) is an assignment of values to \( \tilde{x} \). The evaluation of the expression exp relative to the assignment \( \sigma \) is denoted \( \sigma(\text{exp}) \).
- A transition of \( N \) is a labeled transition
  \[ \sigma \xrightarrow{e} \sigma' \]
  such that there is a labeled guarded command \( g \xrightarrow{e} [\tilde{y} := \tilde{v}] \) where we write \( \tilde{y} \) for \( y_1, \ldots, y_n \) and \( \tilde{v} \) for \( v_1, \ldots, v_n \), in \( G \), for which \( \sigma(g) \) is true and \( \sigma' \) differs from \( \sigma \) only in the values of \( y_i \), i.e. \( \sigma'(u) = \sigma(v_i) \) if \( u = y_i \) and \( \sigma'(u) = \sigma(u) \) otherwise. If \( e \) is of type output and depends on state variables, then \( e \) is interpreted according to \( \sigma \).
- A transition \( \sigma \xrightarrow{e} \sigma' \) is enabled in state \( \sigma \).

Let \( N \) be the transition system \( \langle I, O, \tilde{x}, \Phi, G \rangle \). We define the resting states of \( N \) to be the states in which no transition labeled with output or silent event is enabled. We represent resting states by a predicate \( \mathcal{R}_N \) over the state variables in \( N \) such that \( \sigma(\mathcal{R}_N) \) is true if and only if \( \sigma \) is a resting state of \( N \). The predicate \( \mathcal{R}_N \) can be obtained as the conjunction of the negation of the guards for each guarded command labeled with an output or silent event.

It will be convenient to consider communication events as instantaneous events. It will also be convenient to assume that when two processes communicate, an output event in one process is equivalent with an input event in the other process. However, these two assumptions contradict the asynchronous nature of communication: when a message is transmitted it may not be received until later. In order to resolve this conflict we follow [Mi84], [Jo85], [Jo87] and regard the reception of a message by a process to occur when the message is transmitted by the sending process. The message is not necessarily received at the destination, but it is bound to be received at the destination at some later moment. It follows that the occurrences of input events are controlled by the environment of the process rather than by the process itself. Hence, in each state of the process all input events must be enabled. This requirement is formalized as follows:

For each state \( \sigma \) and input event \( e \)
there is a state \( \sigma' \) such that \( \sigma \xrightarrow{e} \sigma' \) is enabled

The standard way to achieve this is to use a state variable, \( buf_{ch} \), for each input channel \( ch \) and include

\[ \text{true} \xrightarrow{ch(d)} [buf_{ch} := buf_{ch} \cdot d] \]

in \( G \). Here \( buf_{ch} := buf_{ch} \cdot d \) means appending \( d \) to the end of \( buf_{ch} \). In this command
as in the following we consider \( d \) to be implicitly universal quantified; formally this command represents a set of commands (one for each possible value of \( d \)).

We now define the *denotation* of a transition system. This denotation will be used in the next section where we define the implementation relation. The denotation of a transition system is the set of traces, i.e., sequences of communication events a transition system may perform, together with the set of quiescent traces, i.e., sequences of communication events after which the transition system may reach a resting state.

**Definition.** Let \( N \) be the transition system \( \langle I, O, \tilde{x}, \Phi, G \rangle \). A *computation* of \( N \) is a finite sequence of transitions

\[
\sigma^0 \xrightarrow{e_1} \sigma^1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} \sigma^n
\]

which fulfills conditions (i) and (ii) below. If the computation also fulfills condition (iii), then the computation is a *quiescent computation*.

(i) **Initialization.** The first state of the sequence, \( \sigma^0 \), satisfies \( \Phi \).

(ii) **State to state sequencing.** Each transition \( \sigma^{i-1} \xrightarrow{e_i} \sigma^i \) is a transition of \( N \).

(iii) **Quiescence.** The last state of the sequence, \( \sigma^n \), satisfies \( R_N \). \( \square \)

**Definition.** Let \( N = \langle I, O, \tilde{x}, \Phi, G \rangle \) be a transition system.

(i) A *trace* of \( N \) is the sequence of communication events (i.e. non-\( \tau \) events) in a computation of \( N \).

(ii) A *quiescent trace* of \( N \) is the sequence of communication events (i.e. non-\( \tau \) events) in a quiescent computation of \( N \).

(iii) The *denotation* of \( N \), denoted by \( [N] \), is the tuple \( \langle I(N), O(N), T(N), Q(N) \rangle \), where

- \( I(N) = I \), the set of input events of \( N \)
- \( O(N) = O \), the set of output events of \( N \)
- \( T(N) \) is the set of traces of \( N \)
- \( Q(N) \) is the set of quiescent traces of \( N \) \( \square \)

Note that the set of quiescent traces, \( Q(N) \), is included in the set of traces, \( T(N) \). Intuitively, \( T(N) \) represents safety properties such as: "the event \( a \) is always preceded by the event \( b \)" or "messages are delivered in the same order as they were received". The properties that can be expressed in this way are safety properties, with the notable exception of deadlock properties. The set of quiescent traces, \( Q(N) \), represents termination or deadlock properties such as "the network may deadlock after the sequence of events \( s \)".

