Automated Protocol Validation in *Argos,* assertion proving and scatter searching

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Argos is a validation language for data communication protocols. To validate a protocol a model in *Argos* is constructed consisting of a control flow specification and a formal description of the correctness requirements. This model can be compiled into a minimized lower level description that is based on a formal model of communicating finite state machines. An automated protocol validator *trace* uses these minimized descriptions to perform a partial symbolic execution of the protocol to establish its correctness for the given requirements.

1. Introduction

In the last few years some experience has been gained with the capabilities and the restrictions of automated protocol validators [2,3,4,8,10,11,12,13]. The first validation methods required considerable effort from the user to translate an abstract protocol specification into the formal code used in the validation process. The analyses were often run in batch mode and the interpretation of the results again required considerable human ingenuity. It was soon found that the time and space complexity of even the best known exhaustive search techniques prohibits their application to larger protocols. Similarly, an apparent conflict between the need for *general* validation tools, applicable to a wide range of protocols, and the need to test protocols on more *specific* protocol−dependent properties prohibited the development of tools that could be used effectively in a design environment.

We will describe a protocol validation methodology that overcomes most of these problems. A protocol is specified in a high level language *Argos*, which is compiled into the code accepted by the analyzer. The output of the analyzer is presented in interpreted form, as execution histories that either violate user−specified *specific* requirements or *general* correctness criteria such as completeness and absence of deadlock.

The analyzer was written in C and is portable across Unix† systems with virtual memory. In exhaustive searches the new validator can be several orders of magnitude faster than the earlier tools that were non−portable and required custom−coded input. For protocols that are beyond the scope of exhaustive searches the validator can perform *scatter* searches: fast scans over the state space in an effort to localize design errors in seconds where a full search would take days.

2. The Problem

Consider a protocol for two processes, each having one hundred states and each accessing five local variables. Message buffers are restricted to five slots each, and the effective range of the local variables is assumed to be limited to ten values. The number of distinct messages exchanged is ten.

_____

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

Intuitively the simplest way to analyze the working of this protocol is to perform an exhaustive search of all reachable system states. Let us consider the worst case. There are $10^{5.2}$ possible states of the protocol variables. Each process can be in one of 10^2 different states, so two processes can generate 10^4 states. Finally, each buffer can hold between zero and five messages, where each message can be one out of ten possible messages. The total number of system states in the worst case then is:

$$10^{10} \cdot 10^4 \cdot \left[\sum_{i=0}^{5} 10^i \right]^2$$

or in the order of 10^{24} different states. If we assume, quite unrealistically, that each state can be encoded in 1 byte of memory and can be analyzed in 10^{-6} sec of CPU time we would still need a machine with at least 10^{15} times as much memory as currently available, and would need roughly 10^{11} *years* of CPU time to perform the analysis. By any standard, this can be called an inadequate strategy. To improve it we will either have to abstract from protocol specifics or restrict the scope of the analysis.

One method to abstract from protocol specifics is to use *assertions* in a formal logic, such as Floyd–Hoare logic or temporal logic. We then annotate the specification with suppositions that can establish the observance of the protocol requirements in which we are interested. The key phrase here is clearly 'in which we are interested.' It allows us to abstract from the details in which we are not interested. To automate this approach, however, is a challenging but open problem.

Another method is to construct a partial model of the protocol, leaving out as many details as we can get away with, and then analyze the model instead of the original protocol. A few models have proven to be useful for these experiments. Well known are the Petri Net model and its derivatives, and the Finite State Machine model. There is, however, an unfortunate tradeoff between the descriptive clarity and flexibility of a model on the one hand and its analytical power on the other hand.

Traditionally, this tradeoff has been interpreted to mean that we can either choose a restrictive model that allows us to validate trivial properties with polynomial time algorithms, or a more powerful model that would allow us to validate more interesting properties with nonpolynomial time algorithms.

This interpretation is based on the assumption that the protocol to be validated is error free. If the protocol has bugs, the assumption is further that the validation should produce an exhaustive listing of *all* the errors present. This leads to a validation strategy where one tries to establish *conformance* to the correctness criteria by exhaustive analysis. If exhaustive analysis within the finite amount of time that we are willing to spend on it becomes infeasible, the strategy fails to produce an answer on the correctness of the protocol.

In practice, and especially in a design process, however, neither of the above assumptions is true. If we do assume that the protocol submitted to a validator has errors and that a designer is interested in seeing any nonempty subset these we can improve the validation strategy substantially. The validator can scan the state space with a heuristic searching algorithm in an effort to find a typical *violation* of the requirements in as short a time as possible. The objective of such a partial analysis, or *scattersearch*, is then to establish the presence rather than the absence of errors. If it fails to locate an error it fails to produce an answer on the correctness of the protocol.

For a protocol that generates a state space of 10^8 reachable states we would need in the order of 10^6 seconds, or at least 12 days of CPU time for an exhaustive analysis. In the design phase, knowing or assuming that our first protocol design has errors, it is unlikely that we would be willing to spend more than a few minutes, say 10^3 seconds, on an analysis.

- The most commonly used *breadth first* search method [4,8,12,13] would generate the top 10^{-3} % of the state space tree, which is unlikely to contain errors or a single complete execution sequence.
- The *depth first* search method, used in [10,11], would generate the first 10^{-3} % of all complete execution histories. The detection of an error sequence among the first few generated is still unpredictable since we have no control over the choice of the sequences that are tested.
- The *scatter searching* method, described below, would attempt to select those 10^{-3} % of all execution histories that are most likely to lead to errors. Unlike in the *depth first* method the sequences to be analyzed would be selected from different parts of the tree and not be taken systematically from the one part of the tree that happens to be generated first.

Scatter searching, then, can be a worth–while addition to our range of tools for testing protocol designs. As such, however, it is a bug–finding tool, not a 'validation' tool in the true sense of the word.

It is, of course, important that we be able to verify the reliability of partial searches. We will return to this at the end of section 5, after a more detailed overview of the various searching strategies and a discussion of the different variants of scatter searching that have been implemented.

In the next two sections we will first describe a general language that allows for the specification of both protocols and of protocol correctness requirements. We then describe the extended finite state machine model that can be used for analyzing protocols, and we discuss a compiler that can translate from the language to the machines. The compiler performs static checks for the syntactical correctness and completeness of the specification. The code it produces is targeted to efficient symbolic execution, for instance by precomputing state information that is relevant in the analysis.

