Analyzing Concurrent Systems Using the Concurrency Workbench*

Rance Cleaveland

Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27695-8206
USA
e-mail: rance@csc.ncsu.edu

Abstract. This paper presents a case study illustrating the different analytical facilities provided by the Concurency Workbench, an automatic verification tool for finite-state systems. The system we consider is the Alternating Bit Protocol, a communications protocol designed to ensure error-free data transfer over lossy media. Using the Workbench we investigate several features of the protocol, and we isolate a crucial property (the assumption of which is unstated in the protocol definition) that media must have in order for the protocol to be deadlock-free.

1 Introduction

In recent years researchers have begun investigating the practicality of automating the process of reasoning about finite-state systems. Several tools have been developed [19, 22] to provide support for various approaches to analyzing the correctness such systems, with some tools based on temporal logic model checking [7, 30] and others built around the determination of different behavioral relations between systems [9, 10, 11, 15, 17, 25, 31]. Typically, the tools give users one verification methodology with which to work; the Concurrency Workbench [9, 10, 11] is noteworthy in that it gives users a spectrum of techniques for establishing the correctness of systems. It does so by providing implementations of three "low-level" algorithms that may then be specialized in order to yield routines for checking different correctness criteria. As a result of this philosophy, the tool is flexible and easy to extend.

This paper describes the application of the Concurrency Workbench to the Alternating Bit Protocol [1]. The goal of this exercise is to illustrate how the Workbench in particular, and the verification techniques that it supports in general, may be used to analyze finite-state communicating systems. The remainder of the paper is structured as follows. The next section provides an overview of the Workbench and describes the Alternating Bit Protocol. Section 3 then shows

* Research supported by NSF/DARPA research grant CCR-9014775, NSF research grant CCR-9120995, ONR Young Investigator Award, N00014-92-J-1582, and a NSF Young Investigator Award.

how the protocol may be formalized in the system, while Section 4 describes the results of the different analyses performed on the protocol. The final section concludes with a discussion of work that is underway on the Workbench.

2 Background

This section gives a brief overview of the Concurrency Workbench and introduces the Alternating Bit Protocol. The level of detail provided here is necessarily sketchy, and the interested reader is referred to [1, 9, 10, 11] for more in-depth discussion.

2.1 The Concurrency Workbench

The Concurrency Workbench supports the automated analysis of finite-state systems expressed in Milner's CCS [26]. The tool implements a number of commands for defining networks of processes and analyzing their behavior; it also provides support for three major approaches to verification, which we now describe.

The first such approach is grounded in the theory of process algebra and relies on the use of a *behavioral equivalence* to relate specifications, formulated as finite-state machines, and implementations, also formulated as finite-state machines. The intention is that the specification details the high-level behavior that the system is supposed to exhibit, with the implementation describing the ways in which the behavior is to be realized. In the Workbench, one uses CCS to specify both implementations and specifications in this approach; the CWB then applies the operational semantics of CCS to the systems entered by the user to generate the associated finite-state machines. Verification, then, consists in showing that the implementation and the specification are behaviorally indistinguishable. Different equivalences have been proposed in the literature to capture an appropriate notion of "same behavior" [3, 14, 16, 20, 21, 26], and the Workbench includes algorithms for deciding a number of them. It does so by providing a generic equivalence checker based on the bisimulation equivalence of Milner [26] and Park [28] that can be specialized in various ways to compute other equivalences.

The second major verification technique provided by the Workbench also has roots in process algebra; in this approach, one uses a *behavioral* preorder to relate specifications and implementations. In one preorder-based verification methodology, the specification is viewed as providing the maximum allowable behavior, and an implementation is correct if it is "less" than the specification in the preorder (i.e. if it provides no more behavior than that allowed by the specification). In another approach, specifications are interpreted as providing the minimum allowable behavior, and an implementation is deemed correct if it is larger in the preorder. Different preorders have been proposed as suitable for relating implementations and specifications in this fashion [3, 14, 20, 24, 28, 32].