**Example.** (Merge) A network receives sequences of input along channels \( \alpha \) and \( \beta \) and transmits the received inputs along channel \( \gamma \), preserving the ordering of items among individual channels. The communication events of the network are of the form \( \alpha(d) \), \( \beta(d) \), or \( \gamma(d) \) with \( d \) taken from some set of values. We present a specification stating that the sequences of events \( \gamma(d) \) is the merge of the sequences of events \( \alpha(d) \) and \( \beta(d) \).
A state of the transition system corresponds to the sequences received at \( \alpha \) and \( \beta \), but not yet transmitted at \( \gamma \). We use the following notation for finite sequences:

- \( \emptyset \) denotes the empty sequence.
- \( s \cdot d \) denotes the result of adding the element \( d \) to the end of the sequence \( s \).
- \( tl(s) \) denotes the result of deleting the first element of the sequence \( s \).
- \( hd(s) \) denotes the first element of the sequence \( s \).


<table>
<thead>
<tr>
<th>Input events:</th>
<th>( \alpha(d), \beta(d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output events:</td>
<td>( \gamma(d) )</td>
</tr>
<tr>
<td>Variables:</td>
<td>( buf )</td>
</tr>
<tr>
<td>Initialization:</td>
<td>( buf = \emptyset )</td>
</tr>
<tr>
<td>Commands:</td>
<td>( \text{true} \xrightarrow{\alpha(d)} [buf := buf \cdot d] )</td>
</tr>
<tr>
<td></td>
<td>( \text{true} \xrightarrow{\beta(d)} [buf := buf \cdot d] )</td>
</tr>
<tr>
<td></td>
<td>( buf \neq \emptyset \xrightarrow{\gamma(hd(buf))} [buf := tl(buf)] )</td>
</tr>
</tbody>
</table>

Figure 2.1: Specification of a merge network

The resting condition, \( \mathcal{R} \), for the merge network of figure 2.1 is \( buf = \emptyset \), i.e. the network can only halt in a state such that all messages received along \( \alpha \) and \( \beta \) are transmitted along \( \gamma \).

3. COMPOSITION OF SPECIFICATIONS

A specification \( N \) is intended to specify the externally observable behavior of a network. To be able to investigate the resulting externally observable behavior of a network built from subnetworks, we define an operation, called parallel composition of transition systems.

Assume that we are specifying a network consisting of communicating subnetworks. The subnetworks are interconnected by internal channels and communicate by sending messages via these channels. Channels of the subnetworks that are not connected to other subnetworks are referred to as external channels; via these channels the network communicates with the environment. The output of a message by one subnetwork into an internal channel is an input of some other subnetwork. Thus, communication via the internal channels are internal events of the composed network, and communication via the external channels are external (observable) events of the composed network. This leads to the following operation on the transition systems: The parallel composition of the transition systems \( N_1, \ldots, N_k \) yields a new transition system \( N \). A state of \( N \) is a tuple, whose components are states of the transition systems \( N_1, \ldots, N_k \). A transition of \( N \) is either an action of one component \( N_i \), or it is an action where two components participate in a communication event. In the latter case the transition affects the states of both participating components in \( N \). The event in such a joint transition is internal to the network \( N_i \), and should as such no longer be observable from the outside of the transition system. We achieve this by replacing these communication events by the silent event \( \tau \). To ensure that internal channels interconnect only two processes, and
to ensure that all local variables are unique, we need to define the concept of a set of transition system to be compatible.

Definition. The transition systems \( N_i = \langle I_i, O_i, \tilde{x}_i, \Phi_i, G_i \rangle \) for \( i = 1, \ldots, k \) are compatible if the following holds whenever \( i \neq j \):

- \( O_i \cap O_j = \emptyset \),
- \( I_i \cap I_j = \emptyset \), and
- \( \tilde{x}_i \cap \tilde{x}_j = \emptyset \), i.e. all variable names are distinct.

Definition. Let \( N_i = \langle I_i, O_i, \tilde{x}_i, \Phi_i, G_i \rangle \) for \( i = 1, \ldots, k \) be compatible transition systems. Define \( IE = (\cup_i O_i) \cap (\cup_i I_i) \) to be the set of internal events. The parallel composition of \( N_1, \ldots, N_k \), denoted \( (N_1 \parallel N_2, \ldots \parallel N_k) \) is the transition system \( N = \langle I, O, \tilde{x}, \Phi, G \rangle \), where

- \( O = \cup_i O_i - IE \), i.e. the set of external output events of \( N \).
- \( I = \cup_i I_i - IE \), i.e. the set of external input events of \( N \).
- \( \tilde{x} = \tilde{x}_1 \cup \ldots \cup \tilde{x}_k \).
- \( \Phi = \Phi_1 \land \ldots \land \Phi_k \).

