An Automatic Trace Analysis Tool Generator for Estelle Specifications

S. Alan Ezust and Gregor v. Bochmann

Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC, Canada H3C 3J7
E-mail: {ezust,bochmann}@iro.umontreal.ca

Abstract

This paper describes the development of Tango, an automatic generator of backtracking trace analysis tools for single-process specifications written in the formal description language, Estelle. A tool generated by Tango automatically checks the validity of any execution trace against the given specification, and supports a number of checking options. The approach taken was to modify an Estelle-to-C++ compiler. Discussion about nondeterministic specifications, multiple observation points, and on-line trace analysis follow. Trace analyzers for the protocols LAPD and TP0 have been tested and performance results are evaluated. Issues in the analysis of partial traces are also discussed.

Keywords: Estelle, Trace Analysis, Protocol Conformance Testing, Formal Description Techniques

1 Introduction

Test result analysis involves analyzing with respect to a specification, the observable behaviour of an IUT (Implementation Under Test) in response to a certain number of executed test cases. Usually, the only observable behaviour of an implementation is a "trace", or a log of the interactions sent through the IUT's service access points, hereafter referred to as interaction points, or **IPs** for short. When only the observable interactions are used in test result analysis, this kind of testing is called "black box" testing. An oracle is needed to determine if each trace could have been generated by an implementation which behaves as specified.

A **trace analyzer** provides the function of this oracle and determines, usually by simulation, whether a trace is valid with respect to a formal specification. An invalid trace is a trace which contains an interaction which could not have been generated by an implementation which follows the specification.

Below are some other situations where a trace analyzer could be useful.

- A deterministic implementation which is accepted as "correct" can be used as a an *operational specification* [8] during the development of a formal specification, which then can be used later to generate implementations on other platforms automatically. In this situation, the formal specification can be tested for conformance to the operational specification. Since the operational specification is deterministic, it also can be viewed as a trace analyzer.
- It may be necessary to take two human-generated implementations which are on different platforms and test the interoperability between them, in which case a trace analyzer could act as an "arbiter" and provide diagnostic information about the behaviour of each implementation.
- A specification which is accepted as correct is used as a test verdict checker, to determine if the test case result (pass or fail) attached to a particular branch of the test case is correct with respect to the specification.

Reprinted from Computer Communication Review Volume 25 Number 4 October 1995 Proceedings of ACM SIGCOMM 95 Conference, Cambridge, MA. Pages 175-184

While the specification language, and the trace analysis technique described in this paper, are both applicable to other areas of software testing, our interest is primarily in the area of communication protocols.

Most formal specifications of commonly used protocols are to some degree, nondeterministic. This is because typical protocols are expected to react to external events, and it is possible that multiple events which require responses may occur simultaneously, or so close together that they appear simultaneous. Often, the actual order in which such events are handled is not specified, unless certain events have a higher priority than others. Piggybacking response messages, where a single message may be a response to more than one event, is another common source of nondeterminism in specifications.

A trace analyzer for a nondeterministic protocol specification may need to try multiple possible execution paths before determining the validity of a trace, so trace analysis is not quite as straightforward as protocol simulation. Furthermore, there are additional concerns when developing a trace analyzer which operates on-line, as opposed to one which is running in batch mode. Analyzing the timing of events for conformance to a specification introduces a whole class of difficulties as well.

This paper presents the requirements and the development of a trace analysis tool generator for Estelle specifications. Estelle [10] is a specification language standardized by the ISO and is based on a model of communicating Extended Finite State Machines (EFSMs). Estelle may be viewed as a set of extensions to Pascal which enable the specification of non-deterministic concurrent communicating processes, or

modules. The principles and difficulties of trace analysis discussed in this paper apply as well to trace analysis with respect to specifications written in SDL [1].

The approach taken was to start with an Estelle-to-C++ compiler called Dingo [17], which generates an executable implementation of a protocol specification. We added routines to enable these generated implementations to perform a state-space search on all possible execution paths which consume the inputs and produce the outputs provided in a trace file.

The result of our work, Tango, also known as the Trace ANalysis GeneratOr, does just that. It generates a trace analysis tool based on a single-module Estelle specification, which can analyze traces using relatively small amounts of memory and CPU time. Almost all Estelle programming constructs (with the exception of **delay** clauses and **primitive** functions and procedures) are supported by Tango.

1.1 Related Work

Several trace analyzers have been written for specific protocols such as SNA [5], MAC [13], Class 4 Transport [11] and X.25 [14], but products such as these were, for the most part, developed by humans, had to be tested very thoroughly before they were put to use, and were not easily adaptable for use on other protocols.

Most of the above examples are trace analyzers constructed for a deterministic specification. In the case of deterministic specifications, any correct implementation can be used for trace analysis: It is sufficient to apply the input interactions of the trace to the implementation and compare the output interactions of the trace with the output produced by the implementation; the trace is valid if, and only if, they are the same. This simple approach is not possible for nondeterministic specifications.

A trace analysis tool for nondeterministic specifications written in LOTOS [4] has been described in [2]; it uses a state-space exploration approach similar to the one described in this paper. Like Tango, it concentrates on the (possibly nondeterministic) control flow of the specification and assumes (except for simple value generation by internal interactions) that the data parameters of output interactions can be (deterministically) deduced from the input parameter values.