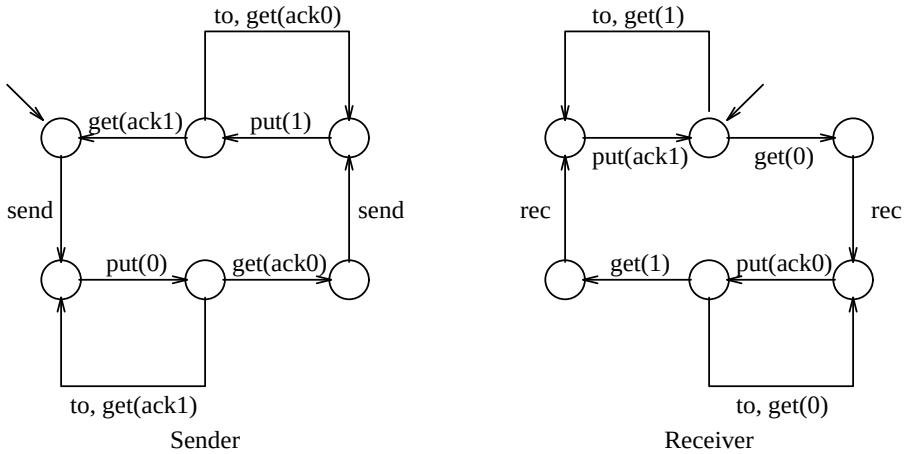
The third approach to verification that the Workbench caters for relies on the use of a modal (temporal) logic as a language for writing specifications. Intuitively, formulas in the logic permit one to make statements about the behavior of a system as it evolves over time; to verify a system, then, one establishes that it satisfies the formulas constituting the specification. In the case of finite-state systems, algorithms that make this determination automatically are called *model checkers*; the Workbench includes a model checker for the *propositional mu-calculus*, which is described in some detail in the paper by Zucker in this volume.

The Workbench also provides other commands for analyzing the operational behavior of agents. One command, for instance, returns all the reachable deadlocked states contained in a system, while others permit users to explore the sequences of transitions available to an agent. The most recent version of the system also includes an interactive simulator for CCS [27].

The Concurrency Workbench consists of approximately 15,000 lines of Standard ML of New Jersey code and is available via anonymous ftp from several sites around the world.

2.2 The Alternating Bit Protocol

The Alternating Bit Protocol [1] was proposed as a method for "achieving reliable full-duplex transmission over half-duplex lines." The protocol is in fact an instance of what today would be called a sliding-window protocol [2], where the send and receive windows are of size one.



"get" and "put" actions correspond to interactions between the protocol entities and the medium. "to" actions represent time-outs.

Fig. 1. The Alternating Bit Protocol.

Figure 1 contains finite-state machines describing the actions to be taken by the sender and receiver in the protocol as described in the original paper. In essence, these work as follows. Both protocol entities maintain single bits, initially 0, that record the "sequence number" of the next message the sender will send and the receiver expects to receive, respectively. When the sender is given a message to transmit to the receiver, it appends the value of its bit, sends this augmented message, and awaits an acknowledgement containing the same bit value from the receiver. If it receives such a response from the receiver, then it increments its bit (mod 2), and the process is repeated for the next message. If the acknowledgement contains the wrong bit value, then the sender retransmits its message and awaits another response from the receiver.

When a message arrives, the receiver compares the "sequence bit" of the message with its bit value. If they match, then the receiver delivers the message, sends an acknowledgement containing the sequence bit of the message, and increments its bit (mod 2). If they do not match, then the receiver sends an acknowledgement containing the erroneous sequence bit and awaits the next message from the sender.

In addition, new versions of the protocol include a time-out facility (to handle lossy media): while waiting for an acknowledgement (message), the sender (receiver) may time-out, in which case it resends the message (acknowledgement) it most recently sent.

3 Formalizing the Protocol in the Workbench

In this section we show how the sender and receiver in the Alternating Bit Protocol may be entered into the Workbench, and we also address the question of formalizing the media over which the protocol entities communicate.

3.1 The Sender and Receiver in CCS

The Workbench provides a version of Milner's Calculus of Communicating Systems (CCS) as the language in which users render their systems. CCS is an example of a process algebra; it provides operators that permit users to build larger systems out of a designated set of atomic actions and other systems.

The atomic actions in CCS correspond intuitively to "communication actions" that systems may engage in. Formally, let Λ be a set of port names; in the Workbench, Λ contains all strings of printing characters beginning with something other than an upper-case letter. Then an action is either:

- an input on port `a`, written as just `a` in the Workbench; or
- an output on port `a`, written as `'a` in the Workbench; or
- an internal action τ, written as `t` in the Workbench.

Given these atomic actions, the following lists the CCS process constructors provided by the Workbench that we will use in this paper.

- a.p, where a is an action and p is a process. This process first engages in a and then behaves like p.
- $p + q$, where p and q are processes. This process may behave like either p or q.
- $p \mid q$. This system consists of p and q running in parallel and executing in an interleaved fashion. If p and q are capable of doing an input and an output (or an output and an input) respectively on the same channel, then they synchronize, producing a τ action, and both p and q change state.
- $p\backslash L$, where $L \subseteq \Lambda$. This process behaves like p except that the ports in L are defined to be local. Thus, other processes are prohibited from interacting with p along these ports.

