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.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

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.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

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

Modelling of Expressed System Behavior Towards Characterization of the Conformance Testing Problem

by

Doug Turner

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

University of Ottawa
OTTAWA, Ontario, Canada
September 4, 1987

ISBN 0-315-46873-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

System specifications characterize externally observable behavior of some valid system implementations, often using formal description techniques. These specifications may involve nondeterminism which allows some variation in the expected behavior of possible system implementations.

This thesis presents a framework through which various issues specific to the process of conformance testing are characterized in terms of a tree model. This model concisely represents a possibly nondeterministic system behavior. A compression applied to this tree model emphasizes important features of a specification that are of particular interest in the testing process. We then define mappings from the ESTELLE, LOTOS, Finite State Machine, and Calculus of Communicating Systems description techniques to this compressed tree model. These mappings provide skeletons for test sequences which can be applied to an implementation in a test for conformance of this implementation to its specification. Heuristics that facilitate the use of such skeletons to derive test sequences which test for valid and defensive behavior of an implementation are then presented.

The tree model and mappings to this model provide a framework which serves as a basis through which we characterize the conformance testing problem and a set of conforming implementations. Specific properties which must be satisfied by a conforming implementation are then presented. Finally, we suggest and discuss solution strategies to the conformance testing problem.

ACKNOWLEDGEMENTS

I wish to express my utmost appreciation for the expert guidance afforded me by my thesis supervisor, Dr. Hasan Ural. His efforts were beyond the call of duty.

A very special thanks goes to Dr. Bob Probert for his guidance, helpful discussions, and valuable suggestions. I wish also to thank the numerous people who have been of help in the University of Ottawa Department of Computer Science. My special thanks to Abdel Obaid and Hazem Elgendy for many helpful discussions, to my fellow graduate students for helping to create a friendly and productive working environment, and to the department secretaries for organizing my various teaching assistantships.

I also thank the Government of Canada, the Government of Alberta, Westcoast Petroleum, Gonzaga University, and the Natural Science and Engineering Research Council of Canada for financial support through my years of study.

A special thanks to Tim Rolfe for showing me how not to index an array. Finally, I thank my parents for making my studies possible and Eliane for the emotional support necessary for such an undertaking.

ABBREVIATIONS

C: The set of capabilities of a specification.

CCS: Calculus of Communicating Systems.

CT: Communication Tree.

CTM: Compressed Tree Model.

C–Vertex: Choice Vertex.

D: Set of defensive test purposes.

dtp_i: A defensive test purpose.

ESTELLE: Extended State Transition Language.

FDT: Formal Description Technique.

FSM: Finite State Machine.

I_i: Implementation conforming to S_i.

ICS: Implementation Conformance Statement.

IMP: A set of implementations.

ISO: International Standards Organization.

IUT: Implementation Under Test.

IXIT: Implementation eXtra Information for Testing.

LOTOS: Language fOr Temporal Ordering Specifications.

NFS: Normal Form Specification.

NFT: Normal Form Transition.

NPL: National Physical Laboratory.

N-Vertex: Non-choice vertex.

OSI: Open Systems Interconnection.

PDU: Protocol Data Unit.

PICS: Protocol Implementation Conformance Statement.

S: A Specification.

S_i: Implementors view of S.

ST: Synchronization Tree.

T: A testing tree with tests for valid and defensive behavior.

T_A: Tests for valid behavior.

T_B: Tests for defensive behavior.

T_i: A set of tests specific to a given implementation.

TM: Tree Model.

TT: Testing Tree.

vtp_i: A valid test purpose.

Contents

List of Figures

1 INTRODUCTION

1.1 Background and Related Work

In a system development process, the desired system functionality is usually formulated as a system specification using some formal description technique (FDT). A system specification S defines the system functionality in terms of interactions that the system exchanges with its environment. In fact, S specifies externally observable behavior of some valid system implementation.

We are particularly interested in systems for which S specifies the desired system functionality in terms of possible valid orderings (sequences) of valid system interactions. Such an S is syntactically finite but may specify possibly infinitely long and/or infinitely many interaction sequences. Examples of these systems include real-time process control, communications protocols, etc. An important characteristic of such a system is that its S may involve some nondeterminism which allows certain implementation choices in the expected behavior of possible valid system implementations. A valid system implementation differs from the other valid implementations by the manner in which it resolves these nondeterministic choices.

The goal in using FDT's is to formally and precisely define the functionality of systems. Various criteria for development of FDT's exist and, due to varying emphasis placed by FDT developers on these criteria, a variety of FDT's are available for systems specifications. Such emphasis by FDT developers can influence the use of FDT's in various phases of the system development process such as automated partial implementation from a given formal specification, verification of designs and implementations with respect to formal specifications, etc.

Of specific interest to us are the relative attributes of given FDT's in terms of their ability to aid in the derivation of test sequences (i.e., sequences of system interactions). Specifically, an FDT aids in the derivation of test sequences if, given a system specification in this FDT, we can fully or partially automate the test sequence derivation process. We are not only interested, however, in an FDT's ability to aid in the derivation of test sequences. The derived set of test sequences should satisfy further criteria specific to the testing process: finiteness, repeatability, determinism, and minimality.

Testing is the controlled execution of an implemetation under test (IUT) for the purpose of revealing errors. Testing **can not** demonstrate the absence of errors in

a given implementation, **only** their presence [Myer79]. It is not the goal of testing to demonstrate absence of errors: this is the purpose of verification.

The testing process is driven by the goal of obtaining a reasonable level of confidence that a given system implementation does behave as specified by its S. It is conceded that exhaustive testing (i.e., application of all possible test sequences) is impossible [Piat80]. Hence, we must, instead, choose a representative subset of all possible test sequences which, when applied to the given implementation, will give us a reasonable level of confidence that the implementation behaves as expected.

There are two basic categories of testing which are differentiated by the method through which test sequences are derived. The first of these, called **white–box testing** (code–based testing, structural testing), assumes the availability of the source code of the IUT. The tester is able to analyse the source code of the IUT and derive test sequences using a variety of techniques that aim to satisfy a chosen white–box criteria. White–box criteria are concerned with coverage of the structural components of the IUT and include statement coverage, branch coverge, path coverage [Wal83], etc. The second category, called **black–box testing** (functional testing, specification–based testing), is our primary concern. Black–box testing assumes that only a specification of system functionality (i.e., S) is available to the tester. The tester analyzes the system specification and derives test sequences from this specification through one or more of a variety of methods (e.g., Equivalence Partitioning, Boundary Value Analysis [Myer79], etc.).

Our primary interest in this thesis is in the area of conformance testing, which is categorized as being a black–box type of testing. In principle, the objective of conformance testing is to establish a claim on whether or not the implementation being tested conforms to its corresponding specification [ISO7]. An implementation conforms to its specification if it does not exhibit behavior contrary to specified legal system behavior.

Much international effort is being expended in the area of conformance testing. The work being done by the International Standards Organization adhoc working group on conformance testing (ISO TC97 SC21 Project 97.21.23) [ISO7] is currently advancing the state of the art of conformance testing (particularly for OSI communications protocols [ISO1]). There are several steps involved in establishing a claim of conformance of an implementation to its specification which are addressed in [ISO7]:

1. Test sequence **derivation** establishes a set of test sequences which can be applied to the IUT. The set of derived test sequences may be either abstract or executable. An **abstract** test sequence contains only the types (e.g., integer) of input values, whereas an executable test sequence contains input values of a given type (e.g., the integer 3). A derivation strategy is (semi-)automated if test sequences can be derived directly from the specification (or, possibly, the implementation) through use of a software tool.

2. Test sequence **selection** involves determination of an appropriate subset of derived test sequences to be applied to the IUT. Such selection may be necessary due to prohibitive cost in applying all derived test sequences (i.e., efficiency concerns) [Matt86]. Test sequence selection can be either **on–line** (occur during running of a test), or **off–line** (occur prior to the running of a test) [Rayn87].

3. Test case **execution** involves application of test sequences to an IUT. Factors involved in execution of a test include configuration of the environment in which to run the test, and acquisition of test result data [Rayn87].

4. **Analysis** of test results assigns a **verdict** to an observed behavior of the IUT as being either valid, invalid, or inconclusive [Rayn87]. An inconclusive verdict is assigned if the purpose of running a given test case is not fulfilled. Research at the National Physical Laboratory (NPL) and the University of Ottawa in this area is currently under way.

ISO TC97 SC21 Project 97.21.23 ad hoc working group on conformance testing is also advancing work on abstract testing architectures and methodologies [ISO7]. An abstract test method describes the points closest to the IUT at which control and observation are performed during testing. Three basic categories of abstract test methods are given [ISO7]:

1. Local Test Methods: Control and observation is specified directly above and below the IUT.

2. Distributed Test Methods: Control and observation is specified directly above and remotely below (i.e., through a communication link) to the IUT.

3. Remote Test Methods: Control and observation is specified remotely below the IUT.

This thesis is also concerned in part with derivation of test sequences from FDT's. Derivation of test sequences is one of the most difficult processes involved in conformance testing. This difficulty is primarily due to the set of criteria which must be fulfilled by derived test sequences:

1. **finiteness:** the sequences must be made finite without sacrificing their ability to reveal errors.

2. **repeatability:** the sequences must be re-applicable in future tests.

3. **minimality:** efficiency considerations are of prime importance in a production environment [Matt86].

4. **determinism:** there can be no question of which test sequences were applied.

Several strategies for derivation of test sequences from FDT's have been proposed. Those techniques which have been proposed for ESTELLE based specifications include methods related to the application of finite state machine (FSM) based testing techniques. For example, [Chow78] presents a test sequence derivation method from FSM specifications. In his method, the set of test sequences required is the concatenation of two sets of sequences. One set visits each state of the FSM while the second set distinguishes each visited state of the FSM. In [Kanu86], FSM based techniques are also used to derive test sequences. These FSM based techniques can be applied to ESTELLE based specifications by disregarding values and variations of parameter fields. [SaBo84] applies three well known FSM based testing techniques (including [Chow78]) to the generation of test sequences for protocol testing. Application of these techniques given in [SaBo84] to ESTELLE is carried out in [Linn86]. However, these techniques only test the control flow of the specification. Recently, a test design method which is able to cope with interaction parameters has been proposed, [SaBo87]. This method assumes that protocol specifications are given in ESTELLE as normal form specifications [SaBo85] and applies certain principles of functional program testing [Howd80] to the generation of test sequences. However, this method requires a considerable amount of effort for the identification of functions and their relationships in the case of complex specifications [SaBo87]. In [Ural87], a method for deriving test sequences semi-automatically

from a protocol specification given in ESTELLE as a normal form specification is presented. The normal form specification is transformed into a graph modelling the flow of both control and data. This graph explicitly identifies associations between definitions and usages of each variable employed in the specification. Based on this information, test sequences are derived to cover all definition and usage pairs satisfying certain constraints. The resulting set of test sequences provides the capability of determining whether an IUT establishes the desired flow of data expressed in the given specification. Thus, this method complements control–flow–based test sequences derivation methods [SaBo84].

Techniques for derivation of test sequences from LOTOS based specifications have also been proposed. [Stee86] presents a method for the derivation of test sequences from a protocol specification given in LOTOS. Actions of the LOTOS specification are transformed into test trees specifying interaction sequences. Nodes of the tree specify external(environment) and internal (implementation) actions. Criteria for selection of test cases are then presented in terms of node types. [Sopp86] discusses a tool for user guided test suite derivation from LOTOS based protocol specifications. The test derivation tool takes as input a LOTOS behavior specification and some information on what type of test must be provided (e.g., tests for dynamic behavior). The tool then interactively derives test suite specifications in LOTOS from this input.

None of these techniques attempt to obtain a set of test sequences which is complete in its coverage for conformance testing purposes. This is in part due to the lack of a formal underlying theory for conformance testing by which an effective, adequate, and complete set of tests can be defined.

In the area of software testing, some attempts have been made to bring some formalism into test completeness. In [Good 75], the basis for a theory of test data selection is proposed. The authors attempt to provide a theoretically sound definition of what constitutes an adequate test. Characterization of what constitutes a *completely effective test strategy* is then given in terms of the proposed theory. In [Wey 80], the theory of [Good 75] is extended and refined. [Good 75] fails to provide procedures to realize *completely effective test strategy*. In [Wey 80], a practical testing strategy based on revealing subdomains is proposed to overcome this shortcoming of [Good 75]. Further, [Wey 80] shows that the theory of [Good 75] is applicable to only white–box (code) based testing strategies. [Wey 80] refines the

theory of [Good 75] such that it may characterize black-box (specification) based testing strategies.

1.2 Motivation and Objectives for Thesis

At present, a framework which can be used as a basis for the characterization of the conformance testing problem is unavailable. We thus intended to make an initial attempt for such a characterization by developing a tree model which would provide the basis for such a framework. Properties of a model specifically desirable for the characterization of conformance testing such as its ability to characterize a set of conforming implementations by a set of tests are satisfied by this model. Specific characteristics of various specification techniques (e.g., nondeterminism) had to be representable in this model.

There are at present no test sequence generation techniques which attempt to obtain a complete set of test sequences that can be used for conformance testing of implementations with respect to their specifications. Since ISO is currently developing two candidate specification techniques for an international specification standard, ESTELLE [ISO8] and LOTOS [ISO9], and since derivation of test sequences from formal specifications is a topic of active research [Stee86], we decided to show mappings from ESTELLE and LOTOS to the tree model developed. As well, we intended to show mappings from the underlying models upon which ESTELLE and LOTOS are based: Finite State Machines, and the Calculus of Communicating Systems [Miln80]. These mappings were to provide us with a skeleton set of test sequences which could be used for conformance testing of implementations with respect to their specifications. From these skeletons, we intended to develop and characterize heuristics through which one can obtain test sequences which could specify tests for valid and defensive behavior of a given implementation of a specification. We also intended to characterize a set of valid implementations in terms of a set of tests. Given these characterizations, solution strategies to the conformance testing problem were then to be discussed.

1.3 Scope and Major Contributions of Thesis

The scope of this thesis is limited to the areas of system specifications and conformance testing. We are concerned only with behavior specified in given specifications,

not with furthering the art of specification techniques. Within conformance testing, we are primarily concerned with the test sequence derivation and test sequence selection problems, and precising what is meant by a conforming implementation. Test architectures, execution of tests, and result analysis are outside the scope of this thesis.

The major contributions of this thesis are as follows:

1. We present a framework that provides a basis for the characterization of the conformance testing problem in terms of a tree model which represents the system behavior expressed by a system specification. This tree model provides a formalism which is graphically clear, emphasizes points of choice in the specification, and, in particular, those points in the specification which are of primary importance to a tester (i.e., points of nondeterminism).

2. We present mappings from the FSM, ESTELLE, CCS, and LOTOS specification techniques to this model through which one can derive skeletons for test sequences that will be employed for conformance testing. We provide heuristics for completing these skeletons so as to obtain test sequences which test for valid and defensive behavior of a possible implementation of a given specification. Parameter variations in syntactically valid test sequences test for both valid and defensive behavior of an implementation. Syntactically invalid test sequences test for further defensive behavior of an implementation. Heuristics for obtaining a set of test sequences specific to a given implementation are also given.