Another approach is described in [12, 20] where the data part is handled through symbolic execution. Applied to Estelle specifications, this approach requires that the specification be manually transformed into *Estelle.y*, a subset of Estelle which does not support state lists, dynamic memory, procedures or functions, and the data structure definitions must be defined in ASN.1. Given a specification in Estelle.y, TESTVAL generates a set of paths satisfying the input and output messages in the test case, and symbolic evaluation is used to detect and delete infeasible paths in that set. The trace fails if the set is empty. This approach is quite elegant from a theoretical standpoint, but certain aspects of the initial transformation are not automated, making the generation of a trace analyzer for an arbitrary Estelle specification less straightforward.

SPIN [9], a verification tool for specifications written in Promela, performs state space exploration on systems of parallel processes, supporting very advanced techniques for reducing the search space/time. SPIN is primarily a specification validator, and from what we have gathered, is not yet used specifically for protocol trace analysis in conformance testing.

1.2 Pet/Dingo

Pet/Dingo, developed at the National Institute of Standards and Technology (NIST), is the second NIST Estelle compiler. The first one, called the NBS Prototype Compiler [18], generated C code and simulated parallelism through a process scheduler. Pet/Dingo is a major step forward, in that it takes an object-oriented approach to specification generation and, for modules which are supposed to be implemented as independent processes, Dingo generates code for independent processes which communicate by sockets (if they are running on the same computer) or Remote Procedure Calls (if they are running on different computers), and synchronize with each other as specified.

PET, or the Portable Estelle Translator, is written in C++ and uses Bison. Bison is a parser similar to Unix's `yacc`, except it has some enhancements in error recovery, and it is produced by the Free Software Foundation.

Pet performs a syntactic and semantic analysis of the Estelle specification, and if the specification has no compiler-detectable errors, Pet outputs an object-oriented static model of the specification.

DINGO, or the Distributed ImplementatioN GeneratOr, can be thought of as the second pass in the code-generation process. The Dingo executable reads the output of Pet into memory, organized as a tree of objects. By traversing this tree of objects, Dingo generates a C++ object hierarchy based on this tree, which can be compiled and linked with a run-time library to produce an executable implementation based on the original specification.

2 Development of Tango

Trace Analysis on nondeterministic specifications can be thought of as a form of state space search, where the search tree consists of nodes (states) and edges (transitions). A trace is "valid" if there exists at least one "solution", or a path (sequence of transitions), from the root of the tree (initial state) to a leaf node (another valid state), which generates all of the interactions in the trace. If the entire state space in the tree is searched, and no solution is found, the trace is "invalid". Usually, a depth-first search (**DFS**) strategy is used for trace analysis, although for parallel or multi-threaded testers with plenty of memory, a breadth-first strategy might be considered as a faster alternative [2]. For on-line trace analysis, simple DFS is not sufficient, as explained further in Section 3.

2.1 Input Requirements

A valid Estelle specification for Tango consists of one EFSM process, or a **module**, with a fully-defined module body, to specify the behaviour of the IUT. The current version of Tango does not support trace analysis of multiple concurrent module specifications, because for our purposes, the added convenience of multiple-module trace analysis was not justified by the increase in algorithmic and state-space complexity of such a tool.

The module which specifies the behaviour of the IUT is called the **TAM**, or the Trace Analysis Module. It should be free of "non-progress cycles"[1], as these can foil DFS algorithms, yielding search trees of infinite depth.

The `delay` statement in Estelle, which is used primarily to specify timeouts in a specification, is not supported by

[1] sequences of transitions which consume no input, produce no output, and end up in the same module state

Tango. The reason for this is that Tango trace files do not contain time stamps, and Tango itself does not keep track of the simulated time during its depth-first search. Estelle is not expressive enough to fully specify performance aspects of a protocol, and implementing a performance model in Tango was not one of our goals in this project. For more information about Estelle performance models, see for instance [3].

2.2 Depth-First Search

DFS performs the following operations, which had to be implemented in Tango-generated TAMs:

- Generate: Generate a list of all of the fireable transitions from the current state to the next possible states
- Update: Execute a transition
- Save: Save the current state, for later possible backtracking
- Restore: Restore a state which was saved earlier, in such a way that subsequent transition executions would behave the same way as they would have when the state was saved.

2.3 TAM States

The state of an Estelle TAM is a composite of the following information:

- Values of module state variables:
  - The Estelle FSM module state, expressed as an ordinal value
  - Global module variables of any size or structure, as defined in the Estelle specification
  - Dynamic memory, which may be allocated or disposed while executing a state transition
- Queue states, to reflect which input and output interactions from the trace file were consumed or produced at the current state

2.4 Runtime Options

After the initial required features were implemented, various enhancements were made to the Tango system to make it more useful in practical applications. These are all referred to as **runtime options**, and they are discussed briefly here.

2.4.1 Initial State Search

Often, an IUT is executing for a while before a trace is collected, in which case the initial state of the IUT is not known. Sometimes, it is desired to analyze such traces.

By default, the TAM fires the `initialize` transition and then starts analyzing the trace. Tango supports an optional initial FSM state search. If the trace is found to be invalid when the TAM begins analysis from the default initial FSM state, the TAM will backtrack to the point right after the `initialize` transition was taken, choose another initial FSM state, and begin the analysis again.

It should be noted that when the DFS begins, the TAM currently assumes that the values of all IUT variables and dynamic memory are initially left as set by the `initialize` transition block. In the event that they were changed in the IUT before the trace was collected, this might cause an "invalid trace" result on a valid trace.

