Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 134

Manual for the Protocol Analyzer ‘Trace’

Gerard J. Holzmann

February 11, 1987

Manual for the Protocol Analyzer ‘Trace’

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Trace is a program that can be used to analyze the consistency of data communication protocols. A protocol is specified in the nondeterministic guarded command language *Argos* that includes case selection, do-loops, variables, expressions, value transfer, procedures, and macros. The analyzer traces deadlocks, unspecified receptions, timing problems, and errors caused by value passing.

The memo describes the specification language *Argos* and explains the error tracing modes provided.

February 11, 1987

Manual for the Protocol Analyzer 'Trace'

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

The first part of this memo gives an informal overview of the language that is used for protocol analysis and discusses general protocol tracing techniques. Part two presents a more systematic overview of the language, gives an overview of the grammar rules and an annotated list of the options that can be used with the tracer. Appendix A gives an example of tracing errors using *Argos* assertion primitives. Appendix B is the manual page for the protocol compiler and analyzer.

To use the analyzer it should be sufficient to read the first part and to use the second part for reference.

Part One

1. Modeling Language

The validation of a communication protocol with *trace* is done in two steps. The first step is to build an abstract model of the protocol in the specification language *Argos*. This model is compiled into an extended finite state machine model with the preprocessor *pret*. The output of *pret* (by default placed into a file *pret*.*out*) can then be analyzed in a variety of ways with *trace*. The syntax of *Argos* is loosely based on *C*, but the language was influenced significantly by the 'guarded command' languages of Dijkstra and Hoare [Dij '75, Hoa '78].

Protocols are modeled in *Argos* by processes that interact by exchanging messages via queues.† The message queue is a primtive data type in *Argos*. Here is a simple example:

```
#define N    2

proc east
{       queue eastq[N];

        eastq?hello −> westq!world
}

proc west
{       queue westq[N];

        if
        :: eastq!hello; westq?world
        :: westq?world; eastq!hello
        fi
}
```

There are two processes, one named 'east' and one named 'west'. The body of a process description contains a declarations part followed by a control flow specification. The declarations part defines the names of local variables and the names and sizes of all message queues the process receives input from. In the

† Note that Dijkstra's language [Dij '75] had no primitives for process interaction. Hoare's language [Hoa '78] was based on the *rendezvous* concept.

example, both process ‘east’ and ‘west’ contain a declaration for a single input queue, named ‘eastq’ and ‘westq’ respectively. Both queues are declared to be of length N, where N is defined to have the value two in a macro definition.

The control flow part is a sequence of statements separated by either semi-colons or arrows (−>). The arrow and the semi-colon are syntactically identical. For communication with other processes there are two types of statements: one for input and one for output.

```
eastq?hello
```

indicates the reception of a message ‘hello’ from queue ‘eastq’. Similarly,

```
westq!world
```

specifies the sending of message ‘world’ to a queue named ‘westq’. The ‘if’ statement provides a mechanism for the specification of nondeterministic case selection. In the example we specified two options: either the second process will start by sending the message ‘hello’ and then wait for the ‘world’ to arrive, or it can try to accept the message ‘world’ first and then respond by sending ‘hello’. A receive statement is only executable when the queue addressed is non-empty and the specified message is indeed the oldest message in that queue. A send statement is only executable when the queue addressed is not full. A non-executable statement simply blocks until it becomes executable. In the example we used send and receive statements to ‘guard’ the options in an ‘if’ statement. An option in such an ‘if’ statement can only be selected if the first statement is executable. If more than one statement is executable one will be selected at random. The execution of an ‘if’ statement terminates when the option selected terminates. Syntactically an option is terminated either by the option separator ‘::’ or by the end of the statement ‘fi’.