3. We give a characterization of all possible conforming implementations of a nondeterministic specification by a set of tests and characterize the conformance testing problem.

4. We suggest solution strategies to the conformance testing problem and discuss the relative merits of these strategies.

1.4 Outline of Thesis

Section 2 gives a tree model which represents possibly nondeterministic system behavior, a corresponding compressed tree model, and discussion on the modelling of

nondeterminism. Section 3 defines mappings from the Finite State Machine, ESTELLE, CCS, and LOTOS specification techniques to the compressed tree model. Section 4 states the conformance testing problem in terms of the compressed tree model, characterizes conforming implementations of a given specification in terms of a set of tests, and discusses possible solution strategies for the conformance testing problem.

2 TREE MODEL

Our intent is to characterize a set of conforming implementations by a set of tests. This set of tests is composed of valid sequences of a given implementation. We use a tree model to concisely specify these valid interaction sequences.

2.1 The Basic Tree Model

S describes the functionality of a system in terms of possible sequences of valid interactions exchanged between the system and its environment. We model the system functionality expressed by a given S as a possibly infinite tree TM=(V,E), where V is a non-empty set of vertices and E is a set of edges. In TM, each interaction is represented by an event and each sequence of valid interactions is represented by a root–to–terminal–vertex path. Thus, each $e \in E$ represents an event and each $v \in V$ represents the root of the subtree defining all possible remaining subsequences of events.

In TM=(V,E),

A) Each event $e \in$ E is of the form e=(P,I,O) where:

- P is a possibly nil predicate that must be satisfied for an event to occur. If not nil, P is of the form: $p(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ (n≥0) are variables. We express legal predicates with the following BNF[1] representation:

  P ::= Bool_Exp
  Bool_Exp ::= (Bool_Exp Log_op Bool_Exp) |
  [NOT] (Var_id "=")(Var_id|Constant_id|Constant) |
  [NOT] (Var_id Rel_op)(Var_id|Constant_id|Constant)
  Log_op ::= OR|AND
  Constant_id ::= {char}
  Var_id ::= {char}
  char::=A| B | ... | Z | a | b | ... | z | 0 | 1 | ... | 9
  Constant ::= (["−"]number|value|Constant_id)
  rel_op ::= <|>|≠|≤|≥
  value ::= number|{char}
  number ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}

[1] ()≡ textual grouping; []≡optional; |≡OR; {}≡1 or more occurences.

- I is a list of inputs from the environment where each input is denoted by ii($x_1, \ldots, x_n$) where "ii" stands for interaction identifier and $x_1, \ldots, x_n$, ($n \geq 0$), are interaction parameters.

- O is a list of outputs to the environment where each output is denoted by ii($x_1, \ldots, x_n$) where "ii" stands for interaction identifier, and $x_1, \ldots, x_n$, ($n \geq 0$), are interaction parameters.

The absence of an input from the environment or the absence of an output to the environment is denoted by "–". A nil predicate is also denoted by "–".

B) Each event $e \in E$ is one of the following four types:

- E_1 denotes both input from and output to the environment. Such an edge can be abstractly labelled by (P,I,O).

- E_2 denotes input from the environment with no output to the environment; i.e.,(P,I,-).

- E_3 denotes no input from the environment with output to the environment; i.e.,(P,-,O).

- E_4 denotes no input from and no output to the environment; i.e.,(P,-,-). This corresponds to an internal interaction that we call a null event (e.g., an internal event in LOTOS[ISO9] or a spontaneous transition which produces no output in ESTELLE[ISO8]).

C) It is assumed that each event may be associated with some uninterruptable action local to the system being specified (e.g., an action that is defined in a BEGIN-END clause of a transition in ESTELLE [ISO8]). These actions can easily be incorporated into this model (i.e., into the representation of events) such that an event becomes a 4–tuple with the inclusion of the action. For the sake of simplicity in our representation of events, we omit the action and make events 3-tuples.

The four edge types are shown in Figure 2.1.

D) Each v ∈ V is one of the following two basic types:

1. **C–Vertex(Choice Vertex)** represents a point in S at which a choice of next event is specified in terms of alternative events. Alternative events specified at a given point in S are modelled in TM as edges emanating from a vertex

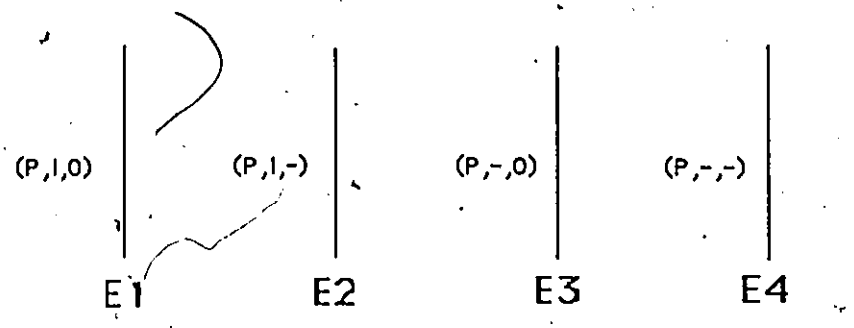


Figure 2.1: Edge Types

which is the point where a choice is specified. We call the alternative events at a choice vertex the **choice_set** of that vertex. The nature of specified alternative events in a choice_set determines the type of that choice, hence the type of that C–vertex. There are three types of C–vertices:

(a) A **nondeterministic vertex** (V_1) is a C–vertex which stands for a choice of next event between two or more alternative events that include at least two events not enabled by input from the environment whose predicates hold (e.g., nil predicates) or at least two events with identical input whose predicates hold.

(b) A **deterministic vertex** (V_2) is a C–vertex which stands for a choice of next event between two or more alternative events that do not include an event that is not enabled by input from the environment and either:

- all events have unique inputs, or
- all events that have identical inputs have predicates that can not hold at the same time.

(c) A **grey vertex** (V_3) is a C–vertex which stands for a choice of next event between two or more alternatives that does not meet the criteria for being either a nondeterministic or a deterministic vertex.

2. **N–Vertex(Non–Choice Vertex)** represents a point in S at which there is no choice of next event. Such vertices represent one of the following:

(a) an intermediate point between successive events in S at which no choice is specified(i.e., choice_set is a single event), or

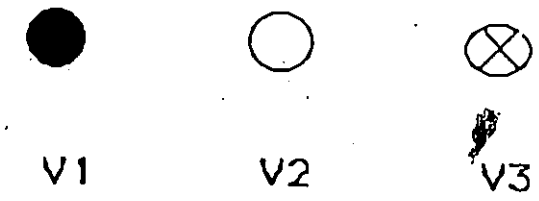


Figure 2.2: Vertex Types

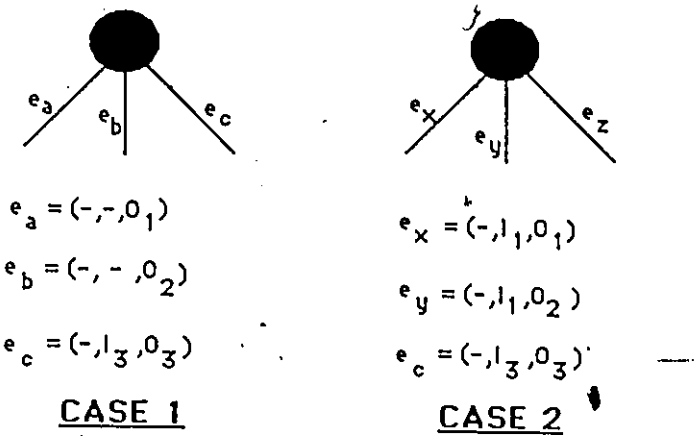


Figure 2.3: Examples of Nondeterministic Vertices

(b) a point in S at which specification syntactically terminates (i.e., choice_set is empty). In this case, the vertex is called a **terminal vertex**.

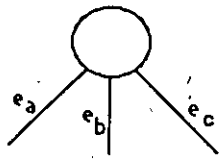
E) Each vertex in TM has an arbitrary label.

F) The root of the TM is any one of the above vertex types.

Figure 2.2 shows the three C-vertex types of TM.

Figure 2.3 gives examples of nondeterministic vertices. For both cases we assume predicates are nil for all alternative events. In case 1, we have nondeterminism due to spontaneous (null input) events e_a and e_b. In case 2, we have nondeterminism due to identical input events e_x and e_y.

Figure 2.4 gives an example of a deterministic vertex. Again, we assume that predicates are nil for all events. We have a deterministic choice due to unique input events e_a, e_b, and e_c.



$e_a = (-, I_1, O_1)$

$e_b = (-, I_2, O_2)$

$e_c = (-, I_3, O_3)$

Figure 2.4: Example of a Deterministic Vertex

Figure 2.5 gives examples of grey vertices. We assume that truth values for predicates P_1, P_2, and P_3 are unknown in both cases. That is, determination of their truth values can not be made using only static information (i.e., information obtained from the syntactic constructs of S). In case 1, we have **possible** nondeterminism due to identical input events e_a and e_c. In case 2, we have **possible** nondeterminism due to spontaneous event e_x in the presence of any one of the inputs I_2 and I_3. Since we can not determine the truth values of P_1, P_2, and P_3, using the static information, we can not distinguish whether the choice is nondeterministic or deterministic. Hence, we have a grey choice which will turn into a deterministic choice or a nondeterministic choice later when some dynamic information (i.e., runtime instantiations) is available.

2.2 Criteria for Determining C–Vertex Types

We enumerate the criteria for determination of choice(or C–vertex) types as follows: Let $e_1, \ldots, e_n$ be the choice_set for a C–vertex $v \in V$. Let $P_1, \ldots, P_n$ be the predicates in events $e_1, \ldots, e_n$; where each e_j, j=1,...,n, contains either "–" (null input) or I(identical input). Also, let X stand for "–"(null output) or O(some output).

1. If P_i and P_j both hold, i≠j, and e_i and e_j are identical input events, then a nondeterministic choice is specified.

2. If P_i and P_j both hold, $i \neq j$, and e_i and e_j are null input events, then a nondeterministic choice is specified.

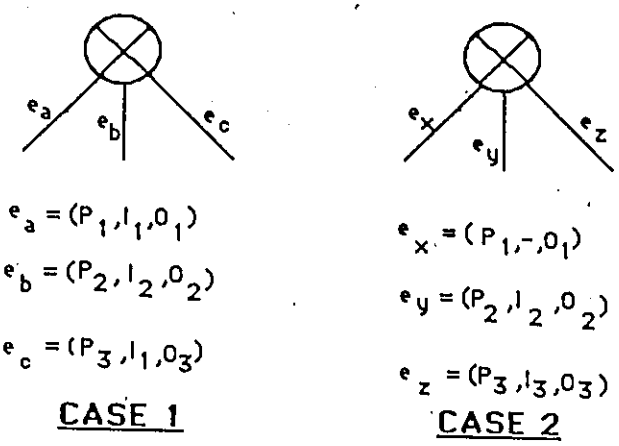


Figure 2.5: Examples of Grey Vertices

3. If P_i does not hold for any e_i=(P_i,–,X), i=1, . . . , n, and no two predicates P_j and $P_k, i \neq j \neq k$ ever hold simultaneously for a pair of $e_j = (P_j, I, X)$ and $e_k = (P_k, I, X)$, then a deterministic choice is specified.

4. Any choice which does not meet the criteria for either a deterministic or nondeterministic choice based on static information in S is a **grey choice**. Specifically,

   - If P_i and P_j **might** hold for any two events e_i and e_j as above, $i \neq j$, then grey choice is specified.
   - If P_i **might** hold for any e_i=(P_i,–,X), and P_j holds for any e_j, $i \neq j$, then grey choice is specified.

In sections 2.4.1 and 2.4.2 we discuss the effects of predicates upon determination of C–vertex types.

2.3 Compressed Tree

In modelling expressed system behavior, we are primarily interested in analysis of points in S at which a choice is specified. TM expresses linear sequences of events which may be rather long. In order to represent the expressed behavior of possible valid implementations of a given S in a compact form and emphasize the points in the expressed behavior where choices are specified, we introduce the following compressed tree model. Section 2.3.1 gives a formal definition of compressed tree

model. Section 2.3.2 gives an example of compressing a TM to obtain a corresponding compressed TM (i.e., CTM).

2.3.1 Formal Definition of Compressed Tree Model

Let TM=(V,E) represent the system behavior expressed by a S.
A sequence of edges $e_1, e_2, \ldots, e_n$ in TM is called a **linear sequence of events** if:

- $e_i = (v_i, v_{i+1}), 1 \leq i \leq n$,
- v_1 = root of TM or C-vertex,
- v_{n+1} = C-vertex or terminal vertex, and
- none of the vertices v_j is a C-vertex, $2 \leq j \leq n$.

CTM=(V', E') is a compressed tree of TM where:

1. The root of CTM is the root of TM.
2. All other vertices $v' \in V'$ are either C-vertices or terminal vertices in TM.
3. Each $e' \in E'$ represents a linear sequence of events $e_1, e_2, \ldots, e_n$ of TM, $n \geq 1$, (i.e., $e' = e_1, e_2, \ldots, e_n$).
4. The label of a $v' \in V'$ is the same as that of the corresponding $v \in V$.

Accordingly, the mapping from TM to CTM is defined as follows:

> For each linear sequence of events $e_1, e_2, \ldots, e_n$ between each pair of vertices v_1 and v_{n+1} in TM, there is an edge $e' \in E'$ such that $e' = (v'_1, v'_{n+1})$ where $v'_1 = v_1$ and $v'_{n+1} = v_{n+1}$.

That is, no vertices other than the root, C-vertices, and terminal vertices of TM appear in CTM. As well, each linear sequence of events in TM is compressed into an edge in CTM as shown in Figure 2.6.

It is important to note that through this mapping the points where choices are specified in the expressed behavior represented by TM are emphasized. All vertices in CTM except the terminal vertices (and possibly the root) are C-vertices. The first events in the linear sequences of events which are represented by edges emanating

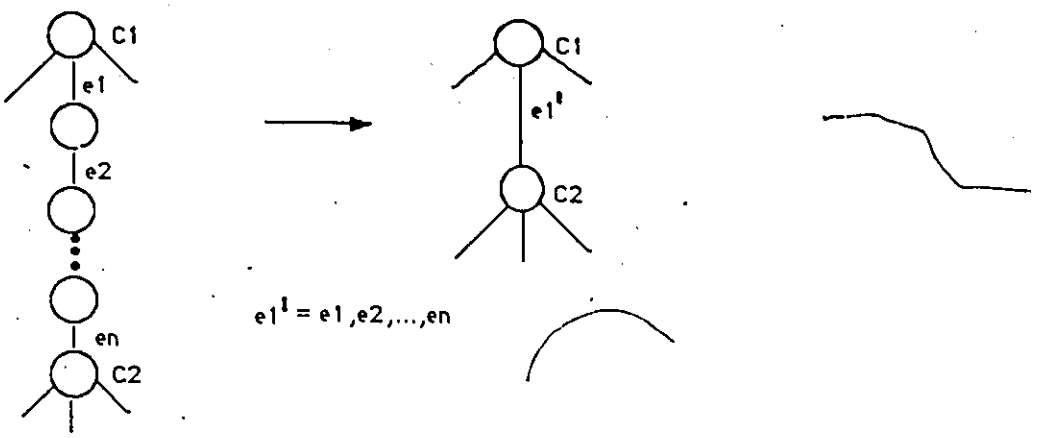


Figure 2.6: Mapping Linear Sequences

from a C–vertex form the alternative events(i.e., **choice_set**) at that choice point. Note that the criteria for choice type determination given in section 2.2 still holds. Note also that if S does not specify any choice, the tree degenerates into a single edge between the root and the terminal vertex.