\( G \) is obtained from \( T_1, \ldots, T_k \) as follows:

(i) If \( e \in IE \), then \( e \) is an event internal to \( N \) but external to two subsystems \( N_i \) and \( N_j \). Let

\[ g_i \xrightarrow{e} [\bar{y}_i := \bar{v}_i] \text{ and } g_j \xrightarrow{e} [\bar{y}_j := \bar{v}_j] \]

be labeled guarded commands in \( G_i \) and \( G_j \) respectively. Then there is a labeled guarded command in \( G \) of the form

\[ g_i \land g_j \xrightarrow{e} [\bar{y}_i := \bar{v}_i, \bar{y}_j := \bar{v}_j] \]

(ii) Transitions with external (i.e. \( e \in I \cup O \)) or silent (i.e. \( e = \tau \)) events are not synchronized. Thus each labeled guarded command with an external or silent event in \( G_i \) is also a labeled guarded command in \( G \).}

Remember that a network must always be prepared to accept input. Thus, a composed network is in a resting state if and only if all subnetworks are resting, i.e.

\[ \mathcal{R}_N = \mathcal{R}_{N_1} \land \ldots \land \mathcal{R}_{N_k} \]

It can be proven that the denotation, \( (I(N), O(N), T(N), Q(N)) \), of the parallel composition \( N \) of the transition systems \( N_1, \ldots, N_k \) can be obtained from the denotations \( (I(N_i), O(N_i), T(N_i), Q(N_i)) \) of \( N_1, \ldots, N_k \) as follows.

- \( O(N) = \cup_i O(N_i) - IE \)
- \( I(N) = \cup_i I(N_i) - IE \)
- \( T(N) = \{ s \setminus IE : s \in (\cup_i E(N_i))^* \land s|E(N_i) \in T(N_i) \text{ for all } i \} \)
- \( Q(N) = \{ s \setminus IE : s \in (\cup_i E(N_i))^* \land s|E(N_i) \in Q(N_i) \text{ for all } i \} \)
where we have used the following notation:

- $E(N)$ denotes the union of input and output events of the transition system $N$, i.e. $E(N) = I(N) \cup O(N)$.
- $s|E$ denotes the restriction of the sequence of communication events $s$ to the set of communication events in $E$, i.e. the maximal subsequence of $s$ containing only events in $E$.
- $s \setminus E$ denotes the subsequence of the communication events $s$ not in the set of communication events $E$, i.e. the result of deleting the events in $E$ form $s$.
- $E^*$ denotes the set of finite sequences of events in $E$.

For this result to hold it is essential that the networks involved always can accept input.

4. CORRECTNESS OF IMPLEMENTATION

In this section we present a rule for proving that an implementation correctly implements a specification. We use transition systems to express both implementations and specifications, and view the implementations as less abstract (or more concrete) specifications. We are interested in verifying only the observable behavior of an implementation. Thus, it is the set of traces and quiescent traces of a transition system that reflects the interesting properties of a network. We will use the philosophy that an implementation correctly implements a specification if everything the implementation can do is allowed by the specification. This leads to the following correctness criterion: Let $N_1$ and $N_2$ be two transition systems. We say that $N_1$ implements $N_2$ if every (quiescent) trace of $N_1$ is, when restricted to the set of communication events of $N_2$, also a (quiescent) trace of $N_2$. This is formalized in the definition below.

**Definition:** Let $N_1$ and $N_2$ be transition systems. We say that $N_1$ implements $N_2$ if the following holds:

(i) $I(N_2) \subseteq I(N_1)$,
(ii) $O(N_2) \subseteq O(N_1)$,
(iii) $T(N_1)|E(N_2) \subseteq T(N_2)$, and
(iv) $Q(N_1)|E(N_2) \subseteq Q(N_2)$

Requirement (iii) states that $N_1$ can only perform sequences of communication events that are also sequences of communication events of $N_2$. Requirement (iv) states that $N_1$ can only halt after a sequence of communication events if $N_2$ can halt after the "same" sequence (i.e. the sequence restricted to the set of communication events of $N_2$). That is, all deadlocks of the implementation must also be deadlocks of the specification. This implies that a property true of traces (quiescent traces) of $N_1$ also is true of traces (quiescent traces) of $N_2$.

We now present a proof rule on the transition systems for proving that $N_1$ implements $N_2$. The main idea is to find a simulation relation between the states of $N_1$ and
the states of $N_2$. The rule is based upon related verification methods in [Jon87] and [LT87].

**Definition.** Let $N_1$ and $N_2$ be transition systems with $I(N_2) \subseteq I(N_1)$, and $O(N_2) \subseteq O(N_1)$. A *simulation relation* $S$ is a relation between the states of $N_1$ and the states of $N_2$ such that

(i) for each initial state $\sigma_1^0$ of $N_1$ there is an initial state $\sigma_2^0$ of $N_2$ such that $S(\sigma_1^0, \sigma_2^0)$ holds.

(ii) whenever $S(\sigma_1, \sigma_2)$ holds and $\sigma_1 \xrightarrow{e} \sigma_1'$ is a transition of $N_1$, then either

(a) $e \in E(N_2)$ and there exists a state $\sigma_2'$ of $N_2$ such that $\sigma_2 \xrightarrow{e} \sigma_2'$ is a transition of $N_2$ such that $S(\sigma_1', \sigma_2')$ holds, or

(b) $e \notin E(N_2)$ and $S(\sigma_1', \sigma_2)$ holds. We refer to this case as $N_2$ doing a *null transition*.

