National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Tango: The Trace ANalysis GeneratOr

S. Alan Ezust
School of Computer Science
McGill University, Montreal

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of M.Sc in Computer Science.

Gregor Bochmann and Gerald Ratzer, Advisors

Copyright © S. Alan Ezust 1995

Final draft printed on: January 18, 1995

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch
Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-05548-5

Canada

Abstract

This thesis describes the development of an automatic generator of backtracking protocol trace analysis tools for single-module specifications written in the formal description language, Estelle. The generated tool automatically checks the validity of any execution trace against the given specification. 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 discussed.

Ce mémoire décrit le développement d'un générateur automatique d'outils pour l'analyse de traces de protocoles de communication non-déterministes, décrits par des specifications formelles Estelle à un seul module. L'outil généré vérifie automatiquement la validité d'une trace d'exécution par rapport à la spécification de référence. L'approche suivie consistait en la modification d'un compilateur Estelle-C++ existant. Une discussion a propos de specifications non-deterministes, de points d'observation multiples, et d'analyse de traces à la volée est presentee par la suite. Des analyseurs de traces pour les protocoles LAPD et TP0 ont été testes, et leurs résultats de performance évalués. Enfin, quelques points reliés à l'analyse de traces partielles sont discutés.

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

Contents

Chapter 1

Introduction

Protocols are sets of rules that govern the interaction of concurrent processes in distributed systems. The design of protocols is related to a number of established fields, such as the design of operating systems, computer networks, and distributed databases.

Typically, books discussing networks, operating systems and distributed databases present protocols which have been accepted as correct by, for example, a large international organization. They rarely explain why the protocols work, what problems they solve, or what pitfalls they avoid [23]. The process of deriving such protocols in the first place, however, is a very involved one, which encompasses different stages in development and testing. Furthermore, when a protocol is accepted as correct, or free of certain kinds of faults, implementing software that follows it precisely, or testing existing software for conformance to the protocol, are two quite complicated steps in systems development.

There are several ways to ways to specify a protocol. A natural-language specification can be easily readable by a human who requires a general understanding of how it works, but it can be imprecise or ambiguous. Such specifications can make the development of an implementation more difficult, or more prone to bugs, which

may cause the implementation to be incompatible with other implementations that claim to follow the same protocol.

FDTs, or **Formal Description Techniques**, are used to specify software in a very precise way. Using a specification language to this end, with very strict syntax and semantics, reduces the problems of imprecise or ambiguous specifications. Specification languages typically resemble high-level programming languages, but they discourage the definition of low-level details which are specific to a particular platform. Specification languages pave the way to automated implementation generation and conformance testing.

The **trace** of a communication program is a record of its inputs and outputs during its execution, which might have been captured on the communication line and saved to a file. The internal behaviour of an implementation is typically hidden from a protocol tester, so in this way, the implementation is viewed as a "black box".

Trace analysis is the act of comparing the observable behaviour of a running implementation to that of its protocol specification. It is one step in protocol conformance testing, and usually involves a simulated execution of different parts of the specification. Formal specifications are especially useful in the context of trace analysis.

Usually, a trace analyzer is based on a specific protocol. An automated technique of generating a trace analyzer based on a formal protocol specification would make it easier to ensure that the trace analyzer follows the specification exactly. Such a technique would considerably facilitate this part of protocol conformance testing.

This thesis chronicles the work done towards the development of a trace analysis tool generator for specifications written in the FDT, Estelle. The approach taken was to start with an Estelle-to-C++ compiler, called Dingo, developed at NIST, and to add all the necessary routines to turn it into a generator of backtracking trace analysis tools. While Estelle is by no means the most popular FDT in use today,

we chose this approach because it was deemed much easier to start with an existing implementation compiler and modify it to suit our needs than to start from scratch, and we were unable to obtain the sourcecode for a similar object-oriented compiler for SDL, a much more popular FDT with similar features to Estelle. However, the principles and difficulties of trace analysis discussed in this thesis apply as well to trace analysis with respect to specifications written in SDL, and a tool does exist which translates SDL specifications into Estelle [41].

Background information and related research on FDTs in general, Estelle in particular, automatic implementation generation techniques, protocol verification, and protocol conformance testing are covered in this thesis as well.

Chapter 2

Formal Specifications

There are several ways to specify the behaviour of a protocol. A natural-language specification can be easily readable by a human who requires a general understanding of how it works, but it can be imprecise or ambiguous. Such specifications allow for different human interpretations of the behaviour during protocol development, and make formal conformance testing impossible.

FDTs, or Formal Description Techniques, are used to specify software in a very precise way. Using a specification language to this end, with very strict syntax and semantics, reduces the problems of imprecise or ambiguous specifications. Such languages leave out the machine-dependent details, but include all the necessary information about the data exchange methods, the timing of events, and valid message criteria.

It is beneficial to write specifications in FDTs because such representations simplify the problems of design and validation [5]. In addition, there are automatic ways of producing executable implementations based on such specifications [6].

This chapter will discuss three different approaches to modelling a protocol. The first one is the Extended Finite State Machine (EFSM) model, which is enforced by languages such as SDL (Specification and Description Language) [1] and ESTeLle

(Extended State Transition Language) [34]. Code written in these languages will have state and transition definitions, with programming extensions such as variables and procedures to make the specification easier for a human programmer. They will often look like high-level procedural languages.

The second model was intended make the passing of messages between communicating processes the most prominent feature in such specifications. Based on *process algebras* [21] [24], the specification language developed within the ISO (International Organization for Standardization) is known as LOTOS (Language of Temporal Ordering Specification) [11]. Program structure, and even flow of control in LOTOS can be compared to what is available in other functional programming languages, except that there is no backtracking.

The third model discussed here is the object-oriented model. Object-oriented organization is quickly becoming the preferred model for data representation in many different areas of computer science. Intuitively, it is quite easy to organize a protocol specification in terms of its components and the methods associated with them. This is the kind of organization enforced by languages such as SDL92 and MonDeL (Montreal Description Language), a specification language developed jointly at CRIM and BNR [3].

For each language discussed in this chapter, only the basic features and some of the major differences will be described. A rigorous comparison of the advantages and disadvantages between using Estelle, LOTOS and SDL can be found in [13]. Additionally, [8] details the formal specification of a simplified transport protocol in each of these languages, comparing the differences between them from a very practical perspective.

The specification languages described in this thesis were developed with the intended application being telecommunications software, but most of them are robust enough to be used to specify the operation of other types of layered, distributed or

concurrent software, such as operating systems and distributed databases.

While they are outside the scope of this thesis, there are other important FDTs and associated tools, some based on Petri Nets and their extensions, others on logical programming languages, used in many large scale projects. In addition to these formal techniques, organizations such as the OSI standardization committees use semi-formal languages (which lack formally defined semantics) such as TTCN [35], the Tree and Tabular Combined Notation, for specifying behaviour, and ASN.1 [30], Abstract Syntax One, for specifying data structures, of communication protocols.

2.1 Extended Finite State Machines

A finite state machine (FSM) is an abstract model consisting of a finite number of states, a finite number of input symbols, and a finite number of output symbols. From each state, it is possible to take a transition into another state, depending on the available input and the "firing" rules of a transition (the rules that govern when a transition can be taken, as specified in the FSM). The reader is assumed to have a basic knowledge of finite state machines. More information on this topic can be found in [24].

Extended FSMs have extensions to the FSM model such as variables and dynamic memory, and programming constructs which can be used to manipulate their values. EFSMs support spontaneous transitions, which can be taken from a particular state regardless of the input, and they also permit nondeterminism, where different transitions can be fired under the same situation from the same state. An EFSM *state* is a composite of the FSM state, and the values of variables and dynamic memory.

Describing a protocol using the EFSM model involves first subdividing the system into a number of communicating modules, or entities, such that each module is an EFSM [5].

One abstract communication mechanism between modules is known as "direct

coupling", where the firing of a transition in one of the modules causes a transition to fire in other modules. Such transitions are executed in parallel. This kind of inter-module communication is not very common, as it does not reflect the way most real communicating components interact. If a transition is not coupled, it can execute independently from the other components.

Another communication mechanism is achieved by using communication pipes, or "channels" as they are called in Estelle and SDL. The message-passing scheme used in both of these languages is asynchronous, meaning that there is queuing and buffering between communicating components. Information received from such a pipe is considered "input" to the module. Similarly, information sent through a pipe is considered "output" from the module.

In a specification for a multi-layered protocol such as TCP/IP, each layer would be represented as a module, with a channel that connects it to each neighboring layer. For example, the module which specifies the Transport layer would have one channel to the application layer module, and another channel to the Internet layer module.

The EFSM model is used in FDTs such as SDL or Estelle. In general, specifications written in SDL or Estelle look very much like programs written in high-level languages, and can be translated into other high-level languages quite naturally.

2.1.1 SDL

SDL development began in 1972 at the CCITT (International Consultative Committee for Telephones and Telegraphs), and the first version was issued in 1976. One of the more recent versions, SDL88, is accepted as the current standard, and contains many features and extensions which are not supported in the older versions [1]. SDL's syntax is modeled after a programming language called CHILL, recommended by CCITT.

An SDL system specification is composed of block descriptions which are com-

posed of process descriptions. The structures are used to produce *system descriptions*, regarding the system as a black box and describing the external observable behaviour only, and *internal structures*, which can be compiled into high level languages automatically [1].

SDL/GR is a graphical representation of SDL, where the states and transitions are represented as a graph of shapes and arcs, complemented by a textual syntax which supplies additional information. Programming in SDL/GR requires a graphical programming environment.

SDL/PR is the phrase representation of SDL, which will allow the programmer to represent all of the structures as text. Generating SDL/PR from SDL/GR is straightforward, although the transformation in the opposite direction requires graphical layout information. According to [1], SDL/GR environments offer a high degree of user-friendliness which greatly expedite specifications in SDL.

The logical flow of control is represented as a flow-chart based on the internal structure. Time constraints are specified by a *timer* construct. Input and output functionality is supported by *channels* over which signals can be sent between processes. Components, or processes, can be dynamically created and destroyed at "interpretation time" (the analog to "runtime" in specification execution) with constructs *create* and *stop*, the latter construct allowing processes to self-terminate (there is no primitive which allows one process to terminate another). The *procedure* construct is similar to the one in Pascal, and also looks very much like a process description in the SDL code, in the sense that it is also an FSM. However, the calling process suspends for the duration of the execution of the called procedure, while a newly created process will execute concurrently with its parent process.

Abstract Data Types are also supported in SDL, allowing the low level details of system implementation to remain as "black boxes" without their behaviour being specified. This solidifies the ability of SDL specifications to remain machine-

independent.

There are features offered in SDL88 such as inheritance and generators (similar to *templates* in C++) which are typically associated with object-oriented languages.

2.1.2 Estelle

A less commonly used language also based on extended finite state machines is known as Estelle, the Extended State Transition Language. Initial development began in 1981, leading to the first formal release in 1987. Defined within ISO, Estelle is a language intended to look very much like Pascal, with extensions to allow definitions of states and transitions. It was designed for specifying distributed systems in general, and communication protocols in particular [12].

A system specified in Estelle consists of *module instances*, where each module represents a concurrent process, as an FSM. A module has a collection of states, transition blocks, variables, procedures, and *interaction points*, which when connected to other interaction points (presumably in other modules), represent connections between them. Information can be passed through channels by way of the interaction points.

Variables and procedures can be declared with scoping rules in the same way as they are in Pascal. Variables can be sharable between module instances, a feature which exists as a convenience to the programmer, but can yield specifications which can be implemented on real systems only with great difficulty, if at all, depending on their capabilities.

Estelle never gained quite as much popularity as SDL, and probably never will, because the development of Estelle programming tools has lagged behind similar developments in SDL. Moreover, SDL has been revised a number of times to make it more object-oriented. However, Estelle is a very clear, concise specification language which does not take very long to learn.

2.1.3 Promela

Promela is a specification language developed at AT&T by Holzmann [23]. In contrast to Estelle, which uses Pascal syntax and semantics whenever possible, Promela uses C syntax and semantics whenever possible.

2.2 Temporal Ordering

2.2.1 LOTOS

LOTOS, also developed within the ISO, allows one to specify systems by defining the temporal relations among the interactions that constitute the externally observable behaviour of a system [11]. Other LOTOS facilities allow the description of data structures and value expressions, based on the formal theory of ADTs (abstract data types).

A LOTOS specification is divided up into units called *processes*, and within each process, there can be a list of *actions*. A process can consist of several sub-processes, so in general, a LOTOS specification is a hierarchy of processes. Some types of actions involve more than one process, such as the sending of a signal from one to another. These are called *interactions*.

Signals are passed between processes through *gates*, which can be explicitly declared as hidden from other processes, to help carry on the black box paradigm. The message passing scheme, unlike SDL or Estelle, is synchronous, or "rendezvous", meaning that there is no queuing of inputs and outputs, but synchronization is required with every interaction. Using rendezvous channels is beneficial because it allows for a more complete specification of interfaces, which would be impossible in Estelle or SDL without including implementation details [7]. Use of asynchronous message passing may also lead to message cross-over in queues associated with the interface, an undesired effect in real-time systems.

Actions can be *observable*, or *unobservable*. The unobservable actions can not be observed by other processes not directly involved in the action. Actions can be combined in a parallel composition, so that they are meant to be executed concurrently. They can also be explicitly synchronized with other actions.

Representation of values or structured data is achieved with abstract data types, which allow one to specify the type of data which is being stored, but not its internal representation. The ADTs of LOTOS are much richer than the ADTs of Estelle or SDL, and are derived from ACT ONE, a specification language for abstract data types [16]. In general, almost every possible implementation-specific detail is abstracted in a LOTOS specification, while specifications written in Estelle or SDL tend to be more implementation-oriented [7]. More details on the ADTs of LOTOS are outside the scope of this thesis, but are available in [11].

Even for an experienced programmer, it is impossible to understand LOTOS specifications without first being familiar with the meanings of the operators. The code is very terse and laden with symbols which form an integral part of the language grammar. However, it is said that once the user has gained a familiarity with the operators, he/she can specify a system in a very natural way, which reflects quite directly the system's structure and behaviour [11]. According to [7], LOTOS has relatively few (compared to SDL and Estelle), but powerful language constructs which make the learning of the complete language easier.

2.3 Object-Oriented Specifications

While Object-Oriented representation of systems and software is not ideal for every application, it is perfectly suited to the area of telecommunication. The objects can represent communicating entities, and message passing between objects is an integral part of an object-oriented language. Information hiding, inheritance, and persistent objects are all useful constructs for building protocol specifications, as well as for

maintaining a certain degree of abstraction.

Another advantage to using objects is that they can represent FSMs in quite a natural way. The machine can be an object, which has an attribute called a "State", and methods to handle each type of incoming interaction.

2.3.1 Mondel

Mondel, unlike the other languages mentioned in this chapter, is not a standard ISO or CCITT FDT. It was developed as a research project between CRIM (Centre de Recherche Informatique de Montreal) and BNR (Bell Northern Research). It was developed because, in the opinion of the developers, none of the existing languages supported concurrency, object-oriented representation and persistent objects while also meeting the requirements for writing system descriptions at the specification and design level [3].

Mondel is an *executable* specification language. A Mondel specification is not as abstract as a LOTOS one, and therefore more explicitly reflects the operation of the system it specifies. Communication among objects is synchronous, achieved through remote procedure calls (RPCs) with return parameters. Persistent objects are accessed through database queries and atomic transactions. The language syntax and semantics were formally defined with the design goal of expediting the implementation of Mondel compilers, and the partial verification of Mondel specifications. Full multiple inheritance, strong typing, and support for assertional specifications of object properties make Mondel a unique and powerful language [3].

A reflective extension to Mondel, called RMondel, is also available. A reflective programming language is one which allows methods of an object instance to be added or removed at runtime.

2.3.2 SDL92

SDL92, the most recent version of SDL, is the object-oriented extension of SDL88 [17]. It is backwards compatible with SDL88, with some very minor exceptions, and supports user-defined operators, export/import variables and procedures, and all of the expected object-oriented programming constructs. A tool has already been developed which translates from SDL92 to C++ [18].

Chapter 3

Estelle

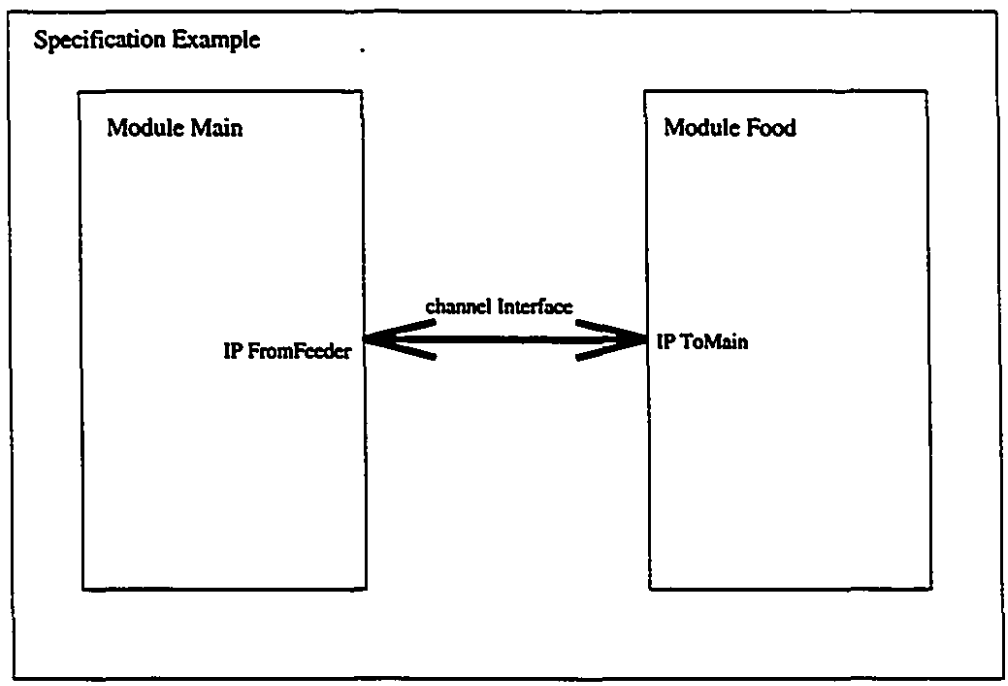

Figure 3.1: Diagram of modules comprising the TriState specification

One of the best ways to learn a new language is by example. This chapter will give an introduction to the syntax and semantics of Estelle by presenting a simple example specification which describes two modules and how they communicate. Further

details on the language can be found in [25].

3.1 The TriState Specification

Figure 3.1 illustrates a system of communicating entities connected by channels. The formal Estelle specification, which is called `TriState`, can be found in Appendix A.1. Please refer to this appendix for line number references. This main module of this specification has three states, and provides us with a very simple example which we will use to illustrate some features of Estelle, and later in this thesis, the features of Pet/Dingo and Tango as well.