In protocol analysis by symbolic execution it is hardly ever necessary to exhaustively search all possible executions to perform a complete analysis. We show that only a subset of the number of feasible process interleavings is relevant to the analysis. Then we describe a modified depth first search algorithm that is used for both complete searches through the state space and for heuristic searches.

3. A Protocol Validation Language

Argos is a simple nondeterministic guarded command language [7,9] that can be used for protocol specification and protocol analysis. Perhaps the most important feature of the language is that there is no difference between a condition and a statement. A condition can appear wherever a statement can appear and vice versa. A statement is said to be *executable* if it evaluates to *true* when it is interpreted as a condition.

The behavior of a distributed system is described by a collection of *process* specifications, where each *process* is an asynchronous agent that can perform *internal* actions and/or synchronize with its environment by performing *external* actions. A process can wait for an event to happen by simply waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

    while (a ≠ b)
            skip;

we can write in *Argos*:

    (a ≠ b);

The boolean test is equivalent to a statement, but it is only executable when the test will return *true*. When it happens to be false execution blocks until it becomes true. To specify control–flow *Argos* has the standard set of structuring tools: case selection, repetition, function call, and macros. In the above case, for instance, we could write:

    if
    :: (a ≠ b) → option1
    :: (a == b) → option2
    fi;

to use the relative values of a and b to choose between two options. Since $(a \neq b)$ is an internal action it is usually embedded in these larger control flow structures.

Internal actions can be boolean tests of or assignments to local variables. External actions are used to communicate messages and values between processes.

The assignment

    $v = e$

where v is a variable and e an expression, is an internal action which always returns *true* and is therefore always executable†. On the other hand, the boolean test

_____

† Note that the assignment does not return the value of expression e.

$$(e)$$

where e is an expression without side effects, is an internal action which returns *false* when expression e returns zero and is therefore only executable if expression e evaluates to a nonzero value. Other types of internal actions are a *jump*, a *call* to a procedure, and a *return* from a procedure. These three actions are always executable and have no side effects. To guarantee this, procedures cannot return values.

The send statement

$$c!m(e)$$

specifies an external action that returns *true* when channel c can accept message m, i.e. whenever channel c is nonfull. When executed, expression e is evaluated and the value returned is attached to the message m. Similarly, the receive statement

$$c?m(v)$$

specifies an external action that returns *true* when channel c can deliver message m, i.e. whenever channel c is nonempty and the oldest message in c is m. When executed, the value attached to message m, if any, is assigned to variable v. As with the *send* statement, the specification of a value transfer is optional. The shorthand forms for sending and receiving are $c!m$ and $c?m$.

Control flow is specified by using *concatenation*, *case selection*, *repetition*, and *jumps*.

The statement separators for specifying concatenations are the traditional semicolon ';' and the arrow '→', which may be used interchangeably. Case selection is specified in an if statement with one or more options:

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

The first statement of an option determines whether the option is executable. Such an option *guard* can be any legal statement: a boolean test, and assignment, a procedure call, a send or a receive, a jump or even another control flow specifier. The repetitive construct

```
do
:: option1
:: option2
...
od
```

also specifies a selection from a list of options, but this time the construct is repeatedly executed until either an explicit *break* statement is encountered or a *jump* transfers control to another part of the program. Note that the repetitive construct is not implicitly terminated when all guards are false, unlike in CSP [9].

The following example specifies a simple filter that receives messages from channel *in* and divides them into two streams named *large* and *small* depending on the values attached.

```
#define N    128
#define size  16

channel in[size], large[size], small[size];

proc split
{       var cargo;

        do
        :: in?mesg(cargo) →
                if
                :: (cargo ≥ N) → large!mesg(cargo)
                :: (cargo <  N) → small!mesg(cargo)
                fi
        od

}
```

A process that merges the two streams back into one, most likely in a different order, and writes it back into channel *in* could be specified as follows.

```
proc merge
{       var cargo;

        do
        ::      if
                :: large?mesg(cargo)
                :: small?mesg(cargo)
                fi;
                in!mesg(cargo)
        od
}
```

So, with an initial contents for channel *in*, specified as

```
channel in[size] = { mesg(256), mesg(13), mesg(4555) };
```

the split and merge filters could busily perform their task forever on. To break a loop on a negative input it is valid, though redundant, to write

```
        do
        :: in?mesg(cargo) →
                if
                :: (cargo <  0) → break
                :: (cargo <  0) → goto out
                :: (cargo ≥ 0) → skip
                fi
        od;
out:    skip
```

In this case the *break* and the *goto* have precisely the same effect and one of the two corresponding options could be deleted. The last option in the if statement above is necessary to avoid the other two options from blocking the control flow when *cargo* is nonnegative.

There is no restriction on the type of statements that can appear as guards in options. If more than one guard turns out to be executable one of them will be selected nondeterministically. If all guards are false the process will block until at least one of them becomes true. In a simple concatenation, therefore, it is valid to write:

in?mesg(cargo); cargo = N; (cargo ≠ N) → larger!mesg(cargo)

Execution can block at three different places in this sequence: at the two external actions, and at the boolean test

(cargo ≠ N) .

In this case the executing process is certain to hang in that test as a result of the assignment that precedes it. The variable is necessarily local, so the execution of no other process could ever reset *cargo*'s value.

Assertions

By default the protocol analyzer that will be discussed in more detail in a later section will check a protocol for the observance of general correctness requirements such as absence of deadlock, and completeness. Assertions are used to test for more specific aspects of a protocol's behavior, The assertions specify global behavior in terms of external actions. For example, the specification

```
assert
{       do
        :: large!mesg; small!mesg
        od
}
```

is a requirement on the order in which messages of the type *mesg* are sent to the two channels *large* and *small*. The assertion is that in each execution sequence a message on channel *large* must precede a message on channel *small*, and that these two actions will always be executed in alternation.
The main restriction to assertion specifications is that they can only refer to external actions, i.e. sends and receives. The control flow constructs, however, are the same as those for process specifications: concatenations, selections, iterations, jumps, procedure calls, and macros. The scope of the assertion, that is the set of external actions that is traced to verify or to violate it, is implicitly defined by the set of external actions specified within the assertion. If an external action occurs at least once in an assertion body all its occurrences in an execution of the protocol are required to comply with it. The appendix gives an example of the use of assertions in the validation process.