**Definition.** A simulation relation $S$ between the states of $N_1$ and $N_2$ is *quiescence preserving* if the following holds:

for all states $\sigma_1$ in $N_1$, if $S(\sigma_1, \sigma_2)$ and $\sigma_1$ is resting then $\sigma_2$ is resting.

<table>
<thead>
<tr>
<th>Rule</th>
<th>For proving $N_1$ implements $N_2$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>Check that $I(N_2) \subseteq I(N_1)$ and $O(N_2) \subseteq O(N_1)$</td>
</tr>
<tr>
<td>(ii)</td>
<td>Find a simulation relation $S(\sigma_1, \sigma_2)$, between the states of $N_1$ and $N_2$.</td>
</tr>
<tr>
<td>(iii)</td>
<td>Check that the simulation relation $S$ is quiescence preserving.</td>
</tr>
</tbody>
</table>

**Proof of soundness.** (Sketch) Given a quiescence preserving simulation relation $S$ between the states of $N_1$ and $N_2$, we can by induction conclude that for each computation $C_1$ of $N_1$ there exists a computation $C_2$ of $N_2$ that simulates $C_1$ step-by-step. By induction on the length of $C_1$ we conclude that for each computation $C_1$ of $N_1$ with the sequence of communication events $s \in T(N_1)$ there is a corresponding computation $C_2$ of $N_2$ with the sequence of communication events $s \mid E(N_2) \in T(N_2)$. By an analogous reasoning we conclude that same holds for the quiescent computations of $N_1$.

**Example.** (Merge 3) As an example we shall prove that the parallel composition of two merge networks yields a network that merges three input channels onto one output channel. A transition system that specifies a merge network, with input channels $\alpha$ and $\beta$ and output channel $\gamma$, was given in figure 2.1. We shall now show that if we compose this merge network with a merge network with input channels $\gamma$ and $\delta$ and output channel $\epsilon$ we get a Merge 3 network with input channels $\alpha$, $\beta$ and $\delta$, and output channel $\epsilon$.

We specify the second merge network in analogy with the merge network in figure
2.1 and obtain their parallel composition according to the rules in section 3 as the
transition system $M_3$ below.

\begin{center}
\begin{tabular}{|l|}
\hline
Input: & $\alpha(d), \beta(d), \delta(d)$ \\
Output: & $\epsilon(d)$ \\
Variables: & $buf_1, buf_2$ \\
Initialization: & $buf_1 := buf_2 = ()$ \\
Commands: & \begin{align}
true \frac{\alpha(d)}{\beta(d)}[buf_1 := buf_1 \bullet d] & \quad (G1) \\
true \frac{\beta(d)}{\delta(d)}[buf_1 := buf_1 \bullet d] & \quad (G2) \\
true \frac{\delta(d)}{\epsilon(d)}[buf_2 := buf_2 \bullet d] & \quad (G3) \\
buf_1 \neq () \frac{\epsilon}{\beta}[buf_2 := buf_2 \bullet hd(buf_1), buf_1 := tl(buf_1)] & \quad (G4) \\
buf_2 \neq () \frac{\epsilon}{\delta}[buf_2 := tl(buf_2)] & \quad (G5)
\end{align}
\end{tabular}
\end{center}

**Figure 4.1:** The parallel composition $M_3$, of two merge networks

The resting condition for $M_3$ is obtained as the conjunction of the resting conditions for the two components, i.e. $R_{M_3} = [buf_1 = buf_2 = ()]$. We shall prove that the transition system given if figure 4.1 implements the transition system $SS$, with the resting condition $R_{SS} = [buf = ()]$, below.

\begin{center}
\begin{tabular}{|l|}
\hline
Input: & $\alpha(d), \beta(d), \delta(d)$ \\
Output: & $\epsilon(d)$ \\
Variables: & $buf$ \\
Initialization: & $buf = ()$ \\
Commands: & \begin{align}
true \frac{\alpha(d)}{\beta(d)}[buf := buf \bullet d] & \quad (S1) \\
true \frac{\beta(d)}{\delta(d)}[buf := buf \bullet d] & \quad (S2) \\
true \frac{\delta(d)}{\alpha(d)}[buf := buf \bullet d] & \quad (S3) \\
buf \neq () \frac{\epsilon}{\delta}[buf := tl(buf)] & \quad (S4)
\end{align}
\end{tabular}
\end{center}

**Figure 4.2:** A specification $SS$, of a 3-way merge network

According to the proof rule for the implementation relation, we must find a quiescence preserving simulation relation $S$ between the states of $M_3$ and $SS$. The simulation relation $S$ is given by:

$$buf \in buf_1 \parallel_{sq} buf_2$$

where the operation $\parallel_{sq}$ (shuffle) on sequences $q$ and $r$ gives the set of interleavings of the sequences $q$ and $r$. Note that if $q$ and $r$ both are empty, then $q \parallel_{sq} r$ is the empty sequence.

