

BACKWARD SYMBOLIC EXECUTION of PROTOCOLS

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A traditional method to validate protocols by state space exploration is to use forward symbolic execution. One of the main problems of this approach is that to find all *undesirable* system states one has to generate all reachable states and evaluate all *desirable* system states as well. The paper discusses an alternative search strategy based on backward symbolic execution. This time we start with a state that we know to be undesirable and execute the protocol backwards, evaluating only undesirable states in an effort to show that they are unreachable.

Keywords:

protocol validation, validation algebra, exhaustive searching, state space explosion.

Proc. 4th IFIP WG6.1 Int. Conf. on Protocol Specification, Testing, and Verification, Skytop, Pa., USA, 1984. North-Holland Publ. pp. 19–30.

1. Introduction

Intuitively one of the simplest techniques for a fully automated validation of communication protocols is to perform an exhaustive search of all reachable system states [1,2]. The number of reachable states, though, can rise rapidly with the number of processes, variables, and buffer slots used, which sets an upper limit to the size of protocols that can be analyzed effectively. The problem to find undesirable states among the reachable states by *forward* execution, however, can be translated into the problem of determining the reachability of the undesirable states by *backward* execution: from the undesirable state back to the initial system state. This principle was first described by Danthine and Bremer in [3]. Their method was to (1) consider each of the communicating machines in isolation and identify the potentially undesirable partial system states. Then (2) to construct all paths per machine that could lead into that state by a backward search. And, finally, (3) to evaluate the compatibility of all combinations of these paths by a forward symbolic execution. In [3] the third and last step was still a *forward* execution, starting in the initial state, which again meant generating desirable system states (redundant in the analysis) in an effort to find the undesirable ones. We will examine here whether, by combining step (2) and (3) we could avoid these problems. We will start the symbolic execution in an undesirable state and start executing backwards immediately, generating only undesirable states, up to the point where no more progress can be made. If at that point we are in the initial system state, we have shown the undesirable state to be reachable. If we are not in the initial state we have shown the undesirable state to be unreachable along the path we tried.

If indeed the number of undesirable system states to be considered in a backward search is smaller than the number of reachable system states to be considered in a forward search, the backward execution method is bound to be faster. Whether or not this is really the case, however, is an unproven proposition.

2. Example

Consider the following message exchange between two processes described here in a simple protocol specification meta-language, derived from Hoare's notation for CSP [4]:

```
queue A = { };
queue B = { };

process A
{   if
    :: A?m1 → B!m2
    :: A?m1 → B!m3
  fi
}

process B
{   A!m1; B?m3
}
```

We have defined two processes named *A* and *B*. The processes communicate by exchanging messages via two queues, both initialized to be empty. For convenience, the queue read by *A* is named *A*, and similarly the queue read by *B* is named *B*.

The notation *target!msg* specifies the sending of a message *msg* to a queue named *target*. Similarly, *source?msg* indicates the receipt of *msg* from queue *source*. The control flow structure:

```
if
:: option1
:: option2
fi
```

is used to specify a *non-deterministic* selection of the execution sequences labeled *option1*, and *option2*. To execute the if structure, one executable option from the range is chosen at random and executed. Usually, the first statement in each option enforces a selection. For example, the receive statement *source?msg* is only *executable* if *msg* is indeed the first message in queue *source*. So we can use a structure:

```
if
:: source?one → remainder1
:: source?two → remainder2
fi
```

as a case switch on the message that we expect to find in queue *source*. If none of the options are executable the execution of the process is blocked.

The behavior of process *A* is non-deterministic. It can respond in two different ways to the reception of the message *m1* from *B*. Clearly, the first option in *A* leads to an unspecified reception in *B*, while the other option leads to a valid execution sequence. The problem is now to detect the possibility of an unspecified reception without analyzing also the valid sequence.

Unspecified Receptions

The first step is to identify the process states in which the unspecified receptions could take place. There are two such states in the above example: one in process *A*, at the start of the non-deterministic selection, and one in process *B*, directly after the first send statement. The unspecified reception in *A* would take place if anything other than an *m1* from process *B* could be in the first slot of *A*'s message queue. The unspecified reception in *B* could take place if anything other than a message *m3* from *A* could arrive.

The first potential protocol error can quickly be ruled out, even by a static check of the protocol specification: *B* will never send anything else than an *m1* to *A*, and no other processes that communicate with *A* are specified.

The second potential error looks more promising. We have to find the states in process *A* that send messages to *B* that differ from *m3*. For each such state found we have to attempt a "backward symbolic

execution" up to the initial system state to see whether or not the combination of process states found corresponds to a reachable system state. The closer we can get to the initial system state, by making more and more assumptions about the behavior of processes other than process *B*, the more time we have to spend in the analysis, but also the more likely the validity of the execution sequence will be.