2.3.2 Example of TM Compression

Consider the TM given in Figure 2.7. Each edge in this TM is abstractly labelled with its corresponding event e_i, i=1,...,10. The allowed sequence of events modelled by the TM in Figure 2.7 can be represented with the following regular expression[Rev 83]:

$$e_1 * e_2 * ((e_4 * (e_5 + e_6)) + e_3 * e_7 * (e_8 + (e_9 * e_{10}))),$$

where "*" denotes concatenation of events in sequence, and "+" denotes choice of next event sequence. As can be seen from Figure 2.7, there are three points in the tree at which choice is specified:

- A is a deterministic choice.
- B is a nondeterministic choice.
- C is a grey choice.

It is these points of choice which are of primary interest in the analysis of the specification.

We obtain the compressed tree given in Figure 2.8 by applying the mapping defined in Section 2.3.1. The regular expression corresponding to this compressed

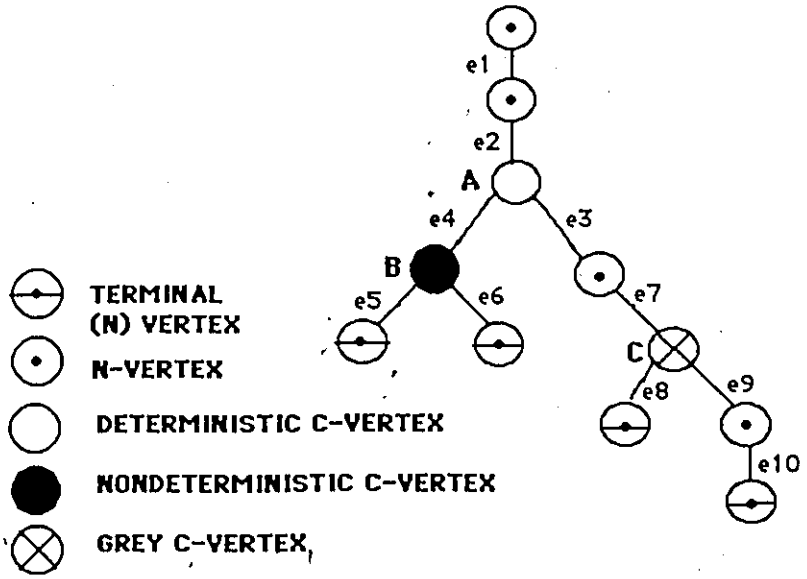


Figure 2.7: An Example Tree Model

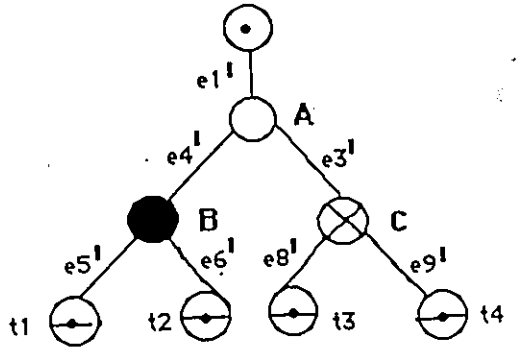


Figure 2.8: Corresponding Compressed Tree

tree of Figure 2.8 is:

$$e_1' * ((e_4' * (e_5' + e_6')) + (e_3' * (e_8' + e_9')))$$

where,

- $e_1' = e_1 * e_2$
- $e_3' = e_3 * e_7$
- $e_4' = e_4$
- $e_5' = e_5$
- $e_6' = e_6$
- $e_8' = e_8$
- $e_9' = e_9 * e_{10}$
- t_i = terminal vertex, $1 \leq i \leq 4$
- choice_set at A = $\{e_3, e_4'\}$
- choice_set at B = $\{e_5', e_6'\}$
- choice_set at C = $\{e_8', e_9\}$

2.4 Modelling of Nondeterminism

Specifically, S can specify nondeterminism in two ways. First, if at some point in S, S specifies one or more events not enabled by input from the environment, then we have possible nondeterminism. Second, if at some point in S, S specifies two or more events enabled by identical input from the environment, then we also have possible nondeterminism. The existence of nondeterminism depends on truth values of predicates in the above specified events. The criteria for choice type determination is given in section 2.2. We now exemplify the modelling of nondeterminism with first a general example and then a specific example from the OSI transport layer [Zimm80].

2.4.1 General Example

Let us elaborate on the role of predicates in possible nondeterministic specifications. The intent here is to show that determination of choice type involves many factors including event types and therefore, on the truth values of predicates. Assume that we are given the following:

- CTM=(V', E').
- e'_1, e'_2, $e'_3 \in E'$ are edges emanating from a vertex $v' \in V'$.
- P_1 ,P_2 ,P_3 are predicates of e'_1 ,e'_2 ,e'_3
- e'_j=(P_j,I,O), j=1,2,3, where I= "–" or "i" which stand for null input and a certain input "i", respectively.

Assume that the truth value of P_3 can be determined by the truth value of P_1 and that no other predicate's truth value can be determined by the others. Then, (P_1,P_2) are independent, (P_2,P_3) are independent, and (P_1,P_3) are *not* independent. Note that if the truth value of P_2 could be determined by the truth value of P_3, then P_1 and P_2 would *not* be independent by transitivity:

$$((\text{ie., } P_1 \Longrightarrow P_3 \Longrightarrow P_2\ ) \Longrightarrow (\ P_1 \Longrightarrow P_2)).$$

Now, P_1 and P_3 are not independent in one of the following four ways:

1. $P_1 \Longrightarrow P_3$
2. $\neg(P_1) \Longrightarrow P_3$
3. $P_1 \Longrightarrow \neg(P_3)$
4. $\neg(P_1) \Longrightarrow \neg(P_3)$

Refer to Figure 2.9 for the following:

- In case 1, e_1 and e_3 alone specify nondeterminism. P_2, though, can also be true and, in this case, e_2 is one of the specified next event nondeterministic choices.
- In case 2, if P_2 is true then nondeterministic choice is specified between e_2 and e_3. If P_2 is false, determinism is specified.

Case	P1	P2	P3	e1	e2	e3	Deterministic	Nondeterministic
1	T	T	T	•	•	•		•
2	F	T	T		•	•		•
	F	F	T			•	•	
3	T	T	F	•	•			•
	T	F	F	•			•	
4	F	T	F		•		•	
	F	F	F				•	

Figure 2.9: Effects of Truth Values of Predicates

- In case 3, if P_2 is true then nondeterministic choice is specified between e_2 and e_1. If P_2 is false, determinism is specified.
- In case 4, P_2 being true implies determinism. If P_2 is false, e_1,e_2, and e_3 are not possible as next events at the current point in the specification.

Hence, we see that only if two of three predicates to identical input events and null input events are false do we have a deterministic choice from possible nondeterministic specification. It is possible to generalize the above argument for the case of an integer number of predicates and events to characterize deterministic, nondeterministic, and grey choice. This characterization is given in section 2.2.

2.4.2 A Specific Example From the OSI Transport Layer

In order to exemplify determination of choice types, we give an example from the OSI Transport Layer [Zimm80]. Consider the state transition diagram given in Figure 2.10. This state transition diagram gives the possible allowed sequences of transport service primitives and transport protocol data units(pdu's). We are primarily interested in the data transfer phase. This phase is represented by the state **data transfer ready** in Figure 2.10. For simplification of the example, we use logically equivalent names for the actual service primitives and pdu's of the transport layer.

Now, during data transfer, a transport protocol entity is capable of four basic events:

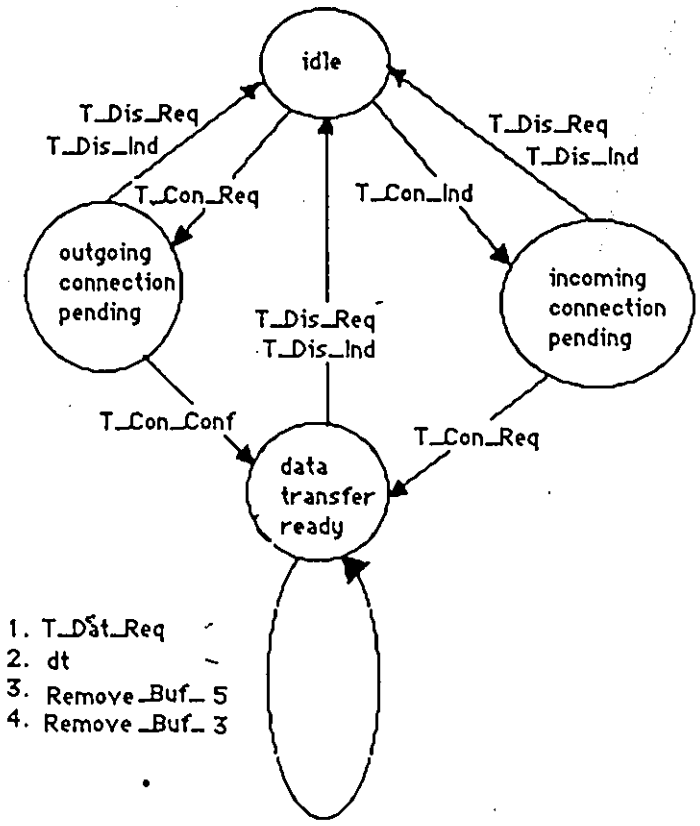


Figure 2.10: Sequences at a Transport Connection Endpoint

1. Receiving from the transport service user a transport data request service primitive (T_Dat_Req).

2. Receiving from the network service provider a transport data transfer pdu (dt).

3. Transmitting to the network service provider any queued transport data request (Remove_Buf_5).

4. Transmitting to the transport service user any queued transport data transfer pdu (Remove_Buf_3).

We represent these four events in the model of a transport protocol entity given in Figure 2.11.

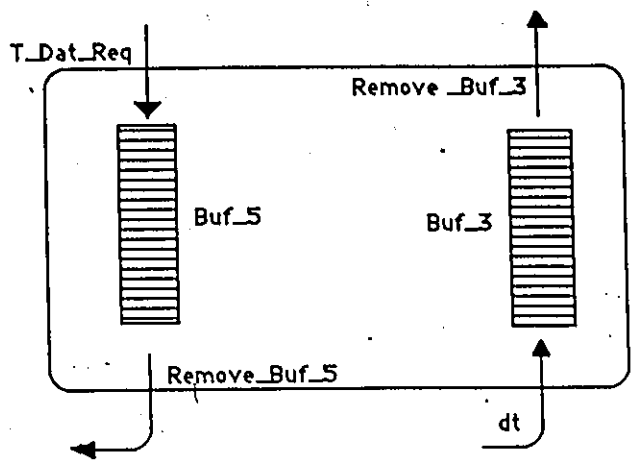


Figure 2.11: Simplified Transport Entity

- Buf_5 is a queue representing transport data requests coming from the session layer. T_Dat_Req is inserted to Buf_5.
- Buf_3 is a queue representing transport data transfer pdu's coming from the network layer. dt is inserted to Buf_3.
- P1 is the predicate: "Buf_5 is not empty".
- P2 is the predicate: "Buf_3 is not empty".
- Remove_Buf_5 is an output to the network layer.
- Remove_Buf_3 is an output to the session layer.

Note that the events T_Dat_Req and dt have no associated predicates. Thus, we assume that these events may occur at their respective ports at any time.

In order to clarify the type of choice specified by our simplified transport protocol entity, we present the possible scenarios for functioning of the modelled entity in terms of the context of its predicates P_1 and P_2. It is the totality of choice types of these scenarios which characterize the type of choice specified by the simplified transport protocol entity. There are four scenarios:

- If P_1 and P_2 are both false, then Buf_5 and Buf_3 are both empty. Here, it is impossible for the transport entity to output to either the lower or upper layer. Thus, if both queues are empty, we have choice between unique input

events T_Dat_Req and dt. Hence, in this case, deterministic choice is present according to the criteria given in Section 2.2.

- If P_1 holds, then the output event Remove_Buf_5 is enabled. Here, we have choice between the single output event Remove_Buf_5 and two unique input events (T_Dat_Req and dt). Hence, grey choice is present as according to the criteria given in Section 2.2.

- If P_2 holds, then the output event Remove_Buf_3 is enabled. Here, we have choice between the single output event Remove_Buf_3 and two unique input events (T_Dat_Req and dt). Hence, grey choice is specified as according to the criteria given in Section 2.2.

- If P_1 and P_2 hold, then both output events Remove_Buf_5 and Remove_Buf_3 are enabled. Here, we have choice between two output events Remove_Buf_5 and Remove_Buf_3 and two input events (T_Dat_Req and dt). Hence, nondeterministic choice is specified as according to the criteria given in Section 2.2.

We see from the example that the determination of whether choice is deterministic or not can not be made until the truth values for P_1 and P_2 are known: output events Pop_Buf_5 and Pop_Buf_3 may or may not be enabled. Our four scenarios show that, depending upon the truth values of these two predicates, deterministic or nondeterministic choice can be present. Thus, in this example, we have a grey choice specification according to the criteria given in Section 2.2 because determination of the choice type depends upon the instantiated values of variables over which the predicates are defined. The instantiated values of these variables are not available from static information of S.

A subtle but important point must be made with respect to those cases in which only a single null input event is enabled (i.e., has true or null predicate) at the same time one or more input events are enabled (i.e., have true or null predicates). If one or more of these input events is available at their corresponding ports, then a nondeterministic choice is present: we can either perform one of the input events or perform the null input event. However, based on static information of S, we can not be certain that one or more of these inputs will be available at the same time when the null input event is enabled. If no input is available, then no choice is present (i.e., only one event, the null input event, is possible). Hence, in those cases where

only a single null input event is enabled, a grey choice is specified: it is uncertain whether or not a nondeterministic choice will be present. On the contrary, if more than one null input event is enabled (i.e., have true or null predicates), then a nondeterministic choice is always present: we can indeterminately perform any one of these null input events.

From this simple transport entity example, we have seen that a specified choice may be dependent upon predicates whose truth values may not be available from static information of S. Such dependence upon predicates of various alternatives can lead to what we call grey choice. We have also seen four scenarios exemplifying deterministic, nondeterministic, and grey choice. This is a very good example for illustrating determination of choice types and will be referred to throughout the remainder of this thesis.

3 COMPRESSED TREE MODEL COVERAGE

Formal Description Techniques(FDT's) are methods of defining the behavior of a system in a language with a formal syntax and semantics. FDT's are important tools for the design, analysis, and specification of information processing systems. It is by means of formal techniques that system descriptions can be produced that are complete, consistent, precise, concise, and unambiguous.