It is a simple exercise to verify that $S$ is a simulation relation. We must verify the following cases:

(i) For the initial states we get $() \in \{() \parallel_{sq} ()\} = \{()\}$ which is true.

(ii) To check that $S$ is preserved by each transition of $M_3$ we see that the transition (G1) is simulated by (S1), (G2) by (S2), (G3) by (S3), (G4) by a null transition of $SS$, and the transition (G5) is simulated by (S4). In order to do this we must verify
the following implications:

\[ [\text{buf} \in \text{buf}_1 \|_\text{sq} \text{buf}_2] \Rightarrow [\text{buf} \circ \text{d} \in (\text{buf}_1 \circ \text{d}) \|_\text{sq} \text{buf}_2] \quad (G1 \text{ and } G2) \]
\[ [\text{buf} \in \text{buf}_1 \|_\text{sq} \text{buf}_2] \Rightarrow [\text{buf} \circ \text{d} \in \text{buf}_1 \|_\text{sq} (\text{buf}_2 \circ \text{d})] \quad (G3) \]
\[ [\text{buf}_1 \not= \emptyset] \wedge [\text{buf} \in \text{buf}_1 \|_\text{sq} \text{buf}_2] \Rightarrow [\text{buf} \in \text{tl(buf}_1) \|_\text{sq} (\text{hd(buf}_1) \circ \text{buf}_2)] \quad (G4) \]
\[ [\text{buf}_2 \not= \emptyset] \wedge [\text{buf} \in \text{buf}_1 \|_\text{sq} \text{buf}_2] \Rightarrow [\text{tl(buf)} \in \text{buf}_1 \|_\text{sq} \text{tl(buf}_2)] \quad (G5) \]

It remains to check that \( S \) is quiescence preserving. We must verify that whenever \( S \)
holds between a state of \( M3 \) and a state of \( SS \) and \( R_M3 \) is true then also \( R_SS \) is true. We must prove that:

\[ [\text{buf} \in \text{buf}_1 \|_\text{sq} \text{buf}_2 \wedge \text{buf}_1 = \text{buf}_2 = \emptyset] \Rightarrow [\text{buf} = \emptyset] \]

which follows immediately from the definition of \( \|_\text{sq} \). We have now verified that the
requirements of proof rule above are fulfilled, and can conclude that \( M3 \) implements \( SS \).

5. COMPLETENESS

To show the completeness of the proof rule in the previous section we must show that
if the transition system \( N_1 \) implements the transition system \( N_2 \) then there exists a
simulation relation between the states of \( N_1 \) and \( N_2 \). However, this is not in general
the case. To guarantee the existence of a simulation relation, we require the transition
system \( N_2 \) to be deterministic. This requirement is necessary to guarantee the
uniqueness of a computation with respect to the sequence of communication events.
Requiring \( N_2 \) to be deterministic means that there exists only one computation of \( N_2 \)
corresponding to each sequence of communication events in the set of traces of \( N_2 \). The
theorem below is a slight modification of a similar result in [Jon87].

Definition. A transition system \( N \) is deterministic if it has only one initial state, no
guarded commands labeled with silent events, and at most one transition of the form \( \sigma \xrightarrow{e} \sigma' \) for each state \( \sigma \) and communication event \( e \).

Theorem. Let \( N_1 \) and \( N_2 \) be two transition system. If \( N_1 \) implements \( N_2 \), and \( N_2 \) is
deterministic then the premises of the proof rule holds.

Proof. Requirement (i) of the proof rule, stating that \( I(N_2) \subseteq I(N_1) \) and \( O(N_2) \subseteq \ O(N_1) \), follows from the fact that \( N_1 \) implements \( N_2 \) implies \( I(N_2) \subseteq I(N_1) \) and \( O(N_2) \subseteq O(N_1) \) according to the definition of the implementation relation. To es-
end requiring (ii) we must show that there exists a simulation relation between
the states of \( N_1 \) and \( N_2 \). We do this by constructing a relation and verify that it satis-
ifies the requirements of a simulation relation. Let \( \text{Com}(C) \) denote the sequence of
communication events in a computation \( C \).

11
Define $S(\sigma_1,\sigma_2)$ to be true if and only if there exists computations $C_1$ and $C_2$ of $N_1$ and $N_2$ respectively, such that:

(i) $\mathsf{Com}(C_1)|E(N_2) = \mathsf{Com}(C_2)$
(ii) $\sigma_1$ is the last state of $C_1$, and
(iii) $\sigma_2$ is the last state of $C_2$.

To verify that $S$ is a simulation relation we check the conditions of the definition of a simulation relation.

(i) Consider the initial states $\sigma_1^0$ and $\sigma_2^0$ of the transition systems $N_1$ and $N_2$. Choose $C_1$ to be $\sigma_1^0$ and $C_2$ to be $\sigma_2^0$ in the above definition of $S$. Clearly it holds that $\mathsf{Com}(C_1) = () = \mathsf{Com}(C_2)$ and we can conclude that $S(\sigma_1^0,\sigma_2^0)$ holds according to the definition of $S$ above.