In the above example, process *A* sends a message different from *m3* in just a single place. From that point on we execute process *A* and process *B* backward, aborting the attempt at any point where we can no longer do so without inconsistencies. We have no problem for the example protocol. Executing *A* backwards we find that it requires the reception of a message *m1* from *B* to reach the initial state. Executing *B* backwards we find that it can generate that message, and by doing so will also reach the initial state. Thus we can demonstrate the feasibility of an invalid execution sequence, without ever seeing the valid one.

Deadlocks

A second type of protocol error that can be detected with a backward search is global system deadlock: a state in which all processes are awaiting new messages and all message queues are empty. There is just one wait state in each process of the example above. The product of the number of wait states in each process is the number of system states that has to be checked for reachability: in this case precisely one state. We can quickly detect that the global deadlock is unfeasible since *B* cannot enter its wait state without sending the message that will release *A* from its wait state. In a more realistic example we may expect up to 10^2 wait states per process and thus 10^4 system states to consider.

3. Reversing Specifications

In more practical applications a protocol specification will not only be larger than the example we considered above, it will also contain more than just send and receive statements. Let us consider a language that contains, apart from definitions and declarations, the following indivisible statements:

qname!mname:

appends message *mname* to the tail of queue *qname*.

qname?mname:

is only executable if message *mname* is currently at the head of the queue addressed. It will delete the message from the queue.

v1 = v2,

where *v1* is a variable and *v2* is either a variable or a constant: assigns the value of *v2* to *v1*.

v++:

adds one to the value of variable *v*.

v— :

subtracts one from the value of variable *v*.

(v1 R v2),

where *R* is one of \equiv , \neq , \geq , \leq , $<$, or $>$: is executable if the two operands (variables or constants) are in relation *R*.

As an example, consider the following specification. The example illustrates the use of protocol variables for two processes executing in a shared memory system.

queue TOA = { msg, msg };

queue TOB = { msg, msg };

queue A = { };

queue B = { };

variable N = 0;

variable M = 0;

```
process A
{
  do
    :: TOB?msg →
      do
        :: (N ≡ 2) → skip
        :: (N ≠ 2) → break
      od;
      N++;
      B!msg
    :: A?msg → M—
  od
}

process B
{
  do
    :: TOA?msg →
      do
        :: (M ≡ 2) → skip
        :: (M ≠ 2) → break
      od;
      M++;
      A!msg
    :: B?msg → N—
  od
}
```

There are again two processes, *A* and *B*, sharing access to two queues, also named *A* and *B*. Process *A* also has access to a private queue named *TOB* with messages that it wants to transfer to process *B*, and process *B* has access to a similar queue named *TOA*. There are two shared variables, *N* and *M*. Variable *N* is incremented by process *A* whenever a message is appended to queue *B*, and decremented by process *B* whenever a message is deleted. Since *N* is initialized to 0, the value of *N* gives an upper bound for the size of queue *B*. Similarly, the value of *M* gives an upper bound for the size of queue *A*.

Do Loops

We have used one new control flow structure here: the *do loop*. The syntax of the do loop is the same as the syntax for the selection structure discussed in the first example. The do loop however will be executed repeatedly until a *break* statement is encountered. In the example we have also used a *skip*, which is equivalent to a null statement.

The purpose of this protocol is to force each process to delay the transfer of messages when the queue read by the other process holds two messages. The inner do-loop of each process can only be terminated by executing the *break* statement, which is conditional on the size of the target queue.

We want to verify whether, starting in the initial state with:

State [S1]:
 $N \equiv 0, M \equiv 0,$
 $TOB = \{ msg, msg \},$
 $TOA = \{ msg, msg \},$ and
 $A = B = \{ \},$

we can reach the deadlock state with:

State [S2]:

$N \equiv 2, M \equiv 2,$
 $TOA = TOB = \{ \},$
 $A = \{ msg, msg \},$ and
 $B = \{ msg, msg \}.$

We want to verify this without looking at valid execution sequences, i.e. sequences that do not lead into an error state. Now, to perform the backward symbolic execution we can simply *reverse* the protocol specification. The unwanted deadlock state will become the new initial state and the old initial state will become an undesirable end state. But, the reversal process can be carried even further. We can pretend that the backward execution is a *fake* forward execution by replacing all send statements by receive statements, replacing conditions by assignments, replacing increments by decrements, and reversing the control flow direction.