Various basic objectives to be satisfied by an FDT are as follows[ISO9]:

Expressive Power: An FDT should be capable of specifying all aspects of the system being described.

Well Definedness: An FDT should have a formal mathematical model that is suitable for analysis of specifications.

Well Structuredness: An FDT should offer means for structuring the description of a specification in a manner that is meaningful and intuitively pleasing.

Powers of Abstraction: An FDT should be completely independent of the method of implementation and offer the means of abstraction from irrelevant details with respect to the context at any point in the specification.

Clarity: An FDT should be readable, clear, understandable, and simple.

There are also objectives to be met by FDT's which are specific to testing [ISO10]:

1. An FDT should be suitable for various stages of the testing process:
   (a) test sequence derivation.
   (b) abstract test description.
   (c) executable test description.
   (d) result analysis.
2. An FDT should assist in determination of whether an observed instance of system behavior satisfies the specification.
3. An FDT should enable the derivation and selection of candidate test specifications from system specifications.

Clearly, there is a large set of criteria which ideally must **all** be satisfied by any FDT. However, due to different emphasis (i.e., **basic** versus **testing** objectives) placed on FDT's by the various FDT developers, a variety of techniques exist. LOTOS, for example, may be superior to ESTELLE in terms of its power of abstraction[ISO9]. ESTELLE, on the other hand, may have greater expressive power[ISO8]. Knowing that various techniques exist, it is our goal to define a mapping to CTM for four of these techniques and ultimately derive test sequences from specifications given using these techniques. Thus, our emphasis for an FDT is clearly its ability to aid in the process of test sequence derivation.

Due to pragmatic considerations, there are several criteria which must be met by our set of derived test sequences. These sequences must be:

Finite: Clearly, we can not apply an infinitely long test sequence to a given implementation: not only will the test never terminate, but also we will never get to the point of analyzing our results.

Repeatable: Given our derived test sequences, an implementation, and an environment with a known context in which to run the test, application of these test sequences should yield identical results in various runs of the test.

Deterministic: There should be no possible ambiguity in analysis of test results due to uncertainty in terms of which test sequences were applied.

Minimal: We want the derived test sequences to be as short as possible. Clearly, the costs incured by application of unecessarily long test sequences are contrary to the goal of efficient testing.

We now proceed to define mappings through which we can derive skeletons for test sequences from various FDT's. Section 3.1 defines a mapping from finite state machines (FSM's) [Rev 83] to CTM. Section 3.2 defines a mapping from Extended State Transition Language ESTELLE [ISO8] Normal Form to CTM. Section 3.3 defines a mapping from the Calculus of Communicating Systems(CCS)[Miln80] to CTM. Section 3.4 defines a mapping from the Language of Temporal Ordering Specifications LOTOS[ISO9] to CTM.

3.1 Tree Model Coverage of FSM

We wish to show that the compressed tree model (CTM) covers the Moore and Mealy Finite State Machines. Section 3.1.1 defines each of these two types of FSM's and shows that CTM covers linear sequences of transitions, deterministic choice behavior, and nondeterministic choice behavior specified in these FSM's. Section 3.1.2 gives an example mapping from FSM to CTM.

3.1.1 CTM Coverage of FSM

A Finite State Machine M is a quintuple M=(S,I,O,δ, λ)
where:

- S is a finite, nonempty set of states;
- I is a finite, nonempty set of inputs;
- O is a finite, nonempty set of outputs;
- δ:I X S $\longrightarrow$ S is the state transition function;
- λ is the output function such that:
  - λ:I X S $\longrightarrow$ O for Mealy machines;
  - λ:S $\longrightarrow$ O for Moore machines.

In any FSM, a linear sequence of state transitions starts at a state S_1 and terminates at a state S_n. Such a sequence is represented by $(t_1, t_2, \ldots, t_{n-1})$, $n \geq 2$, where t_1 is a transition defined at state S_1 and t_{n-1} takes the FSM to state S_n. In a linear sequence of state transitions $(t_1, t_2, \ldots, t_{n-1})$, no state S_i, $2 \leq i \leq n-1$, exists such that:

- $\delta : I_1 X S_i \longrightarrow S_{i+1}$; and
- $\delta : I_2 X S_i \longrightarrow S_k$, $k \neq i+1$.

Linear sequences of transitions in FSM's are mapped into CTM as follows:

> For each linear sequence of transitions $(t_1, t_2, \ldots, t_{n-1})$ between each pair of states S_1 and S_n in the FSM, there is an edge $e' \in E'$ such that $e' = (S_1, S_n)$.

In any FSM, **deterministic choice** is specified at a given state S_i whenever there are two or more alternative transitions:

- $\delta_1 : (S_i, I_1)$
- $\delta_2 : (S_i, I_2)$
- $\vdots$
- $\delta_n : (S_i, I_n)$

such that: $I_k \neq I_j \ \forall$ k≠j and $I_k \neq null$ and j,k=1,...,n. We represent S_i in CTM as a deterministic vertex S_i: For each transition $\delta_1, \ldots, \delta_n$ emanating from state S_i, we create an edge $e_i = (-, I_i, O_i)$, $i = 1, \ldots, n$.

In an FSM, **nondeterministic choice** is specified at a given state S_j whenever there are two or more alternative transitions:

- $\delta_1 : (S_i, I_1)$,
- $\delta_2 : (S_i, I_2)$,
- $\vdots$
- $\delta_n : (S_i, I_n)$

such that $I_i = I_j$ for at least two transitions, $i \neq j$, or I_i=null and I_j=null,i≠ j, $i = 1, \ldots, n$. In such a case it is indeterminate which transition will be taken. We represent S_j in CTM as a nondeterminisitic vertex: For each transition $\delta_1, \ldots, \delta_n$ emanating from state S_i, we create an edge $e_i = (-, I_i, O_i)$, i=1,...,n.

We see that CTM covers transitions, deterministic choice, and nondeterministic choice in Moore and Mealy FSM's. Since the FSM's have no mechanism in their basic forms through which we specify enabling predicates, they are unable to specify grey choice. Hence, CTM covers **every** behavior represented by Moore and Mealy FSM's.

3.1.2 Example FSM to CTM Mapping

Consider the FSM shown in Figure 3.1. This FSM is mapped into the CTM shown in Figure 3.2 where:

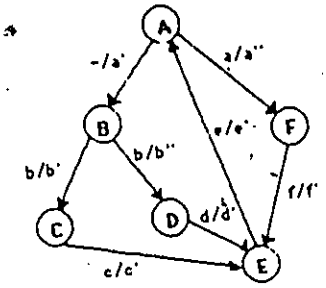


Figure 3.1: A FSM

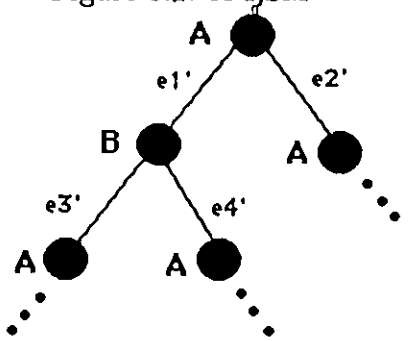


Figure 3.2: Corresponding CTM

- e_1'=(-,-,a')
- e_2'=$((-, a, a'') * (-, f, f') * (-, e, e'))$
- e_3'=$((-, b, b') * (-, c, c') * (-, e, e'))$
- $e_4' = ((-, b, b') * (-, d, d') * (-, e, e'))$

3.2 Tree Model Coverage of Estelle

We wish to show that the compressed tree model(CTM) covers the **Extended State Transition Model (ESTELLE)** [ISO8]. Section 3.2.1 gives a general description of ESTELLE Normal Form Specification(NFS). Section 3.2.2 gives a formal definition of an NFS. Section 3.2.3 defines a mapping to CTM of each of the constructs of NFS defined in Section 3.2.2. Section 3.2.4 gives an example CTM mapping from a NFS.

3.2.1 Normal Form Specifications

In Estelle[ISO8], a protocol may be specified in terms of externally observable behaviors of possibly more than one module. Externally observable behavior of each module is specified in terms of the state space and the possible state transitions of the module. In general, specifications in Estelle may include intermodule interactions and relatively complex transition definitions. Various transformations eliminate intermodule interactions and simplify transition definitions in order to obtain easy-to-analyze, single-module specifications (i.e., normal form specifications) from the original specifications given in Estelle[SaBo85].

A **Normal Form Specification** (NFS) describes the externally observable behavior of a module (e.g., a protocol entity) in terms of **Normal Form Transitions**. A Normal Form Transition (NFT) consists of the following clauses:

1. an optional **WHEN** clause specifying an external input interaction (when this clause is absent the transition is said to be **spontaneous**),
2. a **FROM** clause containing a specific identifier for the present state,
3. an optional **PROVIDED** clause specifying an enabling predicate which must be satisfied for the transition to take place,
4. a **TO** clause containing a specific identifier for the next state,
5. a **BEGIN–END** clause specifying a single path composed of assignment statements, procedure calls, and possibly some output statements defining external output interactions.

3.2.2 Formal Definition of Normal Form Specifications

Formally, a NFS **T** is defined as **T** ={t| t is a **NFT**} where each NFT t consists of the following five components:

1. WHEN[t]
2. FROM[t]
3. PROVIDED[t]

4. TO[t]

5. BEGIN-END[t]

The components of a NFT t are defined as follows:

1. Let **WHEN[t]** be nil for a spontaneous NFT **t** and let **WHEN[t]** for a NFT **t** that is not spontaneous be an input interaction i which is in the form $ii(X_1, \ldots, X_n)$, where ii stands for interaction identifier and $X_1, \ldots, X_n$ $(n \geq 0)$ are interaction parameters. Then, for a given NFS **T**, the set of input interactions I is defined as $I = \cup_{t \in T}\{WHEN[t]\}$.

2. Let **FROM[t]** and **TO[t]** be the present state and the next state of a NFT **t**, respectively. Then, for a given NFS **T**, the set of states **S** is defined as $S = \cup_{t \in T}\{FROM[t], TO[t]\}$. It is assumed that $idle \in S$ and that there exist at least one t_1 and one $t_2 \in T$ such that $FROM[t1] = idle, TO[t2] = idle$, and $t_1 \neq t_2$.

3. Let **PROVIDED[t]** be either nil or an n-ary predicate $p(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ $(n > 0)$ are variables.

4. Let $BEGIN\text{-}END[t]$ be a statement block[2] $b = \langle d_1, \ldots, d_m, o_1, \ldots, o_n \rangle$ where:

   - d_i, $i = 1, \ldots, m$, $m \geq 0$, is an assignment statement a or a procedure call c
   - o_j, $j = 1, \ldots, n, n \geq 0$, is an output statement o

   (a) An assignment statement a is in the form $Y := f(X_1, \ldots, X_n)$, where $n \geq 0$ and $Y, X_1, \ldots, X_n$ are variables.

   (b) an output statement o is in the form **out ii**$(X_1, ..., X_n)$, where out indicates that the statement is an output statement, ii stands for interaction identifier, and $X_1, \ldots, X_n$ $(n \geq 0)$ are interaction parameters.

   (c) a procedure call c is in the form $pn(X_1, \ldots, X_k, Y_1, \ldots, Y_m, Z_1, \ldots, Z_n)$ where pn is the procedure name and $X_1, \ldots, X_k$ $(k \geq 0)$; $Y_1, \ldots, Y_m$ $(m \geq 0)$; and $Z_1, \ldots, Z_n$ $(n \geq 0)$ are variables that stand for actual in-out, output, and input parameters, respectively.

[2] **A statement block** is a set of ordered statements $b = (r_1, \ldots, r_n)$ such that if $n > 1$, r_i physically precedes $r_{i+1}, i = 1, \ldots, n-1$.

3.2.3 CTM Coverage of NFS Constructs

To show CTM coverage of ESTELLE NFS's, we first define how CTM is generated from the NFS in terms of choice and sequence mappings. We then give a mapping for each NFS construct specifying valid behavior to events of CTM.

Let NFS T be $\{t_1, t_2, \ldots, t_n\}$.

I) A TM=(V,E) is constructed from NFS as follows:

1. For each s$\in$ S,

   (a) if the total number of transitions originating from state s is equal to 1, then s is represented by an N–vertex in CTM.

   (b) otherwise, s is represented by a C–vertex in CTM.

   That is, each state s$\in$ S is mapped into a vertex that is the root of the subtree representing all possible subsequent sequences of events.

2. Each t$\in$T is mapped into an event that is represented by an edge e=(P,I,O) between vertices v and w such that FROM[t]=v and TO[t]=w. The components of e are determined as follows:

   (a) P = PROVIDED[t] =

   - "–" if PROVIDED[t] is nil;
   - p($X_1, \ldots, X_n$) if PROVIDED[t] is not nil.

   (b) I = WHEN[t] =

   - "–" if WHEN[t] is nil;
   - ii($X_1, \ldots, X_n$) if WHEN[t] is not nil.

   (c) O=$\bigcup_j o_j$ in BEGIN–END[t]. Thus, the output list of the event in CTM is composed of each output specified for the statement block in the corresponding NFT. A NFT whose statement block does not contain an output statement is mapped to an event where O="–".

II) Apply the compression of Section 2.3 to obtain CTM from TM.

CTM assumes that context information changes flow from its root towards its leaves. Hence, context information changes specified in a NFT statement block via assignment statements and procedure calls are implicitly covered by CTM.

3.2.4 Example Mapping

A NFS for the OSI Class 0 Transport Protocol is given in Figure 3.3. This NFS, taken from[Ural87] consists of nineteen NFTs (i.e.,t_1 to t_{19}) involving four states: idle, wfcc (wait for connection confirmation), wftr (wait for transport connect response), and data (data transfer). The corresponding CTM is given in Figure 3.4: application of the algorithm of Section 4.4.3 yields this CTM[3].