(ii) Assume that $S(\sigma_1,\sigma_2)$ holds and that $\sigma_1 \xrightarrow{e} \sigma'_1$ is a transition of $N_1$. We must show that either:
(a) $e \in E(N_2)$ and there exists a transition $\sigma_2 \xrightarrow{e} \sigma'_2$ of $N_2$ such that $S(\sigma'_1,\sigma'_2)$ holds.

By the definition of $S$, the states $\sigma_1$ and $\sigma_2$ are the last states of two computations $C_1$ and $C_2$ of $N_1$ and $N_2$. Extend $C_1$ by the transition $\sigma_1 \xrightarrow{e} \sigma'_1$ to get $C'_1$. Then $C'_1$ is also a computation of $N_1$, and thus $\mathsf{Com}(C'_1) = \mathsf{Com}(C_1) \circ e \in T(N_1)$. Since $N_1$ implements $N_2$, it follows from the definition of the implementation relation and from $e \in E(N_2)$ that $\mathsf{Com}(C_1) \circ e | E(N_2) = \mathsf{Com}(C_2) \circ e \in T(N_2)$ and there exists a computation $C'_2$ of $N_2$ with the sequence of communication events $\mathsf{Com}(C_2) \circ e$. Let $\sigma'_2$ be the last state of $C'_2$. Since $N_2$ is deterministic the computation $C'_2$ is uniquely determined by the sequence of communication events $\mathsf{Com}(C_2) \circ e$ and extends the computation $C_2$ by the transition $\sigma_2 \xrightarrow{e} \sigma'_2$. Thus we can conclude that $S(\sigma'_1,\sigma'_2)$ holds.

(b) $e \notin E(N_2)$ and $S(\sigma'_1,\sigma_2)$ holds. This follows by an analogous reasoning.

Finally, we must verify that $S$ is quiescence preserving. Assume that $S(\sigma_1,\sigma_2)$ holds and $\sigma_1(\mathcal{R}_{N_1}) = \text{true}$. We must show that $\sigma_2(\mathcal{R}_{N_2}) = \text{true}$.

By the definition of $S$, the states $\sigma_1$ and $\sigma_2$ are the last states of two computations $C_1$ and $C_2$ of $N_1$ and $N_2$. Since $\sigma_1(\mathcal{R}_{N_1}) = \text{true}$ it follows from the definition of $Q(N)$ that $\mathsf{Com}(C_1) \in Q(N_1)$. Since $N_1$ implements $N_2$, it follows from the definition of the implementation relation that $\mathsf{Com}(C_1) | E(N_2) = \mathsf{Com}(C_2) \in Q(N_2)$. Since $N_2$ is deterministic, there is only one computation of $N_2$ with the sequence of communication events $\mathsf{Com}(C_2) \in Q(N_2)$. Thus, by the definition of $Q(N)$, $C(N_2)$ must terminate in a resting state, i.e. $\sigma_2(\mathcal{R}_{N_2}) = \text{true}$.

Thus, $S$ is indeed a quiescence preserving simulation relation.
6. VERIFICATION OF THE ALTERNATING BIT PROTOCOL

To illustrate the use of our verification method we specify and verify the alternating bit protocol (AB-protocol) ([BSW96]). The AB-protocol is a simple data link protocol, which guarantees error-free one way communication across a faulty medium. The AB-protocol is a standard example for which correctness proofs in various styles have been given frequently in the literature ([Bo78], [BK84], [La83], [Pa85], [Jo85]).

6.1 Protocol architecture.

The architecture of the AB-protocol is shown in figure 6.1. The protocol consists of four components: a Sender, a Receiver, and two media. The components are connected with each other and with the environment via six channels. A user can access the AB-protocol via the input channel X and the output channel Y.

![Figure 6.1: The architecture of the alternating bit protocol](image)

6.2 Service specification.

The service specification of the AB-protocol is the service of an unbounded FIFO queue. Each message received on X is delivered on Y in the same order as received. In the service specification we use a buffer variable buf to model the unbounded input channel of the protocol. The resting condition for the service specification is: $R_{SS} = [buf = \{\}]$.

<table>
<thead>
<tr>
<th>Input events:</th>
<th>$X(d)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output events:</td>
<td>$Y(d)$</td>
</tr>
<tr>
<td>Variables:</td>
<td>buf</td>
</tr>
<tr>
<td>Initialization:</td>
<td>$buf = {}$</td>
</tr>
<tr>
<td>Commands:</td>
<td>true $X(d)[buf := buf \cdot d]$ (S1)</td>
</tr>
<tr>
<td></td>
<td>$buf \neq {} \xrightarrow{Y(bd(buf))} [buf := tl(buf)]$ (S2)</td>
</tr>
</tbody>
</table>

![Figure 6.2: Service specification](image)
6.3 Protocol specification.

The protocol specification consists of a sender, $S$, a receiver, $R$, and two media, $MSR$ and $MRS$. The operation of the protocol is as follows. The sender accepts messages to be transmitted on channel $X$. When it has a message to send it adds a one bit sequence number to the message and transmits the message to the medium $MSR$ via the channel $\alpha$. The sender then awaits an acknowledgement with the same sequence number on the channel $\delta$ from the medium $MRS$. When an acknowledgement with the correct sequence number is received the procedure is repeated with the sequence number inverted. A transition system specifying $S$ is given in figure 6.3. The resting condition is: $R_S = [buf_X = () \land buf_\delta = () \land a = s]$.