It is computationally impractical to try all possible initial TAM states. In the case of Estelle, they may be infinite in number due to the fact that Estelle supports dynamic memory allocation. In most situations, it is not sufficient simply to try different initial FSM states, as Tango does. Another approach for handling partial traces is discussed in Section 5.

2.4.2 Interaction Relative Order Checking

The order of the interactions, as they appear in the trace file, can be interpreted in a number of ways. In all cases, if two interactions going in the same direction through the same IP appear in the trace file, the order in which they appear is observed and checked by the trace analysis tool. However, the order of interactions which go through different IPs, or through the same IP but in different directions, can be **observed** (and checked) or **ignored** by the TAM, depending on the runtime options.

In general, enabling these relative order checking options reduces the state space (and therefore, the search time), without any sacrifice in the quality of analysis. However, queues in the IUT can cause Tango to give invalid results on valid traces if the trace information was observed before interactions were placed into a queue.

In the case of full order checking, the inputs and outputs in the trace file must be in an order in which the inputs can be consumed and the outputs can be generated by the Estelle module specification, assuming no input queues. However, in practice, the IUT that has generated the trace file may include input and/or output queues associated with the different IPs observed. The presence of these queues may lead to an order of the interactions in the trace file which is not compatible with the simple Estelle specification (assuming no queues).

For instance, if separate input queues are present for different IPs, the relative order of trace inputs pertaining to different IPs is of no relevance, and hence, should be ignored. Similarly, if the output interactions from different IPs travel through different queues before being recorded in the trace, the relative order of outputs pertaining to different IPs is of no relevance, and should be ignored as well. Finally, for any given IP, if an input or output queue is present in the implementation, it is possible that an input which appears in the trace file before an output, passing through the same IP, is actually consumed *after* the output is generated by the IUT. In this case, the relative order of inputs with respect to outputs is of no relevance.

Therefore, depending on the degree of observability of the IUT, not all of the relative order checking options below are applicable.

Inputs with respect to outputs: Ensures that the next input consumed by a transition precedes any other output interaction at the same IP in the trace. This option should be used under most circumstances.

Outputs with respect to inputs: Ensures that the next output generated by a transition precedes any other input interaction at the same IP in the trace. This option should not used if the implementation that generated the trace includes an input queue for the IP in question.

IP relative order checking: Ensures that the next input consumed by a transition precedes any other input in the trace, and that any output generated precedes any other output in the trace. This option should not be used if the implementation that generated the trace includes input or output queues.

There is a special case handled by Tango when this option is used. If there are multiple outputs sent to different IPs in a single transition block, then Estelle semantics do not specify the actual order these outputs should be generated. Therefore, if the order of these outputs is permuted in the trace, it is still considered a valid trace.

It is clear that the presence of the input and output queues in the implementation reduces its observability. These issues are discussed in more detail in [6]. However, it is important to note that the use of order checking during the trace analysis significantly reduces the state space of the search, because most non-spontaneous transitions become deterministic. Under many situations, using the relative order checking options will yield linear-time trace analysis executions with respect to the length of the trace, as we explain in Section 4. Therefore, if it is possible to observe inputs after they exit, and outputs before they enter queues, using these options will improve performance significantly.

2.4.3 Disabling an IP

Disabling an IP means that outputs sent through that IP during the trace analysis are not checked, but always considered valid. This feature may be useful when the trace itself did not include output observations made at certain IPs, due to practical problems of observability.

While it is possible with this option to use Tango to perform trace analysis when not all outputs from the IUT are available, all *input* interactions arriving at the IUT are needed for a TAM to perform trace analysis, if they affect the observable behaviour of the implementation. This may be considered a significant limitation of Tango, as there are situations where the inputs arriving at some of the IPs of the IUT are not observable, and it is still desired to perform a trace analysis on the interactions passing through other IPs of the IUT. Section 5 discusses some of the problems involved in implementing partial trace analyzers.

3 On-Line Trace Analysis

When a trace analyzer runs on-line, it receives interactions from an IUT while the IUT is executing. Such a program is expected to be able to verify incoming interactions as fast as they arrive. In addition to the speed issue, on-line nondeterministic trace analysis involves a search algorithm which is more sophisticated than DFS, to prevent cases where the TAM is indefinitely waiting for more input to arrive at a particular IP, while the solution may exist elsewhere in the search tree.

Tango generates trace analyzers which implement a multi-threaded depth-first search algorithm, to provide a means for on-line trace analysis. This section will describe some of the design issues and implementation details, but more details can be found in [7].

The way Tango handles on-line trace analysis is by treating the trace file as a "dynamic" trace file. A dynamic trace file is one that can grow during the trace analysis, while a static trace file is one which does not grow, and therefore, can be loaded into memory before the search begins. At any time, another process independent of Tango can append data to a dynamic trace file, which the TAM must check periodically for more data to read. This should make it very easy to interface a Tango trace analyzer with another program that collects trace data from an IUT.

Hereafter, when a TAM is performing on-line trace analysis, we will say that it is running in **dynamic mode**, to distinguish it from a TAM which is only reading static trace files, which we would say is running in **static mode**.