A ‘do’ statement has options just like an ‘if’ statement, but is repeatedly executed until an explicit ‘break’ or ‘goto’ statement is encountered in one of the options. A ‘do’ statement where all the options end with a ‘break’ is equivalent to an ‘if’. In the example, therefore, we could have used for process west:

```
do
:: eastq!hello; westq?world; break
:: westq?world; eastq!hello; break
od
```

Any valid statement, including assignments, procedure calls and boolean expressions, may occur as the first statement in an option. An assignment or a procedure call is by definition always executable. A condition is defined to be executable only if it evaluates to true. The following example program uses that property. It also uses two integer variables (declared with the keyword ‘pvar’), and uses value passing.

```
/*
 * find greatest common divisor [Dij '75]
 */

proc gcd
{       queue in[2];
        pvar x, y;

        do
        :: in?number(x) −> in?number(y) −>
                do
                :: (x > y)  −> x = x - y
                :: (y > x)  −> y = y - x
                :: (x == y) −> out!number(x); break
                od
        od
}
proc user
{       queue out[1];
        pvar x;

        in!number(15); in!number(25); out?number(x)
}
```

Process ‘gcd’ receives two numbers ‘x’ and ‘y’ from queue ‘in’, computes the greatest common divisor, and returns the result to queue ‘out’.

Note that the semantics of the ‘if’ and the ‘do’ statement are different from both [Dij ’75] and [Hoa ’78]. In Dijkstra’s language, the ‘alternative construct’ (our ‘if’ statement) aborts the process if all guards are false. In *Argos* the executing process waits until one of the guards becomes true. Similarly, in Dijkstra’s language the ‘repetitive construct’ (our ‘do’ statement) terminates when all guards are false. In *Argos* a ‘do’ statement only terminates when a ‘break’ or ‘goto’ statement is executed, and it will block when all guards are false.

The protocol compiler *pret* checks the syntax and completeness of specifications such as the above, and constructs a minimized finite state machine description for each process and procedure defined. The compiler is invoked with the command:

```
pret ex.1
```

or, in verbose mode

```
pret −v ex.1
```

where ‘ex.1’ is the name of the file that contains the protocol specification. *Pret* will write one of two files, ‘pret.out’ or ‘pret.err’, in the user’s working directory, depending on the type of errors that were found. If invoked without a filename argument, *trace* will try to open the file named ‘pret.out’. If it is decided that an error reported by *pret* can be ignored, the ‘pret.err’ file can either be renamed ‘pret.out’ before the analyzer is invoked, or trace can be invoked with the explicit filename, as in:

```
trace −v pret.err
```

In verbose mode the protocol compiler will list the number of queues used, their names and *sorts* (i.e. the set of messages that are sent to and received from a queue), the processes and procedures with the number of states in the corresponding state machines. It will also report if any states are unreachable. The first example above gives us:

```
$ pret −v ex.1
Overview:
========
2 queues:
        1       eastq,     sort: hello
        2       westq,     sort: world
2 processes:
        1       east,      3 states
        2       west,      4 states
0 procedures:
0 assertions
$
```

To analyze the pret.out file we can now simply say:

```
$ trace
```

but, we'll come back to the *trace* command later. First, let us see what other tools we have to build the protocol specifications.

Variables and Value Passing

Consider the following example process specification

```
proc sender
{       queue sender[1];
        pvar sbit = 1;
        pvar rbit;

        do
        :: channel!smsg(sbit);
                do
                :: sender?cack(rbit) −>
                        if
                        :: (rbit == sbit) −> sbit = (sbit+1)% 2; break
                        :: (rbit != sbit) −> skip
                        fi
                :: sender?timeout −> break
                od
        od
}
```

It specifies a process 'sender' with one input queue, also named 'sender', and two local variables named 'sbit' and 'rbit'. Variable 'sbit' is initialized to the value 1. By default, the other variable will be assigned an initial value of zero. The control-flow specification consists of two nested do-loops. There is just one option in the outer loop. The 'sender' process will start by sending message 'smsg' to the queue 'channel', which is assumed to be declared elsewhere. Attached to the message is the current value of protocol variable 'sbit'. The 'sender' will then enter the inner loop, waiting for the message 'cack' to arrive. If it arrives, whatever value is attached to the message is captured in variable 'rbit', and depending on the value received one of the two options in the 'if' statement that follows will be selected. If the received value equals the value sent earlier the value of 'sbit' is incremented modulo 2 and the 'sender' process breaks from the inner loop, returning to the outer loop. In the other case we execute a 'skip' statement, which has no effect, and return to the start of the inner loop.†

The second option in the inner loop is a timeout on queue 'sender'. The timeout is included here to specify

† An 'if' statement is not skipped if all the guards are false: the process will hang until at least one guard becomes true.

an option that may be executed if no sequence of operations can deliver a new message (specifically the ‘cack’ message expected) into queue ‘sender’. If the timeout occurs we will break from the inner loop without changing the value of the ‘sbit’ variable, and return to the outer loop to retransmit the current value.