![Figure 6.3: Specification of the sender.](image)

The receiver acknowledges all messages from $MSR$ with unexpected sequence numbers by transmitting the sequence number of the next expected message to the sender via $MRS$. Each message with a sequence number different from the preceding one is transmitted via channel $Y$. The specification is given in figure 6.4, $R_R = [buf_\beta = ()]$.

![Figure 6.4: Specification of the receiver.](image)

The media, $MSR$ and $MRS$ are potentially infinite buffers. Each medium can lose but not reorder messages. Corruption of messages can be taken into account by assuming the existence of some mechanism that will discard corrupted messages and thereby modeling corrupted messages as loss. Specifications of the media are given in figure 6.5 and 6.6. The resting conditions are: $R_{MSR} = [buf_\alpha = ()]$ and $R_{MRS} = [buf_\gamma = ()]$ respectively.
6.4 Verification.

We shall now prove that the network consisting of the four components $S$, $R$, $MSR$ and $MRS$ satisfies the specification $SS$. The first step towards this goal is to construct the parallel composition $(S\|MSR\|R\|MRS)$ denoted $PS$. This is done, by applying the rules in the definition of the parallel composition operation, below. The resting condition for the composition is obtained as the conjunction of the resting conditions for $S$, $R$, $MSR$ and $MRS$, i.e.

$$\mathcal{R}_PS = [buf_X = () \land buf_\delta = () \land s = a \land buf_\beta = () \land buf_\alpha = () \land buf_\gamma = ()]$$

<table>
<thead>
<tr>
<th>Input events: $X(d)$</th>
<th>Output events: $Y(d)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables: $buf_\alpha, buf_\beta, buf_\gamma, buf_\delta, s, a, Next$</td>
<td></td>
</tr>
<tr>
<td>Initialization: $buf_\alpha = buf_\beta = buf_\gamma = (), s = a = Next = 0$</td>
<td></td>
</tr>
<tr>
<td>Commands: $X(d)[buf_x := buf_x \cdot d]$ (G1)</td>
<td></td>
</tr>
<tr>
<td>$buf_\alpha \neq () \land s = a \overset{r}{\rightarrow}[s := s, r := hd(buf_\alpha), buf_x := tl(buf_\alpha)]$ (G2)</td>
<td></td>
</tr>
<tr>
<td>$s \neq a \overset{r}{\rightarrow}[buf_x := buf_\alpha \cdot (r, a)]$ (G3)</td>
<td></td>
</tr>
<tr>
<td>$buf_\alpha \neq () \overset{r}{\rightarrow}[buf_x := tl(buf_\alpha)]$ (G4)</td>
<td></td>
</tr>
<tr>
<td>$buf_\alpha = (d, n) \overset{r}{\rightarrow}[buf_x := tl(buf_\alpha), buf_\beta := buf_\beta \cdot (d, n)]$ (G5)</td>
<td></td>
</tr>
<tr>
<td>$hd(buf_\beta) = (d, Next) \overset{Y(d)}{\rightarrow}[buf_\beta := tl(buf_\beta), Next := \overline{Next}]$ (G6)</td>
<td></td>
</tr>
<tr>
<td>$hd(buf_\beta) = (d, n) \land n \neq \overline{Next} \overset{r}{\rightarrow}[buf_\beta := tl(buf_\beta), buf_\beta := buf_\beta \cdot Next]$ (G7)</td>
<td></td>
</tr>
<tr>
<td>$buf_\gamma \neq () \overset{r}{\rightarrow}[buf_\gamma := tl(buf_\gamma)]$ (G8)</td>
<td></td>
</tr>
<tr>
<td>$hd(buf_\gamma) = n \overset{r}{\rightarrow}[buf_\gamma := tl(buf_\gamma), buf_\delta := buf_\delta \cdot n]$ (G9)</td>
<td></td>
</tr>
<tr>
<td>$buf_\delta \neq () \overset{r}{\rightarrow}[a := hd(buf_\delta), buf_\delta := tl(buf_\delta)]$ (G10)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.7: Parallel composition $(S\|MSR\|R\|MRS)$ of the sender, receiver and the two media.

We now prove that $(S\|R\|MSR\|MRS)$ implements $SS$. The proof is divided into two main steps: we first establish a simulation relation, $S$, then we prove that whenever $S$ holds between two states $\sigma_{PS}$ and $\sigma_{SS}$ in $PS$ and $SS$ respectively, and $\sigma_{PS}(\mathcal{R}_PS)$
holds then also $\sigma_{SS}(R_{SS})$ holds. That is, a resting state of $PS$ can only be simulated by a resting state of $SS$. To establish the $S$, we must first establish an invariant over the states of $PS$. The invariant consists of three parts:

1. $a = s \Rightarrow \begin{cases} s = Next \\ a = Next \end{cases}$
2. $buf_\beta \cdot buf_\alpha \in a^* Next^* (r, z)^*$
3. $buf_\beta \cdot buf_\alpha \in (d, Next)^* (r, z)^*$

The first invariant, (1), relates the sequence numbers of the sender to the sequence number of the receiver. The second and third invariants, (2) and (3), relates the sequence numbers of the receiver and the sender to the contents in the message buffers.