3.1 Multi-Threaded Depth-First Search

In on-line trace analysis, when a TAM has encountered the end of input interactions for a particular IP, the trace analyzer has two choices. It can wait indefinitely for a new input to arrive, or it can "mark" the current state as a state which needs to be checked again, and continue searching other paths in the tree. The former technique allows one to continue using standard DFS, and may be a reasonable one to use for certain specifications with only one IP, but an indefinite delay is not acceptable if there are interactions to consume and check which are waiting in the queues of other IPs.

```
ip A,B;
state S1, S2;
trans
  from S1 to S1 when A.x name T1:
       begin end;
  from S1 to S2 when A.x name T2:
       begin end;
  from S2 to S1 when B.y name T3:
       begin output A.ack; end;
```

Figure 1: Pseudo-Estelle specification **ack**

Imagine that the TAM is performing on-line trace analysis using our specification **ack** in Figure 1. Suppose that the inputs arrived from our IUT at A and B were [x x x] and [y] respectively, and the only output traced so far was [ack]. Logically, we can see that our IUT at some point decided to take T2 when it consumed one of the x interactions from A. However, if our trace analyzer decided to fire T1 three times, consuming all of the interactions arriving at A, it would arrive at a state in the search tree with no possible next transitions to fire, and the output [ack] would not have been verified, nor would the input [y] have been consumed by the TAM.

At this point, if the TAM were performing regular DFS (waiting indefinitely for new input to arrive) and no new inputs arrived, the trace analysis would deadlock.

If the TAM decided to backtrack and analyze other paths in our search tree, it would validate the trace upon execution of the following transitions: T1, T2, T3, T1. However, in the general case, it is not reasonable to assume that a complete solution exists elsewhere in the search tree, and it is possible that the solution began with the transition sequence which was reached earlier. Therefore, it is necessary to save such states, so the TAM can analyze them again when new input arrives.

This technique will hereafter be referred to as "Multi-Threaded Depth-First Search", or MDFS, and is implemented in the current version of Tango. MDFS is similar to standard DFS except that at certain stages in the search, it might be necessary to save a state, and analyze it again later. Each saved state represents a "thread" in the search, which may lead to a solution at a later time in the analysis. The high-level algorithm is described in the next subsections.

3.1.1 Description of Basic MDFS

If an input queue is empty during the generate operation, this means that from the current state, some of the transitions which may have been fireable if input were available, will not be fireable until new input arrives. In this situation, the transition list is considered "incomplete". Hereafter, a node in the search tree with an incomplete transition list will be referred to as a "partially generated node", or a **PG-node** for short. After all of the possible transitions which were generated from a PG-node are searched, it is necessary to save the PG-node for analysis later.

When the rest of the search tree is exhausted, PG-nodes will be the only ones left to search. The oldest PG-node in the tree will be the next state to be analyzed. A re-generate operation will be performed on the state, to determine if additional transitions are fireable. If additional transitions exist, they will be searched at this point. If some input queues are still empty in this state, the node is still considered PG, and will be saved for analysis again later.

3.1.2 Termination Conditions

As long as a PG-node exists in the search tree, MDFS will never terminate. This is because a PG-node needs to be checked again later to determine if there are additional fireable transitions from that state, arising from the arrival of new input.

For traces which contain no invalid interactions, there will *always* be PG-nodes in the search tree. Therefore, MDFS will never terminate with a valid result.

If one of the PG-nodes represents a state where all inputs were consumed and all outputs were verified, the node is called a Partially Generated All-Verified node, or **PGAV-node** for short. If such a node exists in the search tree, this means that the trace is "valid" so far[2]

The TAM may output an "invalid" result, but this will happen only if all of the possible transition sequences are searched, and no PG-nodes remain in the search tree. This can happen only if invalid interactions exist at points in the trace early enough to prevent the consuming or producing of all available inputs or outputs in one of the queues.

So what does it mean if the TAM is cycling through a set of PG-nodes, none of which are PGAV nodes? None of these states have consumed/verified all of the inputs/outputs, but when new input arrives, there might be more transitions to search. Does this mean that the trace is valid so far?

The answer is "maybe". Consider a specification **ip3'**, which is like the one in Figure 2, except that *only* transitions `t1, t2` and `t3` are defined. Imagine that the trace collected so far contains one input from A, `x`, and one output to A, `o`. The interaction `o` will never be generated by our specification **ip3'**. However, the TAM can still non-deterministically continue consuming and verifying `data` interactions which pass through IPs B and C until no more input and output trace

```
ip A,B,C;
state s1, s2
trans
  from s1 to s1 when B.data name t1:
        begin output C.data; end;
  from s1 to s1 when C.data name t2:
        begin output B.data; end;
  from s1 to s1 when A.x name t3:
        begin output A.p; end;

  from s1 to s2 when B.finished name t4:
        begin end;
  from s2 to s1 when A.x name t5:
        begin output A.o end;
```

Figure 2: Pseudo-Estelle specification, **ip3**

data is available for those IPs. When this happens, some PG-nodes exist, and MDFS will indefinitely cycle through them, waiting for more input to arrive at B or C, even though interaction `o` is invalid. As each new `data` interaction arrives for B or C, it is analyzed and verified, and the TAM continues waiting. In this situation, an invalid trace is *not* detected by the TAM running MDFS.

Now consider the specification **ip3** where *all* the transitions in Figure 2 are defined. Here, we can see that once an interaction `finished` arrives at B, then `t4` is fired, the module enters `s2`, `o` can be verified, and the trace will be valid.

Popular protocols are not usually written in such a way that their implementations will leave inputs in queues for a long time without looking at them. If an input arrives while the IUT is in a state which is not expecting that input, the IUT will usually consume the input and send some kind of result indicating that it was an error. Therefore, situations like the one described above rarely happen, so practically speaking, when only PG-nodes which are non-AV exist in the search tree, this means that the trace is "likely to be invalid", but still, no conclusive result can be given.