In Estelle, `specifications` and `modules` are analogous to Pascal programs and procedures respectively, and are declared in the same way. `const`, `type`, and `var` declarations, scoping rules, and most other Pascal constructs are available in Estelle as well. The important syntactical difference between Estelle and Pascal is that in Estelle, all statements must appear inside transition blocks. A transition block is a compound statement executed as a single atomic operation. The main specification and each module body consists of an `initialize` transition, followed by any number of additional transition blocks, followed by the keyword `end`. This is in contrast to a Pascal program which has a main body consisting of just a single compound statement.

Modules use *interaction points* (or **IPs**) to communicate with other modules. A chunk of information sent through an IP is called an *interaction*, and interactions are structured data types, which can contain a parameter list of valid Estelle data structures.

The `channel` declaration on line 10 declares and groups the different kinds of valid interactions by *role*. The role of an IP determines which group of interactions can be transmitted, and which can be received. The roles for our channel called `interface` are `receiver` and `sender`. The `data` interaction can be sent through a `sender` IP,

or received from a `receiver` IP. Conversely, `data_response` and `close_connection` interactions can be sent through the `receiver` IP, and received through the `sender` IP.

The `module` declaration for `Main_type` is on line 72. A module specifies a finite state machine, with states and transitions. This module *class* is specified by an attribute **systemprocess**, and this means that it is a separate communicating process in the specification. The possible Estelle module classes are **systemprocess**, **systemactivity**, **process**, and **activity**.

Since modules can be nested just like Pascal procedures, it is possible to specify a hierarchical tree structure of module definitions. A module A nested inside another module B is considered the *child* of B. The parent of all modules is the **specification**, typically declared on line 1 of most Estelle specifications. The **specification** can also be declared with a class attribute.

The way module instances behave with respect to each other is dependent on the way they are nested and attributed. Modules attributed with **systemprocess** or **systemactivity** are referred to as *system* modules, and specify separate communicating systems within the specification. Modules with attributes are supervising managers of their children instances, and since a system module can not have attributed parents, this means that no supervising control may be imposed on one from its parent module. A **systemprocess** attribute specifies a synchronous parallel module, where the child modules of class **process** all execute transitions in parallel, while a **systemactivity** specifies a nondeterministic module, where the child modules of class **activity** execute transitions without synchronization with respect to each other. More information about module class attributes can be found in [25].

The `IP` declaration appears right after the module declaration, and defines an interaction point called `fromFeeder` which is connected to a channel on the `receiver` end.

On line 89, the valid states are listed for the `Main_type` module.

The transitions for this FSM begin on line 92 with the `initialize` transition, which sets the state to `Liquid` and initializes some variables. Line 105 defines a spontaneous delay-transition: any time the state is `Nonliquid`, the FSM goes to the `Liquid` state after 3 seconds, and sends a `data_response` to its IP.

The other transitions are taken depending on what is available to be read from the input queue of the IP. On line 115, the transition rules for `toGas` are specified. They are described below:

- There is an interaction which has not yet been "consumed" from the `fromFeeder` interaction point, and it is a `data` interaction.
  (This is the `when fromFeeder.data` clause).

- The `I` field of the parameter is greater than 0.
  (This is the `provided parameter.I > 0` clause).

The transition `_toSolid` is similar, except that it is taken when the `I` field is less than or equal to 0. Finally the transition `_ToFinished` will be taken only when the `I` field is equal to 99. Notice however, that this is a nondeterministic transition. Since the `I` field is also greater than 0, the FSM might enter the `Gas` state instead of the `Finished` state. This is an example of a "bad protocol", in the sense that implementations strictly based on this specification may have problems communicating with each other, but the nondeterministic aspect of it provides us with an illustrative example. An implementation which conforms to this specification can send a `data_response` or a `close_connection` message when it receives a data interaction with `parameter.I` set to 99.

The `Feeding_Module` has a similar structure to `Main_Module` except that it sends `data` interactions to its `sender` interaction point, and waits for `data_response` interactions after each one. After 10 `data` interactions, it sends a final `data` interaction

with the `parameter.I` field set to 99, to signal that it is done. Since there are no transitions from the `done` state, this process will deadlock after taking this transition.

On line 142, the `modvar` declaration begins. These variables are declared in the scope of the main specification, rather than in the scope of one of the modules. If a modvar is viewed as a pointer to a module, the `init` statement is analogous to a `new` statement for modules. The call on line 148 causes a new process to be spawned, of type `Main_type`, and the code which it must execute is the body defined as `Main_body` (there can be multiple body definitions for each module type). Unlike a procedure call, `init` statements execute immediately and flow continues to the next statement in the calling block.

The `connect` statement on line 150 defines how the interaction points are connected. Semantic rules of Estelle state that only IPs for channels of the same type but of opposite roles can be connected together. After the initialize transition has been executed, the specification's main process exits, and the two newly spawned processes can communicate freely with each other.

Section 4.2 describes how this specification is translated into C++. Section 4.3 describes how it would execute under Dingo.

Chapter 4

Implementation Generation

Because of the strict syntax and semantic rules inherent in formal specifications, it is not much more difficult to write an automatic implementation generator (also known as an FDT compiler) than it is to write a compiler for a structured high-level programming language. There are different issues and problems which come up when designing an FDT compiler, however.

One problem is that the parallelism in a specification must be accurately reflected in an implementation. Some compilers, when given a specification for multiple independent processes, generate a single-process implementation that schedules the actions to be performed by each module, using heuristics when no synchronization is required. Others generate a program that spawns other processes which communicate with each other through IPC (Inter-Process Communication) for synchronization. Others will simply not implement certain module systems for the sake of simplicity, and leave it up to a programmer to hand-code this aspect of the implementation.

Since timing of events is crucial in many high-speed protocols, an FDT compiler must generate code which will respond to events fast enough, and with enough accuracy, to actually follow the specified behaviour. Often, implementations for high-speed or time-crucial protocols are written manually by programmers because auto-

matically generated implementations are less efficient with CPU time.

Another problem with FDT compilers is that specifications such as the one given in Section 3.1 do not specify low-level details, such as how structured data are encoded and transmitted through channels. Therefore, some consistent way of implementing the physical layer might be supplied by the compiler (as is the case in Dingo), or else must be added manually by a programmer.

Some Estelle compilers which are strong in some areas impose restrictions on the structure of Estelle specifications. Effectively, these are compilers for subsets of the Estelle language. The introduction of [37] mentions a few such compilers which were available before Pet/Dingo was released.

4.1 NIST Integrated Estelle Compiler

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 [39], 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 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.

4.1.1 Pet

PET, or the Portable Estelle Translator, is written C++ and Bison. Bison is a parser similar to the Unix 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.

Pet's C++ class definitions include a hierarchy of def classes. This class hierarchy contains a subclass to describe each possible Estelle construct, and is based on an early definition of Smalltalk [38].

Each def subclass instance represents an identifier declaration or a statement, and contains information about its lexic level, attributes, consistency check functions, and a linked list of its related components, which are pointers to other def classes. Some examples of def subclasses are: `Attribute` for module attributes, `ComGroups`, for comments, and `TypeDef`, for type definitions. The `TypeDef` class would have a linked list of `Decl` subclass instances, one for each type declaration.

When Pet reads an Estelle specification such as TriState, Pet first creates an instance of a def subclass for the Specification declaration, and this is treated as the "root" object in this representation. The root object has a related component list consisting of the def objects representing the definitions of type, interface, feeding_module, main_type, modvar, and each of the specification's transitions. The feeding_module definition object has its own component list with pointers to def objects for the type and variable declarations local to its module. Each typedef object has a component list for each of its type identifiers.

When Pet is finished reading the Estelle specification, it will have in memory, a tree of def objects containing all of the static information that is described by the original Estelle specification. The leaves of this tree are the most simple definition types, which have empty component lists.

Class `StoreObject` will represent a structured object in a storable form, and `DefStore` will store such an object as a file. Pet's class library also includes an object, `restoreFrom`, which will restore the def tree from a file created by `DefStore`, and

this class is used by Dingo for the next phase in code-generation.

4.1.2 Dingo

DINGO, or the Distributed ImplementatioN GeneratOr, can be thought of as the second pass in the code-generation process. The executable program Dingo reads the output of Pet into memory, organized as the same def tree which Pet stored. By traversing the graph of pointers and objects, Dingo generates C++ code to define objects based on each definition in the tree.

The C++ code, after it is compiled, must then be linked with Dingo's Estelle runtime library to generate an executable implementation. This runtime library contains an X Windows graphical interface, which permits the user to examine variable values, a log of transitions taken by each module, and information about each module's current state. In addition, this runtime library contains base classes of certain objects from which Dingo-generated objects inherit, to define the generic aspects of an executable Estelle-based implementation. The code generation program and the run-time library together will be hereafter referred to as the Dingo system, or Dingo for short.

Generic Aspects of Dingo-generated objects

In the Dingo Estelle run-time library, base classes describe generic aspects of most Estelle constructs. For example, a generic Estelle interaction is a chunk of data of unknown structure, but the Estelle specification contains the type declarations which specify the exact structure of each interaction type. Some aspects of an interaction are common to all instances, such as the "point of entry". Other aspects depend on the interaction, but still must be present in all subclasses, so these methods are virtual (defined for the superclass, but re-defined for each subclass). Examples of such methods are: `readParsFrom`, a method which reads a stream of printable characters

and interprets the information as the proper fields of a particular interaction, as well as `printOn`, which will send all the information in an interaction to a stream as a sequence of printable characters. The superclass of all interactions in Dingo is called `_Interact`. Another example of an important superclass is that of the simple interaction point, class `_SIPtype`, which has a queue associated with it, information about the channel to which it is connected, and methods to enqueue and dequeue interactions. The routines for transmitting interactions to the interaction point on the other end of the channel vary depending on where the other module is executing (it could be part of the same CPU process, or another process on the same CPU, or another process on another CPU), so these are inherited methods as well.

To support the different module classes in Estelle (see Section 3.1), each module class is implemented as a subclass of the `_MInstance` class. These subclasses are `_System`, `_Process`, and `_Activity`, while `_System` itself has two subclasses `_SysProcess` and `_SysActivity`. Each module definition in an Estelle specification is defined as the appropriate sub-sub-class of `_MInstance`, depending on its module class attribute.

Each block of code (specification, module body, function, procedure, or transition) as well as each structured object (such as an interaction), can have a block of memory for variables which are accessible within its scope. Each block of memory is called a *frame*. Frames get pushed and popped off of a GRM, or Global Reference Manager, as the execution enters and exits these variable scopes. The GRM (class `_GRManager`) is an object which manages an internal stack of frames, accessible through a method `getFrame` which answers requests for pointers to variables, given the proper frame index information and variable names. The proper frame index required for each call to `getFrame` can be obtained at implementation generation time.

Each Estelle transition is translated into a C++ function of the same name. A transition scheduler called `selAndExec` is generated for each module instance.

The scheduler is a sequence of complicated conditionals which strictly depend on the original specification, and when called during runtime, will select one fireable transition and execute it. An example of `selAndExec` appears in Appendix A.2, and its behaviour is described in sections 4.2.2 and 6.2.3. It is generated by the function `defSelExecFunct` in the file `cxxmod.cxx`.

4.2 The TriState Implementation

A slightly modified version of Pet/Dingo was run on the TriState specification which appears in Appendix A.1, to generate an implementation. A piece of the generated code appears in Appendix A.2. Subsequent line number references apply to the text in that listing.

4.2.1 Transition Code

Observe the functions `_Init_Trans` and `_ToSolid` which correspond to the Estelle transitions of the same names. They are defined on lines 1 and 21 respectively.

Before the Estelle transition statements can be executed, a frame must be declared which contains a structure of pointers to the local variables for this transition. In the case of `Init_Trans`, there is one local variable called `I` of type integer. In the implementation, the variable is declared local to the function, but then a pointer to it is placed in a frame structure which was generated for this particular transition. The `_GRM->enter` statement pushes this frame onto the GRM stack, so that the variable `I` can be accessed from the GRM in other blocks nested in this scope (in this case, there are none), or restored when a recursive function call returns. This is necessary for many reasons, the most important one being that functions can not be nested in C++, while in Estelle, they can.

Variables local to the module `Main_Body`, such as the record `V`, are in the parent

scope of the transition block. Since this transition needs to access V, it makes a call to the GRM getFrame method, as shown on line 8, to obtain the proper address of this record. The local variable called V becomes an alias to the element V in the frame of the parent scope, which is of course, on the GRM stack.

After the variable scoping problems are resolved, the transition code, consisting of Estelle statements which were translated into C++ (a simple process which involves not much more than replacing Pascal operators with C++ operators) can be executed. When these statements are complete, the frame is popped by the statement `__GRM->leave()` on line 18.

4.2.2 Selecting a Transition

The method `_selAndExec`, which begins on line 39, looks fairly complicated, but is relatively simple compared to the _selAndExec methods which can be generated from more interesting protocol specifications. It will execute a transition if the timers agree, and if the transition is "fireable" (that is, if all the transition conditions are true).

If a transition is fireable, a part of code is executed which sets up the proper frames and interaction pointers, as shown on lines 67–71, 84–92, 109–116, and 134–141. `_transBlock` is a pointer to a function, and gets assigned to the location of the proper transition function. Before the transition is executed, a call to the method `_wantToFire` is made, which returns 1 with a probability of $1/N$, where N is the number of fireable transitions at that moment.

Finally, the `_Exec` block (line 150) is reached when a fireable transition wants to fire. Frames get pushed onto the stack, and a call to `_transblock` is made (line 159). The state is updated, the consumed interaction is deleted from memory, the GRM frames are popped, and selAndExec returns.

4.3 The TriState Execution

After a C++ implementation is generated by Dingo, it is possible to compile and execute it. Information provided by a Dingo-generated implementation after execution includes a log of all transitions taken during execution, and a *trace*[1] of all interactions which were sent through interaction points. All line numbers in this section refer to text from Appendix A.3.

The trace of `Feeding_body`, starting on line 3, shows all of the interactions which were sent from this module. Each entry is 3 lines long. The first line of each entry begins with >> followed by the module name. The second line specifies the name of the interaction point and the name of the interaction, separated by a colon. The third line is a text representation of the information held in the interaction parameter list. This is a structured type, which can consist of other structured types. Each structured type is enclosed in {curly brackets}. More information about the trace file format can be found in Appendix C.2.3.

In our example, the `data_type` which is used for sending interaction data, consists of a 10-element array of integers `H`, and three simple-type fields. They are: an integer `I`, a boolean `J` and a character `K`. Booleans appear in trace files as either 1 or 0. Character data appear as decimal integers representing their ASCII codes. We can see on line 5 that the first data packet sent specifies that `H` contains even integers from 2 to 20, `I`, an integer value of 0, `J`, a boolean true, and `K`, the ASCII character number 1 (Ctrl-A). The last interaction sent from `Feeding_body` is listed on line 36 and shows that the `I` field has the value 99, signalling the end of data packets.

The trace for `Main_body` begins on line 42, and shows that 11 `data_response` inter-

[1]The trace file generation routines were never present in Dingo, but were added to Tango during the early stages of development. Now, when Tango is given a normal Estelle specification to compile, it generates an implementation which behaves the same way as a Dingo-generated implementation, but also produces a module trace.

actions were sent through the interaction point `fromFeeder`. This is the correct number of responses, since 11 data interactions were received. The log files clearly show `Feeding_body` taking alternate transitions `send_packet` and `finished_waiting`, while `Main_body` is oscillating between states `gas`, `liquid` and `solid`. The final transition taken by `Main_body` was `ToFinished`, and a `close_connection` was sent to the IP, although it could have just as easily been a transition `toGas` as the specification is nondeterministic and can take either transition when the `I` parameter is 99 (see Section 3.1).

The trace obtained from this execution will later be used as test data for trace analysis, but it could also be used as a test case for an IUT.

Chapter 5

Protocol Verification and Conformance

Communication software, like most large pieces of software, goes through a development life cycle which resembles that of software engineering. Protocols, however, need to be tested much more rigorously than other more popular, traditional programs, because rather than interfacing with humans, a piece of communication software interfaces with another program. Slight deviations in behaviour which might be tolerated by a human user can cause major communication problems between software.

Communication protocol verification and conformance testing each comprise a stage in the communication software development life cycle. The first, *protocol verification*, typically performed during the development of the specification, is used to verify that certain properties of correctness hold in the specification, such as exhibition of desired behaviour, proper handling of invalid input sequences, and lack of deadlocks. The second, *conformance testing*, applied during the implementation of the software, involves testing conformance of the implementation to the specification.

5.1 Verification Methods

There are several approaches to protocol verification. They can be grouped under two categories: *dynamic analysis* and *static analysis* [7].

5.1.1 Dynamic Analysis

Dynamic analysis can be classified as either *exhaustive* or *simulative* [7]. The most popular exhaustive method is called **reachability analysis**, especially useful for FSM-based FDTs. It involves exhaustively exploring all the possible interaction sequences of two (or more) FSM-based modules in a protocol specification. A composite, or global state of the system is defined as a combination of the states of each module involved. From a given initial state, all possible events are generated, leading to a number of new global states. This process is repeated for each of the newly generated states, until no new states are generated. For FSMs, this is a finite process, since there is a finite number of possible global states. This method determines all of the possible outcomes that the protocol may achieve.

Reachability analysis is useful for detecting situations where the processing of a receivable message is not defined, or where the transmission medium capacity is exceeded. Deadlocks (global states with no exits) are easy to catch as well. [43] provides more detailed descriptions of reachability analysis techniques. They are, however, difficult to apply to some EFSM-based specifications of the size and complexity found in most practical applications, because the information comprising an EFSM state can include variables, as well as dynamic memory, making the number of true "states" infinite. Even without supporting dynamic memory, EFSM-based reachability analysis has huge memory and processing requirements.

Simulation analysis restricts the verification to only selected paths among the possible executions. Simulation is useful for real-size specifications, as the memory and processing requirements for simulation are not as great as those of reachability

analysis. The process of deciding which paths to simulate can involve random or probability-based exploration.

5.1.2 Static Analysis

Some tools exist which perform a static analysis of the text of the specification. These tools are useful for finding clerical errors related to scope rules, type conformance, and other semantic conditions. Compilers which translate from a formal specification language to another high-level language often perform this kind of analysis as part of the code generation process. Some forms of static flow analysis are also possible but are limited in utility compared to reachability analysis.

Another approach, program proofs, involves the formulation of assertions which reflect the desired correctness properties of a protocol. Sometimes, these properties are supplied by the specification, but often it is up to the verifier to formulate them. This approach is suitable for dealing with the full range of protocol properties to be verified, not only the general properties such as deadlocks and missing transitions. Ideally, any property for which an appropriate assertion can be formulated can be verified, but this process is rather difficult to automate, and usually requires a good deal of ingenuity on the part of the verifier [10].

5.2 Conformance Testing

Conformance testing involves comparing the behaviour of an Implementation Under Test (IUT) to that of its specification. Automated tools are used to achieve this goal. Typically, there are three stages in conformance testing:

1. **Test Suite Generation:** A set of test cases is generated from a formal (or sometimes an informal) specification. A test case is typically a collection of

interactions to be fed into an IUT, often composed with additional information about the IUT's expected response.