By the way, if we attempt to compile the above, incomplete, protocol specification, the result will be:

```
$ pret −v incomplete
sender: queue not addressed
warning: queue ‘channel’ undeclared
warning: queue ‘channel’ unknown owner
queue sender: mesg cack is received but not sent
queue channel: mesg smsg is sent but not received

Overview:
========
2 queues:
        1       sender,    sort: cack
        2       channel,   sort: smsg
1 processes:
        1       sender,    5 states (1 unreachable state)
0 procedures:
0 assertions
output written to ‘pret.err’
$
```

The first message warns us that no message is ever sent to queue ‘sender’. Even worse, queue channel is not declared by any process (unknown ‘owner’). By default the compiler will assign it a size of two slots. The unreachable state reported for process sender, however, is quite harmless. Note that there is no ‘break’ statement in the outer loop of the process so that it’s endstate (just before the closing brace) cannot be reached. Ignoring the errors and tracing the incomplete protocol specification will immediately reveal the obvious deadlock that will occur after process sender has filled up queue channel with two messages and blocks after a timeout. The output, an example time sequence, will look somewhat like this:

```
$ trace −v pret.err
exhaustive search

deadlock
queues:
        sender          channel

                        [smsg](1),
        tau,
                        [smsg](1),
        tau,

5 states, 1 deadlocks, 0 loops
$
```

The real output is a little more elaborate, but we will come back to that later. For an explanation of the specific format used, see appendix A. ‘Trace’ lists a time-ordered buffer history: that is a sequence of send and receive operations that leads up to the error state. Every column corresponds to a queue, and every newline indicates a time step. A normal, unbracketed message is a message that was both sent and received. Every bracketed message is a message sent but not yet received. A deadlock state is a state where no further progress is possible. The message ‘tau’ indicates the occurence of a timeout on the corresponding queue.

Boolean conditions on protocol variables can be used as guards, but there is no equivalent of the ‘else’ for these conditions, so all possibilities must be spelled out. Furthermore, since a condition may be evaluated more than once before it becomes ‘executable’ (i.e. ‘true’) a condition may not have side effects.

The following specification describes a slow ‘division with remainder’ process:

```
/*
 * Slow division with remainder [Hoa ’78]
 */

proc div
{       queue in[2];
        pvar x, y, rem, quot;

        in?number(x); in?number(y);
        quot = 0; rem = x;

        do
        :: (rem >= y) −> rem −= y; quot++
        :: (rem <  y) −> out!number(quot); out!number(rem); break
        od
}

proc user
{       queue out[2];
        pvar x, y;

        in!number(25); in!number(3);
        out?number(x); out?number(y)
}
```

Compiling this in non-verbose mode, and then analyzing it produces:

```
$ pret ex.2; trace −lf
exhaustive search, depth bound 120
endstate:
        in = {number(25),number(3),}
        out = {number(8),number(1),}
execution sequence:
        number(25),number(3),number(8),number(1),
$
```

Which tells us that there is just one possible execution for a division of 25 by 3. It will return a quotient of 8 and remainder 1 to the user. (For the meaning of the tracing flags ‘l’ and ‘f’ see part two.) Of course, a faster equivalent of the slow divider would be the less inspiring program:

```
proc div
{       queue in[2];
        pvar x, y;

        in?number(x); in?number(y);
        out!number(x/y); out!number(x%y)
}
```

Assignments can be written in C-style, including abbreviations such as ‘a++’ for ‘a = a + 1’ and ‘a *= b’ for ‘a = a * b’. No more than one variable may be assigned a value per expression. A condition or parameter field in procedure calls may not include assignments.