Recursive and nonterminating processes may also be defined. Let

$$X_1 = p_1, \ldots, X_n = p_n$$

be a sequence of equations (where, in the Workbench, the left-hand side of an equation may be any sequence of printing characters beginning with an uppercase letter). Then in the process

$$p \text{ where } X_1 = p_1 \text{ and } \cdots \text{ and } X_n = p_n \text{ end}$$

p may contain references to the X_i; these references are resolved by replacing X_i by p_i whenever the former is encountered as p executes.

The formal semantics of these operators may be found in [26]; they are given in terms of a relation $p \xrightarrow{a} q$, which may be read as "p performs action a and then behaves like q." Using the semantics, transition graphs may be generated for any CCS term.

```
bi Sender S0 \
    where S0  = send.S0' \
    and   S0' = 'put0.(get_ack0.S1 + get_ack1.S0' + t.S0') \
    and   S1  = send.S1' \
    and   S1' = 'put1.(get_ack1.S0 + get_ack0.S1' + t.S1') \
    end

bi Receiver R0 \
    where R0  = get0.'rec.'put_ack0.R1 \
                + get1.'put_ack1.R0 \
                + t.'put_ack1.R0 \
    and   R1  = get1.'rec.'put_ack1.R0 \
                + get0.'put_ack0.R1 \
                + t.'put_ack0.R1 \
    end
```

Fig. 2. Workbench Definitions for the Sender and Receiver.

We now turn to specifying the sender and receiver of the Alternating Bit Protocol in CCS. Figure 2 contains two Workbench-style definitions for these protocol entities. The `bi` keyword indicates that the identifier to follow is to be bound to the CCS expression that appears next. The \ at the end of some of the lines is a continuation marker indicating that the CCS expression continues on the next line. So the identifier `Sender` is bound to the CCS process

```
S0 \
where S0  = send.S0' \
and   S0' = 'put0.(get_ack0.S1 + get_ack1.S0' + t.S0') \
and   S1  = send.S1' \
and   S1' = 'put1.(get_ack1.S0 + get_ack0.S1' + t.S1') \
end
```

and similarly for `Receiver`. The reader should compare these definitions with those given in Figure 1. Note that time-out transitions here are modeled using internal `t` actions.

3.2 Modeling the Medium

The original definition of the Alternating Bit Protocol places no requirements on the media over which the sender and receiver communicate other than they are to be "half-duplex" lines. In order to analyze the protocol using the Workbench, however, it is necessary to develop a model of the medium in CCS. In this section we in fact present several models of lossy and safe media, since as we will see in the next section the correctness of the protocol in fact depends on the specific model of the medium adopted.