2. **Test Case Execution:** The test cases generated from the previous step are fed into an IUT, and the results are collected.

3. **Test Result Analysis** The observable outputs from the IUT are analyzed with respect to the test case's expected results.

5.2.1 Test Suite Generation

Test suite generation for FSM-based specifications involves the generation of a collection of test cases [20]. Some automated techniques for test case generation incorporate some form of state space search, but manually-generated test suites are also common.

A test case is typically a sequence of inputs which could be "fed" into an IUT, perhaps augmented with information about the expected observable response from the IUT. Sometimes, the test case is expressed as a tree of interactions, where the nodes traversed while following a path from the root to a leaf represent an input sequence to be fed into the IUT. When the test case is expressed as a graph with cycles, generating a set of input sequences is a little more complicated, but still straightforward.

If an IUT based on the specification fails a test case, it can be said that the IUT is non-conforming. However, we can not say anything about an IUT that passes our test cases, unless we can prove that the test cases we chose to execute are complete enough to cover the faults for which we are testing.

A *complete* test suite for an FSM is one that covers every possible fault in an implementation. An IUT which passes every test case in a complete test suite is free of faults. Such test suites are inordinately long, and usually infinite in length. Exhaustive testing, or executing every test case in a complete test suite, is impracticable

from both a theoretical and a practical standpoint.

Fault models are used to avoid exhaustive testing and to reduce the size of a test suite, while still finding most faults in an IUT. A fault model characterizes a subset of possible "mutant", or non-conforming implementations of the specification in behavioral terms [31]. By definition, executing a test suite which "covers" a particular fault model is guaranteed to show any existing faults of that type in an IUT. Some examples of the types of faults used in FSM testing [4] are:

Output faults: An output fault exists when, for corresponding initial states and inputs, an IUT outputs something which does not follow the FSM specification. This fault model is used in all test-coverage techniques.

Transfer faults: A transfer fault exists if, for the corresponding initial states and input, the IUT enters a different state than that specified by the FSM.

Transfer faults with additional states: There are certain situations when an IUT can enter a state which does not correspond to one in the FSM, in which case, additional states must be added to the fault model to reflect possible IUT behaviors. When the IUT enters such a state, this is a transfer fault to an additional state.

If n is the number of states in the specification, it is assumed that all possible IUTs have at most m states, where m might be greater than n. As $(m - n)$ increases linearly, the number of test cases in the suite grows exponentially, so this is an expensive fault model to use [20].

Additional transitions: For nondeterministic machines, there can exist multiple actions defined for a particular input from a particular state. In these cases, the fault model would include additional transitions to reflect possible IUT behaviors.

One of the important issues in fault coverage is that of generating a test suite which is both manageably short and reasonably thorough, two conflicting goals which force one to make a tradeoff of one in favor of the other [20]. In addition, it is difficult to prove that, given a test suite which covers a particular fault model for a particular specification, there does not exist another test suite which is smaller (contains fewer test cases) and covers the same model. Thus, eliminating redundant test cases is also an interesting problem.

Automated fault coverage techniques, when used in conformance testing, have the advantage of being very thorough in finding faults in an implementation, but most existing methods are suited for deterministic, minimal (without redundant states and transitions), fully-specified (for any input sequence, from any state, some response is specified) specifications.

It is possible to generate a suite that covers faults in nondeterministic specifications. Typically, such a technique needs to model an arbitrary nondeterministic FSM as a deterministic FSM [19], or a minimal FSM, or an "observable"[1] FSM [28], to satisfy assumptions which were made in the proof of the technique. A problem which arises from using such techniques is that the tests are not necessarily repeatable; that is, for a given sequence of input, there may be different resulting expected output sequences depending on the internal choices of the specification / implementation [20].

5.2.2 Test Case Execution

This step in conformance testing is fairly straightforward, involving the feeding of test case inputs into an IUT and capturing the inputs and outputs in a trace. The IUT specification, test case, and the obtained trace can provide enough information

[1] An observable FSM is one where an input/output *pair* a/b uniquely identifies a transition from a particular state (i.e. no other transitions with the same input/output pair can exist from the same state). An observable FSM can still be nondeterministic, as multiple transitions from the same state can take a as input.

to determine if the IUT passed or failed the test.

However, if the IUT is nondeterministic, complete guaranteed fault detection is theoretically impossible. For example, if it is desired to test the IUT for a number of nondeterministic reactions to a given input sequence t, the test case must be applied to the IUT repeatedly until all of the behaviors are exhibited. If the IUT has a fault, such that one of these possible specified reactions to t is never exhibited, this means that no number of repetitions of the test case will give conclusive evidence that this fault exists, for it may be that the IUT simply chose not to take a particular nondeterministic transition each time the test was executed.

To get around this problem, one usually makes a so-called *complete testing assumption*, which states that after a finite number of repetitions of a particular test case, if a certain behaviour which exists in the specification is not exhibited in the IUT, then there exists a fault in the IUT. The quality of the test increases with the number of repetitions of the test case. New techniques [19] [28] for automated test suite generation of nondeterministic protocols make this assumption.

5.2.3 Test Result Analysis

Test result analysis involves analyzing the IUT's observable behaviour in response to each test case executed with respect to the specification. Usually, the only observable behaviour of an implementation is a "trace", or a log of the interactions sent through the IUT's interaction points. 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 follows the specification.

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* [22] 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 trace is correct with respect to the specification.

Trace analyzers can run in real-time, monitoring an implementation as it is executing, or they can run in a batch-mode, processing traces which were collected during previous implementation executions. Some communicate with other modules over a network, others simulate the execution in one process. Different "test architectures" are used under different situations [9], such as when the IUT communicates with more than one module at a time.

Nondeterminism

When the specification is nondeterministic, trace analysis may require backtracking. Imagine a simple multiplexer specified by a nondeterministic EFSM M with one FSM state, and three interaction points, A, B, and C. The trace to be analyzed contains

2 input interactions: "a" arriving at A, and "b" arriving at B. The 2 interactions which appear in the output trace through interaction C are "ab". M has three nondeterministic transitions which look like this:

1. If available, read input from A and store in queue.

2. If available, read input from B and store in queue.

3. If available, output the next element in the queue to C.

Assume the queue structure is a dynamic data structure of infinite size.

If the two inputs arrive at the same time, there are two possible outputs for this module: "ab" and "ba". The possible transition sequences which generate "ab" are: [1,2,3,3], and [1,3,2,3]. The possible transition sequences which generate "ba" are: [2,1,3,3] and [2,3,1,3].

Since all four transition sequences are possible, a trace analyzer can begin by attempting one of them, and determine if it was the right choice. If the trace analyzer picked [2,3,1,3] as the first sequence to attempt, it would need to backtrack after executing transition 1, due to the output mismatch. In order to try another transition sequence [2,1,3,3], the trace analyzer would need to backtrack to the state right after taking transition 2, restoring all the variable values, as well as the queue state (which only contained "b"), to be what they were when transition 2 was executed for the first time.

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, depth-first search 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 realtime trace analysis, simple depth-first search is not sufficient, as explained further in chapter 7.

Non-Progress Cycles

In some specifications, there can exist a sequence of spontaneous transitions, from a particular FSM state to the same state again, which produces no output. If input is available to be read at this time, this is called a non-progress cycle, and can result in a search tree of infinite depth. When such a cycle exists in the specification, performing a complete state space search may be impossible. Sometimes an assumption is made that the length of a non-progress sequence of transitions can not exceed a certain number during a trace analysis, in order to force a verdict on any test case, but this can still result in a state space explosion of unmanageable size, and if the trace is not found to be valid, then all that can be said about the trace is that given the constraints on the search, no solution was found.

Sometimes, a non-progress cycle is entered while waiting for a time-dependent event. For trace analyzers which do not keep track of the time of events, such a cycle should be removed from their specifications. In other situations, infinitely cycling through the same states without making any progress is called a "livelock" which, from the black-box perspective, is indistinguishable from a deadlock. Protocol specifications should be free of possible livelocks before trace analyzers (and, of course, implementations) are written for these protocols.

Diagnostic Information

Providing useful information about the IUT fault in the event that the trace is invalid is a non-trivial task for the trace analyzer designer. It is easy to provide information about which transitions in the specification were attempted when "following"

the invalid trace. It is also straightforward to implement the error messages like "interaction i in trace t mismatches with the interaction generated from specification at this point in the search". However, for nondeterministic specifications, it can be that faults in an IUT have nothing to do with the transitions taken in the specification during a trace analysis, and often more useful information is desired in any case. Tetra [2], a trace analysis tool for LOTOS specifications, implements an error explanation search routine that can guess at what is wrong with an invalid trace, if it is due to a mismatched, missing, or extra interaction in the trace.

Initial IUT States

One of the most daunting problems a trace analyzer can face arises from the fact that the initial IUT state is not always known at the start of the trace. In the EFSM model, where variables and other parameters comprise the EFSM state, the number of possible FSM states multiplied by the number of possible variable values is the number of possible initial EFSM states. It may take an inordinate amount of time to determine the initial EFSM state, given that there are so many possible ones from which to choose. If the EFSM model also supports dynamic memory, this can yield an infinite number of possible initial EFSM states. At the moment, most existing trace analyzers assume that traces start when an IUT is in an initial state, or in one of a small set of possible initial states.

Chapter 6

Tango

6.1 Introduction

Several trace analyzers have been written for specific protocols such as as SNA [14], MAC [29], Class 4 Transport [26] and X.25 [32], 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. A more general purpose tool, that can be used to analyze traces of any protocol specified in a particular specification language, is presented here.

This chapter delineates the requirements and the development of a trace analysis tool generator for Estelle specifications and static trace files. A static trace file is one that does not grow during the trace analysis. For on-line trace analysis, the size of the trace to be analyzed is not known, and grows during the analysis. A procedure for analyzing such traces is discussed in chapter 7.

The approach taken was to modify Dingo (see Section 4.1.2 or [36]) in such a way that instead of simply generating an implementation that could be executed, the compiler would generate a trace analyzer for one module in the specification. The trace analyzer, in turn, would fire transitions depending on the trace information

supplied as the input to that module, and compare its generated output to the traced output, backtracking when necessary.

The result of our work, Tango, also known as the Trace ANalysis GeneratOr, does just that. It generates a trace analysis tool based on any single-module Estelle specification, which can analyze traces using relatively small amounts of memory and CPU time.

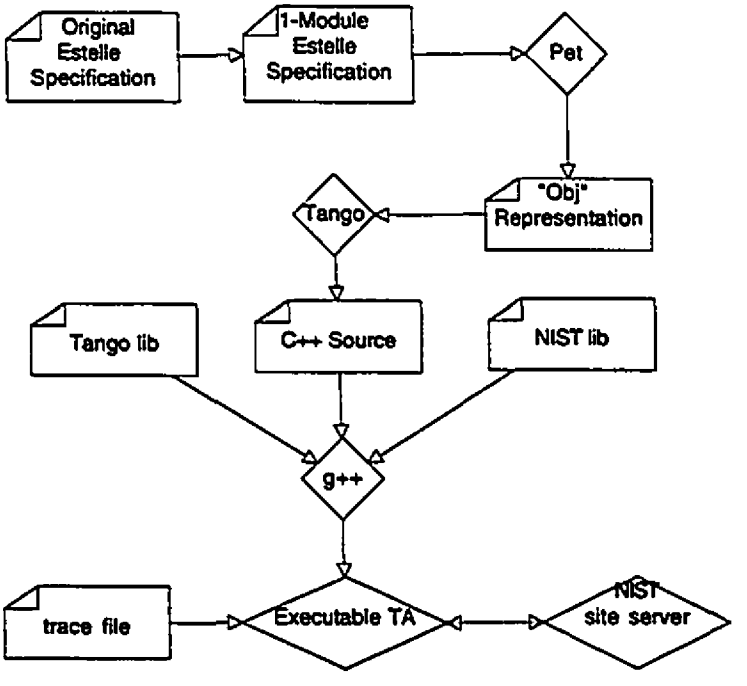

Figure 6.1: The Tango System

Related Work

A trace analysis tool for specifications written in LOTOS [11] 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, TESTVAL, is described in [27] [42]. Applied to Estelle specifications, this approach requires that the specification be manually transformed into *Estelle.y* which contains no 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 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.

6.2 Requirements Specification

A valid specification for Tango must contain only one module body, which specifies the behaviour of the module to be tested in the IUT. This module is called the TAM, or the Trace Analysis Module. Trace analysis on multiple modules executing concurrently is more complicated, since observable behaviour from one module considered invalid with respect to the trace can be a result of the invalid behaviour of another module in the system.

The TAM can have any number of interaction points, and these must be connected by channels to "feeding" modules in the `modvar` section of the specification. Feeding modules, when executed, will read information from a trace file and feed the proper interactions as inputs to the TAM.

The single-module specification with feeders must be translated into an object-oriented static representation by Pet (see Section 4.1.1) and then translated into C++ by Tango. The C++ source is then compiled by the GNU G++ compiler into an executable TAM.

Implementing a backtracking trace-analyzer by adding routines to Dingo required the following:

- *Saving TAM states.* The information comprising a TAM state includes the FSM state, an image of all variables, queue states, and dynamic data. Any time the TAM state must be saved, a copy of this information is placed into an object called `_state_info`, which is then pushed onto a stack for later retrieval.

- *Restoring TAM states.* All the information which was stored in the `_state_info` must be elegantly and quickly copied into the proper places for the TAM execution to continue at the point where it was when the image was saved.

- *Loading Trace Files.* Trace information, both inputs and outputs, stored in a text file, must be read in by the trace analyzer and interpreted as `_Interact` objects. The interactions from the output trace must be stored somewhere and readily accessible by the TAM for comparing to the generated output.

- *Generating feeding modules.* In addition to a TAM, Tango must generate a "feeding" module process for each one declared in the specification. Feeding modules run in parallel with the TAM. They read input trace information from the trace file and send it into the TAM's input queues. This way, the interactions are available to be read from the interaction points inside the TAM.

- *Comparing interactions for equality.* No routines were generated for comparing structured objects for equality by Dingo, so the generation of == operators for each structured object was added to Tango.

- *Generating a list of fireable transitions.* When a module is about to fire a transition, the Dingo-generated implementation chooses one of the fireable transitions and forgets about the rest (see Section 4.2.2). A TAM must generate a list of

all fireable transitions, save this list for later possible backtracking, and then the TAM may fire one of the transitions from this list.

- *Depth-First Search.* After the above features are implemented, it is fairly straightforward to write a depth-first search routine, which generates and searches through a tree of possible transition sequences for a path which satisfies the trace.

Some of these steps are described in more detail below.

6.2.1 Saving and Restoring TAM States

Non-dynamic Variables

All non-dynamic variables are accessible through a Global Reference Manager (GRM), which is an object belonging to each Module Instance object that is executing, and it contains a stack of pointers to **frames**, each frame containing pointers to structured data objects.

The frame on the bottom of the GRM stack contains pointers to all global variables, and other frames are pushed onto the stack when scopes are entered, and popped when scopes are exited. The actual variables occupy automatic memory (memory allocated when a function is called, and freed when the function returns) in the generated C++ methods for each transition block. In order to save an image of all variables on the GRM stack, the following steps must be taken:

- Another GRM is created, with frames which are exactly the same structure as the ones in the active GRM.
- For each new frame on the new GRM, memory is allocated for each of the frame variable pointers.

- The values are copied from the memory pointed to by the frames on the active GRM into the newly allocated memory for the frames in the new GRM.
- A pointer to the new GRM is placed into the `_state_info` object.

A bottom-up approach was taken to solve these problems.

Frames are custom-generated for every possible transition block and interaction. Transition blocks without local variables have frames without any data pointers in them, but they still appear in the generated implementation. Below are some frames which were generated by Dingo when run on the TriState specification in Appendix A.1.

```
// frame for the _Data interaction
struct __frame_Data {
   _Data_type*  Parameter;
};

//  Frame for Module _Feeding_body
struct __frame_Feeding_body {
  _Interface* ToMain;
  _Integer* Num_packets;
  _Integer* I;
  _Data_type* P;
};

// Frame for transition _Send_packet
struct __frame_Send_packet {
  _Integer* I;
};
```

As shown above, all local variables in `_Feeding_body`, `P`, `I`, and `Num_packets` as well as channel pointers, appear in `_frame_Feeding_body`. For the purposes of backtracking, only the variables must be copied.

One advantage of object-oriented languages is that all Estelle structured variables can be translated into objects with constructors (for creating a new instance of the

same type), and assignment operators (for copying values from one object to another of the same class). These methods were already implemented for Dingo-generated implementations. Getting frames to allocate and copy themselves is a little more complicated, but can be achieved using similar programming constructs.

The first step involves giving all frames a common superclass, `__frame_generic` with virtual methods for the following:

- `alloc_decls`: For each variable pointer in the frame, call its class constructor, to allocate memory for that type of variable.
- `dup_decls(__frame_generic*)`: Copy data from locations pointed to by "this" frame into locations pointed to by the frame of the argument.
- `free_decls`: De-allocate the objects pointed to by each variable pointer (useful when a `__state_info` object is no longer necessary).
- `clone_frame`: returns a pointer to a clone of the frame, by calling the frame's constructor, alloc_decls, and dup_decls.

Next, it is necessary to make Tango generate frame objects which inherit from `__frame_generic`, and custom methods to override each of the virtual ones above. Below is an example of what Tango generated for the `__Feeding_Body` module:

```
struct __frame_Feeding_body : public __frame_generic {
  _Interface* ToMain;
  void alloc_decls();
  void free_decls();
  void dup_decls(__frame_generic*);
  __frame_generic *clone_frame();
  _Integer* Num_packets;
  _Integer* I;
  _Data_type* P;
};
void __frame_Feeding_body::alloc_decls(){
  Num_packets = new _Integer;
```

```
  I = new _Integer;
  P = new _Data_type;
}
void __frame_Feeding_body::free_decls(){
  delete Num_packets;
  delete I;
  delete P;
}
void __frame_Feeding_body::dup_decls(__frame_generic *destination){
  __frame_Feeding_body *dest = (__frame_Feeding_body*) destination;
  *(dest->Num_packets) = *Num_packets;
  *(dest->I) = *I;
  *(dest->P) = *P;
}
__frame_generic *__frame_Feeding_body::clone_frame(){
  __frame_Feeding_body *retval = new __frame_Feeding_body;
  retval->alloc_decls();
  dup_decls(retval);
  return (retval);
}
```

The final step, making a method for the GRM that will clone itself, is relatively simple, and involves not much more than cloning each frame on the GRM stack.

Dynamic Variables

A dynamic memory manager (class `_DRManager`) maintains a linked list of `_DREntry` objects. Each entry contains an address, a size, and a data pointer. Each time a call to **new** is made on an Estelle data structure pointer, an entry is added to the list which contains the address and size of the new memory block, and the data field remains set to `null`. Each time a call to **dispose** is made on an Estelle data structure pointer, the corresponding entry in the list is removed. This way, only the memory blocks with corresponding entries in the `_DRManager` are the ones used by the module at any given moment.