Implementation

The version of the validation language that has been implemented for the protocol analyzer has a number of restrictions, some of which were made to simplify the compiler and some to simplify the analyzer. Perhaps the main restriction to the language as implemented is that it requires all processes and all channel names for a protocol to be defined at compile time: there is no facility for the dynamic creation of processes or channels. Another main restriction, that was made to alleviate the memory requirements for the protocol analyzer, is that the specification language has only one type of data element: the short integer. There are no arrays or record structures. Further, though the language allows for procedures to call other procedures, it rules out direct or indirect recursive calls, and provides no means for procedures to return values to a caller.

4. Finite State Machines

Having a protocol validation language, as described above, it is relatively straightforward to develop an interpreter to perform a simple exhaustive search for protocol errors. As indicated in section 2, however, that strategy inherently limits the class of protocols that can effectively be analyzed to a small subset of what is feasible. The alternative we will explore is to translate the description into a form that is targeted at efficient state space searching methods. In this section a formalism for finite state machines is defined that is used to optimize state space searches.

A finite state machine is usually defined as an abstract demon that can accept input signals, generate output signals and change its inner state in accordance with some predefined plan. In the definitions given below, we will extend this basic model with variables and with channels that map the output signals of one machine upon the input signals of another. We will also replace the conventional notion of the finite state machine as an automaton that can only perform input, output and state transitions with a more general

notion of an automaton performing internal or external *actions* that may or may not have a *side effect* on its environment.

A protocol, then, is a collection of processes, channels and variables.

> A *protocol* is a tuple (P, B, V, M_b, M_v), where:
> P is a finite set of *processes*,
> B is a finite set of *channels*,
> V is a finite set of *variables*,
> M_b is a mapping of P onto B. and
> M_v is a mapping of P onto V.

Mapping M_b defines via which channels a process can receive input, and similarly M_v defines which variables a process can access. Each process must be able to send to all channels and to receive input from a subset of the channels. No channel, however, need be read by more than one process. The mapping M_b therefore defines a partitioning of set B into a number of disjoint subsets, where each subset contains the input channels of a single process in P. Process interactions are restricted to message passing, so the same rules can be enforced for the access to variables: each process should be able to access zero or more variables, but no more than one process may access any single variable. M_v therefore also partitions set V into disjoint subsets, each set containing the local variables of a single process.

The choice not to include the definition of variables and channels in the definition of processes is important. It allows us to divide a protocol model into automata representing process behavior and a single *context* automaton representing the environment. The context automaton is defined by the state of variables and channels and in turn defines the executability of actions in the process automata. As a result, the size of a finite state machine description for a given protocol, measured in numbers of states, can be smaller. In the extreme case the improvement can be the difference between the size of a Cartesian product and a sum of two sets of states.

As part of the definition of a channel, we use the notion of a *channel sort*: the set of all messages that can be sent to a given channel.

A *channel* is a triple (S, C, N), where:

> S is the *channel sort*,
> C is the *channel contents*, an ordered set of elements from S, and
> N is the number of messages the channel can maximally hold.

Note that a channel can be in only a finite number of states

$$\sum_{i=0}^{N} |S|^i$$

where $|S|$ is the cardinal number of set S.

Similarly, a variable can be defined to have both a value and a finite *range* of possible values. Each variable then can only be in a finite number of different states.†

A *process* is defined as a tuple (Q, q_s, q_e, A, T), where:

> Q is a finite, non−empty set of states,
> q_s is the *startstate*, and q_e is the *endstate*, elements of Q,
> A is a set of *actions*, and
> T is the state transition relation: a mapping of $Q \times A$ into the powerset of Q.

Since the language to be represented is nondeterministic the state transition relation T can define a subset of Q for every element of $Q \times A$. This nondeterminism is illustrated by the following valid protocol fragment in *Argos*

---

† In the validation language the range of the variables is 2^{15}.

```
state1: if
        :: a?m → goto state2
        :: a?m → goto state3
        fi;
state2: ...
```

which gives us

$$T(state1, a?m) = \{state2, state3\}.$$

Executability

For each state–action pair (q,a) in $Q \times A$, a fixed set of rules determines whether the action a is executable or not. The execution rules, discussed below, define a mapping E of $Q \times A$ onto the set *true*, *false*. If a is executable in state q we define $E(q,a) = true$; If a turns out to be non–executable $E(q,a) = false$ and we will set $T(q,a) = \varnothing$.

Performing an action will in general change the *environment* of a process. Receiving a message, for instance, will change the state of a channel. The execution of an action can also affect the executability of other actions, both in the executing process and in other processes. By sending a message we can enable another process to execute the matching receive action, and by changing the value of a variable we can enable other actions within the same process to pass a boolean test. We distinguish between actions that do, and actions that do not change the state of the environment of other processes. The first class corresponds to the *external* actions of the validation language, such as *send* and *receive*. The second class corresponds to the *internal* actions, including assignments to and testing of local variables.

A receive action *qname?ack* is only executable when the message *ack* is the first message in channel *qname*. Similarly, a send action *qname!data* is executable whenever the channel *qname* is not full. We could define an assignment to a variable to be unexecutable if it attempts to assign a value outside the variable's range. Such a definition would make it easy to model, for instance, semaphores that synchronize parallel processes executing in a shared memory system. However, since we are modeling distributed systems, we instead define a modulo arithmetic on variables. An attempt to assign a value outside its range will result in an assignment *modulo* its range.

The null action *skip* and assignments to variables, then, are always executable. With the guarded command language that we use for protocol specification in mind, however, we define a *condition* to be executable when it is true and unexecutable when it is false. Though it seems odd that we will have to evaluate a condition to find out if it can be 'executed,' this definition will guarantee that a condition can not be passed unless it holds, and thus that conditions that are used as 'guards' in selection and loop structures can be modeled with the same ease as send and receive actions.

Symbolic Execution

Given an initial state q for a process, it can be 'executed' by nondeterministically selecting an executable action a, executing it, and changing the process state to an element of $T(q,a)$. Similarly, the execution of a protocol system is a three–step process: (1) selecting an action from any process, (2) executing the action, and (3) changing the state of the executing process. The explicit *selection* of a candidate action (step 1) allows us to perform the partial searches referred to in section 2.