More on Timeouts and Defaults

We have briefly discussed timeouts above. There are two pseudo ‘messages’ that can be specified in input statements. They are:

```
from?timeout
```

for a timeout on the contents of queue ‘from’, and

```
from?default
```

to specify that *any* input from queue ‘from’ is acceptable as input. A ‘timeout’, however, cannot specify a parameter. By default a ‘timeout’ option is only assumed to be applicable when no other input from the queue specified is possible. This interpretation can be overruled with a runtime option "−n" in the analysis phase. The option specifies that a timeout can be selected whenever the corresponding queue is empty. The consequences of either choice will be considered in more detail below when the tracing modes of the protocol analyzer are discussed.

Procedures

In its simplest form a procedure can be just a fragment of code that is called from a process body, perhaps indirectly via still other procedures.

```
send()
{       out!msg; in?ack
}

proc west
{
        send()
}
```

The procedure can take arguments. Unless otherwise specified, parameters are assumed to be of type ‘pvar’. For instance:

```
send(val)
{       out!msg(val);
        in?ack
}
```

With somewhat more work, we can also pass queue and message names into a procedure. The mechanism for this is the ‘qset’. The ‘qset’ declares a queue name and the message names that are to be bound to this queue name.

```
pname(ms, pv)
        qset ms { q : msg1, msg2 };
        pvar pv;
{
        do
        :: q!msg1(pv); q!msg2(pv)
        :: q!msg2(pv); q!msg1(pv)
        od
}
```

The formal parameter ‘ms’ is declared as a ‘qset’. The ‘qset’ declaration is enclosed in curly braces; it consist of a queue name, followed by a colon, followed by a comma separated list of message names. The qset must also be declared in the calling procedure and initiated with the actual queue name and corresponding message names.

As a rule, message names can only be passed to procedures via qset’s, and indeed the *only* application of ‘qsets’ is to pass such parameters into procedures. The qset can be declared either locally or globally. The queue in a ‘qset’ declaration need not be declared yet when the ‘qset’ is defined.

Not every queue name used in a procedure has to be passed as a parameter via a qset. It is always allowed to send to any queue. If the queue name used is not specified as a parameter, the name is interpreted as a global name of a queue declared elsewhere. It is also allowed to receive from queues that were not passed as parameters, but the protocol compiler will complain if the name of the queue is not declared in one and only one of the calling processes. It is not valid, though, to use message name parameters in combination with global queue names or vice versa.

Arrays

One of the more recent additions to *Argos* is the inclusion of process arrays, queue arrays and variable arrays. The following example defines an array of eight processes, and eight messages queues, with five message slots in each queue.

```
#define NUMPROCS    8
#define BUFDEPTH    5
#define active      1
#define inactive    0

queue process_q[NUMPROCS][BUFDEPTH];

proc  db_server[NUMPROCS]
{
        pvar i, state;

        i=0;
        do
        :: (i <  NUMPROCS) -> process_q[i]!hi_there(_PROCID); i++
        :: (i == NUMPROCS) -> break
        od;

        do
        :: process_q[_PROCID]?hi_there(i)  -> state = active
        :: process_q[_PROCID]?destroyed(i) ->
                        if
                        :: (i != _PROCID) -> state = inactive
                        :: (i == _PROCID) -> skip
                        fi
        od
}
```

There is a predefined local variable '_PROCID' in each process that holds the process index (a value between zero and seven in the example).

Compiling this description prints the following:

```
Overview:
========
1 queue:
        1       process_q[8], sort: hi_there/*2, hi_there/*3, destroyed/*3
8 processes:
        1x8   db_server, 9 states (1 unreachable state)
0 procedures:
0 assertions
```

The multiplexed queue is listed as one queue, with an array index. The process array is indicated by the count '1x8'. The magic behind the slash of the message names is a reference to the queue being indexed (in general an expression).