When the TAM state must be saved, a copy of the `_DRManager` list is made, and

placed into the _state_info object, leaving the original one unchanged. Then, the linked list in the copy is traversed, memory is allocated for the data field of each entry, and finally, memory is copied from the location pointed to by address into the newly allocated space. The contents of all dynamic memory in use that that moment is then available for future possible restoration.

When the TAM state must be restored, the _DRManager copies the data back in the other direction.

This approach is fairly simplistic, and works only when the TAM never de-allocates memory between a save and a restore. In the event that the TAM does free memory, a memory fault will result from copying the saved info from the old copy of the _DRManager into its former location in memory. Since the former memory location might be used for something else, the memory can not simply be re-allocated on demand.

Two solutions to this problem are proposed here:

- Never de-allocate dynamic memory, so it will always be possible to copy a dynamic record back into its former memory location on backtracking.
- Keep track of all pointers to dynamic memory, and when re-allocation of disposed memory is necessary, adjust all pointers to the old memory location so they point to the new memory location.

The first approach results in a considerable amount wasted memory at runtime, but is very simple to implement. This was the approach taken in the first working version of Tango.

The second approach, however, was implemented in the current version of Tango. One field was added to the _DREntry object, status, which can be either deleted or active. The Tango code-generation routines had to be modified so the following operations were supported:

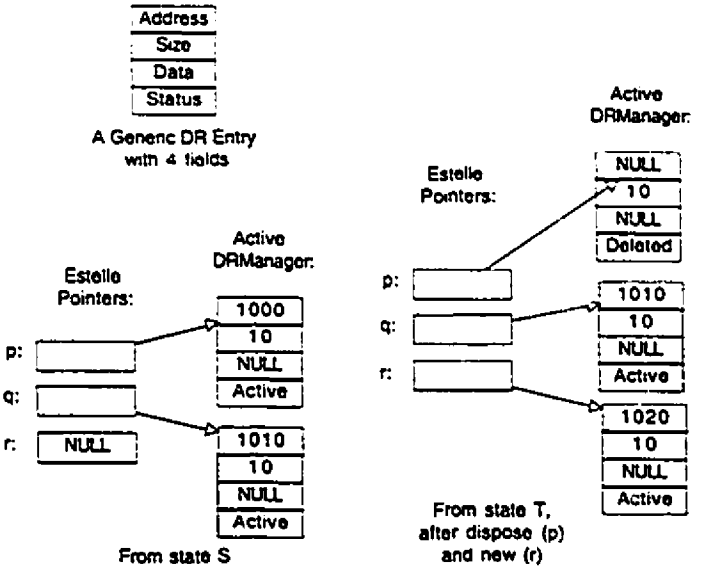

Figure 6.2: An example DR Manager

- Any de-reference of a pointer in Estelle becomes a *double* de-reference in the C++ implementation.

- **new(p)** As before, a new entry is created in the DRManager, and the DREntry's address field will point to the new dynamic record. Instead of setting p to be another direct pointer to the same dynamic record, p will point to the DREntry's address pointer. The status will be set to active.

- **dispose(p)** De-allocates the memory pointed to by the DREntry's address, and marks the status field of that entry as deleted.

- On backtracking, two DRManagers are consulted before data is restored: the saved one, referred to as the *source*, and the active one, in the MInstance, referred to as the *target*. The following cases must be handled:

  1. Memory pointed to by a DREntry in the target which does not have a

corresponding DREntry in the source to be restored should be de-allocated. These are objects which were allocated after the state to be restored was saved, and are no longer necessary.

2. Any DREntry marked as `deleted` in the target which has a corresponding `active` DREntry in the source must have its `address` pointer re-allocated, and its status pointer set to `active` again. These are elements which were de-allocated after the state to be restored was saved. Since each Estelle pointer is a pointer to a pointer in the DRManager, rather than a pointer to the dynamic memory itself, a double de-reference will result in the correct area of memory being accessed.
3. Any DREntry in the target which has an active corresponding DREntry in the source can have the memory pointed to by its `address` field over-written as before.

In a depth first search, restoring a state implies restoring to an ascending state in the search tree. During a restore, it is acceptable to remove all DREntries in the target DRManager which were allocated between the save and the restore, as they will never be needed again. Therefore, it is possible to maintain a one-to-one correspondence between the n elements in the source list, and the first n elements in the target list, where n is the number of `new` statements executed while taking transitions which form a path from the root node to the node to be restored.

Because of the one-to-one correspondence in depth first search, it is possible to write a restore operation which performs the above task and requires only one traversal of each list.

Queue States

Since the channels are asynchronous, there is a queue associated with each interaction point, containing all of the inputs which arrived but were not yet consumed. The

queue state is an important part of the TAM state.

Some different approaches to saving a queue state will be discussed. At first one might try creating a copy of the queue, where each element in the newly created queue is a copy of an element in the queue to be saved. Then, upon restoring, the active queue along with all of its elements is destroyed, and the copy is put in its place. When there is frequent backtracking, this approach, with its high computation and memory requirements, is not very practical. In any case, Estelle does not permit the changing of data elements in the queue, so saving the data and restoring it is a waste of resources.

A more practical approach would just save the information required to have the next enqueue and dequeue operations perform as they would have when the state was saved. Therefore, the de-queued elements must be available somewhere for future restoration.

The queue in Dingo is an object of type `DList`, a double-linked list, with methods for traversing, adding, and removing elements anywhere in the list. `DList` also has methods **enqueue** and **dequeue** which, when they are the only methods used for adding and removing elements from this list, provide an implementation of a FIFO queue. Dingo implements a channel queue (in class `_SIPType`) using this `DList`.

In Tango, the method for **dequeue** was re-written, so that instead of actually removing the element from the `DList`, a private member field of `DList` called **next**, which points to an element in the list, is consulted to determine the next element to be dequeued, a *copy* of that element is returned, and the **next** pointer is updated to point to the following element in that list [1].

Using this approach, no element is actually removed from the DList, and to restore the queue state, all that is necessary is the restoring of the value of the **next** pointer, and removing the elements which were enqueued since the save-state operation.

[1] Later in the development of Tango, it became necessary to change the dequeue method again, so it would not copy the interaction! The reasoning is explained in Section 6.2.3

In fact, the task is easier than described above. Since, during trace analysis, all interactions which are sent out through interaction points from the TAM are analyzed immediately for conformance to the trace, this means that interactions sent by the TAM are never placed onto a queue, and thus, never need to be removed on backtracking. Therefore, the only information needed in the `_state_info` for each queue is the `next` pointer. History of the enqueued elements will always be available in the queue itself.

6.2.2 Trace Files

While the IUT is being traced, the inputs sent to the IUT as well as the outputs received by IUT must be saved into a file. These interactions are all necessary as input to the TAM when the trace is to be analyzed. The format of a trace file is usually simpler when it describes the interactions going through just one channel, but because the TAM can have any number of bi-directional channels, Tango trace files must contain information about from which channel each interaction was traced.

An example of the format for the TAM's input is described in Section 4.3, shown in Appendix A.3, and the format is specified informally in Appendix C.2.3. This format was chosen because Dingo already has methods for the stream input/output of the interaction parameter lists, and those methods are used for reading and writing trace files in Tango.

Each module spawned by a Tango specification has a trace file manager (class `tf_man`). The `tf_man` keeps an array of `ip_queue` structures, one for each interaction point in that module. Before a module begins execution, `tf_man`'s constructor automatically reads each interaction in the trace file. If the interaction is an output originating from that module, it is enqueued into its corresponding `ip_queue`.

When the TAM executes an output statement, a call to `tf_man::check_output` is made. This method determines if the interaction being sent through a particular

interaction point exactly matches the interaction which appeared next in the trace (which is stored in one of the tf_man ip_queues). If there is a match, the proper tf_man.ip_queue next pointer is updated so that a subsequent call to check_output will compare the output to the following interaction in the trace file for that IP.

The object tf_man must be able to save and restore its state in the event of backtracking. Since tf_man uses the same DList structure for its queues as SIPType, a list of next pointers, one for each ip_queue, is sufficient for saving and restoring tf_man states.

6.2.3 Generating a List of Fireable Transitions

The Dingo-generated routine for each module body, __selAndExec, is used as a model for the TAM method, __generateFireable. The method generated for TriState is shown in Appendix A.4. __generateFireable uses the same conditionals for determining if a transition is fireable as its __selAndExec counterpart, but instead of setting local variables for the transition frames, interaction, and transition block, a structure of type _trans_info is created, and this structure must contain all of the information required for executing a fireable transition. For each transition which could have been fired by __selAndExec, a _trans_info object is appended to a double-linked list, and a pointer to that list is returned by __generateFireable.

It was during the development of __generateFireable that it became apparent that some parts of Dingo were not written in a very object-oriented fashion. For example, in the __selAndExec method featured in Appendix A.2, the interaction pointer __cinter is assigned to point to the first interaction in the FromFeeder queue, on line 77. Then, this interaction is deleted directly by __selAndExec on line 159. This means that no matter how thoroughly the methods for __SIPType protect the interactions in the queue, it is still possible that the interactions will be de-allocated without the knowledge or consent of __SIPType. All delete statements which applied

to interactions were removed from Tango to prevent this from happening. After this was done, it became obvious that the dequeue operation of `DList` did not need to return a pointer to a *copy* of the interaction being dequeued, but a pointer to the actual interaction itself, as it did before (see Section 6.2.1). Otherwise, considerable amounts of memory would be wasted.

The frame pointers in `_selAndExec` point to frames which are declared *locally* to `_selAndExec`, on lines 51 and 52. Saving pointers to these frames and exiting from `_selAndExec` will cause memory errors, since these pointers will point to de-allocated automatic memory. `_generateFireable` must place pointers to *copies* of each frame in the `_trans_info` fields, and thus, `clone_frame` is used to this end, as shown in lines 56, 85, and 114 of Appendix A.4.

Since a `_trans_info` object should be destroyed after executing its transition, a destructor for `_trans_info` which de-allocates its pointers to frame copies is necessary.

6.2.4 Depth-First Search

The algorithm for depth first search looks like this:

1. Check if all inputs were consumed and all outputs verified. If so, output a successful result and exit
2. Generate fireable transitions. If there is more than one possible transition, save the current TAM state.
3. Choose one of the not taken transitions from the list, mark it as "taken" and execute it.
4. If there were no possible transitions, backtrack and goto step 3
5. If outputs from this transition were invalid, backtrack and goto step 3
6. Goto step 1

Backtracking involves finding the deepest point in the search which contains unexplored transitions, and restoring the state, and the list of fireable transitions from

that state. In the event that all possible transitions are searched, the TAM outputs a "trace invalid" result, and exits.

The routine which performs the depth first search is the method called `startTAExec` and is defined for the class `_System`, used for all generated TAMs.

6.3 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 using them is described in the Tango Tutorial, Appendix C.2.4.

6.3.1 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 be infinite in number due to the fact that Estelle supports dynamic memory allocation. It is frequently not sufficient simply to try different

initial FSM states, as Tango does. Another approach for handling partial traces is discussed in Section 9.2.3.

6.3.2 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 interaction point 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 interaction points, or through the same interaction point but in different directions, can be **observed** (and checked) or **ignored** by the TAM, depending on the runtime options.

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 generated by the Estelle module specification, assuming no input or output queues. However, in practice, the implementation under test that has generated the trace file may include input and/or output queues associated with the different interaction points 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 interaction points, the relative order of trace inputs pertaining to different interaction points is of no relevance. Similarly, if the output interactions from different interaction points travel through different queues before being recorded in the trace, the relative order of outputs pertaining to different interaction points is of no relevance. Finally, for any given interaction point, if an input or output queue is present in the implementation, an input in the trace may precede the next output to be generated (in the case where this input was already provided by the environment, but not yet processed by the module, at the time when the output was generated).

Tango supports the following options for relative order checking:

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

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

IP relative order checking: Checks 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.

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 [15]. It is also 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. This will often yield linear-time trace analysis executions with respect to the length of the trace (see Section 8.2.2).

Temporal Information

Currently, there is no facility in Tango to keep track of real-time relationships between events. This means that implementations which exhibit real-time behaviors, such as time-outs, can not be checked for conformance to a specification by a Tango-generated TAM. Handling tracefiles with time stamps attached to each interaction would be a possible future enhancement to Tango, and would also eliminate some problems faced

by protocol testers who currently can only incorporate different degrees of relative order information in the trace files used by a TAM.

6.3.3 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 9.2.3 discusses some of the problems involved in implementing partial trace analyzers.

6.4 The TriState Trace Analysis

This section describes the execution of a Tango-generated TAM based on TriState (see Appendix A.1). The trace fed to the TAM as input is the same trace from Appendix A.2. This means, of course, that the TAM and the IUT are both strictly based on the same specification, so the trace *must* be valid. The trace analysis progress report, or log file, is shown in Appendix A.5. This section describes how to understand the contents of the log file.

Before each transition is executed, a log entry of the following format appears:

```
currentState = s
```

```
depth: d  check_again: c  Transitions: t
<transname 1>
<transname 2>
....
<transname t>
```

s is the name of the current DFS state. d is the search tree depth. c is a boolean, indicating whether the node is a Partially Generated node and needs to be checked again later[2], and t is the number of fireable transitions from this state. For each fireable transition, the transition name is listed below that line.

When a transition is chosen, a message like this will appear in the log file:

`::: Executing transition: _ToSolid`

If this is a non-spontaneous transition, the input consumed results in the following message in the logfile:

`--> Input :` *interaction_name* [*index*] `from` *IP_name*

Where *index* is an ordinal value attached to the interaction which indicates how many interactions appeared in the trace file before this one.

Outputs which are produced during the execution of the transition code result in a similar `Output :` message in the log file.

In our example, it is not until depth 22 that more than one fireable transition is generated, and all the transitions before that match the logfile of the IUT execution from Appendix A.2. However, at depth 34, as it was mentioned in Section 3.1, after receiving the data packet with `parameter.I` set to 99, it is possible for an implementation to take the transition `_ToGas` or `_ToFinished`. The TAM arbitrarily decided to try `_ToGas` first, as we can see on line 158. The following spontaneous transition `_ToLiquid` sends a `_Data_response` through its interaction point. This does not match the `_Close_Connection` interaction which appears next in the trace, thus we can see the message on line 166, and the TAM backtracks. Back at depth

[2]This is relevant only to dynamic trace files. See Section 7.2.2 for more information.

22 on line 170, we can again see the list of transitions from that point, and here `_ToGas` is marked as "already tried", so the TAM takes `_ToFinished`, outputs a `_Close_Connection` which matches the trace, and all outputs are verified. Therefore, the trace is valid, as expected.

Some statistics on the search appear at the bottom of the log file. The meaning of each field is explained below.

CPU Time (seconds): This is the amount of CPU time used by the TAM process. Achieved by using the `clock(3C)` function.

Trans executed: The number of transitions executed during the search. This can also be thought of as the number of edges searched in the tree. During DFS, this is the sum of generates and restores.

Generates: The number of times a call was made to `_generateFireable`. During DFS, this is the number of vertices in the search tree.

Depth: The depth of the most recently searched node in the tree.

Max Depth: The maximum depth achieved in the tree during the search.

Restores: The number of state restores during the search. During DFS, this is the number of backtracks.

Saves: The number of state saves during the search. During DFS, this is the number of nodes with more than 1 child in the search tree.

Trans per second: This is the number of executed transitions divided by the CPU time, in seconds.

Chapter 7

On-Line Trace Analysis

7.1 Introduction

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.

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**.

7.2 Requirements Specification

The following features were added or re-written in Tango 1.5 to implement a multi-threaded trace analysis:

- Multiple, dynamic trace files
- Multi-Threaded Depth-First search
- Dynamic memory restore
- Queue state save/restore routines

7.2.1 Multiple, Dynamic Trace Files

In static-mode, Tango requires only one trace file to analyze, which may contain interactions between the IUT module and all of the other modules it communicates with. In dynamic mode, chances are that the trace is being taken from multiple observation points (channels between the IUT and its sibling modules). If Tango required all trace data from multiple observation points to be merged into one trace file, this could make the interface between the IUT and the TAM a little more complicated, so Tango supports multiple trace files, as a convenience to the tester.

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.

If interactions from the same observation point appear in different trace files during a trace analysis, they will be placed in their proper IP queues in the order

they are read from the trace files. If interactions from different observation points appear in different trace files, the runtime option `ip_relative` should not be used, as there is no way for Tango to determine the IP relative order of interactions when they appear in different trace files.

7.2.2 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 7.1: Pseudo-Estelle specification **ack**

Imagine that the TAM is performing on-line trace analysis using our specification **ack** in Figure 7.1. 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] be 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 later in this section.

Implementation Details of Standard DFS

The DFS search tree is implemented as a double-ended queue of `dfs_info` objects, where each object can be thought of as a "node", or a "vertex" in the search tree. Each object contains a pointer to a `_state_info` object, and a pointer to a linked list of `trans_info` objects. Each `trans_info` contains all the necessary information to execute a transition from that state, and can be thought of as an "edge" in the search tree. After each transition is executed, its `trans_info` object is marked as "already taken" so that it is not searched again.

During a "generate" (a DFS operation that generates all the possible transitions coming from the current state), a new dfs_info is created, and is set to contain the

linked list of fireable transitions from this state. If the number of transitions is greater than one, the current TAM state is saved. The object is then appended to the "bottom" of the search tree.

After all edges under the current node have been searched, the node can be removed from the search tree data structure, and the next "bottom" node is consulted for the next transition to search.

Implementation Details of Basic MDFS

If an input queue is empty during transition generation, 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. In basic MDFS, the TAM will place this node on the "top" of the search tree, rather than the bottom, so that it will only be searched after the rest of the search tree has been exhausted.

When the rest of the tree is exhausted, PG-nodes will be the only ones left to search. The bottom PG-Node will be restored and the TAM must make another call to `_generateFireable`, to determine if there are additional transitions which are fireable from the current state which were not already tried before. If there are newly generated transitions, they will be searched next. If some input queues are still empty in this state, the node is still considered PG, and will be placed on the top of the search tree again after the newly generated transitions have been explored.

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.