Optimization

A simple way to translate a description in the validation language into a finite state machine specification is to assign a *state* in set Q to every single statement in the process specification. Statements that start an option in an 'if' or in a 'do' structure (guards), however, are to be treated specially. The guards are part of the 'if' or 'do' statement and do not get a separate entry in set Q. The transition relation T can then be defined in the obvious way to reproduce the control flow of the original specification. Unfortunately, the finite state machines produced in this manner are far from minimal. For one thing, they contain many *transit states* that merely connect two other states with the null action *skip*, e.g. to connect the end of a loop 'od' to its start 'do'. Suppressing the transit–states during compilation saves many useless transitions in the

analysis, where every execution step counts. To suppress all transit states we can use a simple one–pass scan of the states in the transition table:

```
for (all action–state pairs (q,a))
for (all states s in T(a,q))
{       t = s;
        while (transitstate(t))
                t = T(skip,t);
        replace s by t in T(a,q);
}
```

The number of states can be reduced still further by combining equivalent states. The definition of an appropriate equivalence relation, however, has to be chosen with some care. Consider the following protocol fragment.

```
if
:: a?m → b?m
:: a?m → skip
fi;
more...
```

Under a standard notion of *language* equivalence, it can be reduced to:

```
a?m; if :: b?m :: skip fi; more...
```

But, this reduction has changed the behavior of the protocol. In particular, for an input sequence "$a!m$; $b!d$" the original version can deadlock but the reduced version cannot. A stronger notion of equivalence, that avoids this problem, can be defined recursively as follows.

- Two states q_1 and q_2 are (1) *equivalent* if $E(q_1, a) = E(q_2, a)$ for all a.
- Two states q_1 and q_2 are (k) *equivalent* if they are (1) *equivalent* and for each immediate successor state of either state q_1 or q_2 there is a $(k{-}1)$ *equivalent* immediate successor of the other state q_2 or q_1.
- Two states are *strongly equivalent* if they are (k) *equivalent* for all $k \geq 1$.

Using a standard algorithm [1] to partition the set of states under this equivalence relation we can reduce each finite state machine to a strongly equivalent one with a minimal number of states. With this definition we can reduce the redundant:

```
do
::      do
        :: receiver?msg1 → channel!ack1; break
        :: receiver?msg0 → channel!ack0
        od;
        do
        :: receiver?msg0 → channel!ack0; break
        :: receiver?msg1 → channel!ack1
        od
od
```

corresponding to a finite state machine of seven states, to the strongly equivalent:

```
do
:: receiver?msg1 → channel!ack1
:: receiver?msg0 → channel!ack0
od
```

corresponding to a reduced machine of four states, with the same deadlock behavior.

Assertions

We have defined assertions as restricted processes. This means that the assertion primitives can be compiled into a restricted class of state machines and minimized as such, with the same algorithm that is used for the compilation of the protocol processes. The protocol validator can use the assertion state machines to *monitor* the external actions on which they are defined, and perhaps even to try and select those executions in a partial search that have the best chance of violating the correctness requirements they express. The assertion defines a constraint on the execution of the protocol system. This simplest way to use this constraint is to verify that no protocol execution can violate the assertions (see also section 5 below). An alternative could be to constrain a symbolic execution to the sequences covered by the assertion, for instance, to verify the capability of a protocol to perform a given service. This option, however, remains to be explored.

5. State Space Searching

The most commonly used method to perform protocol analysis is to attempt to search the complete *system state space* with an iterative *breadth first* algorithm:

```
breadthfirst()
{      R = { initial state, };      /* set of reachable states */
       A = { };                     /* set of analyzed states  */
       do {
              analyze();
       } while (set R nonempty);
}
analyze()
{      while (R is non−empty)
       {      select an arbitrary state q from R;

              if (q is an errorstate)
                     reporterror();

              delete q from R;
              add q to A;

              for (all executable actions a)
              for (all states s in T(a,q))
              {
                     if (s is not in A or R)
                            add s to R;
       }      }      }
```

We know that the state space generation process must terminate since the size of all channels, the range of all variables, and the number of process states is restricted. As illustrated in the introduction, however, their cross product could well be unmanageably large.

If we succeed in sufficiently bounding the depth of each execution sequence, we can replace the *breadth first* algorithm, with a simpler recursive *depth first* method:

```
depthfirst()
{       q = initial state;
        analyze(q);
}
analyze(q)
{       if (q is an errorstate)
                reporterror();
        else
                for (all executable actions a)
                for (all states s in T(a,q))
                        analyze(s);
}
```

The recursion depth of this algorithm is not necessarily bounded, since even finite systems can have infinite executions. By guaranteeing that the same state is never analyzed twice, however, we can bound the search depth straightforwardly:

```
boundeddepthfirst()
{       P = { };       /* set of states in one execution sequence */
        q = initial state;
        analyze(q);
}
analyze(q)
{       add q to P;

        if (q is an errorstate)
                reporterror();
        else
                for (all executable actions a)
                for (all states s in T(a,q))
                        if (s is not in P)
                                analyze(s);

        delete q from P;
}
```

The maximum size of set P is determined by only the *depth* of the state space tree, while the combined size of set R and set A from the *breadth first* algorithm is determined by its *volume*. At each execution step, set P only contains those states that lead from initial state to the current state. The memory requirements can be reduced even further by observing that not every system state can be found at the start of an execution loop, and therefore it is not necessary to remember every state in set P. The validation language strictly defines which states can start a loop. The state of the protocol consists of the states of variables, channels and processes. The state of the processes in this set of sets is defined by a tuple $(q_1, q_2, ..., q_n)$ with n is the number of processes. Now, a system state that includes $(q_1, q_2, ..., q_n)$ can only be found at the start of an execution loop through the system state space if for at least one $1 \leq i \leq n$ q_i corresponds to a 'labeled' state in the language specification of process q_i, where a 'labeled' state is either a state that is prefixed with a true label (i.e. the target of a goto jump) or the start of a 'do' loop. We can therefore upgrade our algorithm still further:

```
analyze(q)
{       if (q is labeled)
                add q to P;
        if (q is an errorstate)
                reporterror();
        else
                for (all executable actions a)
                for (all states s in T(a,q))
                        if (s is not in P)
                                analyze(s);
        if (q is labeled)
                delete q from P;
}
```

The 'labeled' states in each process can be precomputed by the protocol compiler.

Assertion Checking by Symbolic Execution

For assertion checking we have to perform a controlled execution of the finite state machine generated for the assertion primitives. If an action is within the scope of an assertion, the state of the corresponding state machine is to be updated as a side effect of the execution of that action, as if the assertion machine itself generated the action. Since the assertion primitives do not themselves access variables or channels the 'state' of an assertion machine is uniquely defined by its control–flow state. The 'execution' of an assertion machine then costs very little in the search algorithm. Wherever the behavior of a normal finite state machine, modeling the behavior of a process, is non–deterministic, the search algorithm described above will inspect every possible execution in turn, one after the other in arbitrary order. With the assertion machines, however, we can define the current state to be a true *set* of legal states. If the next–state function T defines two possible successors for state q under external action a, we replace q by *both* successors in the current state set. When the protocol system reaches an endstate, compliance with the assertion can be established by verifying that the current state set of the assertion machine includes its endstate. If this is not true the assertion is violated and the current execution sequence can be listed as a counterexample. Similarly, if, before the protocol has reached an endstate, the assertion machine can not be executed for an action that is within its scope, the assertion has been violated and a counter example can be produced. With little overhead or added complexity, we can thus exploit the finite state machine model to combine the depth–first search with assertion checking capabilities.