Arrays of variables are trivial. They are declared and used in the obvious way. For instance:

```
queue west[4][2];
pvar  varname[8];
...
west[2]!appel(varname[ varname[0] ] - varname[3]);
```

Shared Variables

Be warned that ‘trace’ is meant to find errors caused by message passing, not the access of shared variables. Global variables can freely be used, but *not* for synchronization, as in:

```
pvar sema = 1;

proc nogood1
{       do
        :: (sema == 1) −>
                sema = 0;
                skip;           /* critical section-1 */
                sema = 1;
                skip            /* non-critical section-1 */
        od

}

proc nogood2
{       do
        :: (sema == 1) −>
                sema = 0;
                skip;           /* critical section-2 */
                sema = 1;
                skip            /* non-critical section-2 */
        od

}
```

Note that the test and set sequence above is not indivisible.

With the more recent versions of ‘trace’ errors in specifications of this type can still be traced, but only with a completely exhaustive search (untruncated), which can be painfully slow (flag −e, see trace options).

Assertions

By default ‘trace’ will check a protocol for the observance of general correctness requirements such as absence of deadlock, and completeness. Assertions can be 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 assertion thus defines a constraint on the execution of the protocol.

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. Appendix A gives a more elaborate example of the usage of assertions.

Restrictions

The version of *Argos* that has been implemented 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: a short integer, or an array of short integers. Further, though the language allows for procedures to call other procedures, it rules out direct or indirect recursive calls, and it provides no means for procedures to return values to a caller.

2. Tracing

In its full generality protocol analysis is really an attempt to solve an intractable problem [Bra ’80, Cun ’81]. This means that there is no general method to perform exhaustive analyses of arbitrarily complex protocols that is guaranteed to complete within a given length of time. Assuming that we still do need tools to analyze protocol designs, and preferably automated tools, the best we can do is to simplify the analytical models that we will subject to analysis or, where no further simplification is feasible, to restrict the scope of the analyses. *Trace* is a tool for exactly this purpose. For relatively small protocols, that generate state spaces of 10^5 states or less, exhaustive analysis is feasible. For larger protocols, generating 10^6 and up, exhaustive analysis is generally out of the question. What we need here is a tool that can ‘probe’ the state space for errors, in such away that if the protocol contains any design error, a series of brief trace attempts will find it. The tracing program exploits heuristic search techniques to select the suspect execution sequences from all possible sequences in a partial search.

The *trace* program can be used for both exhaustive and partial searches.

For a quick overview of all options you can type:

```
trace −?
```

The output will look something like this:

```
usage: trace [−option [N]]
−a show all prefixes leading into old states
−b ‘blast mode’ (quick, very partial search)
−c  N  perform class N validation (N: 0..5)
−e ‘exhaustive search’ (untruncated)
−f or −F alternative formats for printing queue histories
−j stop at the first buffer lock found
−k  N  restrict the state space cache to N thousand states
−l show normal execution sequences and loops, but no prefixes
−m  N  restrict search depth to N steps
−n don’t use timeout heuristics
−q  N  restrict queue sizes to  N slots
−r  N  restrict the runtime to  N minutes
−R  N  report on progress every N minutes
−s show the transition tables (different if combined with ‘v’)
−v verbose − print execution times, etc.
−x perform a scatter search
no flag: try a sensible partial search
```

Protocols of a complexity comparable to the CCITT recommended X.21 protocol, the Arpanet 3-way

handshake protocol, or the alternating bit protocol, can be analyzed exhaustively within 5 seconds of CPU time. When performing a partial search on much larger protocols, the search heuristics can identify design errors in state spaces of which the size is usually beyond the scope of automated validation tools [We ’82].

Default Search Mode

If *trace* is invoked without arguments it will make an estimate of a reasonable search depth and run a default partial search of the protocol. The default search will not always find all errors in a specification, but it is often a useful starting point for a series of tests that can help to identify the flaws. For the first example protocol we discussed,

```
proc east
{       queue eastq[2];

        eastq?hello; westq!world
}

proc west
{       queue westq[2];

        if
        :: eastq!hello; westq?world
        :: westq?world; eastq!hello
        fi
}
```

the default analysis produces:

```
$ pret ex.1
$ trace
default search: −vxjqm 2 14
        time: 0.42s u + 0.60s sys = 1.02s

        1 states cached, 0 states zapped
        search depth reached: 4; memory used: 0
        0 locks, 0 loops, 1 terminating executions, and 0 prefixes
$
```