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? At none of these states have all inputs/outputs been consumed/verified, 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 7.2, 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 nondeterministically continue consuming and verifying data interactions which pass through IPs B and C until no more input and output trace 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

```
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 7.2: Pseudo-Estelle specification, **ip3**

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 7.2 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 situations like this can 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 input 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. See Section C.2.4 for more information.

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. 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 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 path to the 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 Tango version 1.52.

Degenerate Cases

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` option will prevent this degenerate MDFS case from occurring. See Section C.2.4 for more information.

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.

7.2.3 Queue States

A small feature was added to the queue state save/restore routines to handle the case where the end of all interactions of a particular queue is reached. When this happens, additional information needs to be saved in the TAM state about the "most recently dequeued" interaction on that IP, so that when this state is restored, newly received interactions will be accessible to the TAM.

7.2.4 Dynamic Memory Restore

The Dynamic Memory Restore routine which was written for static-mode Tango assumed that the search would be depth first. This assumption is useful because it implies that any restore brings the state from a child to its parent (or another ascendent), rather than to an arbitrary cousin or descendent in the search tree. During this kind of restore, all entries in the DRManager which were added since the save can be removed during the restore. This maintains a one-to-one correspondence between the n entries which are in the DRManager to be restored, and the first n entries of the

active DRManager. The algorithm for a depth-first search dynamic memory restore requires only one traversal of each DRManager linked list, so it is performed in linear time, with respect to n.

When restoring from one state to a non-ascendent node in the search tree, there is no longer a correspondence between elements in the two DRManagers. Furthermore, it is not permitted to remove entries from the active DRManager when restoring to an ascendent, if there exist PG-nodes in the search tree which are descendents of the state to be restored, because they will be needed again in the active DRManager when those PG-nodes are restored.

Therefore, non-ascendent restores, and restores to states which are ascendents of PG-nodes in the tree, require m searches through a data structure with n elements in it, where n is the number of elements in the active DRManager, and m is the number of elements in the DRManager to be restored.

In standard DFS, the number of entries in the active DRManager grew with a_1, the number of allocations performed while executing the transitions which form a path in the search tree from the root node to the current one being searched. In MDFS, however, the number of entries in the active DRManager grows with a_2, the number of memory allocations performed during the **entire search** up to that point, through any path in the tree. a_2 depends on the degree of nondeterminism in the specification, and can be exponential with respect to a_1.

In theory, the faster DFS restore routine can still be used in MDFS to restore to a parent node, which is not the ascendent of a PG-node in the search tree, but in the interest of simplicity, the current version of Tango performs the more general-purpose dynamic restore algorithm for every restore during an MDFS. Even if Tango performed regular DFS restores when possible, dynamic memory restores would be, on average, considerably slower under MDFS than under regular DFS.

Since the DRManager data structures are unsorted linked lists, a restore under

MDFS takes $O(m \times n)$ entry key compares, even if only a small percentage of these entries need to have their memory re-allocated, restored, or de-allocated.

If Tango is used frequently for on-line trace analysis and one is looking for areas to improve Tango's performance, this is an area worth investigating further. Implementing a heap-like data structure for the DRManager, or using the faster DFS restore algorithm whenever possible in MDFS, are the suggested enhancements.

Chapter 8

Practical Uses of Tango

8.1 Test Case Generation

Any Estelle specification which has no `external` module body definitions can be put through Pet and Tango to generate an executable implementation which will behave identically to a Dingo-generated implementation.

The implementation generation feature of Tango is useful for generating test cases. Connecting the module which specifies the IUT to other "testing" modules that force the module to exhibit certain behaviors, compiling it with Pet/Tango, and executing it under the NIST Site Server, will yield trace files which can be used as test cases for an IUT, or as sample input to a Tango-generated trace analyzer.

An important difference between a Tango-generated trace file and one that was captured from one or more observation points during protocol testing, is that the points of observation for a Tango-generated trace file are *inside* each module that is being executed. This means that relative order information between interactions sent from the same module, but to different IPs, is available in the generated tracefile. Furthermore, if an interaction i, listed in the Tango-generated trace file as input to the TAM, appears before another interaction o, listed in the trace file as an output

from that TAM, this means that i was *consumed* by the implementation before o was produced. This means that using the I/O relative checking option on a Tango-generated trace file will work fine, even though the channels are asynchronous.

8.2 Trace Analysis Performance Results

The current version of Tango has been tested on some simple example specifications, as well as 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 [1]. The machine used for testing was a 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 an entire second to search only 10 transitions.

8.2.1 LAPD

Using the LAPD specification developed at CNET [33], and using Tango in implementation generation mode, we generated 7 valid traces, by sending various length-ed

[1] For more information on these protocols see [40]

DI	CPUT	TE	GE	RE	SA	CPUT	TE	GE	RE	SA
	NR					IO				
5	4.1	34	21	15	17	2.9	28	19	9	13
10	7.6	64	36	30	32	5.5	53	34	19	28
15	11.0	94	51	45	47	10.9	78	49	29	43
25	18.4	154	81	75	77	16.3	128	79	49	73
50	34.4	284	148	138	144	30.8	237	146	91	140
75	52.2	414	215	201	211	50.7	346	213	133	207
100	71.7	579	296	285	292	62.8	483	294	189	288
DI	CPUT	TE	GE	RE	SA	CPUT	TE	GE	RE	SA
	IP					FULL				
5	1.6	24	19	5	7	0.7	20	19	1	3
10	3.0	44	34	10	17	1.6	35	34	1	8
15	5.0	64	49	15	29	2.3	50	49	1	15
25	7.7	104	79	25	46	3.5	80	79	1	22
50	13.3	192	146	46	95	6.8	147	146	1	50
75	21.0	280	213	67	135	9.5	214	213	1	69
100	30.2	389	294	95	191	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 8.1: Execution times of a TAM on LAPD traces of various sizes

sequences of data interactions 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 8.1.

A few comments and observations on this table appear below:

- The number of interactions LAPD sends to the network layer is not always the number of interactions LAPD receives from the user module. This is because sometimes the user module closes the connection before the network module has had a chance to request all of the data being held for it by the LAPD module. Coincidentally, the number of restores when performing IP relative output checking only, is the number of data interactions sent by the LAPD module. The sum of this and the **DI** value gives us the total number of data interactions in the trace file.

- Using no relative checking, or I/O relative checking only, the transitions per second remains roughly the same, around 9. The ratio of backtracks to transitions taken also remains the same, although the search space is reduced slightly when I/O relative checking is enabled.

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 between the LAPD module and the module which represents the network layer, because the interactions passing between the user module and the LAPD module are not necessarily observable. The current version of Tango can not analyze such traces, but we address this problem in Section 9.2.3, on partial trace files.

8.2.2 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 module 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 (see Appendix B) 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 into `buffer2`.
- `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 into `buffer1`.
- `T16`: If nonempty, send an interaction from `buffer1` to the upper 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 [2], 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. This is an area suitable for further research.

The results of executing a TAM on an *invalid* TP0 trace are shown in Figure 8.2.

[2]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 8.2: Execution times of a TAM on invalid TP0 traces

The 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 using different relative checking option combinations Then, larger traces which were edited similarly were 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 TP0 can output a disconnect indication at any time, even if data remains in its buffers after the disconnect request is received by the TP0 module. In other words, the transition which receives the disconnect request and outputs a disconnect indication is `t17`, and it becomes fireable from the data state, in addition to the

fireable transitions described above. Enabling the relative checking options on these invalid traces reduced the average fanout from 2.4 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.

Chapter 9

Conclusions

9.1 Summary

In this thesis, the concepts of formal protocol specification, validation, and testing were presented. Issues in automatic generation of implementations based on formal specifications were discussed. The steps required to transform an implementation generator into a trace analyzer generator were chronicled, and a fully-functional tool to generate trace analyzers for single-module Estelle specifications was developed and tested.

Tango provides a means to analyze traces of any single-module protocol specified in Estelle, supporting almost all[1] 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 [36]. 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 arises from the nondeterminism of the specification. In this respect, it is

[1] With the exception of `delay` statements

important to note that the input and output queues that may be part of the IUT 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.

9.2 Possible Areas of Future Work

This section describes some current problems in trace analysis which were not solved by the latest version of Tango.

9.2.1 Time-Stamped Interactions in Trace Data

Without information about the time passed between interactions in a trace, it is impossible to determine if certain time-specific behaviors in the IUT are exhibited as specified. The current version of Tango has no way of handling time information in trace files. A future version of Tango might support this.

9.2.2 Invalid Trace Error Diagnostic Searching

There are other trace analysis tools which attempt to provide more useful diagnostics in the event that an invalid trace is encountered [2]. They can determine if the trace is invalid due to a missing or extra transition. A future version of Tango might implement a similar kind of search.

9.2.3 Analysis of Partial Traces

For the purposes of this thesis, 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 an lysis significantly more difficult. In the case where the initial module state is unknown, certain variables will be undefined, 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 a very high-order 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.

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.

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.

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" 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.

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.

Appendix A

Listings

A.1 Estelle Specification of TriState

```
 1: Specification TriState;  timescale seconds;
 2:   type
 3:     data_type = record
 4:       H : array[1..10] of integer;
 5:       I : integer;
 6:       j : boolean;
 7:       K : char;
 8:     end;
 9:
10: channel interface (receiver, sender);
11:    by sender:
12:      data(parameter:data_type);
13:    by receiver:
14:      data_response;
15:      close_connection;
16:
17: {------------------------- Feeding_Module ------------------------}
18:
19:   module Feeding_Module systemprocess;
20:     ip
21:        toMain : interface (sender) individual queue;
22:   end;
23:
24:   body Feeding_body for Feeding_Module;
25:     var
26:       num_packets, i : integer;
27:       p : data_type;
28:
```

```
29:   state SENDING, waiting, DONE;
30:
31:   initialize
32:     to SENDING
33:     name toSending:
34:       begin
35:         num_packets := 0;
36:       end;
37:
38:   trans
39:     from SENDING to DONE
40:       provided num_packets > 10
41:       name toDone:
42:         begin
43:           p.i := 99;
44:           p.k := 'a';
45:           output toMain.DATA(p);
46:         end;
47:
48:     from SENDING to WAITING
49:       {The below provided clause eliminates non-determinism}
50:       provided num_packets < 11
51:       var
52:         i : integer;
53:       name send_packet:
54:         begin
55:            for i := 1 to 10 do p.h[i] := num_packets + i*2;
56:            p.I := (num_packets mod 2 * 2 - 1) * num_packets;
57:            p.j := (num_packets mod 2) = 0;
58:            p.k := succ (p.k);
59:            output toMain.DATA(p);
60:         end;
61:
62:     from WAITING to SENDING
63:       when toMain.data_response
64:       name finished_waiting:
65:         begin
66:           num_packets := num_packets + 1;
67:         end;
68:   end; { feeding_body }
69:
70: {----------------------Main_type module-------------------}
71:
72:   module Main_type systemprocess;
73:        ip
74:          fromFeeder : interface(receiver)    individual queue;
75:   end;
76:
77:   body Main_body for Main_type;
```

```
 78:
 79:   type
 80:      rect = record
 81:        ishot: boolean;
 82:        isfinished, iscold : boolean;
 83:        end;
 84:
 85:   var
 86:      v : rect;
 87:      D : data_type;
 88:
 89:   state SOLID, LIQUID, GAS, FINISHED;
 90:   stateset NONLIQUID = [SOLID, GAS];
 91:
 92:   initialize
 93:      to LIQUID
 94:      var
 95:        i : integer;
 96:      name INIT_Trans:
 97:      begin  { initialize variables }
 98:        v.ishot := TRUE;
 99:        v.iscold := TRUE;
100:        v.isfinished := FALSE;
101:       end;
102:
103:   trans
104:
105:     from NONLIQUID to LIQUID
106:       delay(3)
107:       name toLiquid:
108:         begin
109:           v.ishot := FALSE;
110:           v.iscold :=  FALSE;
111:           output fromFeeder.data_response;
112:         end;
113:
114:     from LIQUID to GAS
115:       when fromFeeder.data
116:       provided parameter.I > 0
117:       name toGas:
118:         begin
119:           v.ishot := TRUE;
120:         end;
121:
122:     from LIQUID to SOLID
123:       when fromFeeder.data
124:       provided parameter.I <= 0
125:       name toSolid:
126:         begin
```

```
127:           v.iscold := TRUE;
128:         end;
129:
130:     to FINISHED
131:       when fromFeeder.data
132:       provided parameter.I = 99
133:       name toFinished:
134:         begin
135:           v.isfinished := TRUE;
136:    output fromFeeder.close_connection;
137:         end;
138:   end;
139:
140:
141: {---------------------- SPECIFICATION BODY ----------------------}
142: modvar
143:     { module-variable-declaration-part of the specification }
144:     Main: Main_type;
145:     Food : Feeding_Module;
146: initialize
147:   begin { module initialization }
148:         init Main with Main_body;
149:         init Food with Feeding_body;
150:         connect Main.fromFeeder to Food.toMain
151:   end; {of specification body}
152:
153: end. {of specification TriState}
```

A.2 Some Dingo-Generated Routines from TriState

```
 1: void _INIT_Trans( __stackElem* __bRef:_INIT_Trans,
 2:                   __MInstance* __MI, __GRManager* __GRM) {
 3:   // define and push local vars; import globals
 4:   __frame_INIT_Trans __frame;
 5:   _Integer I;
 6:   __frame.I = &I;
 7:   __GRM->enter( &__frame, __bRef_INIT_Trans);
 8:     _Rect& V = *((__frame_Main_body*) __GRM->getFrame(1))->V;
 9:   __MI->logString( ">>> executing transition _INIT_Trans");
10:   // from Estelle source;
11:      {
12:        V.Ishot = __TRUE;
13:        V.Iscold = __TRUE;
14:        V.Isfinished = __FALSE;
15:        ;
16:      }
```

```
17:   // pop context;
18:   __GRM->leave();
19: }
20:
21: void _ToSolid( __stackElem* __bRef_ToSolid,__MInstance* __MI,
22:                __GRManager* __GRM) {
23:   // define and push local vars; import globals
24:   __frame_ToSolid __frame;
25:   __GRM->enter( &__frame, __bRef_ToSolid);
26:     _Rect& V = *((__frame_Main_body*) __GRM->getFrame(2))->V;
27: #ifdef LOGT
28:   __MI->logString( ">>> executing transition _ToSolid");
29: #endif
30:   // from Estelle source;
31:      {
32:        V.Iscold = __TRUE;
33:        ;
34:      }
35:   // pop context;
36:   __GRM->leave();
37: }
38:
39: int __MI_Main_body::__selAndExec( int __dt) {
40:  // selects and executes a trans; if none is selected returns 0;
41:  // also sets mayExecuteNext to indicate when this module could
42:  // execute a transition without further inputs;
43:  int __ok;
44:  __SIPType* __sip = 0;
45:  __Timer *__tim=0, *__ctim=0;
46:  __Interact* __inter, *__cinter;
47:  __PTB __transBlock = 0;
48:  int __toState;
49:  void* __frame1;
50:  void* __frame2;
51:  _Data __wcs_Data;
52:  __frame_Data __wcf_Data;
53:  __wcf_Data.Parameter = & __wcs_Data.Parameter;
54: #ifdef OPTIMIZE
55:  __mayExecuteNext = -__MAXNOLOOP;
56: #else
57:  __mayExecuteNext = 0;
58: #endif
59:  {
60:   __ok = 1;
61:      __ctim = &__timers._ToLiquid;
62:      __ok = __ctim->fireable() && (!__ctim->optional() ||
63:             __wantToConsiderOpt("_ToLiquid"));
64:      __mayExecuteNext = __ctim->isSet() ?
65:        max(__mayExecuteNext,__ctim->beforeFireable()):__mayExecuteNext;
```

```
66:       if (__ok) {
67:         __frame1 = __frame2 = 0;
68:         __inter = 0;
69:         __tim = __ctim;
70:         __toState = _LIQUID;
71:         __transBlock = &_ToLiquid;
72:         if (__wantToFire("_ToLiquid")) goto __EXEC;
73:       }
74:   }
75:   {
76:    __ok = ( (__currentState==_LIQUID));
77:    if (__ok &&(__cinter=FromFeeder.firstIs(62))) {
78:      __frame_Data __pwcFrame;
79:      _Data_type& Parameter=((_Data*) __cinter)->Parameter;
80:      __pwcFrame.Parameter =& Parameter;
81:      __GRM->enter( &__pwcFrame, __GRM->getBackRef(-1));
82:       __ok = ((Parameter.I > 0));
83:          if (__ok) {
84:            __tim = 0;
85:            __toState = _GAS;
86:            __inter = __cinter;
87:            __wcs_Data = *( _Data*) __cinter;
88:            __frame1 = &__wcf_Data;
89:            __frame2 = 0;
90:            __sip = &FromFeeder;
91:            __transBlock = &_ToGas;
92:            if (__wantToFire("_ToGas")) {
93:               __GRM->leave();
94:               goto __EXEC;
95:            }
96:          }
97:          __GRM->leave();
98:        }
99:   }
100:  {
101:   __ok = ( (__currentState==_LIQUID));
102:   if (__ok &&(__cinter=FromFeeder.firstIs(62))) {
103:     __frame_Data __pwcFrame;
104:     _Data_type& Parameter=((_Data*) __cinter)->Parameter;
105:     __pwcFrame.Parameter =& Parameter;
106:     __GRM->enter( &__pwcFrame, __GRM->getBackRef(-1));
107:      __ok = ((Parameter.I <= 0));
108:         if (__ok) {
109:           __tim = 0;
110:           __toState = _SOLID;
111:           __inter = __cinter;
112:           __wcs_Data = *( _Data*) __cinter;
113:           __frame1 = &__wcf_Data;
114:           __frame2 = 0;
```

```
115:           __sip = &FromFeeder;
116:           __transBlock = &_ToSolid;
117:           if (__wantToFire("_ToSolid")) {
118:              __GRM->leave();
119:              goto __EXEC;
120:           }
121:         }
122:         __GRM->leave();
123:       }
124:  }
125:  {
126:   __ok = ( (__currentState==_LIQUID));
127:   if (__ok &&(__cinter=FromFeeder.firstIs(62))) {
128:     __frame_Data __pwcFrame;
129:     _Data_type& Parameter=((_Data*) __cinter)->Parameter;
130:     __pwcFrame.Parameter =& Parameter;
131:     __GRM->enter( &__pwcFrame, __GRM->getBackRef(-1));
132:      __ok = ((Parameter.I == 99));
133:        if (__ok) {
134:          __tim = 0;
135:          __toState = _FINISHED;
136:          __inter = __cinter;
137:          __wcs_Data = *( _Data*) __cinter;
138:          __frame1 = &__wcf_Data;
139:          __frame2 = 0;
140:          __sip = &FromFeeder;
141:          __transBlock = &_ToFinished;
142:          if (__wantToFire("_ToFinished")) {
143:             __GRM->leave();
144:             goto __EXEC;
145:          }
146:        }
147:        __GRM->leave();
148:      }
149:  }
150:  __EXEC:
151:    if (__transBlock) {
152:       waitForResumeIfSingleStep( this);
153:       __mayExecuteNext = 0;
154:       if ( __frame1) __GRM->enter(__frame1,__GRM->getBackRef(-1));
155:       if ( __frame2) __GRM->enter(__frame2,__GRM->getBackRef(-1));
156:       if (__inter) __sip->dequeue();
157:       __transBlock( __GRM->getBackRef(-1),this,__GRM);
158:       __switchState( __toState);
159:       if (__inter) delete(__inter);
160:       if ( __frame1) __GRM->leave();
161:       if ( __frame2) __GRM->leave();
162:       if (__tim) __tim->reset();
163:       __localExecution();
```

```
164:        __childrenUpdate( __dt);
165:        return 1;
166:    }
167:    else  return 0;
168: }
```

A.3 Execution of TriState Log and Trace Files