It is possible that the operator would like to "force" a termination verdict on the TAM which is executing MDFS, so this feature is supported in Tango, by the use of an "end-of-file" marker in the trace file. Once the TAM is notified that there will be no more data to arrive in any of its dynamic trace files, the PG-nodes in the search tree become fully-generated nodes. At this point, it is possible to exhaust the search tree and report a conclusive result.

3.1.3 Multi-Threaded DFS with Dynamic Node-Reordering

One disadvantage of using basic MDFS becomes apparent when analyzing long valid traces of highly nondeterministic specifications. It is possible that when the end of input is encountered, even if for only one of the IPs, the path from the root of the tree to the current PG-node is a partial solution (that is, part of a full solution, if one exists) for validating the trace in progress. In the case where a PGAV-node exists, it is almost *certain* that the path from the root to that node is a partial solution to the full trace analysis. By taking PG-nodes, placing them on the top of the tree, and forcing the TAM to analyze all of the other possible paths, the TAM might end up searching through a very large tree before getting back to the PG-nodes.

Since search trees of nondeterministic specifications may

[2] It might be acceptable, when a PGAV node is found, to go through the search tree and remove *all* nodes which are not PGAV nodes. In a sense, this is like breaking the trace into smaller pieces, viewing the fragments which were analyzed so far as "piecewise valid" and viewing the remaining parts as unanalyzed partial traces. This would have the effect of reducing the time and memory required for a search. However, it is possible that Tango will give an invalid result on a valid trace, and the frequency of this kind of result depends on the protocol. One might find that the tradeoff is justified in some circumstances.

grow exponentially in size with the length of the trace to be analyzed, this could cause the TAM to spend an inordinate amount of time searching the rest of the tree, which may or may not contain another partial solution, while the path which is most likely to be part of the solution will not be searched until the rest of the search tree is exhausted.

An enhanced version of MDFS incorporates dynamic node-reordering in the search tree, and solves this problem. Any time new input arrives, the search tree is reordered so that PG-nodes are placed at the bottom of the tree, and thus will be searched immediately after the new input arrives, putting the rest of the search tree "on hold". This algorithm was implemented in the current version of Tango.

3.2 Memory Requirements and Performance Problems

3.2.1 Degenerate Cases when using MDFS

Some protocol specifications have multiple IPs of which, during a typical test case execution, not all are in use. In such cases, the unused IPs will have empty queues during the entire search. Therefore, we encounter a situation where *each* state which is generated during the MDFS becomes a PG-node, and thus must be saved, for possible future re-generation. In this case, MDFS will waste all of the available memory very quickly.

If it is known before the trace analysis that no inputs will *ever* arrive at a particular IP, using the `disable_ip` run-time option will prevent this degenerate MDFS case from occurring.

However, if the first interaction passing through a particular IP arrives very late in the trace analysis, or if the input queue for that IP is empty for most but not all of the time, disabling the IP is not an option. Still, most of the nodes searched will be PG-Nodes in MDFS, and saving the TAM state info for each of them will require large amounts of memory. Tango is not well suited for on-line analysis of this particular combination of trace and specification types, and it is suggested that one uses Tango in static mode under these circumstances.

3.2.2 Dynamic Memory in Specifications

The saving and restoring operations on dynamic memory during MDFS require substantially more memory and CPU time than they do in standard DFS, and may cause significant performance degradation during on-line trace analysis. It is suggested that a highly nondeterministic Tango-generated trace analyzer which makes heavy use of dynamic memory be used to analyze static trace files only, or that the specification be rewritten without the use of dynamic memory for generation of an on-line trace analyzer. The reasoning, and the low-level details of dynamic memory state saves and restores can be found in [7].

4 Practical Results

The current version of Tango has been tested on TP0, the "Class 0 Transport Protocol", a specification of an OSI transport layer, for networks with very reliable network layers, and the LAPD protocol, also known as CCITT Recommendation Q.921, for the Link Layer of an ISDN. These experiments were performed on SUN 4 with 32Mb of memory.

One way of measuring the performance of a Tango-generated trace analyzer is in terms of *transitions per second*, or the number of edges searched in the search tree per CPU second. This value depends on many factors, such as the amount of memory used by variables and dynamic records, the frequency of backtracking, and the number of transition declarations in the TAM's specification. For simple test-specifications with under 10 transition declarations, TAMs can search up to 250 transitions per second. For a slightly more interesting specification like TP0 (19 transition declarations), the TAM can search between 40 and 60 transitions per second. However, while analyzing traces of behemoth-like specifications such as LAPD (over 800 transition declarations), a TAM can take a second to search only 10 transitions.

4.1 LAPD

Using the LAPD specification developed at CNET [15], we used Tango to generate an implementation. We executed this implementation 7 times, to collect 7 valid traces. Each trace differs from each other by the number of data packets sent from the User module (layer 3) to the LAPD module (layer 2).

We then generated a trace analyzer based on the same specification, and ran it four times on each of these obtained traces, using different relative order checking options each time. The execution results can be found in Figure 3.

As indicated by our results, trace analysis was significantly faster when we enabled relative order checking options. This is because many nondeterministic choices became deterministic ones, thereby reducing the state space of the search.

One problem we encountered when analyzing LAPD traces is that often, it is desired to analyze only the packets transmitted at the lower interface of the LAPD module, between the LAPD module and the physical line, because the interactions passing between the user module and the LAPD module are not necessarily observable. The current version of Tango could analyze such traces based on a modified protocol specification which includes only the interactions at the lower interface, but another approach to this problem is discussed in Section 5, on partial trace files.