Running the bounded *depth first* algorithm to completion, achieves an exhaustive exploration of the system state space, equal to the one obtained with the *breadth first* search method, but using less space to store intermediate results.

Apart from the space complexity of the exhaustive search algorithm, however, we have another problem: time complexity. In the next section we will explore methods we can use to overcome that problem.

Reducing Run Time

Let us first note that, no matter how fast or how sophisticated our search algorithm will be, there will always be cases where exhaustive analysis is no longer feasible. The limits can be moved, but not removed [4,5,6]. Reduction strategies that preserve the scope of an exhaustive analysis are necessary and precious but can never be sufficient. For large protocols an exhaustive analysis generally requires not only unacceptable but also unpredictable amounts of time. Below we will discuss some strategies that can be used to optimize symbolic executions. Then, we will discuss a *scatter* search strategy that can be used to analyze state spaces of which the size precludes analysis by traditional means.

The State Space Cache

We have seen above that the depth first search method can be executed with a state space that only contains states encountered along a single execution path. If an execution path joins a previously analyzed sequence, though, this search strategy will do double work. To avoid it, we can store all states encountered along each

execution sequence and create a complete state space, as in the *breadth first* search strategy, while maintaining the *depth first* search discipline. The problem, however, remains that such a 'complete' state space may well be too large to store. Fortunately, with the depth first search we can be selective about the states that are stored: the state space does not have to be complete as long as it contains at least the minimal subset of states that specifies the current execution sequence. We can therefore create a restricted *cache* of selected system states that are likely to return in different parts of the tree. Initially, we can simply store all system states encountered into the cache. When the cache fills up, however, we may have to delete old states to accommodate new ones. The revised search algorithm now looks as follows:

```
modified_depthfirst()
{       A = { };      /* subset of the previously analyzed states */
        P = { };      /* set of states in one execution sequence  */
        q = initial state;
        analyze(q);
}


analyze(q)
{
        if (q is labeled)
                add q to P;

        if (q is an errorstate)
                reporterror();
        else
                for (all executable actions a)
                for (all states s in T(a,q))
                        if (s is not in P or A)
                                analyze(s);

        if (q is labeled)
        {       delete q from P;
                if (size of A is maximum)
                {       select a state s from A
                        delete s from A;
                }
                add q to A;
        }       }
```

The size of the cache itself is irrelevant to the scope of the analysis. But extending the size of the cache to the maximum that can fit in main memory can avoid double work and thereby decrease the runtime of an analysis.

The question remains what the selection criterion should be for determining which states can be deleted when the cache overflows. One readily available piece of information on the probability that a state will be revisited in a different part of the state space tree is the number of times that it was visited before. Figure 1 shows the results of measurements on two medium size protocols.
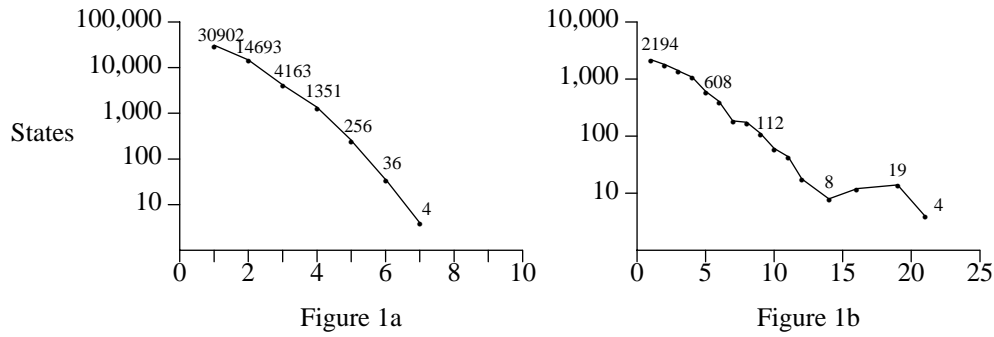
<div align="center">Figure 1a   Figure 1b</div>

<div align="center">*Frequency of Occurrence in State Space*</div>
<div align="center">*Figure 1*</div>

The probabilities that a given states are revisited, given that they were visited N times before, for these two protocols, is shown in figure 2.
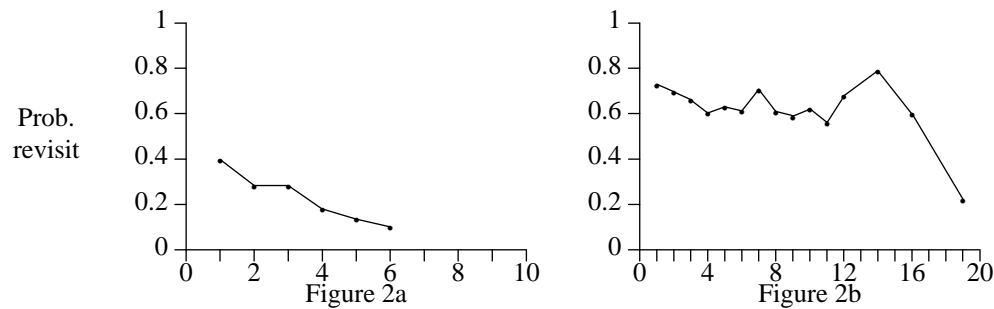
<div align="center">Figure 2a   Figure 2b</div>

<div align="center">*Number of previous visits*</div>
<div align="center">*Figure 2*</div>

For the protocol in Figure 2a the best strategy seems to be to replace those states that were visited most frequently before, since these have the least chance of being visited again later in a search. It must, however, be noted that the absolute number of states that are visited more than twice is relatively small. This strategy can therefore result in the elimination of *all* states visited more than once, thereby invalidating the assumptions we made for a replacement strategy. For the protocol in Figure 2b, the probability of a return to a previous state is largely indepedent of the number of previous visits to that state.

The performance of five different replacement strategies where measured for a range of medium sized protocols. The first two strategies required a complete scan of the state space cache to select a state that was either visited

    (a) most frequently, or
    (b) least frequently

before.