By default, *trace* will read in the compiled protocol specification from the file ‘pret.out’. The search mode selected is echoed to the standard error output; in this case the combination of flags and parameters selected is "−vxjqm 2 14". The number of options seems discouraging, but we will quickly see that each option fulfills a simple and useful function. The "−v" flag, for instance, selects the ‘verbose’ output which prints the timing results and some data on the number of states in the state space and the types of execution sequences examined. Discarding all the other options, we can invoke a full search on this trivial protocol by selecting just the "−v" option:

```
$ trace −v
exhaustive search, depth bound 56
        time: 0.42s u + 0.52s sys = 0.93s

        1 states cached, 0 states zapped
        search depth reached: 4; memory used: 0
        0 locks, 0 loops, 1 terminating executions, and 0 prefixes
$
```

The state space cache held just one state. There are no errors; the execution terminates in a normal ‘endstate’ (i.e. at the closing brace in each process description). In this simple case we do not win or lose much with any of the different search modes.

When a deadlock or a loop is reported, ‘trace’ will list a time-ordered buffer history: that is the sequence of send and receive operations that leads up to the error state. See appendix A for some examples.

Getting Started: Validation Classes

There is a small set of options that can quickly get you started when analyzing a protocol. Six classes of protocol validation have been pre-defined. The are invoked with the flag "−c" where the argument indicates the validation class chosen. A validation *class 0* for instance, is invoked by typing

```
$ trace −c 0
```

It is the fastest partial search available, and will generally complete within a few seconds of CPU time. If this search does not produce an error we can move one step up to a *class 1* validation, by typing

```
$ trace −c 1
```

which is somewhat slower, but explores more cases. This sequence can be continued up to a *class 5* test, which will perform an extensive search for errors. For some protocols, however, analysis in the classes 3 and up may no longer be feasible within a reasonable period of CPU time. To overcome this, more specific error tracing modes can then be specified by the user to probe the state space.

If the runtime of the analysis exceeds two minutes real time *trace* will print an intermediate report on the standard error output on the progress made, and it will keep doing this every two minutes for the duration of the run (cf. the "−r" flag below). As an example, in an attempt to run an exhaustive analysis on an extremely large state space one of the reports was:

```
seconds depth  states  returns zapped loops   locks  memory
61097.0   743  889820   624516 842946   1382  4377 2909184
```

The first column gives the run time in seconds (see "−r" flag). The second column gives the maximum search depth reached (see "−m" flag). Then the number of states, returns and zapped states is listed ("−k flag"); followed by the number of execution loops and deadlocks discovered. The last column gives the total size of the allocated memory. In the above example we specified a state space of 46874 (45 * 1024) states which gives a cache of roughly 3 Mbyte. † If the analysis run is aborted by sending the program an interrupt signal, it will report the progress made up to the point where it was interrupted. The equivalent of each validation class in terms of the other option flags available is as follows:

validation class:	equivalent options:	description:
0	−vjbl	‘blast search, default depth N’
1	−vjxm 2N	‘partial search, depth 2N’
2	−vjx	‘partial search, default depth’
3	−vjm 1.5N	‘exhaustive search, depth 1.5N’
4	−vjm 2N	‘exhaustive search, depth 2N’
5	−vj	‘exhaustive search, default depth’

The number N in this list is the sum of the effective number of states per process. The meaning of the other flags is explained in more detail in part two.

---

† The analysis completed after 80 hours of computation on a VAX 750, having generated and analyzed roughly 5.3 million states. It shows that a full analysis in a large state space may sometimes still be feasible, but only for a considerable runtime penalty. For comparison: a partial search in the same state space traced 25% of the errors and completed in 2 minutes of CPU time.

Part Two

3. Language Summary

3.1. Lexical Conventions

White space (blanks, tabs, newlines, and comments) is ignored, except where it serves to separate tokens.

3.2. Comments

Any string started with /* and terminated with */ is a comment. Comments may be nested.

3.3. Identifiers

An identifier is a single letter followed by zero or more letters, digits or underscores. Only the first fourteen characters of an identifier are significant.