4.2 TP0

The performance of a Tango-generated trace analyzer depends on many factors, such as the length of the trace data, the degree of nondeterminism in the specification, and, in the cases of highly nondeterministic specifications, the "luck of the draw". Often, the time required to analyze a valid trace is proportional to the length of the trace to be analyzed, but the time required to analyze an *invalid* trace where the first n interactions are valid, depends more on the degree of nondeterminism in the specification, and can be exponential with respect to n.

For example, the TP0 specification defines a module which communicates with two other modules, an "upper tester" and a "lower tester". The lower module represents the network layer, while the upper module represents the user layer. When a data interaction from one module is received by TP0, it is saved into a buffer of "infinite" length and, at some later time, sent along to the other module. The specification enters a state known as `data` after the initial handshaking is complete between the modules above and below it. At this point, the upper and lower modules can simultaneously send data to each other. To summarize, from the `data` state, TP0 can do the following:

- `T13:` If available, read a data interaction from the upper module, and place it into `buffer2`.

DI	CPUT	TE	GE	RE	SA
	NR				
5	4.1	34	21	15	17
10	7.6	64	36	30	32
15	11.0	94	51	45	47
25	18.4	154	81	75	77
50	34.4	284	148	138	144
75	52.2	414	215	201	211
100	71.7	579	296	285	292
	IO				
5	2.9	28	19	9	13
10	5.5	53	34	19	28
15	10.9	78	49	29	43
25	16.3	128	79	49	73
50	30.8	237	146	91	140
75	50.7	346	213	133	207
100	62.8	483	294	189	288
DI	**CPUT**	**TE**	**GE**	**RE**	**SA**
	IP				
5	1.6	24	19	5	7
10	3.0	44	34	10	17
15	5.0	64	49	15	29
25	7.7	104	79	25	46
50	13.3	192	146	46	95
75	21.0	280	213	67	135
100	30.2	389	294	95	191
	FULL				
5	0.7	20	19	1	3
10	1.6	35	34	1	8
15	2.3	50	49	1	15
25	3.5	80	79	1	22
50	6.8	147	146	1	50
75	9.5	214	213	1	69
100	12.8	295	294	1	97

Key:

DI	# of data interactions sent by the User module to LAPD module
CPUT	CPU time, in seconds
TE	Transitions executed during search
RE	Restores, or backtracks performed during search
SA	Number of State Saves during search
GE	Number of Generates during search
NR	Relative Order Checking Disabled
IO	I/O and O/I relative order checking only
IP	IP relative order checking only
FULL	All relative order checking options enabled

Figure 3: Execution times of a TAM on LAPD traces of various sizes

- `T14:` If nonempty, send an interaction from `buffer2` to the lower module.
- `T15:` If available, read a data interaction from the lower module, and place it into `buffer1`.
- `T16:` If nonempty, send an interaction from `buffer1` to the upper module.
- `T17:` If a disconnect request is received from the upper module, send a disconnect indication to the lower module.

Imagine a trace to be analyzed which contains the initial handshaking, followed by 20 interactions sent from the lower module and 20 interactions sent from the upper module. To analyze this trace, the search tree depth would be at least 80, because each interaction (there are 40) sent from one end to the other requires the TP0 to read/enqueue (one transition) and dequeue/output (one transition).

During most of the analysis, the TP0 module is in the `data` state, and from this state there will be usually at least two, and sometimes as many as four of the above transitions which are fireable.

A quick calculation will show that if there were, on average, only 2.4 transitions fireable from each `data` state [3], a search tree of depth 80 would contain 2.6×10^{30} transitions. At 150 transitions per second, it could take 4.8×10^{20} years to analyze an invalid trace!

This problem arises from the fact that a trace which has a bad or missing interaction near the end of it gives rise to an exponential number of "partial solutions", each one causing the trace analyzer to search very deeply into the tree before encountering the bad or missing interaction.

For *valid* traces, however, it should be apparent that taking *any* sequence of transitions (T13 through T16) which consume input when available from the IPs, and output interactions when available from the TP0 queues, would eventually consume all inputs and verify all outputs. In other words, there are an exponential number of solutions with respect to the length of the trace, and finding one of them requires no backtracking. Therefore, the search time would be linear with respect to the length of the trace.

A logical question to ask might be: if the order of these transitions does not matter, how can we avoid checking all of the possible permutations? In fact, it is impractical to analyze long invalid traces of specifications such as TP0 without having an answer to this question. Perhaps what is necessary is some form of control and data flow-analysis which would show that taking one permutation of transitions is equivalent to taking a class of others. This would provide a means to "trim" the search tree before or during analysis. It would be interesting to determine whether the methods developed for the verification of specifications [19] are applicable in this context. Another useful approach might be to keep information about which states were reached during the search in a hash table, to prevent the analysis of the same state twice.

The results of executing a TAM on *invalid* TP0 traces are shown in Figure 4. The first trace contains three data interactions sent by the upper tester, and three sent by the lower tester, and was obtained by executing Tango in implementation generation mode. One parameter in the last data interaction of the trace file was edited slightly to cause a mismatch. The same trace was analyzed four times, each time

[3] This is the average fanout in a Tango search tree of depth 13 analyzing TP0

Depth	RCM	CPUT	TE	GE	RE	SA
13	None	1469.5	88329	36687	51642	34440
13	IO and OI	1.3	173	104	69	69
13	IP only	6.7	984	495	489	428
13	Full	0.9	173	104	69	69
21	Full	32.1	4021	2258	1763	1763
29	Full	2658	122202	65575	56627	56627

Depth = Depth of search tree
RCM = Relative Checking Mode

Figure 4: Execution times of a TAM on invalid TP0 traces

using different relative order checking option combinations. After this, longer invalid traces were generated in a similar fashion and analyzed using full relative order checking.