In the third strategy the states were divided dynamically in classes according to the number of times they had been visited before in the search. To replace a state the state space cache was scanned round–robin until a state was found that belonged to the currently

    (c) largest class

of states under this criterion. In the fourth strategy the number of previous visits to a state was ignored entirely. The cache is viewed as a circular buffer. To replace a state with this strategy we blindly picked the one at the pointer and advanced the pointer to the next one in the circle to make sure that the oldest states in the cache are selected first:

    (d) blind, round–robin selection.

In the last strategy we used the depth in the state space tree at which a state was last encountered. It is based on the observation that states near the root of the tree are also roots of the largest subtrees. To replace a state, therefore, it should be advantageous to select a victim as deep in the tree as possible, having the

(e) smallest subtree.

To avoid having to make a complete scan of the state space to find the deepest state in the tree, this strategy was implemented by a round–robin search for a state that was at least in the bottom half of the current tree.

The results of two of the tests are summarized in Figure 3. In the first test we used a protocol that generated a state space of 4523 unique states, and analyzed it with a cache of 1000 states. The protocol from the second test generated 8139 unique states and was analyzed with a cache of 6000 states.
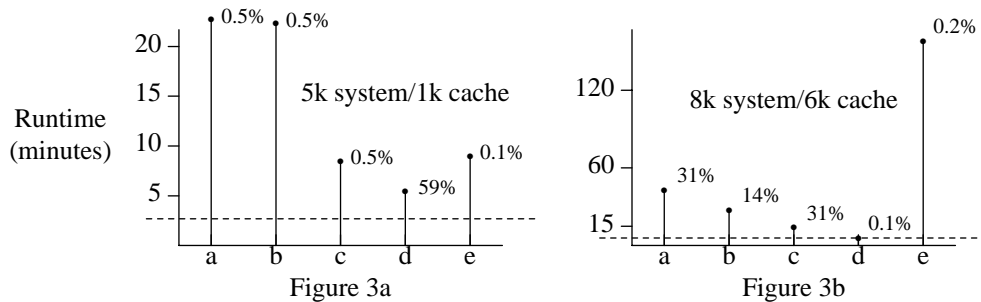
Figure 3a   Figure 3b

Replacement Strategy
Figure 3

The amount of double work incurred with each strategy is shown as percentages in the dot charts. The runtime of strategies **(a)** and **(b)** is dominated by the frequently recurring scan of the state space (88% of the runtime in strategy **(a)**). Strategy **(c)** requires only partial scans of the state space, which makes it faster than **(b)** even though it is not always quite as succesful in avoiding double work (Figure 3b). The worst performance in Figure 3b, is obtained with strategy **(e)**. For this protocol it turns out that the states that are revisited most frequently are all in the bottom half of the state space tree. Remarkably, the simplest strategy **(d)** is consistently the fastest, independent of its quality.

The dashed lines in Figure 3 give the runtime for the algorithm when a cache size is used that can accommodate all unique states. The penalty for reducing the cache by 80% in Figure 3a, or by 25% in Figure 3b is negligible. The effect of larger reductions depends largely on the structure of the state space tree, which is protocol dependent. For the protocol in Figure 3b, reducing the cache to 3000 states increases the runtime more than 100 times, which invalidates it as a useful strategy. In the next section we will therefore consider other, more succesful, strategies to restrict the size of the state space.

Scatter Searching

If we drop the requirement that reductions must maintain the scope of analysis, we will need criteria for deciding which sequences are to be analyzed and which ignored. It is relatively straightforward to give preference to the shortest complete execution sequences and to defer analysis for longer sequences corresponding too, for instance, ten–fold overlapping executions of processes. In the protocol validator that was implemented for specifications written in *Argos* an optional depth–bound is used that sets an upper limit the depth of recursion in the search algorithm. The depth bound is by default the sum of the number of states in all state machines, multiplied with a constant factor ten, but it can be set to any value as a parameter of an analysis run.

An obvious candidate to restrict the amount of work to be done in a search further is to trim the number of options in the inner loop. Our basic tool for this is to add a selection *criterion* that determines which executions will, and which will not, be analyzed:

        for (all executable actions a)
        for (all states s in $T(a,q)$)
            if (*criterion* is met $\cap$ s is not in P or A)
                analyze(s);

In some cases we can still skip analyzing an option without restricting the scope of the analysis. Consider the case where the combined set of action state pairs of all processes contains just two executable actions: and internal action a in machine A and an external action b in machine B. There are two possible orders in

which these two actions could be executed, corresponding to the two sequences

a; b and b; a.

The two sequences each lead into a new states that form the root of two subtrees in the state space graph. But it is easy to see that these two states must be equal. The execution of the internal action a did not change the environment for the remote machine B, so neither the executability nor the result of b can be any different when a is executed first or last. Similarly, the execution of a is independent of the environment affected by B and also its executability and result is independent of whether b preceded it or not. In this case then, it suffices to search one of the two possible interleavings and to ignore the other. The more vigorous restrictions to be discussed below, however, do pay a penalty in the scope of an analysis for limiting the runtime of the analysis. We discuss them in order of severity.

Channel Limits

The capacity of a communication channel for holding messages can have an important effect on the size of a state space. Reducing the capacity of a channel N by one can reduce the size of the state space, and speed up the analysis, by a factor of maximally

$$\sum_{i=0}^{N} |S|^i \ - \ \sum_{i=0}^{N-1} |S|^i \ = \ |S|^N$$

(cf. section 4). If the complexity of the analysis precludes it from running to completion a relatively painless method is therefore to analyze the protocol with reduced channel sizes. The analysis will not be complete, but produces results faster. Fortunately, many, if not all, protocol design errors of interest manifest themselves independently of the absolute size of message channels.

Noninterleaved Actions

The size of the state space tree is determined by the branching factor at each node in the tree. By restricting the branching factor uniformly for all nodes we can select executions from all parts of the state space tree, touching upon every part of the protocol in a short scan, hopefully catching most of the more fundamental flaws in a design. The basic tool for restricting the branching factor of the tree is to extend the set of noninterleaved actions. Above we have discussed a case where one specific interleaving of actions could be ignored since it would predictably lead into sequences that were analyzed before. By relaxing the conditions under which actions need to be interleaved for a complete analysis we can obtain major reductions in search time for a relatively small penalty in scope. Below we give three specific examples that were used in the protocol validator *trace*.