```
 1: Trace of _Feeding_body

 3: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
 4: ToMain:_Data
 5: {  { { 2 4 6 8 10 12 14 16 18 20 } 0 1 1}}
 6: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
 7: ToMain:_Data
 8: {  { { 3 5 7 9 11 13 15 17 19 21 } 1 0 2}}
 9: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
10: ToMain:_Data
11: {  { { 4 6 8 10 12 14 16 18 20 22 } -2 1 3}}
12: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
13: ToMain:_Data
14: {  { { 5 7 9 11 13 15 17 19 21 23 } 3 0 4}}
15: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
16: ToMain:_Data
17: {  { { 6 8 10 12 14 16 18 20 22 24 } -4 1 5}}
18: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
19: ToMain:_Data
20: {  { { 7 9 11 13 15 17 19 21 23 25 } 5 0 6}}
21: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
22: ToMain:_Data
23: {~{ { 8 10 12 14 16 18 20 22 24 26 } -6 1 7}}
24: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
25: ToMain:_Data
26: {  { { 9 11 13 15 17 19 21 23 25 27 } 7 0 8}}
27: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
28: ToMain:_Data
29: {  { { 10 12 14 16 18 20 22 24 26 28 } -8 1 9}}
30: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
31: ToMain:_Data
32: {  { { 11 13 15 17 19 21 23 25 27 29 } 9 0 10}}
33: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
34: ToMain:_Data
35: {  { { 12 14 16 18 20 22 24 26 28 30 } -10 1 11}}
36: >> _Feeding_body@18857-0+2055~champlain.IRO.UMontreal.CA
37: ToMain:_Data
38: {  { { 12 14 16 18 20 22 24 26 28 30 } 99 1 97}}
```

```
39:
40: Trace of _Main_body
41:
42: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
43: FromFeeder:_Data_response
44:
45: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
46: FromFeeder:_Data_response
47:
48: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
49: FromFeeder:_Data_response
50:
51: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
52: FromFeeder:_Data_response
53:
54: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
55: FromFeeder:_Data_response
56:
57: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
58: FromFeeder:_Data_response
59:
60: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
61: FromFeeder:_Data_response
62:
63: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
64: FromFeeder:_Data_response
65:
66: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
67: FromFeeder:_Data_response
68:
69: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
70: FromFeeder:_Data_response
71:
72: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
73: FromFeeder:_Data_response
74:
75: >> _Main_body@18856-0+2052~champlain.IRO.UMontreal.CA
76: FromFeeder:_Close_connection
77:
78: Log file for _Feeding_body
79:
80: >>> executing transition _ToSending
81: >>> executing transition _Send_packet
82: >>> executing transition _Finished_waiting
83: >>> executing transition _Send_packet
84: >>> executing transition _Finished_waiting
85: >>> executing transition _Send_packet
86: >>> executing transition _Finished_waiting
87: >>> executing transition _Send_packet
```

```
 88: >>> executing transition _Finished_waiting
 89: >>> executing transition _Send_packet
 90: >>> executing transition _Finished_waiting
 91: >>> executing transition _Send_packet
 92: >>> executing transition _Finished_waiting
 93: >>> executing transition _Send_packet
 94: >>> executing transition _Finished_waiting
 95: >>> executing transition _Send_packet
 96: >>> executing transition _Finished_waiting
 97: >>> executing transition _Send_packet
 98: >>> executing transition _Finished_waiting
 99: >>> executing transition _Send_packet
100: >>> executing transition _Finished_waiting
101: >>> executing transition _Send_packet
102: >>> executing transition _Finished_waiting
103: >>> executing transition _ToDone
104:
105: Log file for _Main_body
106:
107: >>> executing transition _INIT_Trans
108: >>> executing transition _ToSolid
109: >>> executing transition _ToLiquid
110: >>> executing transition _ToGas
111: >>> executing transition _ToLiquid
112: >>> executing transition _ToSolid
113: >>> executing transition _ToLiquid
114: >>> executing transition _ToGas
115: >>> executing transition _ToLiquid
116: >>> executing transition _ToSolid
117: >>> executing transition _ToLiquid
118: >>> executing transition _ToGas
119: >>> executing transition _ToLiquid
120: >>> executing transition _ToSolid
121: >>> executing transition _ToLiquid
122: >>> executing transition _ToGas
123: >>> executing transition _ToLiquid
124: >>> executing transition _ToSolid
125: >>> executing transition _ToLiquid
126: >>> executing transition _ToGas
127: >>> executing transition _ToLiquid
128: >>> executing transition _ToSolid
129: >>> executing transition _ToLiquid
130: >>> executing transition _ToFinished
```

A.4 An Example of TANGO's generateFireable() Method

```
 1: PDList __MI_Main_body::__generateFireable() {
 2:  // Finds all fireable trans, returns them in a PDList
 3:  // also sets mayExecuteNext to indicate when this module could
 4:  // execute a transition without further inputs;
 5:  int __ok;
 6:  __trans_info *ti;
 7:  PDList retval = 0;
 8:  __Timer *__tim=0, *__ctim=0;
 9:  __Interact* __inter, *__cinter;
10:  _Data __wcs_Data;
11:  __frame_Data __wcf_Data;
12:  __wcf_Data.Parameter = & __wcs_Data.Parameter;
13: #ifdef OPTIMIZE
14:  __mayExecuteNext = -__MAXNOLOOP;
15: #else
16:  __mayExecuteNext = 0;
17: #endif
18:   ti = 0;
19:  {
20:   __ok = 1;
21:      __ctim = &__timers._ToLiquid;
22:      __ok = __ctim->fireable() && (!__ctim->optional() ||
23:             __wantToConsiderOpt("_ToLiquid"));
24:      __mayExecuteNext = __ctim->isSet()?
25:           max(__mayExecuteNext,__ctim->beforeFireable()):__mayExecuteNext;
26:      if (__ok) {
27:        ti = new __trans_info;
28:        ti->__frame1 = ti->__frame2 = 0;
29:        ti->__inter = 0;
30:        ti->__tim = __ctim;
31:        ti->__toState = _LIQUID;
32:        ti->__transBlock = &_ToLiquid;
33:        ti->transName = "_ToLiquid";
34:      }
35:  }
36:  // Place trans info onto move array:
37:    if (ti) {
38:       if (retval==0) retval = new DList;
39:       retval->append(ti);
40:       }
41:   ti = 0;
42:  {
43:   __ok = ( (__currentState==_LIQUID));
44:   if (__ok &&(__cinter=FromFeeder.firstIs(64))) {
45:     __frame_Data __pwcFrame;
46:     _Data_type& Parameter=((_Data*) __cinter)->Parameter;
47:     __pwcFrame.Parameter =& Parameter;
48:     __GRM->enter( &__pwcFrame, __GRM->getBackRef(-1));
```

```
49:      __ok = ((Parameter.I > 0));
50:         if (__ok) {
51:           ti = new __trans_info;
52:           ti->__tim = 0;
53:           ti->__toState = _GAS;
54:           ti->__inter = __cinter;
55:           __wcs_Data = *( _Data*) __cinter;
56:           ti->__frame1 = __wcf_Data.clone_frame();
57:           ti->__frame2 = 0;
58:           ti->__sip = &FromFeeder;
59:           ti->__transBlock = &_ToGas;
60:           ti->transName = "_ToGas";
61:         }
62:         __GRM->leave();
63:       }
64:  }
65:  // Place trans info onto move array:
66:    if (ti) {
67:       if (retval==0) retval = new DList;
68:       retval->append(ti);
69:       }
70:   ti = 0;
71:  {
72:   __ok = ( (__currentState==_LIQUID));
73:   if (__ok &&(__cinter=FromFeeder.firstIs(64))) {
74:     __frame_Data __pwcFrame;
75:     _Data_type& Parameter=((_Data*) __cinter)->Parameter;
76:     __pwcFrame.Parameter =& Parameter;
77:     __GRM->enter( &__pwcFrame, __GRM->getBackRef(-1));
78:      __ok = ((Parameter.I <= 0));
79:         if (__ok) {
80:           ti = new __trans_info;
81:           ti->__tim = 0;
82:           ti->__toState = _SOLID;
83:           ti->__inter = __cinter;
84:           __wcs_Data = *( _Data*) __cinter;
85:           ti->__frame1 = __wcf_Data.clone_frame();
86:           ti->__frame2 = 0;
87:           ti->__sip = &FromFeeder;
88:           ti->__transBlock = &_ToSolid;
89:           ti->transName = "_ToSolid";
90:         }
91:         __GRM->leave();
92:       }
93:  }
94:  // Place trans info onto move array:
95:    if (ti) {
96:       if (retval==0) retval = new DList;
97:       retval->append(ti);
```

```
 98:       }
 99:   ti = 0;
100:  {
101:   __ok = ( (__currentState==_LIQUID));
102:   if (__ok &&(__cinter=FromFeeder.firstIs(64))) {
103:     __frame_Data __pwcFrame;
104:     _Data_type& Parameter=((_Data*) __cinter)->Parameter;
105:     __pwcFrame.Parameter =& Parameter;
106:     __GRM->enter( &__pwcFrame, __GRM->getBackRef(-1));
107:      __ok = ((Parameter.I == 99));
108:         if (__ok) {
109:           ti = new __trans_info;
110:           ti->__tim = 0;
111:           ti->__toState = _FINISHED;
112:           ti->__inter = __cinter;
113:           __wcs_Data = *( _Data*) __cinter;
114:           ti->__frame1 = __wcf_Data.clone_frame();
115:           ti->__frame2 = 0;
116:           ti->__sip = &FromFeeder;
117:           ti->__transBlock = &_ToFinished;
118:           ti->transName = "_ToFinished";
119:         }
120:         __GRM->leave();
121:       }
122:  }
123:  // Place trans info onto move array:
124:    if (ti) {
125:       if (retval==0) retval = new DList;
126:       retval->append(ti);
127:       }
128:   return (retval);
129: }
```

A.5 TriState TAM Log, Analyzing Trace from Appendix A.3

```
0: Log file for _Main_body@23517-0+2611~champlain.IRO.UMontreal.CA
1: >>> Executing Transition _INIT_Trans
2: No tango.cfg found- using default settings.
3:
4: TANGO version 1.55 Trace analysis log.
5: Runtime Options:
6:    1    debug_trace level (debugging info during trace)
7:    off  io_relative (In/Out Relative checking)
8:    off  ip_relative (Interaction Point Relative checking)
9:    off  debug_load (Debug info during trace file load)
```

```
10:    off  search_init_state (initial state search mode)
11:    off  initial_suspend (wait for user to open window and resume)
12:    off  all_ips (wait for inputs to arrive at all IPs before commence search)
13:    off  continuous_read (keep waiting for input)
14:
15: Trace files to read:
16:   Static    : trace
17:
18: IP Queue Stats:
19:    FromFeeder: 12 outputs to verify
20: Executed startTAExec
21:
22: currentState = LIQUID
23: depth:0  check_again: 0  Transitions: 1
24:   _ToSolid
25:  ::: Executing Transition: _ToSolid
26:  --> Input   : _Data  [1]   from FromFeeder
27:
28: currentState = SOLID
29: depth:1  check_again: 0  Transitions: 1
30:   _ToLiquid
31:  ::: Executing Transition: _ToLiquid
32:  <-- Output  : _Data_response    to FromFeeder
33:
34: currentState = LIQUID
35: depth:2  check_again: 0  Transitions: 1
36:   _ToGas
37:  ::: Executing Transition: _ToGas
38:  --> Input   : _Data  [2]   from FromFeeder
39:
40: currentState = GAS
41: depth:3  check_again: 0  Transitions: 1
42:   _ToLiquid
43:  ::: Executing Transition: _ToLiquid
44:  <-- Output  : _Data_response    to FromFeeder
45:
46: currentState = LIQUID
47: depth:4  check_again: 0  Transitions: 1
48:   _ToSolid
49:  ::: Executing Transition: _ToSolid
50:  --> Input   : _Data  [3]   from FromFeeder
51:
52: currentState = SOLID
53: depth:5  check_again: 0  Transitions: 1
54:   _ToLiquid
55:  ::: Executing Transition: _ToLiquid
56:  <-- Output  : _Data_response    to FromFeeder
57:
58: currentState = LIQUID
```

```
 59: depth:6  check_again: 0  Transitions: 1
 60:   _ToGas
 61:  ::: Executing Transition: _ToGas
 62:  --> Input   : _Data  [4]   from FromFeeder
 63:
 64: currentState = GAS
 65: depth:7  check_again: 0  Transitions: 1
 66:   _ToLiquid
 67:  ::: Executing Transition: _ToLiquid
 68:  <-- Output  : _Data_response    to FromFeeder
 69:
 70: currentState = LIQUID
 71: depth:8  check_again: 0  Transitions: 1
 72:   _ToSolid
 73:  ::: Executing Transition: _ToSolid
 74:  --> Input   : _Data  [5]   from FromFeeder
 75:
 76: currentState = SOLID
 77: depth:9  check_again: 0  Transitions: 1
 78:   _ToLiquid
 79:  ::: Executing Transition: _ToLiquid
 80:  <-- Output  : _Data_response    to FromFeeder
 81:
 82: currentState = LIQUID
 83: depth:10  check_again: 0  Transitions: 1
 84:   _ToGas
 85:  ::: Executing Transition: _ToGas
 86:  --> Input   : _Data  [6]   from FromFeeder
 87:
 88: currentState = GAS
 89: depth:11  check_again: 0  Transitions: 1
 90:   _ToLiquid
 91:  ::: Executing Transition: _ToLiquid
 92:  <-- Output  : _Data_response    to FromFeeder
 93:
 94: currentState = LIQUID
 95: depth:12  check_again: 0  Transitions: 1
 96:   _ToSolid
 97:  ::: Executing Transition: _ToSolid
 98:  --> Input   : _Data  [7]   from FromFeeder
 99:
100: currentState = SOLID
101: depth:13  check_again: 0  Transitions: 1
102:   _ToLiquid
103:  ::: Executing Transition: _ToLiquid
104:  <-- Output  : _Data_response    to FromFeeder
105:
106: currentState = LIQUID
107: depth:14  check_again: 0  Transitions: 1
```

```
108:   _ToGas
109:  ::: Executing Transition: _ToGas
110:  --> Input   : _Data  [8]   from FromFeeder
111:
112: currentState = GAS
113: depth:15  check_again: 0  Transitions: 1
114:   _ToLiquid
115:  ::: Executing Transition: _ToLiquid
116:  <-- Output  : _Data_response    to FromFeeder
117:
118: currentState = LIQUID
119: depth:16  check_again: 0  Transitions: 1
120:   _ToSolid
121:  ::: Executing Transition: _ToSolid
122:  --> Input   : _Data  [9]   from FromFeeder
123:
124: currentState = SOLID
125: depth:17  check_again: 0  Transitions: 1
126:   _ToLiquid
127:  ::: Executing Transition: _ToLiquid
128:  <-- Output  : _Data_response    to FromFeeder
129:
130: currentState = LIQUID
131: depth:18  check_again: 0  Transitions: 1
132:   _ToGas
133:  ::: Executing Transition: _ToGas
134:  --> Input   : _Data  [10]   from FromFeeder
135:
136: currentState = GAS
137: depth:19  check_again: 0  Transitions: 1
138:   _ToLiquid
139:  ::: Executing Transition: _ToLiquid
140:  <-- Output  : _Data_response    to FromFeeder
141:
142: currentState = LIQUID
143: depth:20  check_again: 0  Transitions: 1
144:   _ToSolid
145:  ::: Executing Transition: _ToSolid
146:  --> Input   : _Data  [11]   from FromFeeder
147:
148: currentState = SOLID
149: depth:21  check_again: 0  Transitions: 1
150:   _ToLiquid
151:  ::: Executing Transition: _ToLiquid
152:  <-- Output  : _Data_response    to FromFeeder
153:
154: currentState = LIQUID
155: depth:22  check_again: 0  Transitions: 2
156:   _ToGas
```

```
157:   _ToFinished
158:  ::: Executing Transition: _ToGas
159:  --> Input   : _Data  [12]   from FromFeeder
160:
161: currentState = GAS
162: depth:23  check_again: 0  Transitions: 1
163:   _ToLiquid
164:  ::: Executing Transition: _ToLiquid
165:  <-- Output  : _Data_response    to FromFeeder
166:  !!! Mismatch: _Close_connection[24]
167: Backtracking...
168:
169: currentState = LIQUID
170: depth:22  check_again: 0  Transitions: 2
171:   _ToGas [Already tried]
172:   _ToFinished
173:  ::: Executing Transition: _ToFinished
174:  --> Input   : _Data  [12]   from FromFeeder
175:  <-- Output  : _Close_connection    to FromFeeder
176: All outputs were verified at this state
177:
178:
179:  CPU time (seconds): .116662
180:  Trans executed: 25. Generates: 24. Depth: 22. Max Depth: 23.
181:  Restores: 1. Saves: 1.  Transitions per second: 214.294
182:
183: Normal termination of program...
184:
```

Appendix B

Transport Protocol 0, Specified in Estelle

```
Specification TP0;
default individual queue;
timescale seconds;

    { Primitive functions removed by SAE, replaced by pascal procedures }

    { This is the top level module body (specification)
      The specification has the attribute systemprocess
      and all its children ( tp0 ) are processes.
      The time scale for delays is in seconds.                }

type  { ... is used to specify that an implementer
              must define these types for his environment.}
reason_type       = (none, user_init,
                {from X.214, Transport Service disconnect reasons of a tdind:}
                remote_TS_user_invoked, local_TS_provider_invoked,
                {from X.214, Transport Service user (add. info when
                disc_reason=local_TS_provider_invoked) reasons of a tdind:}
                lack_resource, qts_below_min, misbehaviour_TS_provider,
                called_TS_user_unknown, called_TS_user_unavailable, unknown_reason,
                {from X.224, Transport Protocol reasons of a DR:}
                not_specified, congestion_TSAP,
                no_session_attached, address_unknown,
                {from X.213, Network Service reasons:}
                disc_normal_condition, disc_abnormal_condition,
                conn_reject_permanent_cond, conn_reject_transient_cond,
                conn_reject_QOS_na_transient, conn_reject_QOS_na_permanent,
```

```
                conn_reject_bad_info_user_data);

ref_type        = 0..65535; { 0..2**16-1 }
tpdu_size_type  = (no_size, s128, s256);
option_type     = (normal, other_option);
addr_type       = packed array [1..25] of char;
data_type       = packed array [1..128] of char;

qts_type        = (low, medium, high);  {Quality of Transport Service: must be a
                                         list of parameters as described in X.214,
                                         but to simplify we define as this}

{ Channel definitions for communication between the processes }

channel U_access_point(User,Provider);

    by Provider:
        tdind(t_disc_reason: reason_type; ts_user_reason: reason_type);
        tcind(to_t_addr:  addr_type; from_t_addr: addr_type; qts_pro: qts_type);
        tccon(qts_res: qts_type);
        tdati(tsdu_fragment: data_type);
    by User:
        tcreq(to_t_addr: addr_type; from_t_addr: addr_type; qts_req: qts_type);
        tcres(qts_req: qts_type);
        tdreq(ts_user_reason: reason_type);
        tdatr(tsdu_fragment: data_type);

channel N_access_point(User,Provider);

    by User, Provider:
        cr(source_ref: ref_type; option: option_type; calling_addr: addr_type;
           called_addr: addr_type; max_tpdu_size: tpdu_size_type);
        dt(user_data: data_type);
        cc(dest_ref: ref_type; source_ref: ref_type; calling_addr: addr_type;
           called_addr: addr_type; max_tpdu_size: tpdu_size_type);
        dr(dest_ref: ref_type; disconnect_reason: reason_type;
           add_clear_reason: reason_type);
                { All add_reason in a DR is user defined according to X.224 }
    by User:
        ndreq(disc_reason: reason_type);
    by Provider:
        ndind;
        nrind;