If relative order information on the interactions in the trace file is available to the tester, enabling the Tango relative order checking options will force the TAM to analyze only the transition sequences which have "progress" transitions appearing in the same order as the interactions they consume or produce in the trace. In effect, the TAM will eliminate permutations of observable and input-consuming transitions from the search tree. In the case of TP0, there are no non-progress transitions, but when analyzing traces where only the last data interaction is invalid, there are still some nondeterministic possibilities near the leaves of the search tree. This is because after receiving a disconnect request, TP0 can output a disconnect indication at any time, even if data remains in its buffers. In other words, `t17`, becomes fireable from the `data` state, in addition to the other transitions described above. Enabling full relative checking on these invalid traces reduced the average fanout from 2.6 to 1.5 on the search trees we were able to measure, but it should be noted that the fanout would be very close to 1 if the invalid data interaction was early enough in the trace to prevent `t17` from becoming fireable anywhere in the search tree. Thus, in our example, while the search time is still exponential with respect to the length of the trace, searches are significantly faster, and in the general case, will *usually* (but not always) take linear time with respect to the length of the trace.

5 Analysis of Partial Traces

For the purposes of this paper, a **partial trace** has one or both of the following properties:

1. It begins with trace data from an IUT which is not necessarily in its initial state.
2. It does not contain input interactions passing through one or more of the IPs which are used by the TAM based on the IUT.

Analysis of a partial trace file introduces a plethora of unknowns, making analysis significantly more difficult. In the case where the initial module state is unknown, certain variables will be uninitialized, and in the event that their values are used to determine the behaviour of the TAM, the validity of any such behaviour is questionable.

In the case where inputs passing through one or more of the TAM's IPs are not supplied by the trace file, the TAM must consider all possible transitions which consume any interaction from these IPs. If an "unknown" interaction has parameters, the values of the parameters are unknown. If the values of unknown parameters are used in parameters of output interactions which must be checked, a true "comparison" of these interactions to the traced interactions is not possible. Furthermore, the average number of fireable transitions from each state will be very high, giving rise to an exponential state space growth.

The implementation of a partial trace analyzer generator requires the addressing of the above problems. An approach to analyzing partial traces is discussed in this section.

5.1 Undefined Variables

Since all Estelle variables are translated into C++ objects, adding an "undefined" attribute to each object is relatively straightforward. The constructors of such objects will initialize this attribute to `true`, and all assignment operators must set it to `false` (unless, of course, they are assigned to be equal to other undefined variables or values).

For all transitions which have `provided` firing rules, each boolean expression in the `provided` clause which tests the value of an undefined variable is assumed to be `true`. For the purpose of comparing generated interactions to traced interactions, parameters of interactions with undefined values are "equal" to all values to which they are compared.

5.2 Undefined Input Queues

Undefined queues have the following properties:

- When determining if all inputs have been consumed, an undefined queue is assumed to be empty.
- If a transition has a `when` clause which is true if an undefined IP has a particular interaction in its queue, then the `when` clause is evaluated to `true`. Before the transition can be fired, a new interaction must be created, of the type defined in the `when` clause, with all its parameter values set to `undefined`.
- The actual queue associated with the undefined IP is always empty, and does not need to be saved or restored during backtracking.

5.3 Control Statements

Some features of Estelle make it impossible to perform a full analysis of partial trace files. If we restrict ourselves to a subset of the Estelle language, which does not support control statements, our problem becomes tenable.

Estelle's control statements are `while, for, repeat, case` and `if/then/else`. Each of these statements requires the comparison of a variable to a value, and the execution of different statements depending on the comparison result. If the variable to be compared is undefined, this can mean that multiple possible paths of execution exist. Where loops are involved, these paths may be infinite in number. In theory, a proper trace analyzer must attempt all possible execution paths to search the entire state space, but because the state space is infinite, supporting loops is impractical.

Applying a straightforward transformation of the specification into a "normal form" [16] which eliminates case and if/then/else statements by adding states and transitions to the specification, will simplify the problem of partial trace file analysis, and allow Tango to analyze partial traces of specifications which do use these constructs.

Fortunately, most Estelle specifications make very infrequent use of loops and conditionals, so in theory, it should be possible to perform partial trace analysis on most Estelle specifications without requiring too much in the way of modification.

5.4 Other Problems

There are still some other problems associated with partial trace analysis which must be addressed. For example, DFS will be foiled by the existence of transition cycles which only read input from unobservable IPs. If we were analyzing a TP0 trace where one of the IPs was unobservable, a possible execution path would be to repeatedly fire the transition which reads and enqueues input from that unobservable IP, yielding a search tree of infinite depth. Another problem arises from the use of an undefined variable as an array index, especially in an array of interaction points. This situation could arise during the analysis of a de-multiplexer, or a router, where an input IP is unobservable. In this situation, an incoming undefined interaction parameter specifies the *destination* IP of the next output, which normally must be known for a comparison of generated output to traced output. There may be other problems similar to the ones mentioned above which will make partial trace analysis of some specifications very difficult, if not impossible.