We define the following abbreviations for **WHEN** components and **output interactions** of the NFS of Figure 3.3 to aid in readability of Figure 3.4:

- A = tcreq(to.t.addr,from.t.addr,qts.req)
- B = tdind(ts.disc.reason,ts.user.reason)
- C = cr(source.ref,option,calling.t.addr,called.t.addr,max.tpdu.size)
- D = dr(dest.refer,disconnect.reason)
- E = dr(disconnect.reason,add.clear.reason)
- F = ndreq(disc.reason)
- G = tdind(ts.user.reason,ts.disc.reason)
- H = cc(max.tpdu.size)
- J = tccon(qts.res)
- K = tdatr(tsdu.fragment)
- L = dt(user.data)
- M = tdati(tsdu.fragment)
- N = ndind()
- P = tdind(ts.disc.reason)
- Q = tdreq(ts.user.reason)
- R = ndreq(disc.reason)
- S = tcres(qts.req)
- T = cc(dest.refer,source.ref,calling.t.addr,called.t.addr,max.tpdu.size)

[3]In order to fit the CTM into the space provided, we use bold labels to denote continuation of the CTM: vertices with identical bold labels correspond to one another

- V = tcind(to.t.addr,from.t.addr,qts.pro)
- W = dr(dest.refer,disconnect.reason,add.clear.reason)
- Y = nrind()

NFS for OSI Class 0 Transport Protocol

```
WHEN tcreq(to.t.addr, from.t.addr, qts.req)
FROM idle
PROVIDED tcreq.in.qts.req = ok
TO wfcc
t1: BEGIN
   local.refer := ...;
   tpdu.size   := ...;
   calling.t.addr := tcreq.in.from.t.addr;
   called.t.addr  := tcreq.in.to.t.addr;
   cr.out.source.ref := local.refer;
   cr.out.option     := 'normal';
   cr.out.calling.t.addr := calling.t.addr;
   cr.out.called.t.addr  := called.t.addr;
   cr.out.max.tpdu.size  := tpdu.size;
   out cr(source.ref, option, calling.t.addr,
       called.t.addr, max.tpdu.size);
   END;
```

```
WHEN tcreq(to.t.addr, from.t.addr, qts.req)
FROM idle
PROVIDED tcreq.in.qts.req <> ok
TO idle
t2: BEGIN
   tdind.out.ts.disc.reason := ...;
   tdind.out.ts.user.reason := ...;
   out tdind(ts.disc.reason, ts.user.reason);
   END;
```

```
WHEN cr(source.ref, option, calling.t.addr,
        called.t.addr, max.tpdu.size)
FROM idle
PROVIDED cr.in.max.tpdu.size = nil
      and cr.in.option = ok
TO wftr
t3: BEGIN
   remote.refer := cr.in.source.ref;
   tpdu.size    := cr.in.max.tpdu.size;
   calling.t.addr := cr.in.calling.t.addr;
   called.t.addr  := cr.in.called.t.addr;
   qts.estimate   := ...;
   tcind.out.to.t.addr   := called.t.addr;
   tcind.out.from.t.addr := calling.t.addr;
   tcind.out.qts.pro     := qts.estimate;
   out tcind(to.t.addr, from.t.addr, qts.pro);
   END;
```

```
WHEN cr(source.ref, option, calling.t.addr,
        called.t.addr, max.tpdu.size)
FROM idle
PROVIDED cr.in.max.tpdu.size <> nil
      and cr.in.option = ok
TO wftr
t4: BEGIN
   remote.refer := cr.in.source.ref;
   tpdu.size    := ...;
   calling.t.addr := cr.in.calling.t.addr;
   called.t.addr  := cr.in.called.t.addr;
   qts.estimate   := ...;
   tcind.out.to.t.addr   := called.t.addr;
   tcind.out.from.t.addr := calling.t.addr;
   tcind.out.qts.pro     := qts.estimate;
   out tcind(to.t.addr, from.t.addr, qts.pro);
   END;
```

```
WHEN cr(source.ref, option, calling.t.addr
        called.t.addr, max.tpdu.size)
FROM idle
PROVIDED cr.in.option <> ok
TO idle
t5: BEGIN
   dr.out.dest.refer := cr.in.source.ref;
   dr.out.disconnect.reason := ...;
   out dr(dest.refer, disconnect.reason);
   END;
```

```
WHEN cc(max.tpdu.size)
FROM wfcc
PROVIDED cc.in.max.tpdu.size <> nil
TO data
t6: BEGIN
   qts.estimate := ...;
   tccon.out.qts.res := qts.estimate;
   in.buffer  := nil;
   out.buffer := nil;
   out tccon(qts.res);
   END;
```

```
WHEN cc(max.tpdu.size)
FROM wfcc
PROVIDED cc.in.max.tpdu.size = nil
TO data
t7: BEGIN
   qts.estimate := ...;
   in.buffer    := nil;
   out.buffer   := nil;
   tccon.out.qts.res := qts.estimate;
   out tccon(qts.res);
   end
```

```
WHEN dr(disconnect.reason, add.clear.reason)
FROM wfcc
PROVIDED dr.in.disconnect.reason = "user.init"
TO idle
t8: BEGIN
   ndreq.out.disc.reason    := dr.in.disconnect.reason;
   tdind.out.ts.disc.reason := dr.in.disconnect.reason;
   tdind.out.ts.user.reason := dr.in.add.clear.reason;
   out ndreq(disc.reason);
   out tdind(ts.user.reason, ts.disc.reason);
```

Figure 3.3: OSI Class 0 Transport Protocol NFS

```
    END;
-----------
WHEN dr(disconnect.reason, add.clear.reason)
FROM wfcc
PROVIDED dr.in.disconnect.reason <> "user.init";
TO idle
t9: BEGIN
    ndreq.out.disc.reason     := dr.in.disconnect.reason;
    tdind.out.ts.disc reason := dr.in.disconnect.reason;
    out ndreq(disc.reason);
    out tdind(ts.disc.reason);
    END;
-----------
WHEN tcres(qts.req)
FROM wftr
PROVIDED tcres.in.qts.req <= qts.estimate
TO data
t10:BEGIN
    local.refer       := ...;
    cc.out.dest.refer := remote.refer;
    cc.out.source.ref  := local.refer;
    cc.out.calling.t.addr := calling.t.addr;
    cc.out.called.t.addr  := called.t.addr;
    cc.out.max.tpdu.size  := tpdu.size;
    in.buffer  := nil;
    out.buffer := nil;
    out cc(dest.refer, source.ref, calling.t.addr,
          called.t.addr, max.tpdu.size);
    END;
-----------
WHEN tcres(qts.req)
FROM wftr
PROVIDED tcres.in.qts.req > qts.estimate
TO idle
t11:BEGIN
    dr.out.dest.refer := remote.refer;
    dr.out.disconnect.reason := ...;
    dr.out.add.clear.reason  := ...;
    tdind.in.ts.disc.reason  := ...;
    out dr(dest.refer, disconnect.reason, add.clear.reason);
    out tdind(ts.disc.reason);
    END;
-----------
WHEN tdreq(ts.user.reason)
FROM wftr
TO idle
t12:BEGIN
    dr.out.disconnect.reason := ...;
    dr.out.add.clear.reason  := tdreq.in.ts.user.reason;
    dr.out.dest.refer := remote.refer;
    out dr(dest.refer, disconnect.reason, add.clear.reason)
    END;
-----------
WHEN tdatr(tsdu.fragment)
FROM data
TO data
t13:BEGIN
    insert(out.buffer, tdatr.in.tsdu.fragment);
    END;
-----------
FROM data
PROVIDED out.buffer <> nil
TO data
t14:BEGIN
    remove(out.buffer, dt.out.user.data);
    out dt(user.data);
    END;
-----------
WHEN dt(user.data)
FROM data
TO data
t15:BEGIN
    insert(in.buffer, dt.in.user.data);
    END;
-----------
FROM data
PROVIDED in.buffer <> nil
TO data
t16:BEGIN
    remove(in.buffer, tdati.out.tsdu.fragment);
    out tdati(tsdu.fragment);
    END;
-----------
WHEN tdreq(ts.user.reason)
FROM data
TO idle
t17:BEGIN
    ndreq.out.disc.reason := tdreq.in.ts.user.reason;
    out ndreq(disc.reason);
    END;
-----------
WHEN ndind()
FROM data
TO idle
t18:BEGIN
    tdind.out.ts.disc.reasson := ...;
    out tdind(ts.disc.reason);
    END;
-----------
WHEN nrind()
FROM data
TO idle
t19:BEGIN
    tdind.out.ts.disc.reason := ...;
    out tdind(ts.disc.reason);
    END;
```

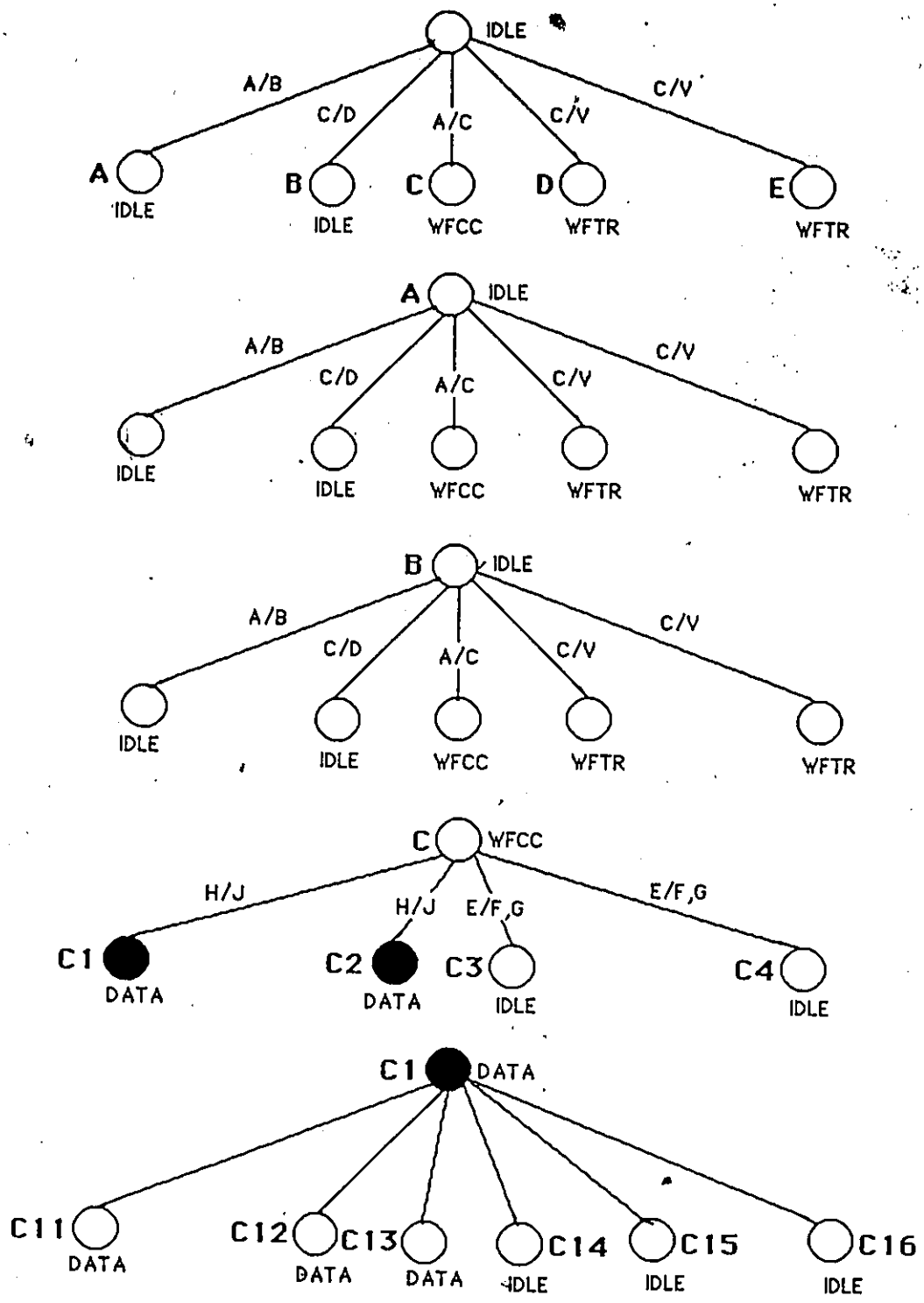
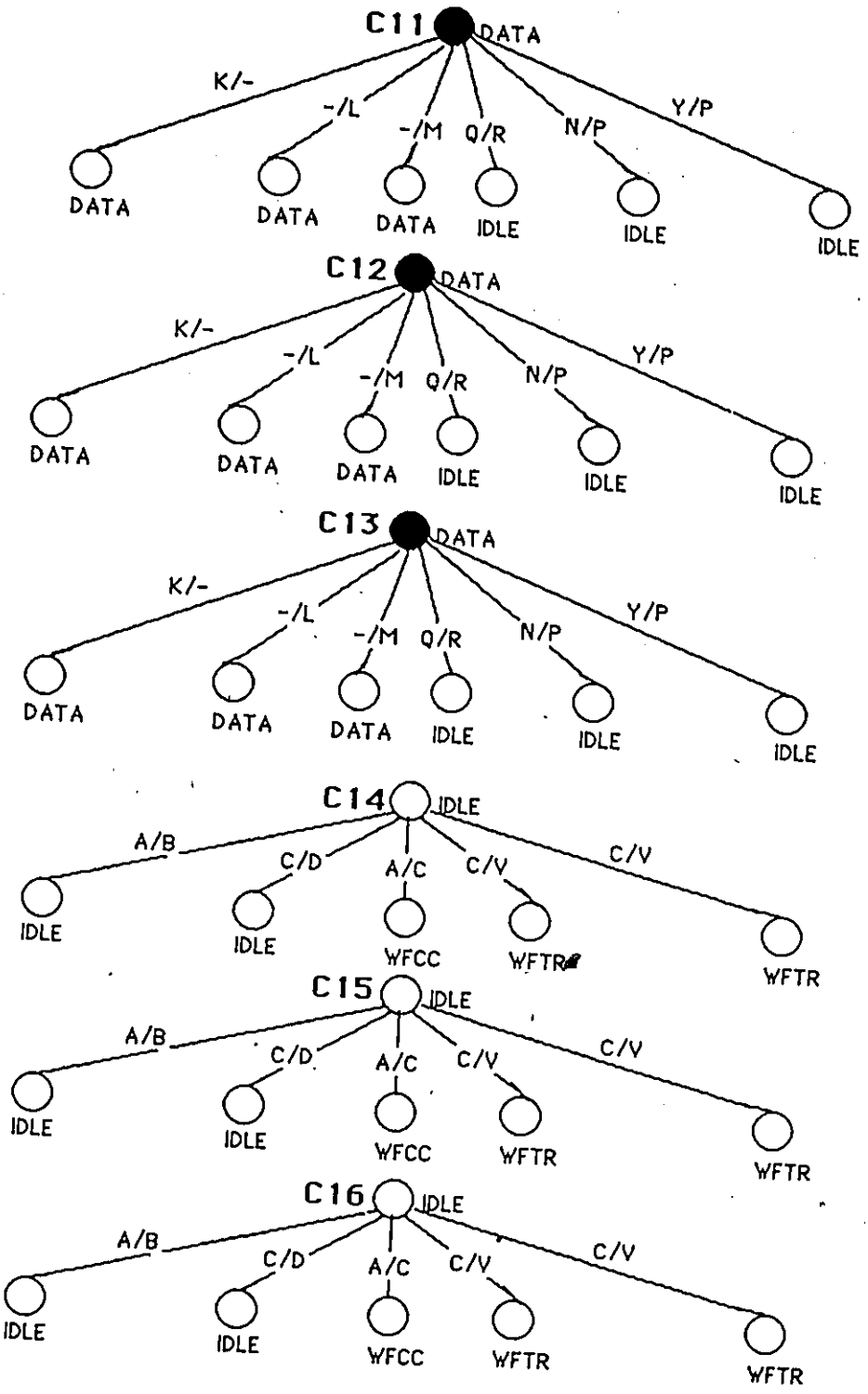
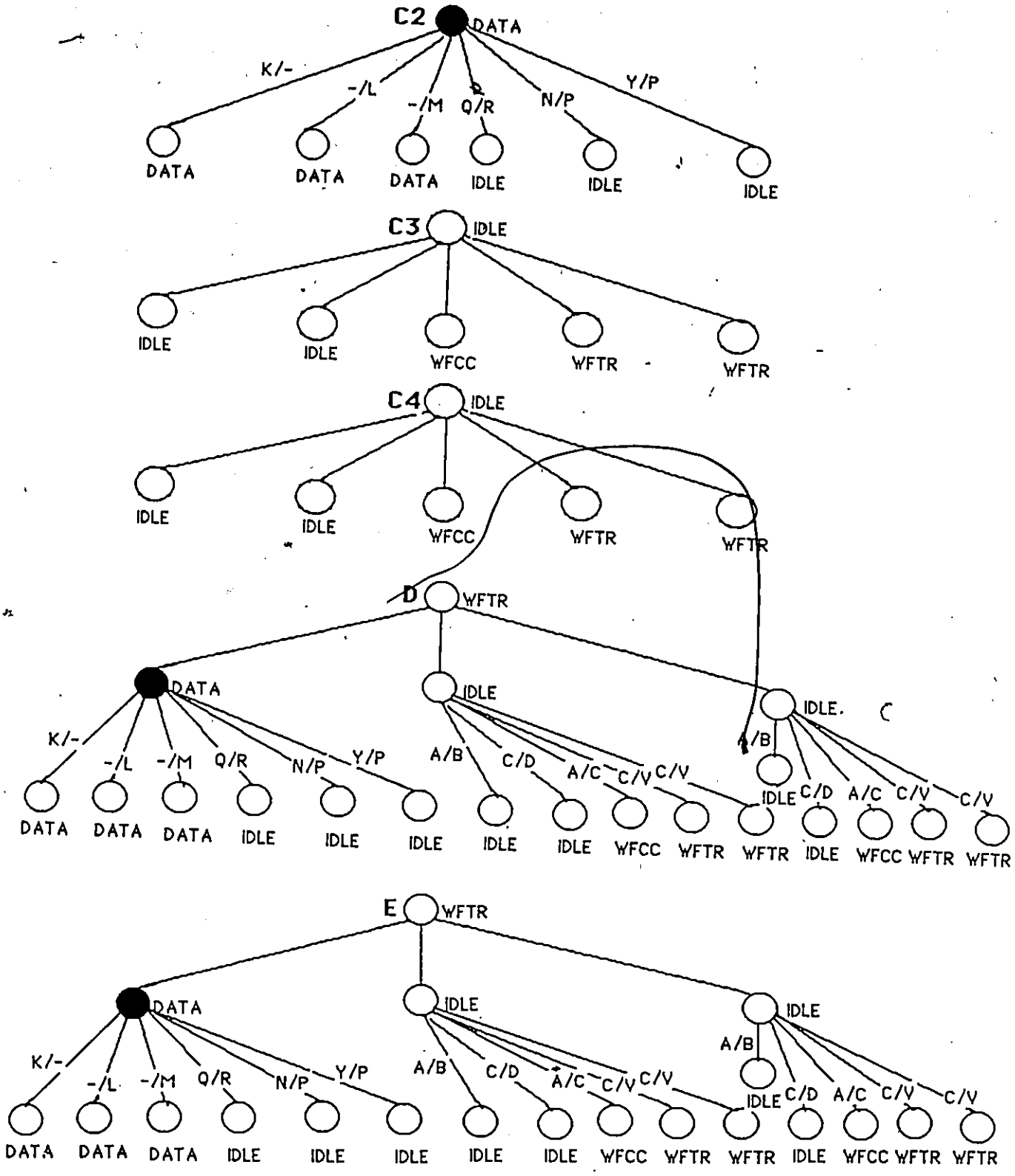