The first example concerns the treatment of *timeouts*. The purpose of an exhaustive analysis is, of course, to examine even unlikely sequences of events to spot the errors that a protocol designer could miss. For timeouts this means that we would like to perform an analysis that is independent of timing considerations. In such an analysis, only the possibility of a timeout will be taken into account, but not it's probability [13]. In practice, validating a protocol for all feasible timings causes valuable reports on the logical inconsistencies of a protocol to be lost in a plethora of reports on errors that may result from choosing inappropriate timeout values. To filter out the errors that are not caused by timing problems *trace* therefore allows the user to require that timeout options be only used to resolve locks, never to generate traffic. The scope of the analysis is restricted, but in a controlled way that, under the assumption of a proper choice for the timeout values, maintains the characteristic behavior of the protocol.

Another particularly interesting method, that succeeds surprisingly well in identifying errors in larger state spaces in seconds where complete searches takes hours, is to restrict the branching factor for each process to one. The branching factor is then maximally equal to the number of state machines in the protocol. The single option that can maximally be explored in each state should then of course be chosen with some care to optimize our chances of finding errors. Since deadlocks are the most common type of protocol errors searched for, the program *trace* will give preference to those actions that can bring the system closer to a deadlock state. This condition is satisfied, for instance, for the execution of a receive action, since it will reduce the number of messages queued in the channels and moves closer to a state where all channels are empty and all state machines are waiting for input.

A third method that is faster still, but not quite as successful in finding nontrivial protocol errors is to set the branching factor for each system state to one. The result will be the analysis of a single complete execution sequence. Only if a protocol is desperately wrong will this be helpful as an analysis tool. It is however a useful mode to start an analysis session that is performed by slowly extending the scope of an analysis, while repeating the analysis runs up to the point where either an exhaustive analysis is completed or proves to be infeasible within a reasonable amount of time.

Foldings

A final method to restrict the run time of an analysis is to define equivalence relations, or *foldings*, on the set of system states. The user could for instance decide that the state of one specific process, message channel, or set of variables is less relevant to the proper execution of the protocol. By defining a folding of the state space that maps all states that only differ in the state of an irrelevant protocol part onto the same class we can reduce the complexity of the analysis. The option was implemented in the protocol validator *trace* but has proven to be a less useful tool. Unlike the methods discussed above, the effect of a folding on an analysis is less predictable, and its usefulness quickly deteriorates.

The Scope of a Partial Analysis

The objective of a partial search is to use a small fraction of the runtime of an exhaustive search to find a large fraction of the errors. There is a minimal fraction of the runtime that has to be spent to find at least the first error in a protocol. Above that threshold, the fraction of all errors found should quickly approach unity. The rate at which the number of errors found approaches the number of errors present, then, can be used to measure the quality of a partial search.
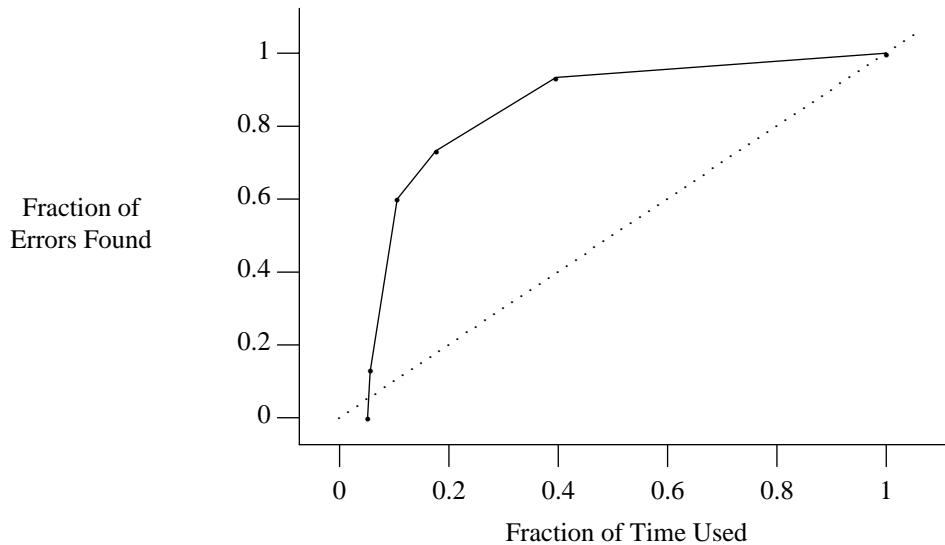

Figure 4

The curve in Figure 4 was compiled from a series of tests on a small protocol with a well–known set of errors (X.21). In these tests we experimented with depth bounds and with restrictions on the branching factors in the state space tree in a limited size cache, and compared the number of errors found in each test to the class of errors traced by an exhaustive search. As yet, there is insufficient data to conclude that the performance from Figure 4 can consistently be achieved, or is perhaps too pessimistic. From these first tests, however, we have every indication that the scatter search technique can be effective.

6. Conclusions

Protocol validation by symbolic execution is inherently a time– and space–consuming problem. So far, however, most protocol validation tools are based on symbolic execution algorithms. We have discussed strategies for expanding the capabilities of symbolic execution tools by reducing the time and space

requirements. For reducing space complexity we have discussed a modified depth–first search algorithm, state space foldings, and the storage of subsets of the state space in a limited size cache. For reducing time complexity we have discussed a range of scatter search techniques, such as search depth bounding, restricting channel capacities, and the restriction of branching factors in the state space tree.

The protocol validation language *Argos* allows us to specify both protocol specifications and correctness requirements in a single higher level specification format that can be checked for syntactic correctness and optimized for analysis by a compiler. The analyzer we have described was written in C and is portable across Unix systems with virtual memory. Small to medium sized protocols can be specified with these tools in a few minutes. Compilation and analysis of these specifications take rarely more than a few minutes of CPU times. The largest protocol analyzed with these tools so far is the control protocol for a dataswitch consisting of four processes and six message channels, specified in an *Argos* description of 270 lines. An exhaustive analysis was performed, with a state space cache of 150,000 states (6.5 Mbyte). With the search depth restricted to 230 steps the analysis completed in 9 hours of CPU time, analyzing a state space 16% larger than the cache (175748 states of which 172402 were unique states). The first deadlock (out of a series of hundreds) was found after 20 minutes of CPU time. A scatter search, with the same depth bound completes in less than 3 minutes of CPU time, and produces an equivalent listing of the main types of protocol errors, by inspecting 4523 or 2.5% of the states. The first deadlock with the scatter search is reported in 3 seconds of CPU time.