3.4. Keywords

The following identifiers are reserved for use as keywords:

any	assert
break	default
do	error
fi	goto
if	od
proc	pvar
qset	queue
skip	timeout

The keyword ‘any’ is syntactically equivalent to the keyword ‘default’. ‘Error’ is a special assertion primitive that works like ‘assert’ but specifies sequences of event that are not supposed to happen. It’s usage is not described in this manual.

3.5. Constants

A constant is a sequence of digits. Any constant is assumed to be a decimal integer.

3.6. Expressions

The following operators can be used to build expressions.

+, −, *, , %,	arithmetic operators
>, >=, <, <=, ==, !=, !,	relational operators
&&, ||	logical AND and OR.

Apart from these operators, the abbreviations ‘a++’ and ‘a−−’ can be used to increment or decrement the value of a variable ‘a’.

An assignment is of the general type ‘a = expression’, where ‘a’ is a protocol variable. There may be only one (explicit or implicit) assignment to a variable per assignment statement.

A condition may be any expression enclosed in round braces, but without implied assignments (e.g. ‘a++’ is not acceptable in a condition). A condition may appear wherever a statement can appear.

3.7. Declarations

The following objects must declared: procedures, variables, queues, and qsets. Except for queues, all objects must be declared before they can be referenced. A queue must be declared before the first message is received from it, but not necessarily before messages are sent to it. Queues, variables and qsets can be declared either locally, within a procedure or a process, or globally. Procedures can only be declared globally. Local declarations must appear at the start of a procedure or process body.

3.7.1. Variables

A variable declaration is started by the keyword ‘pvar’ followed by one or more identifiers, and is terminated by a semicolon:

```
pvar name1, name2, name3;
```

There is only one type of variable and it is assumed to be a short integer. By default all variables are initialized to zero. An explicit initial value can be specified with an initializer:

```
pvar name1 = name2;
```

where ‘name2’ is either a constant, the name of a previously declared variable, or an expression.

An array of ‘pvar’s can be declared as follows:

```
pvar name1[N];
```

where ‘N’ is a constant.

3.7.2. Queues

A queue declaration is started by the keyword ‘queue’ followed by one or more queue specifiers, as follows:

```
queue name1[N1], name2[N2], name3[N3];
```

where N1, N2, and N3 are constants. The constants specify the sizes of the queues. By default all queues are initialized to be empty. An explicit initial contents can be specified with a queue initializer:

```
queue name1[N1] = { mesg1, mesg2, mesg3 };
```

The queue specifier is followed by an equal sign and one or more message names separated by commas and enclosed in curly braces. Queues are either declared globally or by a process that reads from it. A queue may not be declared globally, however, if more than one process uses it fopr input. In that case at least one of these processes must contain the queue declaration.

An array of queues, all of the same length, can be declared as follows:

```
queue name1[NQ][QL];
```

where ‘NQ’ is a constant defining the number of queues in the array and ‘QL’ is a constant specifying the queue length. A queue array cannot be initialized.

3.7.3. Qsets

A queue set is used to attach a symbolic name to a combination of one or more messages and the queue to which they can be sent. The only use of a qset declaration is to pass message names to a procedure. A qset is declared as follows:

```
qset name1 { name2 : mesg1, mesg2, mesg3 };
```

where ‘name1’ is the name of the qset declared, ‘name2’ is the name of the queue, and remaining names are names of messages that can be sent to queue ‘name2’.

3.7.4. Procedures

A procedure is declared by a name, followed by a list of parameter names, enclosed in parentheses, followed by a list of declarations for the parameter names, followed by the procedure body enclosed in curly braces. The shortest procedure declaration is:

```
name()
{       skip
}
```

which declares a procedure ‘name’ without parameters and a body consisting of only the null statement ‘skip’. A more interesting example:

```
name1 ( param1, param2 )
        pvar param1;
        qset param2 { name2 : mesg1, mesg2 };
{
        /* local declarations, if any      */
        /* sequence of statements          */
}
```

declares a procedure ‘name1’ with two parameters. The first parameter is declared to be a variable, and the second to be a qset. The names given to the queue and the messages in the queue set are irrelevant, as long as the number of messages named matches the number of messages declared in the corresponding qset of the calling process or procedure. Parameter declarations that are omitted are assumed to be variables. Procedures can call other procedures but may not be recursive. Actual parameters in a procedure call are either names (of variables or qsets) or expressions (to match variables in the formal parameter list). Arguments are passed by value. Procedures do not return values.†