Motivation

Consider the following sequence of message exchanges on an imaginary queue Q :

$Q!x; Q!y; Q?x; Q?y$

Executing forwards we must verify that the messages x and y are retrieved from the queue in the same order as they were appended. Executing backwards, starting with the last operation, we start by *claiming* that a message y was appended at some earlier time, and then proceed by asserting that message x was appended before that. Our backward search is facilitated if we consider the *claim* on the send sequence to be the contents of a second pseudo queue \bar{Q} . Receive statements on Q then become send statements to queue \bar{Q} , and similarly, send statements on Q translate into receive statements on \bar{Q} . So, if we read the above sequence backwards and replace each statement by its opposite we obtain:

$\bar{Q}!y; \bar{Q}!x; \bar{Q}?y; \bar{Q}?x$

and maintain the same *fifo* queue discipline as in forward symbolic executions.

Next, let us consider conditions and assignments. If we back up across a condition

$(N \equiv 5)$

we know that we have been executing a path that required the variable N to have the value 5. In other words, we *assert* that N did in fact have the value 5 before we continue executing backwards. But, instead of asserting that an assignment has taken place at some time earlier, we can also *assign* the value 5 to a pseudo variable \bar{N} while executing backwards.

$(N \equiv 5)$ translates into $\bar{N} = 5;$

Similarly, a real assignment, such as

$N = 4;$

tells us that variable N really had the value 4 in the path that we just executed backwards. If this conflicts with the values we had asserted before we know that the path is unexecutable. Therefore, the assignment really becomes a condition when reversed: we can back up across the above assignment only if N really had the value 4. Therefore, we have the elegant property that

$N = 4;$ translates into $(\bar{N} \equiv 4)$

We can similarly convince ourselves that increments translate into decrements and vice versa. Of course, not everything is quite as simple. What happens, for instance, to conditions of the type:

$(N \geq 5)$ or $(N \neq 2)$

states\actions	$\overline{TOA!msg}$	$\overline{M=2}$	$(\overline{M=1}2)$	$\overline{N-}$	$\overline{A?msg}$	$\overline{B!msg}$	$\overline{N++}$
0			1				
1	2	1					
2					4		3
3						2	
4				0			

By showing that we can reach the endstate [S2] from state [S1], taking the residual buffer contents of the deadlock state as our initial buffer contents, we can prove the reachability of the deadlock. In this case the proof is trivial: we can cycle twice through a sequence on states (0,0), (1,1), (2,2), (4,4) to reach the unwanted state. Of course, it will not always be the case that the shortest path to the unwanted end state will be discovered first, especially with an automated analyzer. In this case there is a redundant path via state (3,3). It is therefore likely that an automated proof may still require some amount of redundant searching through the set of states that is claimed to be unreachable.

4. Limitations

A first drawback of the backward search strategy, when applied to the detection of *unspecified receptions* is that for every "valid" message that can be received that are very many "invalid" ones that may come from the other processes. In the first steps of a backward search we then have to consider many different assumptions about matching progress states in other processes. The backward method can therefore only prove superior to the forward method if the number of trails to follow very quickly converges to the real invalid execution sequences, that is, within a few executions steps backwards.

Another problem that restricts the applicability of backward search techniques is that one may have to make many additional assumptions about execution sequences to bound the length of the backward searches. Note that, if we would allow for arbitrary initial message queue content to remain as "left-overs" of a backward search after the initial state is reached, the length of the sequences cannot be bounded at all. For the reachability of the target end-state it would be sufficient to assume an initial queue contents that leads the target process into that state, even without further actions from the other processes.

The range of errors to be found will also be smaller than is possible for a forward search method. Note that in the first example we have checked for the possibility that process *B* sent any message other than *m1* to process *A* and found that it couldn't. What we did not check is whether process *B* will ever send message *m1*, for if it would fail to do so, process *A* might hang. There seems to be no easy way to verify this by backward searches.

Better or Worse ?

Since the number of unreachable states and their connectivity will differ for each protocol considered, the performance of the backward search method can only be assessed by implementing it. A simple backward symbolic execution program, restricted to deadlock analyses, was therefore written, and tested on a number of small examples. Protocol specifications written in the meta-language used here are translated into either forward or backward transition tables by a 'protocol compiler,' similar to the one written for the Pandora system [5]. The tables are then analyzed by a simple symbolic execution program that uses some of the reductions that can be motivated with the validation algebra described in [6]. Unfortunately, the first results do not give us much reason to believe that a major improvement in search times can be realized.

As an example, the deadlocks in the first example protocol listed in the appendix are traced by forward symbolic execution in 4.4 seconds of CPU time (on a VAX-750 running a UNIX† version 8 operating system). In the backward search 47 different potential error states have to be inspected, and the same errors are traced in 3.9 seconds of CPU time. The backward execution that demonstrates the reachability of the one deadlock among the 46 non-deadlocks alone, however, completes in 0.3 seconds. The second example protocol from the appendix is analyzed by a forward execution in just 0.3 sec., while the backward search on 5 potential errors requires 1.1 sec. Especially in cases where the number of potential error states is large, the deadlock analysis by an exhaustive forward symbolic execution can be considerably faster than the

† UNIX is a trademark of AT&T Bell Laboratories.

combined search times for backward symbolic executions. Therefore, to make the backward search method faster than the forward search method one would at least need a better way to identify potential error states. So far, no such method has been developed.