The typical use of the validation tools described in this paper then is as follows. We start by building a restricted model of a protocol and its correctness requirements in the language *Argos*. The specification is checked for syntactical correctness, and for a small range of completeness criteria that can be verified at compile time. If the description passes these tests it is minimized and compiled into finite state machines of the type discussed in section 4. The protocol validator *trace* is then invoked, in a first run with the most restrictive setting for all analysis parameters, for instance with channel capacities restricted to one slot, with a large class of noninterleaved actions, and with a depth bound equal to the sum of the states in all automata generated by the compiler. Largely independent of the complexity of the protocol analyzed, this first analysis run will complete in a few seconds of CPU time. If no errors were found in this first restrictive scatter search the restrictions are weakened and the run is repeated, taking slightly longer while analyzing a slightly larger fraction of the state space. This iterative search process can continue until either the first error is found, or until an exhaustive search is completed. The third case is, of course, to find no errors and still be unable to complete an exhaustive search. In that case, our only hope would be to return to the first step of this validation process and to try and define a more restricted, but functionally equivalent, version of the protocol itself, by changing the model built in *Argos*.

7. References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D., "The design and analysis of computer algorithms," Addison–Wesley Publishing Co., 1974.

2. Bochmann, G., & Sunshine, C.A., "Formal methods in communication protocol design," IEEE Trans. on Communications, Vol. COM–28, No. 4, April 1980, pp. 624–631.

3. Brand, D., and Joyner, W.H. Jr., Verification of protocols using symbolic execution. Computer Networks, Vol. 2 (1978), pp. 351–360.

4. Brand, D., & Zafiropulo, P., "Synthesis of protocols for an unlimited number of processes," Proc. Computer Network Protocols Conf., IEEE 1980, pp. 29–40.

5. Cunha, P.R.F., & Maibaum, T.S.E., "A synchronization calculus for message oriented programming," Proc. Int. Conf. on Distributed Systems, IEEE 1981, pp. 433−445.

6. DeTreville, J., "On finding deadlocks in protocols," March 22, 1982, unpublished technical memorandum, Bell Laboratories.

7. Dijkstra, E.W., "Guarded commands, nondeterminacy and formal derivation of programs," Comm. ACM, Vol. 18, No. 8, Aug. 1975, pp. 453−457.

8. Hajek, J., "Automatically verified data transfer protocols," Proc. 4th ICCC, Kyoto, Sept. 1978, pp. 749−756.

9. Hoare, C.A.R., "Communicating sequential processes," Comm. ACM, Vol. 21, No. 8, August 1978, pp. 666−677.

10. Holzmann, G.J., "A Theory for protocol validation," August 1982, IEEE Trans. on Computers, Vol. C−31, No.8, pp. 730−738.

11. Holzmann, G.J., "The Pandora system − an interactive system for the design of data communication protocols," Computer Networks, Vol. 8, No. 2, pp 71−81.

12. West, C., "Applications and limitations of automated protocol validation," Proc. 2nd IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification, USC/ISI, Idyllwild, CA. May 1982, pp 361−373.

13. 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.

APPENDIX

The following is an example of a simple alternating bit protocol specified in *Argos*. The specification consists of four different processes: a sender, a receiver, a processes modeling the behavior of a communication link that can lose messages, and a user process that stores the messages that the receiver claimed to have received correctly from the sender process.

```
channel sender[1], receiver[1], link[1], user[1];

proc sender
{       do
        ::      link!msg1;
                do
                :: sender?ack1 → break
                :: sender?ack0 → skip
                :: sender?timeout → link!msg1
                od;
                link!msg0;
                do
                :: sender?ack0 → break
                :: sender?ack1 → skip
                :: sender?timeout → link!msg0
                od
        od
}


proc receiver
{       do
        ::      do
                :: receiver?msg1 → link!ack1; user!msg1; break /* accept */
                :: receiver?msg0 → link!ack0   /* reject */
                od;
                do
                :: receiver?msg0 → link!ack0; user!msg0; break /* accept */
                :: receiver?msg1 → link!ack1 /* reject */
                od
        od
}


proc user
{       do
        :: user?default → skip /* receive any message and store it */
        od
}
```

```
proc link
{       do
        :: link?msg0 → if :: receiver!msg0 :: skip fi  /* transfer or lose */
        :: link?msg1 → if :: receiver!msg1 :: skip fi
        :: link?ack0 → if :: sender!ack0 :: skip fi
        :: link?ack1 → if :: sender!ack1 :: skip fi
        od
}
```

When analyzing this protocol we would like to establish that the link processes will correctly see messages with alternating sequence numbers. A first attempt to express this in an assertion in *Argos* could be:

```
assert
{       link!msg1;
        link!msg0
}
```

Submitting this specification to the analyzer produces a violation of the assertion in 1.2 of CPU time. The error report produced by *trace* looks as follows:

queue:	channel	sender	receiver
event:			
1	msg1,		
2		tau,	
3	[msg1],		

Every column corresponds to a message channel. A message in a column represents a message sent to the corresponding channel. A bracketed name represents a message sent but not received. The counter example to the assertion produced by *trace* shows that the sender can timeout and retransmit a message with the same sequence number as the last message sent to the link process. We can try again by stating that at least the receiver should see messages with an alternating sequence number:

```
assert
{       receiver!msg1;
        receiver!msg0
}
```

But, again *trace* produces a counter example, this time in 1.4 sec.:

queue:	receiver	sender	channel	user
event:				
1			msg1,	
2	msg1,			
3			ack1,	
4				msg1,
5		tau,		
6			msg1,	
7	[msg1],			

We try again.

```
assert
{       user!msg1;
        user!msg0
}
```

and after 1.6 seconds *trace* reports:

queue:	user	sender	channel	receiver

```
event:
 1                              msg1,
 2                                      msg1,
 3                      ack1,
 4       msg1,
 5              tau,
 6                      msg1,
 7                              msg1,
 8                      ack1,
 9              ack1,
10                      msg0,
11                              msg0,
12                      ack0,
13       msg0,
14              tau,
15                      msg0,
16                              msg0,
17                      ack0,
18              ack0,
19                      msg1,
20                              msg1,
21                      ack1,
22       [msg1],
```

Note that the sender and receiver can loop through their specifications, while the assertion stated that the sequence could occur only once in any given execution sequence. The counter example showed that this is not true. Finally, we can try specifying what we meant to say all along:

```
        assert
        {       do
                :: user!msg1; user!msg0
                od
        }
```

An exhaustive validation by *trace* now completes in 1.7 sec. announcing that it was unable to violate the assertions or to find deadlocks or an incompleteness in the specification.