**Proof.** We first prove that I1, I2 and I3 are invariants over $PS$. To do this we must verify that I1, I2 and I3 are preserved by all transitions of $PS$. We omit the details. As an illustration, we prove that the labeled guarded command G2 does not alter the truth of the invariants. G2 deletes one element from $buf_x$ and updates the variables $s$ and $r$. We must prove the following implication:

$$\begin{align*}
(a = s) \Rightarrow \begin{cases}
   s = Next \\
a = Next
\end{cases}
\quad \xrightarrow{G2} \quad \begin{cases}
   buf_\beta \cdot buf_\alpha \in (d, Next)^* (r, z)^* \\
   s \neq a
\end{cases}
\end{align*}$$

The simulation relation, $S$, is given by:

$$buf = \begin{cases} \text{if} \quad (Next = s) \quad \text{then} \quad buf_x \quad \text{else} \quad r \circ buf_x \end{cases}$$

This relation relates $buf$ with the state of $PS$. The buffer $buf$ contains the sequence of messages received at $X$ but not yet delivered at $Y$. We must prove that $S$ satisfies the requirements of a simulation relation. For the initial states we get: $() = ()$, which is true. It remains to verify that whenever $PS$ can perform a computation step, then this computation step can be simulated by a computation step of $SS$ preserving $S$.

G1 is simulated by S1 and G6 is simulated by S2. The rest of the transitions in $PS$ are simulated by null transitions of $SS$. We must prove, for each transition in $PS$ that changes a variable in $S$, that this transition and the simulating transition in $SS$ preserve the truth of $S$. We omit the details. As an illustration we verify that G1 is simulated by S1. G1 adds one element to buffer $buf_x$. We must verify that $S$ holds regardless of the values of $s$ and $Next$. We have two cases:

(i) $Next = s$. In this case we must verify the following implication:

$$\{buf = buf_x\} \Rightarrow \{buf \circ d = buf_x \circ d\}$$

(ii) $Next \neq s$. In this case we must verify the following implication:

$$\{buf = r \circ buf_x\} \Rightarrow \{buf \circ d = r \circ buf_x \circ d\}$$
The last step in the proof is to verify that $S$ is quiescence preserving, i.e. resting states of $PS$ can only be simulated by resting states of $SS$. We must prove that if $S$ holds between two states $\sigma_{PS}$ and $\sigma_{SS}$ of $PS$ and $SS$ respectively, and $\sigma_{PS}(R_{PS}) = true$, then $\sigma_{SS}(R_{SS}) = true$. We verify this by proving the implication:

$$\{R_{PS} \land S\} \Rightarrow R_{SS}$$

As a matter of fact, the truth of the above implication is not dependent of all the conjunctions in $R_{PS}$. As is shown below, only the conjuncts $buf_x = ()$ and $s = a$ are needed. To prove the implication above we must verify that:

$$\begin{cases} 
buf = \text{if } (s = a) \text{ then } buf_x \text{ else } r \cdot buf_x \\
buf_x = () \\
s = a
\end{cases} \Rightarrow \{buf = ()\}$$

Thus we have proven that $S$ fulfills the requirements of a quiescence preserving simulation relation and can conclude that $(S||MSR||R||MRS)$ implements $SS$.

7. CONCLUSION

We have presented a method for specification and verification of networks of non-deterministic processes that communicate by asynchronous message-passing. The method is based on traces and has the following features:

(i) Operational specification. The specification of a network as a transition system has the form of an abstract program. With this approach, the operational intuition often helps to convey the meaning of a specification.

(ii) Our verification method is based on the operational intuition of simulations. We say that a transition system, $N_1$, implements another transition system, $N_2$, if we can find a simulation relation between the states of $N_1$ and $N_2$. Our verification method captures safety properties and absence of deadlock, and is shown to be complete under certain assumptions. Note the difference from the bisimulation in CCS ([Mi83]) where two processes are considered as equal if there exists a bisimulation between the states of the processes, i.e. in our terminology; the processes implement each other.

We only consider the finite behavior of processes, and can not model divergence or properties related to fairness. Furthermore, we do not distinguish between deadlock and normal termination, both are modelled as resting states of transition systems. However, when proving that the transition system $N_1$ implements the transition system $N_2$, we can view the resting states of $N_2$ as normal termination. If we can find a quiescence preserving simulation relation between the states of $N_1$ and $N_2$, then we can be sure that $N_1$ will not halt in any state other then the states prescribed by $N_2$. An other approach it to split the resting states into two sets: one describing deadlock, the other normal termination. This is the approach in ([NDGO86]), where a system is deadlocked if a
finite set of its components are deadlocked, and totally deadlocked if every component is deadlocked.

8. REFERENCES