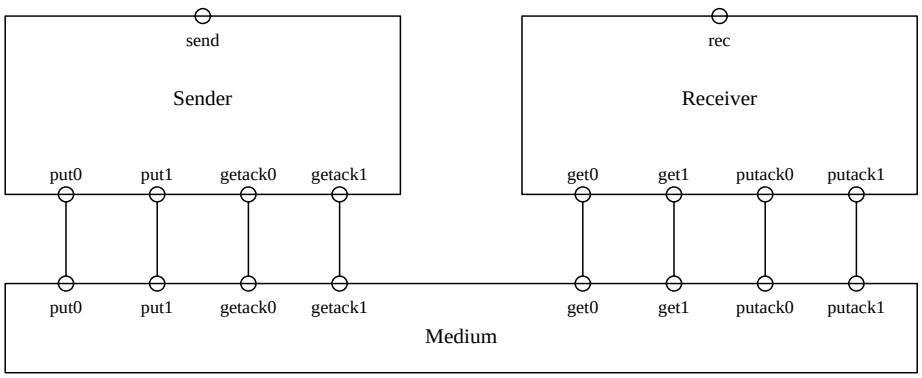
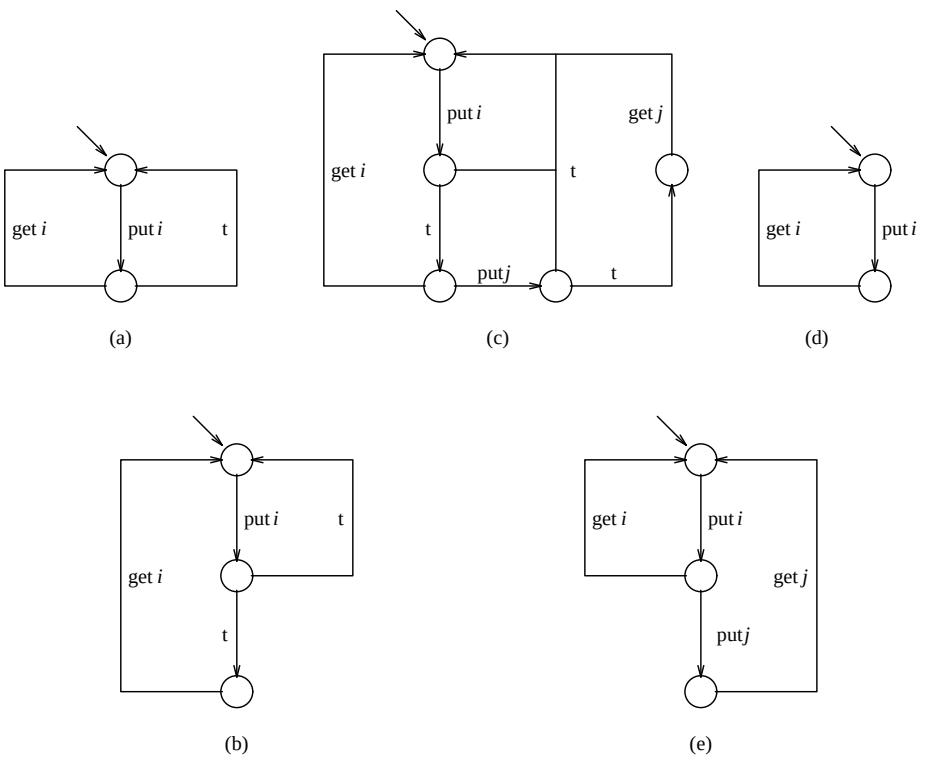


Fig. 3. The Architecture of the Alternating Bit Protocol.

In order to define these media models we must first understand how they are to be used by the protocol entities. We begin by looking at the network

architecture of the Alternating Bit Protocol, which appears in Figure 3. The system consists of three components: a sender, a receiver and a *medium*. The first two were described in the previous section; here we concentrate on the medium, for which we give five models that differ in subtle yet important ways.



In the above, $i, j \in \{\texttt{0}, \texttt{1}, \texttt{ack0}, \texttt{ack1}\}$. Action `t` is the internal action. Media (a), (b) and (c) are lossy, while (d) and (e) are safe.

Fig. 4. Media models for the Alternating Bit Protocol.

Figure 3 and the definitions of the sender and receiver indicate that the medium must be able to respond to inputs on channels `put0`, `put1`, `putack0` and `putack1` and deliver outputs on `get0`, `get1`, `getack0` and `getack1`. Figure 4 presents the finite-state automata representations of the media models we consider. Each is capable of buffering at most one "message", where a message may either be `0`, `1`, `ack0` or `ack1`. The first three are also *faulty*, meaning that as a result of an internal move on the part of the medium, any buffered message may be lost. The first two differ only in the way in which the medium may lose the message; the first may lose a message at any time until the message is retrieved by the receiver, while the second, after deciding that the message is

to be delivered, remains in a state in which the delivery cannot be pre-empted. The third faulty medium, and the second safe medium, also permit *overwriting*; if the medium is full (i.e. buffering one message), and another message arrives, then the buffered message is discarded and the new message takes its place. The Workbench account of each of these media appears in Figure 5.

4 Analyzing the Protocol Using the Workbench

We now turn to an analysis of the Alternating Bit Protocol using the Concurrency Workbench. We focus on using the equivalence checking facility provided by the tool; accordingly, we begin by formalizing the specification of the protocol as a process, and then we compare the protocol (in the context of the different media models discussed in the last section) with the specification to see if they provide the same behavior. We show that with respect to certain media models, the protocol is incorrect, and we use the model-checking and other features of the Workbench to locate the source of the error.

4.1 Specifying the Protocol

Intuitively, the "service" that the Alternating Bit Protocol is to provide is that of a reliable communications channel in which at most one message is outstanding at any given time. This suggests the simple two-state specification given in Figure 6, which requires that the protocol alternately provide `send` and `'rec` actions, with `send` enabled initially.