Figure 3.4: Corresponding CTM


C11
DATA
K/-
-/L
-/M
Q/R
N/P
Y/P
DATA
DATA
DATA
IDLE
IDLE
IDLE
C12
DATA
K/-
-/L
-/M
Q/R
N/P
Y/P
DATA
DATA
DATA
IDLE
IDLE
IDLE
C13
DATA
K/-
-/L
-/M
Q/R
N/P
Y/P
DATA
DATA
DATA
IDLE
IDLE
IDLE
C14
IDLE
A/B
C/D
A/C
C/V
C/V
IDLE
IDLE
WFCC
WFTR
WFTR
C15
IDLE
A/B
C/D
A/C
C/V
C/V
IDLE
IDLE
WFCC
WFTR
WFTR
C16
IDLE
A/B
C/D
A/C
C/V
C/V
IDLE
IDLE
WFCC
WFTR
WFTR


C2
DATA
K/-
-/L
-/M
Q/R
N/P
Y/P
DATA
DATA
DATA
IDLE
IDLE
IDLE
C3
IDLE
IDLE
IDLE
WFCC
WFTR
WFTR
C4
IDLE
IDLE
IDLE
WFCC
WFTR
WFTR
D
WFTR
DATA
IDLE
IDLE.
K/-
-/L
-/M
Q/R
N/P
Y/P
A/B
C/D
A/C
C/V
C/V
A/B
IDLE
C/D
A/C
C/V
C/V
DATA DATA DATA IDLE IDLE IDLE IDLE IDLE WFCC WFTR WFTR IDLE WFCC WFTR WFTR
E
WFTR
DATA
IDLE
IDLE
K/-
-/L
-/M
Q/R
N/P
Y/P
A/B
C/D
A/C
C/V
C/V
A/B
IDLE
C/D
A/C
C/V
C/V
DATA DATA DATA IDLE IDLE IDLE IDLE IDLE WFCC WFTR WFTR IDLE WFCC WFTR WFTR

3.3 Tree Model Coverage of CCS

We wish to show that the compressed tree model (CTM) covers the Calculus of Communicating Systems (CCS) developed by Milner [Miln80]. To aid the readers understanding, we develop the coverage in three steps:

1. In Section 3.3.1, we approach the problem conceptually in order to give an intuitive feel for what the coverage involves.

2. In Section 3.3.2, we show a mapping from CCS to a tree representation which expresses behavior of only a limited subset of CCS. In Section 3.3.3 this tree representation is then shown to be covered by CTM. This will serve as a first simplified approximation to mapping from CCS to CTM with the intent of aiding the reader in understanding how various constructs are mapped.

3. In Section 3.3.4, we show a mapping from CCS to a tree representation which fully represents behavior expressed in CCS. In Section 3.3.5 this tree representation is then mapped to CTM in order to show CTM coverage of CCS.

Accompanying figures are provided to help the reader visualize each mapping. Section 3.3.6 gives an example mapping from a CCS specification to its corresponding CTM. Conclusions follow.

3.3.1 Conceptual Coverage

CTM models valid behavior of possible system implementations as described by S in terms of possible valid orderings of events e = (P,I,O). The terms of CCS are behaviors and CCS models functionality through manipulation of guards to these behaviors. CCS has two basic types of guards: unrestricted which specify input and output events and internal(τ) which specify unobservable events.

In CCS, positive labels bind values to variables and negative labels qualify values:

- input(I) in CTM is equivalent to an α-v experiment (v a value submitted by the environment) in CCS,

- output(O) in CTM is equivalent to a $\bar{\beta}$-v experiment (v a value received by the environment) in CCS.

- Null events (P,–,–) in CTM are equivalent to τ events in CCS.

Thus, the three types of guards to behaviors in CCS that are covered by CTM are:

- αx-tuple,
- $\bar{\beta}$E-tuple, and
- τ;

where x-tuple and E-tuple are tuples of variables(input in CTM) and values(output in CTM), respectively.

To gain an intuitive feel for CTM coverage of CCS, we stress the following:

- CTM uses deterministic C-vertices to model deterministic choices. CCS does not explicitly model deterministic or nondeterministic choice. Instead, CCS has a general choice operator: summation "+". Semantic rules on the "+" operator distinguish deterministic from nondeterministic behavior specification. CTM's deterministic vertices cover deterministic summation operations in CCS.
- CTM also covers nondeterministic summation operations in CCS. A CCS summation is nondeterministic if it either specifies two internal events(τ) or specifies two identical experiments with different outcomes from a given state. This is equivalent to the criteria set forth in Section 2.2 for establishing a nondeterministic C-vertex in CTM.
- CTM uses grey C-vertices to model points in S at which determination of whether or not a choice is deterministic is sensitive to the context variables in predicates of alternative events. In CCS, the conditional operator "If E then B else B'"; where E is a boolean expression and B and B' are behavior expressions, can specify grey choice. A conditional operation such as this can specify grey choice since the truth value of E can not always be determined using only syntactic information. Thus, CTM covers grey choice specified in CCS.
- CTM uses terminal N-vertices to denote a point at which S terminates. This is equivalent to the inaction operator *NIL* in CCS. Thus, CTM covers termination of specification in CCS.

- CTM abstracts linear sequences of events. CCS can specify abstraction of behavior directly by specifying behavior identifiers for the abstracted behavior. For example, we may specify the following in CCS:

  $$\bar{\alpha}\text{E-tuple.B.}$$

  Here, we have abstracted specifics of behavior B. CTM covers this type of abstraction by representing the abstracted behavior as a subtree which stands for subsequent behavior. Thus, CTM covers the abstraction capabilities of CCS.

- CCS forbids unguarded recursive definitions: it does not allow infinite sequences of identical behaviors which come about due to the absence of guards. This in turn means that CCS can not define an infinite S of this type. We have no such restriction on S in CTM. Clearly, CTM can cover the restricted specifications of CCS since CTM imposes no limits on the depth of the tree.

This section has conceptually shown that specifications in CCS can be modelled using CTM. We note that Milner's attempt to define CCS in the algebra of communication trees (CT's) [Miln80] is strong evidence for the usefulness of a tree based model such as CTM. We now proceed to define a series of mappings showing CTM coverage of CCS.

3.3.2 Mapping CCS to ST's

Towards a first approximation for showing compressed tree model (CTM) coverage of CCS, we show a mapping of CCS to synchronization trees (ST's) [MILN80].

ST's are an extension of rigid synchonization trees (RST's) [Miln80]. The elementary algebra of RST's and ST's are the same. Definitions needed for the following discussion are:

- A **label** is a member of a given (fixed) label set Λ.

- A sort is a subset of Λ.

- A ST of sort L is a rooted, unordered, finitely branching tree, each of whose edges is labelled by a member of $(L \bigcup \tau)$. The vertices of a ST are abstract representations of states. The label "τ" stands for a silent (unobserved) transition.

- A ST is a representation of a transition graph[Miln80].

To show a mapping from CCS to ST, we give a mapping for each CCS behavior type. Refer to Figure 3.6 on page 59 for the following:

Inaction

Inaction specified by *NIL* is represented in ST as a single vertex (black circle). Label sets in both CCS and ST for this behavior are empty.

Summation

The summation operator "+" represents choice between two or more behaviors. The resulting behavior expression has a label set which is the union of label sets of all components of the choice operator(s). In ST, this choice is represented as a vertex with out-degree equalling the component number of behaviors in the CCS summation operation(s). Each edge emanating from a choice vertex in ST represents the first event of the alternative behaviors represented as trees in the corresponding CCS specification. Let

- t1=$a_1, \ldots, a_n$
- t2=$b_1, \ldots, b_n$

specify two behaviors in CCS. t1+t2 is mapped into ST as C_1=(V_1,C_SET_1); where

- C_1 is the choice definition,
- V_1 is a vertex representing the point in the specification where choice is specified,
- C_SET_1 is the set of edges emanating from V_1. Here, C_SET_1 is (a_1,b_1).

We label each edge in C_Set_i, i = (1,..., number of points of choice in the CCS specification), with the corresponding CCS label; namely, a_1 and b_1 for C_SET_1. Remaining sequences of behavior $a_2, \ldots, a_n$ and $b_2, \ldots, b_n$ are mapped as linear sequences in ST. The label set in ST is the same as that in CCS.

Events

Events are guards to behavior expressions and are of three types: binding, qualifying, and internal. Binding events, which denote input from the environment, are represented as positive labels and bind values to tuples of free variables. Qualifying events, which denote output to the environment, are represented as negative labels

and qualify tuples. Internal events are represented with the label τ. Since ST's are unable to specify input to or output from the environment, the mapping from CCS to ST is incomplete. If, though, we treat the guards in CCS as generic labels (i.e., positive and negative labels are not differentiated), then we simply map guards in CCS to edges in ST with the generic label. "τ" guards in CCS map directly to edges in ST with the label "τ".

Event Sequences

Sequences of events in CCS are separated in ST by vertices representing abstract states. Let $e_1, \ldots, e_n$ be a sequence of events in a CCS specification. This sequence is represented in ST as $((v_1,\ldots,v_{n+1}),(e_1, \ldots, e_n))$, where v_j, j=1,...,n+1, represents the current abstract state, and e_i, i=1,...,n, is the edge corresponding to the event e_i between vertices v_i and v_{i+1}. Each edge is labelled with its corresponding event. Label sets are preserved for the mapping of events from CCS to ST since they are treated as generic entities.

Composition

The composition operator "|" in CCS is used to represent the parallel composition ("running") of two or more behaviors. It is also used in ST's to express parallel composition of component ST's. Thus, the mapping of composition from CCS to ST's is direct. Let

- t_1=$a_1, \ldots, a_n$;
- t_2=$b_1, \ldots, b_n$

specify two behaviors in CCS. ST can represent their parallel composition in two ways.

1. If, in ST, we linearize the composite behavior of t_1 and t_2, then at every abstract state vertex in ST we give choice of next event between next possible parallel events. Thus, for our first choice, we have $C_1 = (V_1, C\_SET_1)$, where C_SET_1=(a_1,b_1). The subtree of a_1 will have choice (V_2, C_SET_2), where C_SET_2=(a_2,b_1), and so on. If two or more members of C_SET_i are co-labels (matching positive and negative labels) and we are linearizing the composition, then we must specify a τ edge for each such pair in C_SET_i. We add such τ choices to C_Set_i. Thus, if

   - $C_k = (V_k, (a, \bar{a}))$,

then we have edges emanating from V_k with labels

- a,
- $\bar{a}$, and
- τ;
- $C\_SET_k = C\_Set_k \bigcup \tau$.

2. If we do not wish to enumerate possible choices, we specify parallel composition of component subtrees using "|". Thus, if we have enumerated choices a_1, b_1, then the component subtree of a_1 can be represented as:

$$(a_2, \ldots, a_n)|(b_1, \ldots, b_n) = t_1'|t_2.$$

Label sets are preserved for the mapping from CCS to ST's (assuming CCS labels are treated generically).

Restriction

The restriction operator "\" is used to specify forbidden behavior. Arguments of the restriction operator are labels. Behavior specified using a given label is forbidden if this label is restricted. In ST's, any edges with a restricted label are pruned along with their subsequent subtrees. The restriction operator can be used in ST's to express forbidden behavior within component subtrees. Thus, the mapping of restriction from CCS to ST's is direct. Let the CCS label set be L, restricted labels be $l_1, \ldots, l_n$. Then, the restricted label set is L – $(l_1, \ldots, l_n)$. Label sets of the CCS specification and ST representation are equivalent through restriction.

Relabelling

The relabelling operator "B[S]" for behavior B is used to give new labels to the current behavior label set. Any relabelled behavior in CCS is directly represented in ST under the new label. Thus, the mapping from CCS to ST is direct under relabelling. ST represents only the new label set (i.e., abstracts the relabelling operation).