{ Module header definitions }

module TESTER_type systemprocess; {This is the feeding module}
        ip
          U: U_access_point(User)        individual queue;
```

```
          L: N_access_point(Provider)        individual queue;
end;
body TESTER_body for TESTER_type; external;


module TP0_type systemprocess;
        ip {interaction point list }
          U:  U_access_point(Provider) individual queue;
          L:  N_access_point(User)     individual queue;
end; { of module header definition }

    { The module has two interaction points named U and L;
      the roles of the module are named:
          Provider with respect to U, and
          User with respect to L.          }

{ The body for TP0 is defined below: }

body body_tp0 for TP0_type;

type

{ nodeptr, nodetype, qtype all added by SAE in July 94 }

   nodeptr = ^nodetype;
   nodetype = record
     data : data_type;
     next : nodeptr;
     end;

   qtype = record
     fi, fo : nodeptr;
     count : integer;
     end;

   buffer_type = qtype;

 var
    in_buffer, out_buffer:          buffer_type;
    local_refer, remote_refer:      ref_type;
    tpdu_size:                      tpdu_size_type;
    qts_estimate:                   qts_type;
    calling_t_addr, called_t_addr:  addr_type;
    tsdu_fragment, user_data:       data_type;
    ts_disc_reason, ts_user_reason,
    disconnect_reason, add_clear_reason: reason_type;

 state idle, wfcc, wftr, data; { state definition part }
```

```
function qts_OK(qts_req: qts_type): boolean;
  begin
  { accepting qts low and medium but not high }
    if qts_req <= medium then qts_OK := true else qts_OK := false
end;

function option_OK(option: option_type): boolean;
  begin
    if option = normal then option_OK := true
       else option_OK := false
end;

procedure assign_local_ref(var local_ref: ref_type );
  begin
     local_ref := 1; { 0 is forbidden because it means there are no local_ref assigned }
  end;

procedure assign_tpdu_size(var tpdu_size: tpdu_size_type);
  begin
    tpdu_size := s128;
  end;

procedure assign_d_reason(var ts_disc_reason: reason_type; new_reason: reason_type);
  begin
    ts_disc_reason := new_reason;
  end;

procedure assign_u_reason(var ts_user_reason: reason_type; new_reason: reason_type);
  begin
    ts_user_reason := new_reason;
  end;

procedure assign_reason(var disconnect_reason: reason_type; new_reason: reason_type);
  begin
    disconnect_reason := new_reason;
  end;

procedure assign_ac_reason(var add_clear_reason: reason_type; new_reason: reason_type);
  begin
    add_clear_reason := new_reason;
  end;

procedure assign_qts(var qts_estimate: qts_type);
  begin
    qts_estimate := medium;
  end;

{procedures remove, add, init_buffer, and insert re-written in Estelle
  by SAE; originally they were C++ primitive functions by Daniel Ouimet}
```

```
procedure remove( var q: buffer_type; var fragment: data_type);
   var
    n : nodeptr;
   begin
     n := q.fo;
     if q.fo <> NIL then
       begin
         fragment := n^.data;
         q.count := q.count - 1;
         q.fo := q.fo^.next;
         if q.fo = NIL then q.fi := NIL;
         dispose(n);
       end;
     end;

 procedure add (var q : qtype; var p : nodeptr);
   begin
     p^.next := NIL;
     if (q.fi <> NIL) then q.fi^.next := p;
     q.fi := p;
     q.count := q.count + 1;
     if q.fo = NIL then q.fo := p;
   end;

procedure init_buffer(var q:buffer_type);
  begin
    q.fi := NIL;
    q.fo := NIL;
    q.count := 0;
  end;

procedure insert(var q: buffer_type; var i: data_type);
   var
     n : nodeptr;
   begin
     new (n);
     n^.data := i;
     add (q,n);
   end;


initialize { initialization-part of the alternating bit process }
to idle  { initialize major state variable to idle }
  begin  { initialize variables }
   local_refer        := 0;
   init_buffer(in_buffer);
   init_buffer(out_buffer);
  end;
```

```
{-------------------------------------------------------------------------}
{ NFS for Class 0 Transport Protocol
  As defined by Hassan Ural and Bo Yang in "A test sequence selection
  method for protocol testing" in IEEE Trans. on Communications, Vol. 39,
  No. 4, April 1991.
  Many corrections to the syntax were made by Daniel Ouimet in
        December 93 and January 94.
}

trans {  transition-declaration-part of the TP0 process   }
  WHEN U.tcreq(to_t_addr, from_t_addr, qts_req)
  FROM idle
  PROVIDED qts_OK(qts_req)
  TO   wfcc
  NAME t1: BEGIN
    assign_local_ref(local_refer);
    assign_tpdu_size(tpdu_size);
    calling_t_addr := from_t_addr;
    called_t_addr  := to_t_addr;
    output L.cr (local_refer, normal, calling_t_addr, called_t_addr, tpdu_size)
  END;

trans
  WHEN U.tcreq {(nil, nil, qts_req)}
  FROM idle
  PROVIDED not qts_OK(qts_req)
  TO   idle
  NAME t2: BEGIN
    assign_d_reason(ts_disc_reason, local_TS_provider_invoked);
    assign_u_reason(ts_user_reason, qts_below_min);
    output U.tdind(ts_disc_reason, ts_user_reason);
  END;

trans
  WHEN L.cr (source_ref, option, calling_addr, called_addr, max_tpdu_size)
  FROM idle
  PROVIDED (max_tpdu_size <> no_size)  and option_OK(option)
  TO   wftr
  NAME t3: BEGIN
    remote_refer := source_ref;
    tpdu_size    := max_tpdu_size;
    calling_t_addr := calling_addr;
    called_t_addr  := called_addr;
    assign_qts(qts_estimate);
    output U.tcind(called_t_addr, calling_t_addr, qts_estimate);
  END;

trans
```

```
  WHEN L.cr (source_ref, option, calling_addr, called_addr, max_tpdu_size)
  FROM idle
  PROVIDED (max_tpdu_size = no_size)  and option_OK(option)
  TO   wftr
  NAME t4: BEGIN
    remote_refer := source_ref;
    assign_tpdu_size(tpdu_size);
    calling_t_addr := calling_addr;
    called_t_addr  := called_addr;
    assign_qts(qts_estimate);
    output U.tcind(called_t_addr, calling_t_addr, qts_estimate);
  END;

trans
  WHEN L.cr
  FROM idle
  PROVIDED not option_OK(option)
  TO   idle
  NAME t5: BEGIN
    assign_reason(disconnect_reason, not_specified);
    output L.dr(source_ref, disconnect_reason, none);
  END;

trans
  WHEN L.cc
  FROM wfcc
  PROVIDED max_tpdu_size <> no_size
  TO   data
  NAME t6: BEGIN
    assign_qts(qts_estimate);
    output U.tccon(qts_estimate);
  END;

trans
  WHEN L.cc
  FROM wfcc
  PROVIDED max_tpdu_size = no_size
  TO   data
  NAME t7: BEGIN
    assign_qts(qts_estimate);
    output U.tccon(qts_estimate);
  END;

trans
  WHEN L.dr
  FROM wfcc
  PROVIDED disconnect_reason = user_init
  TO   idle
  NAME t8: BEGIN
```

```
    output L.ndreq(disconnect_reason);
    output U.tdind(add_clear_reason, disconnect_reason);
  END;

trans
  WHEN L.dr
  FROM wfcc
  PROVIDED disconnect_reason <> user_init
  TO   idle
  NAME t9: BEGIN
    output L.ndreq(disconnect_reason);
    output U.tdind(none, disconnect_reason);
  END;

trans
  WHEN U.tcres (qts_req)
  FROM wftr
  PROVIDED qts_req <= qts_estimate
  TO   data
  NAME t10: BEGIN
    assign_local_ref(local_refer);
    output L.cc (remote_refer, local_refer, calling_t_addr, called_t_addr, tpdu_size);
  END;

trans
  WHEN U.tcres(qts_req)
  FROM wftr
  PROVIDED qts_req > qts_estimate
  TO   idle
  NAME t11: BEGIN
    assign_reason(disconnect_reason, not_specified);
    assign_ac_reason(add_clear_reason, qts_below_min);
    assign_d_reason(ts_disc_reason, qts_below_min);
    output L.dr(remote_refer, disconnect_reason, add_clear_reason);
    output U.tdind(none, ts_disc_reason);
  END;

trans
  WHEN U.tdreq(ts_user_reason)
  FROM wftr
  TO   idle
  NAME t12: BEGIN
    assign_reason(disconnect_reason, ts_user_reason);
    output L.dr (remote_refer, disconnect_reason, add_clear_reason)
  END;

trans
  WHEN U.tdatr(tsdu_fragment)
  FROM data
```

```
  TO   data
  NAME t13: BEGIN
    insert(out_buffer, tsdu_fragment);
  END;

trans
  FROM data
  TO   data
  PROVIDED out_buffer.count > 0 {not buffer_empty(out_buffer)}
  NAME t14: BEGIN
    remove(out_buffer, user_data);
    output L.dt(user_data);
  END;

trans
  WHEN L.dt(user_data)
  FROM data
  TO   data
  NAME t15: BEGIN
    insert(in_buffer, user_data);
  END;

trans
  FROM data
  PROVIDED in_buffer.count > 0 {buffer_empty(in_buffer)}
  TO   data
  NAME t16: BEGIN
    remove(in_buffer, tsdu_fragment);
    output U.tdati(tsdu_fragment);
  END;

trans
  WHEN U.tdreq(ts_user_reason)
  FROM data
  TO   idle
  NAME t17: BEGIN
    output L.ndreq(ts_user_reason);
  END;

trans
  WHEN L.ndind
  FROM data
  TO   idle
  NAME t18: BEGIN
    assign_d_reason(ts_disc_reason, local_TS_provider_invoked);
    output U.tdind(none, ts_disc_reason);
  END;

trans
```

```
  WHEN L.nrind
  FROM data
  TO   idle
  NAME t19: BEGIN
    assign_d_reason(ts_disc_reason, remote_TS_user_invoked);
    output U.tdind(none, ts_disc_reason);
  END;
end; {  of the body_tp0}


modvar
    { module-variable-declaration-part of the specification }

    TP0_var: TP0_type;
    TESTER_var: TESTER_type;

initialize { initialization-part of the specification }

 begin { module initialization }

        begin
          init TP0_var with body_tp0;
          init TESTER_var with TESTER_body;
          connect TESTER_var.U to TP0_var.U;
          connect TESTER_var.L to TP0_var.L;
        end;
 end; { of module initialization within the
        specification's initialization-part }

end.  { End of specification; the specification has no transition part }
```

Appendix C

Tango Tutorial, Version 1.5

Tango can be used in two ways. The first, "implementation generation mode", will produce an executable implementation which behaves in the same way as a Dingo-generated implementation, except that traces are generated for each module, containing inputs and outputs involving that module. The second use of Tango is as a trace analysis tool generator, where the module which corresponds to the IUT is referred to as the TAM, or Trace Analysis Module. Each use is described in a section of this Appendix.

C.1 Implementation Generation

Any Estelle specification which has no `external` module body definitions can be put through Pet and Tango to generate an executable implementation which will behave identically to a Dingo-generated implementation. For more information on how such implementations can be used, see [36].

During an implementation execution, each *module* (as opposed to each channel) maintains a trace file, named *_modulename@pid.index~hostname*`.tra`, where *pid* is the process id, *index* is a unique string to distinguish it from other modules of the same name, and *hostname* is the name of the host executing the module process. Each interaction sent from, or received by module A is represented as an entry in module A's trace file. If module A receives an interaction from an IP, and that IP is connected to another IP in module B, then an identical entry for that interaction can be found in the trace file for module B as well.

The module which represents the IUT will generate a trace file which can be used as sample input for the TAM, since the trace file format used by the Tango-generated implementations is the same as the format used by a TAM.

C.2 Trace Analysis

Starting with an Estelle protocol specification and a trace of inputs to, and outputs from, one module of the IUT, this tutorial will provide step-by-step instructions on how to generate a trace-analyzer from the specification and how to analyze the trace given.

This section describes the following steps:

1. Creating a single-module test system specification.
2. Generating an executable Trace Analyzer
3. Formatting the tracefile.
4. Executing the Trace Analyzer
5. Viewing the results of the analysis

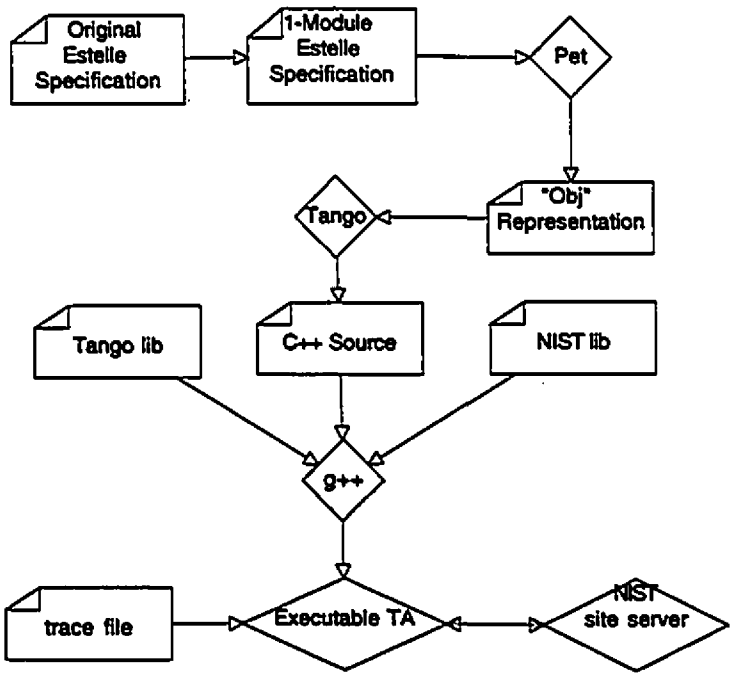

Figure C.1: The Tango System

C.2.1 Creating a Single-Module Specification

During implementation execution, an IUT communicates with other entities, hereafter referred to as modules. The IUT, and the modules it communicates with, comprise the **test system**. In figure 3.1, the test system is `Specification Example`, which includes two modules, `Main` and `Food`. The module specifying the IUT is `Module Main`.

The specification which is given as input to Tango must reflect the structure of the test system. Even though the module `Food` is not being tested, it still must be declared as a module in Tango's input specification, because it communicates with `Main` during testing.

Modules which communicate with the IUT module, when they are part of the trace analysis tool, "feed" inputs to the TAM at the beginning of the trace analysis. Thus, they are referred to as "feeding modules" in this thesis. Since the behaviour of feeding modules is defined in Tango, it does not need to be defined in the formal specification. For this reason, the convention of defining bodies of feeding modules as **external** has been adopted in Tango. The example specification in listing 1 has a module which is called `Feeding_Module` and can be transformed into a proper Tango feeding module as shown below:

```
module Feeding_Module systemprocess;
  ip
     toMain : interface (sender) individual queue;
end;

body Feeding_body for Feeding_Module; external;
{ --- end of feeding module --- }
```

Notice that the only declarations for `Feeding_Module` which were carried over from listing 1 are the `ip` declarations, which were not changed at all.

There are a number of possible faults which can exist in a specification yielding a trace analysis tool which will not work. It is recommended that the suggestions below are followed to ensure a working trace analyzer generation.

- Place external body definitions *first*, and the TAM specification *last*. Tango knows to generate a TAM instead of a regular Dingo implementation of a module by the existence of an `external` module body definition in the specification. Because Tango is a one-pass compiler, all external body definitions (i.e. feeding modules) must be declared above the IUT module specification. If no feeding modules appear before the IUT module specification, Tango generates an executable implementation rather than a trace analyzer for the IUT module,

not realizing that feeding module declarations appear afterwards. Think of the `external` keyword as a compiler directive to Tango.

- Make sure that only one module has a non-external body. If more than one module with a non-external body appears in the TAM specification, Tango will generate *independent* TAMs for each module. Regardless of whether the non-external modules are connected by channels, all the outputs from each module will be analyzed against the trace information rather than being sent to other modules. Therefore, unexpected results will occur.

- Avoid **primitive** functions. For backtracking to work, all functions and procedures which affect the TAM state must be defined in the Estelle specification. Functions and Procedures declared as `primitive` and defined in code elsewhere, which cause side effects in the running operation of the implementation, would cause unpredictable results when linked with a TAM.

- Remove **delay** clauses. It is recommended all `delay` statements be removed from the specification. They will only slow down the trace analysis. Time-dependent behaviors can not be checked by a TAM in any case.

- Double-check your connections. If the modules are not connected properly in the `modvar` section, the TAM may deadlock before performing any analysis.

- Set all module attributes properly. The last step in preparing the single-module specification is to set all the module attributes to `systemprocess`, and to remove the root specification attribute, if there is one.

C.2.2 Generating the Executable TAM

After the TAM specification is ready to be compiled, the following steps must be taken.

1. Generate an object-oriented static representation with **pet.**
   `pet -o` *objfile estellefile*

2. Generate C++ code from the the output produced by pet.
   `tango` *objfile*

   Tango generates a makefile template, called *specname*`.make.tmpl`, where *specname* is the name of the specification as described in the Estelle source.

3. `make` environment variables are left blank in the template, and must be filled with proper pathnames, as specified in the comments of the makefile template.

   Example:

```
LIBPATH= /home/champlain2/ezust/pde
XLIBDIR= /usr/local/lib/
MIWINDIR= /home/champlain2/ezust/pde
INCLPATH= /home/champlain2/ezust/pde
GNUINCLUDEDIR= /usr/local/lib/g++-include
```

Additionally, the LOCMODS environment variable should contain the names of the main specification, the TAM, and all feeding modules, separated by spaces. This specifies that all the modules are to be executed locally on the same machine.

Example:

```
LOCMODS = _Example _Main_body _Feeding_body
```

More information about the makefile template can be found in section 3.11 of [36].

4. If the previous steps were successful, the **make** command should compile and link the specification and create a set of executables, one for each module, and one for the root specification, called *_Specname*.

C.2.3 Formatting the Tracefile

The primary Tango trace file, which must be located in the same directory as the executable trace analysis tool, must be called "**trace**". It is a standard text file, with each interaction taking 3 lines of text.

Each entry looks like this:

```
>> _Module_name
Ip:_Interaction_type
{ arguments for the interaction}
```

While in the initial stages of generating trace files, it may be useful to see Tango's trace file debugging information, as it will often help the user to determine where incorrectly formatted information appears in the tracefile. Debugging information will be sent to the log file when the runtime option, **debug_load**, is specified. See Section C.2.4 for more information.

Module Name Specifier

The first line must begin with ">> _" followed by the name of the module that is sending the interaction. In Tango trace files, the module name is defined as the **module body** name, as it appears in the Estelle specification. If the test system has multiple instantiations of the same module body for feeding modules, then a unique string must be concatenated to the end of the module name for each instance, to distinguish it from other instances of that module.

For example, in an Estelle specification that declares an array of module instances like this:

```
Module Alternating_bit_type systemactivity;
  ...
body Alternating_bit_body for Alternating_bit_type; external;
  ...

modvar
    Alternating_bit: array [1..2] of Alternating_bit_type;

initialize
   begin { module initialization }
       init Alternating_bit[1] with Alternating_bit_body;
       init Alternating_bit[2] with Alternating_bit_body;
  ...
```

Dingo names each instance of _Alternating_bit uniquely as shown below:

```
>> _Alternating_bit_body@10489-0+3698
>> _Alternating_bit_body@10489-1+3698
```

When the modules run under the same process, the process id (which follows the @ sign) is the same for both modules, and the index of the array follows the - sign, as shown above. When they run as independent processes, the process ID differs, and the index of each module is 0. As long as a unique string is appended consistently to the name of each module, it does not matter to Tango what the string is.

Errors If a module name in a tracefile entry does not match any module in the TAM system specification, the entry will be ignored. No error message will be displayed.

Interaction Point/Interaction Specifier

The second line must begin with the interaction point name, as specified in the Estelle specification for the module which sent the interaction. If a specification uses an array of IPs, the index must be specified by concatenating an underscore followed by the 5-digit index, padded with leading zeros, to the end of the IP name.

Following the IP name is a ":_" followed by the name of the interaction, as specified in the Estelle specification.

Examples of valid second lines:

`N_00002:_DATA_response` (sends a `DATA_response` to interaction point `N[2]`)
`U:_RECEIVE_response` (sends a `RECEIVE_response` to interaction point `U`)

Errors If the IP specifier does not match the name of an IP in the module specification, an error message "`ip_index: Unable to find match for` *ip_name*" will appear in the module's log file, where *ip_name* is the name of of the IP specifier in the tracefile.

If the interaction type is not a valid interaction for the corresponding IP, an error message "`unable to create interact from` *interact*" will appear in the logfile, where *interact* is the name of the interaction type in the tracefile. During the loading of a tracefile, Tango does not check to ensure that the role of an IP is the right role for the interaction to be loaded. Needless to say, if an interaction does not match the proper role of the IP, and it appears in the trace file, the trace will be detected as invalid by the TAM.

The Interaction Parameter line

The third line of each trace file entry contains the parameters of the interaction. They are enclosed in curly brackets {}.

For simple types, such as integers, the values are printed as ASCII decimals. For boolean types, values are printed as '0' or '1'. For characters, values are printed as decimal numbers, in ASCII code i.e. the letter A is represented as the string '98', B is '99', etc... For Enumerated type variables, their ordinal values are printed as decimal integer strings, just like characters.

For records, the value of each field must be listed in the order it appears in the specification, and the entire record contents must be enclosed in curly brackets.

For arrays, likewise, the contents of the array must appear in between curly brackets, with the elements listed in index-order.

For strings (packed array of characters), a sequence of ASCII-decimal numbers representing each character in the string should appear in ascending order by index, and the entire string must be enclosed in double quote characters (").

For example, an interaction which has a parameter of the following `data_type`:

```
type
  data_type = record
    H : array[1..10] of integer;
    I : integer;
    J : boolean;
    K : char;
    L : packed array[1..4] of char;
    end;
```

might have a parameter line which looks like this:

```
{ { { 2 4 6 8 10 12 14 16 18 20 } 1 0 98 " 98 99 100 101 " } }
```

where `H[1] = 2, H[2] = 4, H[10] = 20, I = 99, J is false,` and the packed array `K = 'ABCD'`.

There should be no newline character in the middle of a data parameter trace file field. The parameter line can be as long as necessary to specify the values of every field in the parameter list.

Errors When curly brackets or string delimiters mismatch, or are missing when required, the trace analyzer may enter into an infinite loop and hang, without outputting an error message.

C.2.4 Runtime Options

TANGO supports a number of runtime options which provide flexibility and power to the user of a TAM. All runtime options are read by a TAM just before trace analysis begins.

Runtime options can be specified in a file called `tango.cfg`, which must appear in the same directory as the TAM executable. The file format is a standard textfile, which can be edited using any text editor. Each option is specified in the file on a line by itself. The order in which the options appear does not matter to Tango. Any invalid options are ignored, yielding informative error messages in the TAM log file.

Tango will ignore text on a line followed by a # character, so comments in the configuration file can be placed after it, or entire lines can be "commented out" in this fashion.

The current version of Tango supports the following runtime options.

Relative Order

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 interaction point 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* interaction points, or through the same interaction point but in different directions, can be **observed** (and checked) or **ignored** by the TAM, depending on the runtime options. Depending on the architecture of the IUT and its observation points, certain options should be used. See Section 6.3.2 for more information.

io_relative, or Inputs with respect to Outputs One form of relative order checking is that of inputs with respect to outputs passing through the same channel. For example, if the next input interaction waiting in the queue of a particular IP is i, and it appears in the trace file after an output o, passing through the same channel, which was not generated yet by the TAM, then consuming i before outputting o violates the I/O relative order of i and o. Enabling the I/O relative checking option will prevent the TAM from consuming i before outputting o.

oi_relative, or Outputs with respect to Inputs O/I relative order checking means that if o appears after i in the tracefile, and the TAM attempts to execute a transition which produces o before consuming i, this will cause an O/I relative order output mismatch.

ip_relative, or Interactions Passing Through Different IPs If the order of the interactions in the trace file reflects the relative order of interactions which passed through **different** interaction points, the TAM will respect this order during trace analysis when this runtime option is used.

Using this option ensures the following:

- The "next" input to be read from an IP x is only readable if it appears in the tracefile before "next" inputs to be read from all other IPs.
- An output o to an IP x will have an "ip relative" output mismatch if an output which has not been verified yet, due to be sent through another IP, appears before o in the tracefile.

debug_load, or Load Tracefile Debugging Info:

Debugging information can be sent to the log file of each module, during the reading of the trace file. To set this option on, place the following line in the configuration file: `debug_load`

Enabling this option will give the user a better idea of which interactions are invalid in the trace file, if the user is unsure that the format of the trace file is correct.

debug_trace, or Trace Analysis Debugging Info:

During trace analysis, it is possible to have the TAM include different degrees of detail in the debugging information of the log file. Using the runtime option `debug_trace=`, the user can set the *debug level* to reflect the desired quantity of output which will appear in the log file. Higher debugging levels slow down the actual search, and in some cases, the difference in search time on the same trace but using different debugging levels can be as great as 15%.

At the end of the analysis, as well as after every 1000 transitions taken, statistics about performance and number of transitions searched, regardless of the value of this runtime option, will be listed in the logfile. Other messages which will be sent to the logfile include: the final result (valid or invalid), and any output messages from the C++ implementation code.

Using a `debug_trace=0` option, *no other messages* will be sent to the log file. Using a debug trace level of 0 is useful if Tango is executing too many transitions for a more detailed log file to fit under your disk quota. This also ensures the fastest search possible using Tango.

A debug level of 1, the system default, is useful if it is desired to follow the path of execution without having too many extra details. The following additional information will be sent to the logfile.

- Search progress information, including the state, search tree depth, and a list of fireable transitions which were generated at each state.
- Interactions which were tested for conformance to the trace, and information about whether they match.
- backtracking status

With `debug_trace=2`, the following information, above and beyond what is listed when the debug level is 1, will be sent to the logfile.

- Frame contents for each generated transition
- Parameter contents for each interaction
- I/O relative order mismatch messages
- IP relative order mismatch messages

initial_suspend, or Initial Wait for User-Resume

If you wish to observe the execution of the trace analysis from the very beginning of the search through the Dingo X-windows site server, you may want to have the TAM initially suspend itself until the user intervenes with either a resume or a single-step event. With the option `initial_suspend`, the TAM will not start its analysis until the user

1. opens a module instance window for the TAM from the `Local Root-Modules` menu in the SiteServer
2. clicks on either the `Suspended` or the `Continuous Mode` menu buttons. The former will begin executing transitions as rapidly as the specification permits, while the latter will execute a single transition each time that menu button is selected.

disable_ip, or Disable Output Checking on an IP

If it is not possible to observe interactions going through some specified IPs in an IUT, it might be desirable to disable output checking on those IPs, while still checking outputs going through all of the other IPs.

Usage:

`disable_ip:`*ip_name*

Where *ip_name* is the symbolic name of that interaction point, as specified in the Estelle TAM specification. It should look exactly the same as the IP name in the trace file for interactions going through that IP.

all_ips, or Wait For Input To Arrive at All IPs

When the TAM and all of the feeding modules are spawned at the same time, it is possible that the TAM might begin its search before inputs have arrived at all of the IPs. This can cause certain transitions which should have been fireable to be non-fireable.

By default, a TAM waits 10 seconds before beginning the search. This runtime option, when enabled, will force the TAM to wait indefinitely, until at least one input has arrived at each IP.

If the `disable_ip` option is used on a particular IP, the TAM will not wait for data to arrive at that IP before commencing its search.

search_init_state, or Search for Valid Initial FSM State

When a TAM begins its analysis, the first transition that gets fired is `initialize`, and whatever state the FSM enters after that is called the *initial state*. In some

circumstances, an obtained trace might not begin with interactions generated by an IUT which was in this initial FSM state. Using this option, the TAM will try all possible initial FSM states with the trace given before outputting an invalid result.

Additional Trace Files

If you wish Tango to read from multiple trace files, additional trace files can be specified in the runtime options file. There are two kinds of trace files: **static** and **dynamic.**

A static trace file is read entirely before the search begins, and is never checked again during the analysis. A dynamic trace file will be periodically checked during the analysis in case additional traced interactions are appended to the file. Dynamic trace files are required for real-time on-line trace analysis.

To specify an additional static trace file, put the following line in tango.cfg:

`static:`*filename*

To specify an additional dynamic trace file, put the following line in tango.cfg:

`dynamic:`*filename*

The default trace file, simply called `trace`, is static, and will always be among the list of tracefiles from which each Tango module will attempt to read. If the default trace file is empty or non-existent, and other valid files were specified, the trace analysis will commence as expected, reading only from the valid trace files.

Since each trace file can contain interactions for any module and any IP in the test system, it is recommended that all interactions for one particular IP be placed in a single trace file, rather than distributed among multiple trace files. Each Tango module will read all of the available interactions in each trace file, starting with the default file, and continuing with each file, in order, as it appears in tango.cfg.

End Of Input Marker: In a dynamic trace file, to signal the end of input, a special interaction must be appended to the file. It begins with the following line:

```
>> END
```

At least two blank (or non-blank) lines must follow this. To ensure termination of the TAM search, these `>> END` markers must appear at the end of each dynamic trace file which is being used. See Section 7.2.2 for more information.

C.2.5 Executing the TAM - User Interface

The executable created in step 2 is not a stand-alone executable; it must be run under the NIST X Windows Dingo Site Server. In the Tango package there is a shell script, called `StartTangoSites`, which executes the Site Server on the local machine. After the Site Server window is active, the user can run the executable for the root module

from the Unix shell, which will in turn spawn processes for the TAM and the feeding modules. The user interface for the site server is described in detail in [36].

Double-clicking on the TAM module in the `Local Root Modules` list will open a module instance window, and from this window one may do the following:

- Suspend and single-step the trace analysis, using the menu buttons `suspended` and `Continuous Mode`
- Examine the `TAM status` from the `Local_Vars` pull-down menu, which will display statistics on the search in progress, as well as other diagnostic information.
- View the logfile as it was when last loaded, from the `Load Module Instance` *module_name* menu button. The logfile is loaded when the window is opened initially, and is re-loaded each time the user clicks on the button bar at the top of this window.

  There is a bug in Dingo which causes the module to crash if you attempt to load its logfile when it is too big to fit into memory. Therefore, if you know that the logfile will exceed a couple of megabytes, and you would like to monitor the `TAM status`, it is recommended that you open the module instance window near the beginning of the analysis, and refrain from clicking on the window's button bar during the search.

When the trace analysis is finished, a message `peer exited or abruptly disconnected` will appear on the standard output. The module instance window will automatically kill itself, if it exists.

C.2.6 Viewing the Results of the Analysis

Each module which is executed under the Site Server keeps its own log file. The filename for a module with name *modulename* is:
_modulename@pid.index~hostname`.log`

The trace analysis results can be viewed in the logfile for the TAM. If the debug_level was greater than 0, each transition the TAM attempted would be listed in the log file, in the order that it was tried. At the end of the file, a message `all outputs verified` or `trace is invalid` indicates the result of the analysis. See Section 6.4 for an example of a log file generated during trace analysis by a TAM.

The statistics given at the end of the trace, and after every 1000 transitions taken, are explained below:

CPU Time (seconds): This is the amount of CPU time used by the TAM process. Achieved by using the `clock(3C)` function.

Trans executed: The number of transitions executed during the search. This can also be thought of as the number of edges searched in the tree. During DFS, this is the sum of generates and restores.

Generates: The number of times a call was made to `_generateFireable`. During DFS, this is the number of vertices in the search tree.

Depth: The depth of the most recently searched node in the tree.

Max Depth: The maximum depth achieved in the tree during the search.

Restores: The number of state restores during the search. During DFS, this is the number of backtracks.

Saves: The number of state saves during the search. During DFS, this is the number of nodes with more than 1 child in the search tree.

Trans per second: This is the number of executed transitions divided by the CPU time, in seconds.

Appendix D

List of Abbreviations

BNR: Bell Northern Research

CCITT: International Consultative Committee for Telephones and Telegraphs.

CRIM: Centre de Recherche Informatique de Montreal

DFS: Depth-First Search

DINGO: Distributed ImplementatioN GeneratOr

EFSM: Extended Finite State Machine. See Section 2.1.

ESTL (or Estelle): Extended State Transition Language

FDT: Formal Description Technique

FSM: Finite State Machine

GRM: Global Reference Manager. See Section 4.1.2.

IP: Interaction Point. See Section 3.1.

ISDN: Integrated Services Digital Network

ISO: International Organization for Standardization

IUT: Implementation Under Test. See Section 5.2.

LAPD: Link Access Procedure D protocol

LOTOS: Language of Temporal Ordering Specification

MDFS: Multi-Threaded Depth-First Search. See Section 7.2.2.

MONDEL: Montreal Description Language

OSI: Open Systems Interconnection

PET: Portable Estelle Translator

PG-Node: A node with a Partially Generated transition list. See Section 7.2.2.

PGAV-Node: A PG-node with all inputs consumed, and all outputs verified. See Section 7.2.2.

SDL: Specification and Description Language.

TAM: Trace Analysis Module. See section 6.2.

TANGO: Trace ANalysis GeneratOr

Bibliography

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

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

[3] Bochmann, Barbeau, Erradi, Lecomte, Mondain-Monval, and Williams. "Mondel: An object-oriented specification language". Technical Report 748, CRIM and University of Montreal IRO Department, 1991.

[4] G. Bochmann, A. Das, R. Dssouli, M. Dubuc, and G. Luo. "Fault models in testing". Technical Report 786, Department IRO, University of Montreal, 1991.

[5] Gregor Bochmann. "Finite state description of communication protocols". *Computer Networks*, 2, North-Holland, 1978.

[6] Gregor Bochmann. "Semiautomatic implementation of communication protocols". *IEEE Transactions on Software Engineering*, SE-13(9), September 1987.

[7] Gregor Bochmann. "Protocol specification for OSI". *Computer Networks and ISDN Systems*, 18, North-Holland, 1989/90.

[8] Gregor Bochmann. "Specifications of a simplified transport protocol using different formal description techniques". Technical Report 623a, Department IRO, University of Montreal, April, 1987, revised June 1988.

[9] Gregor Bochmann, Rachida Dssouli, and J. R. Zhao. "Trace analysis for conformance and arbitration testing". *IEEE Transactions on Software Engineering*, 15(1), November 1989.

[10] Gregor Bochmann and Carl Sunshine. "Formal methods in communication protocol design". *IEEE Transactions on Communications*, 28(4), April 1980.

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

[12] S. Budkowski and P. Dembinski. "An introduction to Estelle: A specification language for distributed systems". *Computer Networks and ISDN Systems*, 14, North-Holland, 1987.

[13] The SPECS Consortium and J. Bruijning. "Evaluation and integration of specification languages". *Computer Networks and ISDN Systems*, 13, 1987.

[14] 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.

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

[16] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[17] Ove Færgemand and Anders Olsen. "Introduction to SDL-92". *Computer Networks and ISDN Systems*, 26, North-Holland Elsevier, 1994.

[18] Joachim Fischer and Eckhardt Holz. "Towards an object-oriented technology for specification and implementation of distributed systems". Technical report, Humboldt-University Berlin, Department of Computer Science, 1993.

[19] S. Fujiwara and G. Bochmann. "Testing non-deterministic state machines with fault coverage". *Protocol Test Systems*, IV, Elsevier Science Publishers B.B. (North-Holland), 1992.

[20] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. "Test selection based on finite state models". Technical Report 716, University of Montreal, 1990.

[21] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall Intl, 1985.

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

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

[24] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[25] *Estelle Tutorial.* Annex D (informative) of ISO 9074, Amendment 1, 1989.

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

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

[28] Gang Luo, Gregor Bochmann, and Alexandre Petrenko. "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-Method". *IEEE Transactions on Software Engineering*, 20(2), February 1994.

[29] 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.

[30] G.W. Neufeld and S. Vuong. "A tutorial on ASN.1". *Computer Networks and ISDN Systems*, North-Holland, 1992.

[31] Alexandre Petrenko, Gregor Bochmann, and Rachida Dssouli. "Conformance relations and test derivation". *Protocol Test Systems*, VI (C-19), Elsevier Science B.V. (North-Holland), 1994.

[32] 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.

[33] Philippe 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.

[34] Harry Rudin. "An informal overview of formal protocol specification". *IEEE Communications Magazine*, 23(3), March 1985.

[35] B. Sarikaya and A. Wiles. "Standard conformance and test specification language TTCN". *Computer Standards and Interfaces*, 14, North-Holland, 1992.

[36] 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.

[37] R. Sijelmassi and B. Strausser. "NIST integrated tool set for Estelle". Technical report, US Department of Commerce, National Institute of Standards and Technology, National Computer Systems Laboratory, Gaithersburg, MD 20899, 1991.

[38] R. Sijelmassi and B. Strausser. "The portable estelle translator: 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.

[39] 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.

[40] Andrew S. Tanenbaum. *Computer Networks, Second Edition.* Prentice Hall, 1988.

[41] Alain Thiboutôt. *Traduction d'un sous-ensemble de SDL en Estelle.* Master's thesis, Université de Montréal, Montreal, Canada, 1994.

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

[43] Zafiropulo et al. "Towards analyzing and synthesizing protocols". *IEEE Transactions on Communications*, 28(4), April 1980.