6 Conclusions

In this paper, the steps required to transform an implementation generator into a trace analyzer generator were summarized, and a fully-functional tool called Tango to generate trace analyzers for single-module Estelle specifications was described. The results from using this tool on different protocols were presented, and an approach to analyzing partial trace files was discussed.

Tango provides a means to analyze traces of any single-module protocol specified in Estelle, supporting almost all[4] of Estelle's programming constructs. It is efficient with memory and CPU time, and handles nondeterminism elegantly. At the same time, Tango can be used to generate implementations which behave the same way as those generated by Dingo [17]. The main shortcoming of Tango is its inability to analyze time-dependent behavior in a specification or an IUT.

The main difficulty of analyzing execution traces with respect to a given specification is the nondeterminism of the latter. In this respect, it is important to note that the input and output queues that may be part of the implementation under test reduce the observability and give rise to additional nondeterminism in the order of the observed interactions. Tango provides options for checking this order as much as possible. As our practical applications have shown, the nondeterminism in many practical protocol specifications is limited enough to make backtracking trace analysis efficiently feasible, at least for valid traces. For invalid traces, the analysis is often much more inefficient due to the inherent parallelism which leads to many different interleavings of events to be explored.

An additional difficulty arises during on-line trace analysis, where the analysis is performed while the end of the trace has not yet been reached. This difficulty is due to the fact that new inputs may occur at different IPs during the search, and certain execution paths of the specification may be blocked because of missing interactions at a given IP, while other execution paths may proceed. This makes a pure depth-first search strategy impossible. We have defined a so-called multi-threaded depth-first strategy which is applicable in these cases.

[4] With the exception of delay statements

Acknowledgements

The author would like to thank Daniel Ouimet and Alexandre Petrenko for invaluable discussions. This work was supported by the Hewlett-Packard-NSERC-CITI Industrial Research Chair on Communication Protocols.

References

[1] F. Belina and D. Hogrefe. "The CCITT specification and description language SDL". *Networks and ISDN Systems*, 16, North-Holland, 1988/89.

[2] O. Bellal, G.v. Bochmann, M. Dubuc, and F. Saba. "Automatic test result analysis for high-level specifications". Technical Report 800, Department IRO, University of Montreal, 1991.

[3] G.v. Bochmann, D. Ouimet, and J. Vaucher. "Performance simulation of communication protocols based on formal specifications". *Transactions of the Society for Computer Simulation*, 9(4):201–225, December 1992.

[4] T. Bolognesi and E. Brinksma. "Introduction to the ISO specification language, LOTOS". *Computer Networks and ISDN Systems.*, 14, Elsevier Science Publishers B.V. (North-Holland), 1987.

[5] R. Cork. "The testing of protocols in SNA products - an overview". In *Proceedings of IFIP WG 6.1 Third Annual Workshop on Protocol Specification, Testing and Verification*, 1983.

[6] R. Dssouli, R. Fournier, and G.v. Bochmann. "Distributed observation and FIFO queues". In *Proceedings of the 3rd International Conference on Formal Description Techniques (FORTE 90)*. North-Holland, November 1990.

[7] S. Alan Ezust. *Tango: The Trace Analysis Generator.* Master's thesis, McGill University, Montreal, Canada, 1995.

[8] D. Hoffman and R. Snodgrass. "Trace specifications: Methodology and models". *IEEE Transactions on Software Engineering*, 14(9), September 1988.

[9] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series, 1991.

[10] *ISO Recommendation 9074, The Extended State Transition Language (Estelle)*, 1989.

[11] C. Jard and G.v. Bochmann. "An approach to testing specifications". *Journal of Systems and Software*, 3(4), December 1983.

[12] M.C. Kim, S. T. Chanson, and S. T.Vuong. "Protocol trace analysis based on formal specifications". Technical report, University of British Columbia, Department of Computer Science, 1991.

[13] R. Molva, M. Diaz, and J. Ayache. "Observer: A run-time checking tool for local area networks". In *Proceedings of IFIP WG 6.1 Fifth Annual Workshop on Protocol Specification, Testing and Verification*, 1985.

[14] R. Probert. "Towards a knowledge-based model for conformance test results analysis". In *Proceedings of the IFIP WG 6.1 Fifth International Workshop on Protocol Specification, Testing and Verification*, 1985.

[15] P. Riou. *Specification of the ISDN Link Access Protocol for D-channel (LAPD) CCITT Recommendation Q.921*, Centre National d'Etudes des Telecommunications (CNET). Available by FTP on louie.udel.edu in pub/grope/estelle-specs, 1989.

[16] B. Sarikaya, G.v. Bochmann, and E. Cerny. "A test design methodology for protocol testing". *IEEE Transactions on Software Engineering*, 13, 1987.

[17] R. Sijelmassi and B. Strausser. "The distributed implementation generator: an overview and user guide". Technical report, US Department of Commerce, National Institute of Standards and Technology, National Computer Systems Laboratory, Systems and Network Architecture Division, Gaithersburg, MD 20899, 1991.

[18] B. Strausser and J.P. Favreau. "User guide for the NBS prototype compiler for estelle". Technical Report ICST/SNA 87/3, National Institute of Standards and Technology, October 1987.

[19] A. Valmari. "A stubborn attack on state space explosion". In *Workshop on Computer-Aided Verification, DIMACS 90*, 1990.

[20] S. T. Vuong, S. Lee, and P. J. Zhou. "La validation des tests de protocole: principles, outils et exemples". *Actes du Colloque Francophone sur l'Ingénierie des Protocoles (CFIP), Montréal Canada*, 1993.