Parameterization

The identifier construct "b$(E_1, \ldots, E_n)$" is used for the parameterization and subsequent instantiation of behaviors. Parameterization requires the use of free variables and instantiation of these variables with actual values. Since ST is unable to represent propagation of values, representation of the CCS identifier construct in ST is impossible. Thus, there is no mapping for the CCS identifer construct to ST's.

Conditional Choice

The conditional construct "*if E then B1 else B2*" is used for specification of choice between two or more behaviors based on boolean predicates. Predicates for the conditional construct are, in general, composed in part of variables. Since ST can not represent variables, there is no mapping for the CCS conditional construct to ST's.

3.3.3 Mapping ST's to CTM

Having mapped CCS to ST's, our first approximation for CTM coverage of CCS is to now show a mapping from ST's to CTM. To do so, we map each construct in ST to a construct in CTM. Refer to Figure 3.7 on page 61 for the following:

Inaction

An inaction is represented in CTM by a null event e=(P,-,-).

Event Sequence and Choice

Since a ST is just a representation of a transition graph[Miln80], each edge in ST is neither an input (I) nor an output (O). However, if we treat each label in ST as an input, we can then show CTM coverage of ST's. Consider a linear sequence of transitions $(t_1, t_2, \ldots, t_n)$ in a ST with the initial vertex v_1 and the terminal vertex v_{n+1}. This sequence can be represented in CTM as a single edge $e_1' = (e_1, \ldots, e_n)$:

- e_i=(-,t_i,-),
- i=1,...,n, and
- "–" denotes no predicate and no output respectively.
- If choice_set of v_{n+1} consists of a single transition then v_{n+1} in CTM is a N–vertex.
- If choice_set of v_{n+1} consists of more than one transition, then v_{n+1} in CTM is a deterministic or nondeterministic C–vertex.

One can employ the criteria of Section 2.2 to determine the vertex type. If in ST, a choice among a set of alternatives at vertex v_k is τ (unobserved), then CTM models this as an edge of type E_4. All other alternatives are modelled as edges of type E_2.

Composition

Composition in ST can be represented by "|". In CTM, choices represented by

a parallel composition must be linearized. Let C=$c_1, \ldots, c_k$ be the possible next events for parallel processes $P_1, \ldots, P_k$ at a given point in a CCS specification. In ST, these next events are represented in the composition $P_1 \mid P_2 \mid, \ldots, \mid P_k$. In CTM, if for all c_i, c_j, i≠j; i, j = $1, \ldots, k$; $c_i \neq c_j$, and $c_i \neq \tau$; then we create a deterministic vertex with out-going edges $(e_1, e_2, \ldots, e_n)$, $e_i = (-, c_i, -)$, $1 \leq i \leq n$. Otherwise, if for any c_i,c_j; i,j = 1,...,k; c_i=c_j, or $c_i = \tau$, then we create nondeterministic vertex with out-going edges $(e_1, \ldots, e_n)$, $e_i = (-, c_i, -)$, $1 \leq i \leq n$.

Restriction

Restricted trees in ST map directly into CTM. Since the mapping from ST to CTM demands composite subtrees representing parallelism be linearized in CTM, restricted branches and their subtrees will be pruned when the mapping is performed. Parallel composition specified in ST must be linearized and any restricted branches pruned in CTM.

3.3.4 Mapping CCS to CT's

It is shown in [Mil 80] that when considering atomic events, it makes no difference whether we think of CCS programs, or of the communication trees (CT's) [Mil 80] which they denote. Thus, when we later show a mapping from CT's to CTM, we can claim CTM coverage of CCS.

CT's, a further extension of ST's, are defined as follows. A CT of sort L is a finite collection (multiset) of pairs, each of which is of the following form:

- (α,f); $\alpha \in$ L, where f is a family of CT's of sort L indexed by the value set appropriate to α; or
- ($\bar{\beta}$,(v,t)); $\bar{\beta} \in$ L, where v is a value appropriate to $\bar{\beta}$ and t is a CT of sort L; or
- (τ,t) where t is a CT of sort L.

As in ST's, the vertices of CT's are abstract representation of states. "τ" again stands for a silent (unobserved) transition.

The primary difference between CT's and ST's is that CT's are able to specify value communication. CT's, unlike ST's, are able to cover all three types of events which can be specified in CCS: binding, qualifying, and internal. Thus, CT's, as

their name implies, are capable of expressing input, output, and internal communication through use of variables.

To show a mapping from CCS to CT, we give a mapping for each CCS behavior type. Refer to Figure 3.8 on page 63 for the following:

Inaction

Inaction specified by NIL is represented in CT as a single vertex(black circle). Label sets in both CCS and CT for this behavior are empty.

Summation

Choice specified using the summation operator "+" is represented in CT as a vertex with out-degree equalling the component number of choice behaviors in the CCS summation operation(s). As in the mapping to ST, each edge emanating from a vertex in CT represents the first event of the alternative behavior trees in the CCS specification. The label set in CT is the same as that in CCS.

CCS is founded on two central ideas: observation and synchronized communication. The behavior expressed by a CCS specification is thought to be observed by an observer external to the specified system. The observer can witness two basic types of communication: input to the system (binding events) and output from the system (qualifying events). Further, CCS can specify internal events of the system which can not be witnessed by an external observer. As well, CCS can specify forbidden behavior in terms of system events; this is called **restriction**. We wish to show mappings of these various CCS features to CT's.

Binding Events

Binding events in CCS specify the binding of values to tuples of free variables. Positive labels bind variables: these variables will take on values of type appropriate to label Λ. We think of a binding event as a Λ-v experiment, where v is the (tuple of) value(s) submitted by the observer of type appropriate to Λ. Binding events in CCS map directly to CT. Let the binding event in CCS be:

$$\text{BA}=\alpha x_1, \ldots, x_n.\text{t}$$

- x_i = free variable to which a value is bound, i=1,...,n.
- label $\alpha \in \Lambda$.
- t the subsequent behavior.

A binding event is represented as an edge $E_1=(V_1,E_2)$ in CT with label α:

- V_1 is the abstract previous state vertex in CT,
- E_2 is an edge in CT along which we specify, at regular intervals, each value V_j, j=1,2,..., which the free variable tuple $x_1, \ldots, x_n$ in BA can have bound to it,
- each V_j is the root vertex of a CT t_j indexed by j.

Thus, depending on the value tuple V_j communicated, we have different subsequent behaviors. If Λ is an infinite-value-domain, then E_2 is represented as being infinite with an arrow head in the direction of increasing index j. The label sets for CCS and CT are identical for binding events.

Qualifying Events

Qualifying events specify qualification of values to be received by the observer. Negative labels qualify variables: qualified variables output values of type appropriate to $\bar{\Lambda}$. We think of qualifying events as $\bar{\Lambda}$-v experiments, where v is the (tuple of) value(s) received by the observer of type appropriate to $\bar{\Lambda}$. Qualifying events in CCS map directly to CT. Let the qualifying event be:

$$\text{QA} = \bar{\alpha}\text{v.t}$$

- v = bound variable of type appropriate to $\bar{\alpha}$;
- $\bar{\alpha} \in \bar{\Lambda}$;
- t the subsequent behavior.

Represent label $\bar{\alpha}$ of QA as an edge E_1=(V_1,V_2) in CT with label $\bar{\alpha}$:

- V_1 is the abstract previous state vertex in CT,
- V_2 is the root of the subsequent behavior tree t which is labelled with bound variable v of QA.

The label sets for CCS and CT are identical for qualifying events.

Internal Events

Internal events map directly to edges in CT with label "τ".

Event Sequences

Sequences of events in CCS are separated in CT, as they are in ST, by vertices representing abstract states.

Composition

Composition of parallel behavior specified in CCS using the composition operator "|" is represented directly in CT using "|". Thus, as with the mapping to ST's, component subtrees running in parallel can be expressed directly in CT. There are four cases.

- For each binding event $\alpha x_1, \ldots, x_n$.t as above composed with behavior u, specify t_j|u for each subtree t_j indexed by value v_j bound to tuple $x_1, \ldots, x_n$ of type appropriate to α in CT.
- For each qualifying event $\bar{\alpha}$V.t as above composed with behavior u, specify t|u for subsequent behavior tree t in CT.
- For each internal event τ.t composed with behavior u, specify t' |u in the component subtree t in CT.
- For each binding event $\alpha x_1, \ldots, x_n$.t and complementary qualifying event $\bar{\alpha}$v.u as above, give a branch labelled τ and component subtree $t_j|\ u'$ in CT.

Linearization of choices for parallel composition in CT is carried out in the same manner for CT's as for ST's. Label sets are preserved for the mapping of composition from CCS to CT.

Restriction

Restriction of behavior is represented in CT, as in ST, using the restriction operator "\". Any branches in CT with a restricted label are pruned along with their component subtrees. The restriction operator is used in CT to express forbidden behavior within component subtrees. The restricted label set in CCS maps directly to CT.

Relabelling

Relabelling of behavior in CCS maps directly to CT by relabelling the branches of CT. CT abstracts the relabelling operation.

Parameterization

The identifier operation is represented directly in CT. Let the CCS behavior identifier be b($e_1, \ldots, e_n$):

- For each e_i, i=1,...,n, which parameterizes a binding event, we create a binding branch in CT.

- For each e_i, i=1,...,n, which parameterizes a qualifying event, we create a qualifying branch in CT.

Occurrence of a behavior identifier in a CCS behavior expression is represented in CT as a component subtree $[[b(e_1, \ldots, e_n)]]$, where [[]] stands for "the CT of", and each e_i, i=1,...,n, has a specific value appropriate to its type[4]. The label set of CT is the same as that of CCS through the identifier operation.

Conditional Choice

The conditional operator "If E then B else B'" maps directly to CT. In CT, the choice is represented as a pair of choice edges with the necessary propagation of free variables. Each choice edge is labelled with both the predicate (condition) necessary for it to be taken (E or $\neg(E)$), and the label of its corresponding event. Thus, we see that the mapping of the CCS conditional operator to CT is the same as the mapping of the CCS summation operator to CT with the exception that the first mapping implies mapping predicates as components of labels in CT. CT abstracts the conditional operator. The CT label set is the same as that of CCS through the conditional operation.

3.3.5 Mapping CT's to CTM

Having mapped CCS to CT's, we now map CT's to CTM. As previously stated, since CT's cover CCS, we can claim CTM coverage of CCS by showing a mapping from CT's to CTM. To map CT to CTM, we map each CT construct to CTM. Refer to Figure 3.9 on page 66 for the following:

Inaction

A NIL event in CT, as in ST, can be represented in CTM as a null event e=(P,-,-).

Binding Events

To map binding events of CT's to CTM, consider a binding event BA = (v_1,e_1,e_2) in CT, where:

- v_1 is the abstract state vertex in CT at which we specify the binding event
- e_1 is the edge in CT with label α specifying the binding event

[4]CCS restricts behavior identifiers by imposing they be **guardedly-well-defined**[Miln80]. Basically, this requirement forbids infinite unguarded recursion in the CCS specification. This requirement ensures a unique CT for the behavior identifier and assigned tuple value.

- e_2=(v_0,...,v_k) is the edge along which we specify value tuples v_0,...,v_k of type appropriate to α.

Represent the binding event in CTM as an edge e'=(v_1, v_2) where:

- e' is (–,α,–)
- v_1 is the initial vertex of the event
- v_2 is the terminal vertex of the event.

We note that CTM statically represents the binding event whereas CT attempts to give a dynamic representation by enumerating the possible set of values which can be bound to the input tuple. However, since CTM assumes flow of context changes from root–to–terminal–leaf paths, the CTM and CT representations for binding events are equivalent.

Qualifying Events

To map qualifying events of CT's to CTM, consider a qualifying event QA=($\bar{\alpha}$,v_1,v_2) in CT where:

- v_1 is the abstract initial state vertex
- v_2 is the terminal vertex of the event labelled with qualified variable v of type appropriate to $\bar{\alpha}$
- $\bar{\alpha}$ is the label of the qualifying event.

We represent QA in CTM with edge e'=(v_1, v_2) where:

- e' is (–,–,$\bar{\alpha}$);
- v_1 is the initial vertex of the event;
- v_2 is the terminal vertex of the event.

Internal Events

The mapping from CT to CTM for internal events is the same as the mapping for qualifying events with the exception that we label edge e' in CTM by –/–.

Event Sequences

Sequences of events in CT can be composed of internal, binding, or qualifying events.

Consider the sequence of transitions $(t_1, t_2, \ldots, t_n)$ in CT with the initial vertex v_1 and the terminal vertex v_{n+1}. This sequence can be represented in CTM by the edge $c_1' = e_1, \ldots, e_n$ where:

- e_i=(-,-,-) if t_i is an internal(τ) event
- e_i=(-,-,(qualified-variable)) if t_i is a qualifying event
- e_i=(-,bound-variable,-) if t_i is a binding event.

If choice_set of v_{n+1} consists of a single transition and is the last vertex in the linear sequence, then v_{n+1} is a terminal N–vertex. If choice_set of v_{n+1} consists of more than one transition in CT, then v_{n+1} is a deterministic or nondeterminjstic C–vertex.

Choice

Choice represented in CT is represented in CTM as either a deterministic, nondeterministic, or grey vertex with corresponding choice_set. The criteria of Section 2.2 can be applied to determine the choice type. However, for sake of clarity, let us consider each case in turn.

Deterministic Choice is mapped into CTM as follows: Let the choice given in CT be C=(CV,($e_1, \ldots, e_n$)); where CV is the abstract state choice vertex and $e_1, \ldots, e_n$ are the alternative events with null predicates. If each e_i is a binding event BA_i=(CV,$E_{1\alpha_i}$,$E_{2\alpha_i}$); where:

- $E_{1\alpha_i}$ is the edge labelled α_i specifying a binding event,
- $E_{2\alpha_i}$ is the edge along which we specify values the tuple of type appropriate to α_i can take on,
- $\alpha_i \neq \alpha_j$, i≠j,
- $i, j = 1, \ldots, n$,

then we specify this choice in CTM as a deterministic C–vertex. Each alternative event is represented by an edge in CTM (–,α_i,–) ; i=1,...,n. Thus, when considering a binding event, if we do not have any two alternatives with identical labels, then we have a deterministic vertex in CTM.

Nondeterministic choice is mapped into CTM as follows: Let the choice given in CT be C=(CV,($e_1,\ldots,e_n$)), as above. If each e_i is a binding event with null predicate BA_i=(CV,$E_{1\alpha_i}$,$E_{2\alpha_i}$) as above; and for some label α_i associated with edge $E_{1\alpha_i}$ and some label α_j associated with edge $E_{1\alpha_j}$, i≠j; α_i=α_j, then we specify this choice in CTM as a nondeterministic C-vertex. Each alternative event is represented by an edge in CTM (-,α_i,-), i=1,...,n. Thus, when considering a binding event, if we do have any two alternatives with identical labels, then we have a nondeterministic vertex in CTM assuming absence of event predicates in CT. Likewise, if we have two or more alternatives which are either internal events or qualifying events with null predicates in CT, then we also have a nondeterministic vertex where each alternative is:

- (-,α_i,-) for a binding event,
- (-,-,$\bar{\alpha}_i$) for a qualifying event, and
- (-,-,-) for an internal event.