As yet, there is insufficient data to either confirm or reject the unproven proposition from the end of section 1, but unless we can improve our method for identifying the potential error states we must conclude that it is false.

Acknowledgements

I thank Doug McIlroy, Rob Pike and Tom Cargill for their careful reading of earlier drafts of this paper. A first version of a program that translates program-form specifications into transition tables was written by B.D. Yeu as part of a lab assignment for his Master's Thesis at the Delft University of Technology.

References

- [1] Brand, D., and Joyner, W.H. Jr., Verification of protocols using symbolic execution. *Computer Networks*, Vol. 2 (1978), pp. 351–360.
- [2] Zafiropulo, P., West, C.H., Rudin, H., Cowan, D.D., and Brand, D., Toward analyzing and synthesizing protocols, *IEEE Trans. Commun. COM-28*, No. 4, (1980), pp. 651–661.
- [3] Danthine, A., and Bremer, J., Modeling and verification of end-to-end transport protocols. *Computer Networks*, Vol. 2 (1978), pp. 381–395.
- [4] Hoare, C.A.R., Communicating sequential processes, *Comm. ACM*, Vol. 21, No. 8, (August 1978), pp. 666–677.
- [5] Holzmann, G.J., and Beukers R.A., The Pandora protocol development system, *Proc. Third Int. Workshop on Protocol Specification, Testing and Verification*, Ruschlikon, Sw., IFIP 1983, North-Holland Publ., Amsterdam, (1983), pp. 357–368.
- [6] Holzmann, G.J., A theory for protocol validation, *IEEE Trans. on Computers*, Vol. C-31, No. 8, (August 1982), pp. 730–738.

APPENDIX

Below we give the listings of the example protocols discussed in section 4. The first protocol is a simple alternating bit protocol, extended with a naive setup procedure that contains a deadlock. The analysis time for forward and backward symbolic executions is roughly the same (4 CPU seconds).

```
queue sender = { };
queue channel = { };
queue receiver = { };

proc sender
{ variable sbit = 1;      /* last sequence number sent   */
  variable rbit = 0;     /* last sequence number received */

  if
  :: sender?setup →
    channel!okay
  :: channel!setup →
    if
    :: sender?okay
    :: sender?setup →
      channel?okay /* deadlocks here */
    fi
  fi;
do
  :: channel!msg(sbit);
  do
    :: sender?ack(rbit) →
      if
      :: (rbit ≡ sbit) →
        sbit + % 2; /* increment mod 2 */
        break
      :: (rbit ≠ sbit) →
        channel!msg(sbit)
      fi
    od
  od
}
```

```
proc receiver
{ variable ebit = 1;      /* sequence number expected */
  variable rbit = 0;      /* sequence number received */

  if
  :: receiver?setup →
    channel!r_okay
  :: channel!r_setup →
    if
    :: receiver?okay
    :: receiver?setup →
      receiver?okay /* deadlocks here */
    fi
  fi;
do
  :: receiver?msg(rbit) →
    channel!ack(rbit);
  if
  :: (rbit ≡ ebit) →
    ebit +% 2 /* increment mod 2 */
  :: (rbit ≠ ebit) →
    skip
  fi
od
}

proc channel
{ variable tbit = 0;      /* sequence number passed on */

do
  :: channel?setup → receiver!setup
  :: channel?okay → receiver!okay; break
  :: channel?r_setup → sender!setup
  :: channel?r_okay → sender!okay; break
od;
do
  :: channel?msg(tbit) →
    receiver!msg(tbit)
  :: channel?ack(tbit) →
    if
    :: skip /* error free transfer */
    :: tbit +% 2 /* introduce error */
    fi;
    sender!ack(tbit)
od
}
```

The second example requires more time for a backward execution (1.1 sec.) than for a forward exhaustive search (0.3 sec.). The reason is that the passive loop in the first process translates into an active loop in the reversed specification, generating arbitrarily large claims on the queue initializations that may lead into the deadlock. The claim is only restricted by the queue sizes which can be specified as an analysis parameter.

```
queue sender = { };
queue receiver = { };

proc sender
{
  receiver!msg1;
  do
    :: sender?ack1 → break
    :: sender?ack0 → skip
  od;
  if
    :: receiver!msg0
    :: skip          /* deletion error causes deadlock */
  fi;
  do
    :: sender?ack0 → break
    :: sender?ack1 → skip
  od
}

proc receiver
{
  do
    :: receiver?msg1 → sender!ack1
    :: receiver?msg0 → sender!ack0; break
  od
}
```