4.2 Verifying the Protocol

We now illustrate how the Workbench may be used to analyze the correctness of the Alternating Bit Protocol. To begin with, we must show how the three pieces of the protocol—the sender, receiver and medium—are to be "glued together". Figure 7 indicates how this is done for each of the five media. The declaration beginning `bsi` defines a set of ports named `LocalPorts`; intuitively, this set contains all ports that are to be local to the protocol. In this case, all ports used by the media are local. Then five different systems are defined; each consists of the sender and receiver running in parallel with one of the five media, with the media ports declared to be local using the \ operator.

The remainder of this section describes an actual session with the Concurrency Workbench during which the five different "versions" of the Alternating Bit Protocol are analyzed. The Workbench is based on a command loop; the user enters a command, awaits the system's response, and then enters his next command. When the tool is first invoked, the user sees the following.

```
The Edinburgh Concurrency Workbench
  (Version 6.0,  August 15, 1991)

Command:
```

```
bi Lossy-Medium1 LM1 \
    where LM1 = put0.(t.LM1 + 'get0.LM1) \
                + put1.(t.LM1 + 'get1.LM1) \
                + put_ack0.(t.LM1 + 'get_ack0.LM1) \
                + put_ack1.(t.LM1 + 'get_ack1.LM1) \
    end

bi Lossy-Medium2 LM2 \
    where LM2 = put0.(t.LM2 + t.'get0.LM2) \
                + put1.(t.LM2 + t.'get1.LM2) \
                + put_ack0.(t.LM2 + t.'get_ack0.LM2) \
                + put_ack1.(t.LM2 + t.'get_ack1.LM2) \
    end

bi Lossy-Medium3 LM3 \
    where LM3 = put0.LM3p0 \
                + put1.LM3p1 \
                + put_ack0.LM3a0 \
                + put_ack1.LM3a1 \
    and LM3p0 = 'get0.LM3 + t.LM3 + LM3 \
    and LM3p1 = 'get1.LM3 + t.LM3 + LM3 \
    and LM3a0 = 'get_ack0.LM3 + t.LM3 + LM3 \
    and LM3a1 = 'get_ack1.LM3 + t.LM3 + LM3 \
    end

bi Safe-Medium1 SM1 \
    where SM1 = put0.'get0.SM1 \
                + put1.'get1.SM1 \
                + put_ack0.'get_ack0.SM1 \
                + put_ack1.'get_ack1.SM1 \
    end

bi Safe-Medium2 SM2 \
    where SM2 = put0.SM2p0 \
                + put1.SM2p1 \
                + put_ack0.SM2a0 \
                + put_ack1.SM2a1 \
    and SM2p0 = 'get0.SM2 + SM2 \
    and SM2p1 = 'get1.SM2 + SM2 \
    and SM2a0 = 'get_ack0.SM2 + SM2 \
    and SM2a1 = 'get_ack1.SM2 + SM2 \
    end
```

Fig. 5. Workbench Accounts of the Media in Figure 4.

```
bi SERVICE send.'rec.SERVICE
```

Fig. 6. The Service Specification of the Alternating Bit Protocol.

```
bsi LocalPorts put0,put1,get0,get1,put_ack0,put_ack1,get_ack0,get_ack1

bi ABP-LOSSY1
(Sender | Lossy-Medium1 | Receiver)\LocalPorts

bi ABP-LOSSY2
(Sender | Lossy-Medium2 | Receiver)\LocalPorts

bi ABP-LOSSY3
(Sender | Lossy-Medium3 | Receiver)\LocalPorts

bi ABP-SAFE1
(Sender | Safe-Medium1 | Receiver)\LocalPorts

bi ABP-SAFE2
(Sender | Safe-Medium2 | Receiver)\LocalPorts
```

Fig. 7. The Alternating Bit Protocol in CCS.

Our first task is to read in CCS definitions of the processes that have been presented up to this point. We could enter them one at a time at the command line; however, it is also possible to store definitions in files that may then be loaded into the Workbench, and this is what we do here. The file containing the relevant CCS is called "abp." To load it into the the Workbench, we execute the command `if abp`, where `if` is short for "input file".

```
Command: if abp
done.
Command:
```

The Workbench responds with `done` when it has finished processing the file; to see what declarations the tool knows about as a result, we can execute the `pe` command (for "print environment"). The system responds by printing out a list of all sets, processes, etc. that have been defined in the current session.

```
Command: pe

  ABP-LOSSY1 = (Sender | Lossy-Medium1 | Receiver)\LocalPorts
  ABP-LOSSY2 = (Sender | Lossy-Medium2 | Receiver)\LocalPorts
  ABP-LOSSY3 = (Sender | Lossy-Medium3 | Receiver)\LocalPorts
  ABP-SAFE1 = (Sender | Safe-Medium1 | Receiver)\LocalPorts
  ABP-SAFE2 = (Sender | Safe-Medium2 | Receiver)\LocalPorts
...
```

We have elided the output of this command in the interest of saving space.