3.8. Processes

A process specification is started with the keyword ‘proc’ followed by a name and a sequence of statements separated by semicolons. The body of a procedure or process is enclosed in curly braces:

```
proc name
{       /* local declarations */
        /* sequence of statements */
}
```

Process specifications, procedure declarations, and global qset or variable declarations can be listed in any order, as longs as the declaration of any object precedes its use elsewhere.

An array of processes can be declared as follows:

```
proc name[NP]
{       /* local declarations */
        /* sequence of statements */
}
```

where ‘NP’ is a constant defining the number of processes. Local to each process in the array is a predefined variable named ‘_PROCID’ that holds the array index of the corresponding process.

3.9. Statements

There are ten types of statements:

selection	cycle	break
send	guard	skip
condition	assignment	
procedure call	goto	

each statement may be preceded by a label (a name followed by a colon). The ‘skip’ is a null statement; it has no effect but may be needed to satisfy syntax requirements. Goto statements can be used to transfer control to any labeled statement within the same process or procedure. Conditions, assignments, and procedure calls have been discussed above. Below we consider the remaining statements: selection, cycle, send and guard.

---

† Where return values turn out to be indispensable they can be imitated with global variables, or by passing a value via a message on a separate queue back to the calling process.

3.9.1. Selection

A selection statement is started with the keyword ‘if’, followed by a list of one or more ‘options’ and terminated with the keyword ‘fi’. Every ‘option’ is started with the flag ‘::’ followed by any sequence of statements. One and only one option from a selection statement will be selected for execution. The first statement of an option determines whether the option can be executed or not. If more than one option is executable, one will be selected at random. Note that this randomness makes the language a nondeterministic one. The tracer, when analyzing a specification, will not make the random choice: it will analyze all possible choices.

3.9.2. Cycle

A cycle or ‘do’ statement is similar to a selection statement, but is executed repeatedly until either a ‘break’ statement is executed or a goto jump will transfer control outside the cycle. The keywords of the cycle are ‘do’ and ‘od’ instead of the ‘if’ and ‘fi’ of selection. The ‘break’ statement will terminate the innermost cycle in which it is executed. The effect of executing a ‘break’ statement outside a cycle is undefined.

3.9.3. Send

The syntax of a send statement is:

```
name1!name2
```

where ‘name1’ is assumed to be the name of a queue, and ‘name2’ the name of a message that is to be appended to that queue. The send statement is not executable (blocks) if the addressed queue is full. If a value is to be passed from sender to receiver, the syntax is:

```
name1!name2(cargo)
```

where ‘cargo’ is a constant, the name of a variable, or an expression returning a value. In the current implementation of *trace*, the value transfered must be in the range: $0 \leq cargo < 2^{15} = 32768$.

3.9.4. Guard

The syntax of the receive statement is:

```
name1?name2
```

where ‘name1’ is again the name of a queue and ‘name2’ the name of a message that the executing process expects to find in that queue. The statement can be executed only if the queue addressed is not empty and if the first message in the queue matches the one specified in the guard. Again

```
name1?name2(cargo)
```

specifies the reception of a message with a value attached. ‘cargo’ must be the name of a variable in which the value transferred is to be written. It is an error to attempt to receive a value when none was transferred and vice versa. Two special types of guards are predefined:

```
name1?timeout
name1?default
```

The first type specifies a timeout on queue ‘name1’. It is executable only if queue ‘name1’ is empty. The second type specifies a default input from ‘name1’, which is executable whenever queue ‘name1’ is not empty. Note that a ‘default’ will delete the first message from the input queue, whatever it is.

3.10. Macros and Include Files

The source text of a specification is processed by the C preprocessor for macro-expansion and file inclusions.

APPENDIX **A**

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.