The Workbench provides commands for computing several different equivalences. In this paper we concentrate on *observational equivalence* [26], which is

designed to be sensitive to the visible actions a system can perform as well as its deadlock behavior while abstracting away from the internal action a system might engage in. Intuitively, for two systems to be observationally equivalent it must be the case that whenever one can engage in a visible action (with some internal computation possibly preceding and following it) and evolve to a particular process, then the other system must also be able to perform this action and evolve to an equivalent process. The interested reader is referred to [26] for a more formal treatment of this equivalence.

In the Workbench the command for computing bisimulation equivalence is `eq`. The result of invoking this command on `SERVICE` and each of the versions of the Alternating Bit Protocol is as follows.

```
Command: eq
Agent: SERVICE
Agent: ABP-LOSSY1
true

Command: eq
Agent: SERVICE
Agent: ABP-LOSSY2
false

Command: eq
Agent: SERVICE
Agent: ABP-LOSSY3
true

Command: eq
Agent: SERVICE
Agent: ABP-SAFE1
false

Command: eq
Agent: SERVICE
Agent: ABP-SAFE2
true
```

So for three media—the first and third lossy media and the second safe medium—the Alternating Bit Protocol is correct, but for the second lossy medium and the first safe medium, it is incorrect! The question that immediately arises concerns the source of this incorrectness. It should be noted that future versions of the Workbench will generate diagnostic information automatically to explain why equivalence checking (and preorder checking) fails [5, 6, 8]; here, however, to determine the source of the error in `ABP-LOSSY2`, we use the method outlined in [11]. We first use the `sort` command to determine which visible actions `ABP-LOSSY2` and `SERVICE` are capable of, since a discrepancy would indicate a reason for inequivalence.

```
Command: sort
Agent: SERVICE
{send,'rec}

Command: sort
Agent: ABP-LOSSY2
{send,'rec}
```

Both systems have the same sort, so this is not the source of the inequivalence. The next check is to determine if the systems are capable of the same sequences of visible actions. If they are not, then this is an explanation of why they are observationally inequivalent. The Workbench command for this is `mayeq`.

```
Command: mayeq
Agent: SERVICE
Agent: ABP-LOSSY2
true
```

The result indicates that `SERVICE` and `ABP-LOSSY2` are in fact capable of the same sequences of visible actions, so this is not the source of the problem.

The next potential source of inequivalence is deadlock; if one of the systems is capable of deadlocking and the other is not, then they would fail to be equivalent. We use a two-step procedure to check this. First, we use the model checker to determine if the systems are capable of deadlocking, and if one is and the other is not, then we invoke a special command provided by the Workbench that prints out all deadlocked states. We use this procedure rather than just invoking the deadlock-finding command directly because the special command is computationally expensive, whereas the detection of the possibility of deadlock using the model checker is very efficient.

In order to use the model checker to determine whether deadlocks exist in a system, we must first devise a formula that is true of processes if and only if they are deadlock-free. Intuitively, a process is *live* if it is capable of some further computation, and it is deadlock-free if every reachable state is live. The property `Live` may be defined as follows, where the command `bpi` enables identifiers to be bound to propositions.

```
Command: bpi Live <->T
```

The proposition `<->T` is a modal one; the modality `<->` indicates that in order for a process to satisfy a formula having it as the outermost constructor, the process must have some transition, labeled by any action, that satisfies the remainder of the formula. In this case, what follows the modality is the formula `T`, or "true"; so a process satisfies `Live` if it is capable of some step of computation.

To define `DeadlockFree`, we first introduce a new proposition constructor using the macro definition facility of the Workbench. The constructor is `AG`, for "always", and its translation into the Workbench's logic is discussed in the papers of Zucker in this volume. The command `bmi` introduces new macros.

```
Command: bmi AG P
Body: max(X. P & [-]X)
```

Here AG is the name of the macro, P is its parameter, and the body, which the system prompts for, is as indicated. Intuitively, a process satisfies AG P if any reachable process satisfies P. We can now formulate our deadlock-freedom formula as follows.

```
Command: bpi DeadlockFree (AG Live)
```

We now check whether SERVICE and ABP-LOSSY are deadlock-free using the command cp, for "check proposition."

```
Command: cp
Agent: SERVICE
Proposition: DeadlockFree
true

Command: cp
Agent: ABP-LOSSY2
Proposition: DeadlockFree
false
```

So ABP-LOSSY2 is capable of deadlock! To determine why, we invoke the command fd, for "find deadlocks".

```
Command: fd
Agent: ABP-LOSSY2
--- send t t t 'rec t t t send t t t t 'rec t t t ---> ((S1'  \
      where S0 = send.S0'  \
        and S0' = 'put0.(get_ack0.S1 + get_ack1.S0' + t.S0')  \
        and S1 = send.S1'  \
        and S1' = 'put1.(get_ack1.S0 + get_ack0.S1' + t.S1')  \
      end
) | ('get_ack1.LM2  \
      where LM2 = put0.(t.LM2 + t.'get0.LM2)  \
                  + put1.(t.LM2 + t.'get1.LM2) \
                  + put_ack0.(t.LM2 + t.'get_ack0.LM2)  \
                  + put_ack1.(t.LM2 + t.'get_ack1.LM2)  \
      end
) | ('put_ack1.R0  \
      where R0 = get0.'rec.'put_ack0.R1  \
                 + get1.'put_ack1.R0 + t.'put_ack1.R0  \
        and R1 = get1.'rec.'put_ack1.R0  \
                 + get0.'put_ack0.R1 + t.'put_ack0.R1  \
      end
))\LocalPorts

...
```

There turn out to be numerous deadlocked states; here we look at one that indicates what the general problem is. In this state, which is reachable via the given sequence of actions, the sender, which is in state `S1'`, wishes to execute `'put1`; the medium, on the other hand, is only willing to execute `'get_ack1`. In other words, we have a collision on the medium; the sender wishes to give something to the medium, which is trying to give something to the sender. The same problem occurs in `ABP-SAFE1`; notice that it cannot occur with any of the other media because they permit contents of the media to be overwritten.

The Alternating Bit Protocol may be altered to "flush" (i.e. receive any outstanding messages) the medium before it sends a message; the resulting altered protocol may be shown correct for all of the five media. The interested reader may find the altered protocol in [11].

5 Conclusions

In this paper we have illustrated some of the analytical capabilities of the Concurrency Workbench, a tool supporting the automatic analysis of finite-state systems, by conducting a case study involving the Alternating Bit Protocol. We showed that the correctness of the protocol in fact hinges on an assumption, unstated in the definition of the protocol, involving the nature of the underlying medium, and we briefly mentioned how the protocol may be changed to remove dependence on this assumption. All the analysis described in the paper was carried out using the Workbench.

The Workbench is being extended in a number of ways, and we briefly mention some of them here. First, some of the analysis routines are being altered to include the automatic generation of diagnostic information when a process fails to meet its specification [5, 6, 8]. Secondly, more efficient verification algorithms have been developed [12, 13], and these are in the process of being implemented in the system. Also, work is underway on a "front-end generator" that will permit users to define the process algebra that they wish to use and then generate a Workbench that supports this process algebra. Finally, we are developing a front end to the system that will enable users to describe their systems, and simulate them, graphically.

Related Work

Other researchers have used the Concurrency Workbench to analyze various communications protocols; here we mention some of them. Parrow [29] used the tool to analyze a simplified version of the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol. Goyer [18] employed the tool to debug the design of the interprocessor communications protocols used in the BHIVE multiprocessor. Finally, Bruns and Anderson [4] studied the Multiprocessor Shared Memory Information Exchange protocol, which is used in control applications in the nuclear power plants; they located a livelock in the protocol (whose existence was already known) and showed how it could be fixed.

References

1. Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links." *Communications of the ACM* 12, n. 5, pp. 260–261, May 1969.
2. Bertsekas, D. and Gallager, R. *Data Networks*. Prentice-Hall, Englewood Cliffs, 1987.
3. Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. "A Theory of Communicating Sequential Processes." *Journal of the ACM*, v. 31, n. 3, July 1984, pp. 560-599.
4. Bruns, G. and Anderson, S. "The Formalization and Analysis of a Communications Protocol." Submitted for publication.
5. Celikkan, U. and Cleaveland, R. "On Computing Diagnostic Information for Preorder Checking." To appear in *Proceedings of the 1992 Workshop on Computer-Aided Verification. Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.
6. Celikkan, U. and Cleaveland, R. "Computing Diagnostic Tests for Incorrect Processes." With U. Celikkan. To appear in *Proceedings of the Twelfth International Symposium on Protocol Specification, Testing and Verification.*
7. Clarke, E.M., Emerson, E., and Sistla, A.P. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications." *ACM Transactions on Programming Languages and Systems*, v. 8, n. 2, pp. 244–263, 1986.
8. Cleaveland, R. "On Automatically Distinguishing Inequivalent Processes." In *Computer-Aided Verification '90*, pp. 463–477. American Mathematical Society, Providence, 1991. Also Lecture Notes in Computer Science 531, Springer-Verlag, Berlin, 1991.
9. Cleaveland, R., Parrow, J. and Steffen, B. "A Semantics-Based Tool for the Verification of Finite-State Systems." In *Proceedings of the Ninth IFIP Symposium on Protocol Specification, Testing and Verification*, June 1989, pp. 287–302. North-Holland, Amsterdam, 1990.
10. Cleaveland, R., Parrow, J. and Steffen, B. "The Concurrency Workbench." In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, pp. 24–37. Lecture Notes in Computer Science 407, Springer-Verlag, Berlin, 1989.
11. Cleaveland, R., Parrow, J. and Steffen, B. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Finite-State Systems." Technical Report 91-26, North Carolina State University, 1991. To appear in *ACM TOPLAS*.
12. Cleaveland, R. and Steffen, B. "Computing Behavioural Relations, Logically." In *Proceedings International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science 510, Springer-Verlag, Berlin, 1991.
13. Cleaveland, R. and Steffen, B. "A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus." In *Proceedings of Computer-Aided Verification '91*, Lecture Notes in Computer Science 575. Springer-Verlag, Berlin, 1992.
14. DeNicola, R. and Hennessy, M.C.B. "Testing Equivalences for Processes." *Theoretical Computer Science*, v. 34, pp. 83–133, 1983.
15. Fernandez, J.-C. *Aldébaran: Une Système de Vérification par Réduction de Processus Communicants*. Ph.D. Thesis, Université de Grenoble, 1988.

16. van Glabbeek, R. and Weijland, P. "Branching Time and Abstraction in Bisimulation Semantics." In *Information Processing 89*, pp. 613–618. North-Holland, Amsterdam, 1989.
17. Godskesen, J.C., Larsen, K.G., and Zeeberg, M. "TAV—Tools for Automatic Verification." Technical Report R89-19, Aalborg University, 1989.
18. Goyer, J.H. *Communications Protocols for the B-HIVE Multicomputer.* Master's Thesis, North Carolina State University, 1991.
19. Har'el, Z. and Kurshan, R.P. "Software for Analytical Development of Communications Protocols." *AT&T Technical Journal*, v. 69, n. 1, pp. 45–59. February, 1990.
20. Hennessy, M.C.B. *Algebraic Theory of Processes.* MIT Press, Boston, 1988.
21. Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.
22. Holzmann, G. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, 1991.
23. Larsen, K.G. "Proof Systems for Hennessy-Milner Logic with Recursion." In *Proceedings of CAAP*, Lecture Notes in Computer Science 299, Springer-Verlag, Berlin, 1988.
24. Larsen, K.G. and Thomsen, B. "Compositional Proofs by Partial Specification of Processes." Report R 87-20, University of Aalborg, July 1987.
25. Malhotra, J., Smolka, S.A., Giacalone, A. and Shapiro, R. "Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems." In *Proceedings of the Workshop on Specification and Verification of Concurrent Systems*, University of Stirling, Scotland, 1988.
26. Milner, R. *Communication and Concurrency.* Prentice Hall 1989.
27. Moller, F. "The Edinburgh Concurrency Workbench (Version 6.0)." Technical Note LFCS-TN-34, University of Edinburgh, 1991.
28. Park, D.M.R. "Concurrency and Automata on Infinite Sequences." In *Proceedings of the* 5^{th} *GI Conference*, Lecture Notes in Computer Science 104. Springer-Verlag, Berlin, 1981.
29. Parrow, J. "Verifying a CSMA/CD-Protocol with CCS." In *Proceeding of the Eighth IFIP Symposium on Protocol Specification, Testing, and Verification*, pp. 373–387. North-Holland, Amsterdam, 1988.
30. Richier, J., Rodriguez, C., Sifakis, J., and Voiron, J.. "Verification in XESAR of the Sliding Window Protocol." In *Proceedings of the Seventh IFIP Symposium on Protocol Specification, Testing, and Verification.* North-Holland, Amsterdam, 1987.
31. Roy, V. and de Simone, R. "Auto/Autograph." In *Computer-Aided Verification '90*, pp. 477–491. American Mathematical Society, Providence, 1991.
32. Walker, D.J. "Bisimulation Equivalence and Divergence in CCS." In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, 1988, pp. 186-192. Computer Society Press, Los Alamitos, 1988.