Edges of the vertex are labelled as explained previously.

Grey choice is mapped into CTM as follows: If at some point in CT, we have choice between two or more alternative events $e_1,\ldots,e_k$ with corresponding predicates $P_1,\ldots,P_k$; and $P_1,\ldots,P_k$ are such that it is not known whether deterministic or nondeterministic choice can be specified depending on the instantiation of predicate values, we represent this choice in CTM as a grey vertex. Alternative events are represented as explained previously.

Composition

Composition of behaviors can be represented directly in CT using the composition operator "|". In CTM, as with the mapping from ST, choices represented by a parallel composition must be linearized. Let choice_set = $(c_1,\ldots,c_n)$ be the set of alternative events for parallel processes $p_1,\ldots,p_n$ at a given point in CT. Each $c_i \in$ choice_set, i=1,...,n, is either a binding event, qualifying event, or internal event. Let:

- bind_set be the subset of choice_set composed of binding events,
- qualify_set be the subset of choice_set composed of qualifying events, and

- internal_set be the subset of choice_set composed of internal events.

We represent the choice of next event in CTM between each c_i as a deterministic, nondeterministic, or grey vertex:

- $Bind\_set_i$ is $\bigcup_{i=1}^{n} (P_i,\alpha_i,\text{-})$; where P_i is the predicate to the binding event and α_i is the label of the binding event,
- Qualify_Set$_j$ is $\bigcup_{j=1}^{n} (P_j,\text{-},\bar{\alpha_j})$; where P_j is the predicate to the qualifying event and $\bar{\alpha_j}$ is the label of the qualifying event; and
- Internal_Set$_k$ is $\bigcup_{k=1}^{n} (P_k,\text{-},\text{-})$; where P_k is the predicate to the internal event.

The following rules hold:

1. If $bind\_set_i \neq bind\_set_l$, i≠l, and if qualify_set and internal_set are both empty, then the choice is represented in CTM by a deterministic vertex as explained previously.
2. If $bind\_set_i = bind\_set_l$, i≠l, with P_i=true and P_l=true, and/or $qualify\_set_j$ and $qualify\_set_m$ exist such that P_j = true and P_m = true , and/or $internal\_set_k$ and $internal\_set_n$ exist such that P_k = true and P_n = true ,and/or $qualify\_set_j$ and $internal\_set_k$ exist such that P_j = true and P_k = true, then the choice is represented in CTM by a nondeterministic vertex as previously explained.
3. Any other choice is represented in CTM by a grey vertex.
4. Alternatives are represented as edges in CTM as explained previously.
5. Only when no parallel choice between two or more behaviors remain can we abstract linear sequences of events in CT into CTM edges.

Restriction

Restricted trees in CT map directly into CTM. Since the mapping from CT to CTM demands composite subtrees be enumerated in CTM, restricted branches and their subtrees will be pruned when the mapping is performed.

Parameterization

CT is able to specify parameterization of component behavior trees using the operator "[[]]". These component behavior subtrees must be enumerated and represented using choice and sequence representations in CTM as explained above.

3.3.6 Example Mapping

Consider the following CCS specification from [Miln80]:

q=$(P_1 \mid P_2 \mid)\backslash\pi\phi$ where:

$P_1 = \pi\alpha_1\beta_1\phi P_1$

$P_2 = \pi\alpha_2\beta_2\phi P_2$

s=$\overline{\pi}\overline{\phi}s$

Then, by CCS expansion theorem[Miln80]:

$$
\begin{aligned}
q &= (\pi\alpha_1\beta_1\phi P_1 \mid \pi\alpha_2\beta_2\phi P_2 \mid \overline{\pi}\overline{\phi}s)\backslash\pi\backslash\phi \\
&= \tau((\alpha_1\beta_1\phi P_1 \mid P_2 \mid \overline{\phi}s)\backslash\pi\backslash\phi) + \\
&\quad \tau((P_1 \mid \alpha_2\beta_2\phi P_2 \mid \overline{\phi}s)\backslash\pi\backslash\phi \\
&= \tau\alpha_1\beta_1((\phi P_1 \mid P_2 \mid \overline{\phi}s)\backslash\pi\backslash\phi) + \\
&\quad \tau\alpha_2\beta_2((P_1 \mid \phi P_2 \mid \overline{\phi}s)\backslash\pi\backslash\phi \\
&= \tau\alpha_1\beta_1\tau((P_1 \mid P_2 \mid s)\backslash\pi\backslash\phi) + \\
&\quad \tau\alpha_2\beta_2\tau((P_1 \mid P_2 \mid s)\backslash\pi\backslash\phi) \\
&= \tau\alpha_1\beta_1\tau q + \tau\alpha_2\beta_2\tau q
\end{aligned}
$$

Clearly, the composite behavior is infinitely recursive since the two derived choice behaviors specify recursive invocations of behavior q. Our key then is to label those vertices in CTM where a recursive instantiation occurs with the instantiated behavior identifier. Producing a finite CTM is then simply a matter of pruning beneath the second occurence of a given vertex label. The labelled CTM corresponding to the expanded CCS behavior expression is given in Figure 3.5 where:

- $e_1' = (-,-,-)^*(-,\alpha_1,-)^*(-,\beta_1,-)^*(-,-,-)$
- $e_1' = (-,-,-)^*(-,\alpha_2,-)^*(-,\beta_2,-)^*(-,-,-)$

3.3.7 Conclusions

We have shown how CTM covers CCS. We first showed a coverage informally in order to give an intuitive feel for the problem. We then proceeded to show four mappings. The mappings from CCS to ST and from ST to CTM gave us our first approximation for coverage. The mappings from CCS to CT and from CT to CTM showed that CTM does cover CCS.

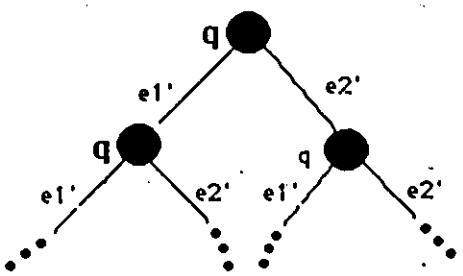


Figure 3.5: Example CCS to CTM Mapping

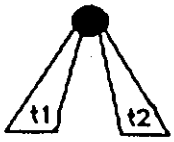
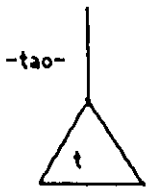
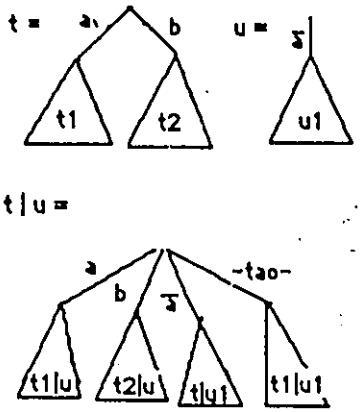
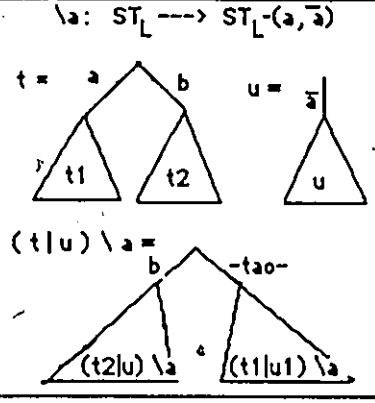
CCS Behavior	ST Representation	Mapping
Nil		Direct. Specifies inaction.
Summation: t1 + t2		Direct. Specifies choice between behavior expressed by t1 and t2.
Action: at. (binding) $\overline{a}$t. (qualifying) -tao-t. (unary)		Incomplete. ST does not specify variable(input) or value(output) expressions. Internal event -tao- maps directly from CCS to ST.
Composition t | u		Direct. Composition of behaviors in CCS is directly expressed in the ST representation using the parallel operator "|".
Restriction (t | u) \ a		Direct. Deny all a and $\overline{a}$ experiments so that t\a is formed by pruning away all branches and sub-branches labelled a or $\overline{a}$.

Figure 3.6: Mapping CCS to ST

Relabelling: B[S]	[S]: ST_L ---> ST_M	Abstracted. Label set L maps to label set M in ST.
Identifier: b(E1,...,En)		No mapping. ST can not represent instantiation and propagation of free variables.
Conditional: if E then B else B'		No mapping. ST can not represent instantiation and propagation of free variables.

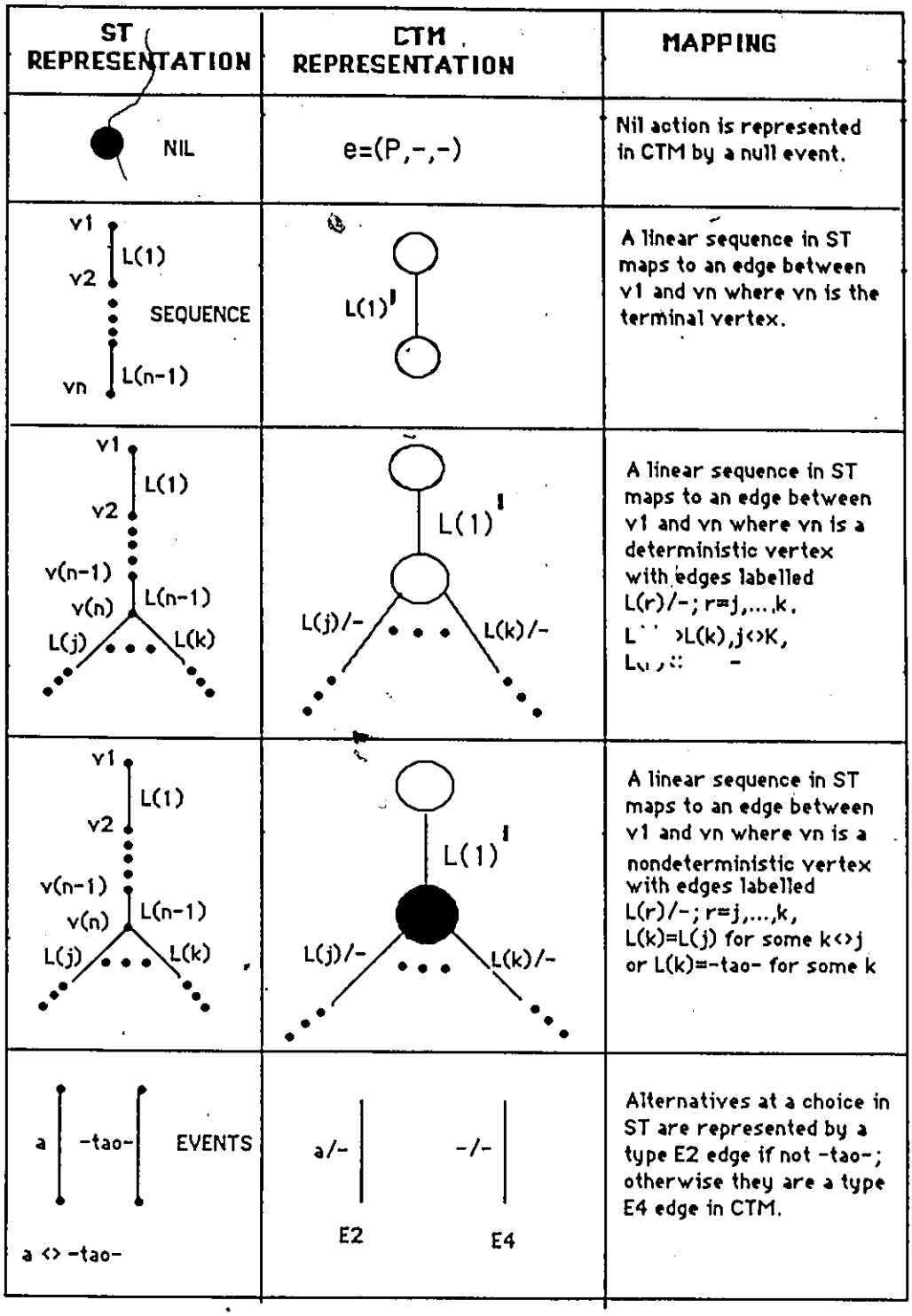
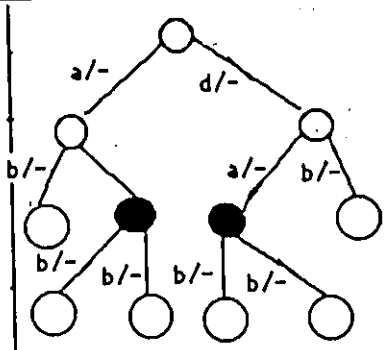
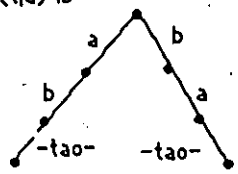
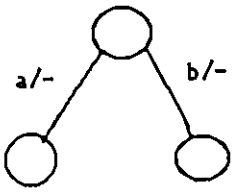


Figure 3.7: Mapping ST to CTM

<table>
<tr>
<td>Let t1 = ab
t2 = db
 

COMPOSITION</td>
<td> 
</td>
<td>Composition represented in ST is linearized in CTM. Choice between 2 or more parallel events is represented as edges emanating from a deterministic vertex if no two events are identical and there is no -tao- event among the choices, otherwise it is represented as edges emanating from a nondeterministic vertex. Events remaining in one process when no events remain in other parallel processes are incorporated into CTM as an edge leading to a terminal vertex of CTM.</td>
</tr>
<tr>
<td>Let t = ac
u = bc
RESTRICTION
(t|u)\b =
 
</td>
<td> 
</td>
<td>Restricted ST's map directly to CTM using choice and sequence mappings.</td>
</tr>
</table>

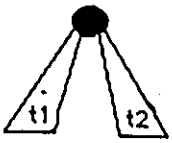
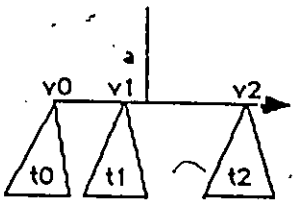
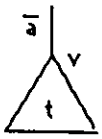
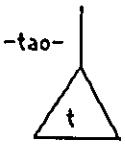
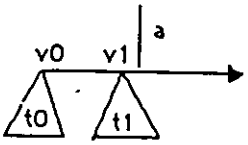
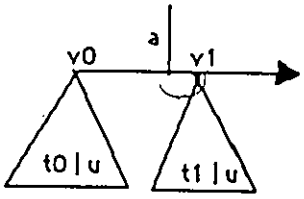
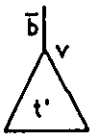
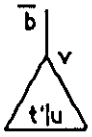
CCS Behavior	CT Representation	Mapping
Nil		Direct. Specifies inaction.
Summation: t1 + t2		Direct. Specifies choice between behavior expressed by t1 and t2.
Binding Event $ax_1 \ldots,x_n.B \quad x_i \text{ in } V_a$		Direct. The value tuple $v_i = x_1,\ldots,x_n$ indexes further expressed behavior trees t0,...,tn. CT is infinite if V_a is an infinite value domain.
Qualifying Event $\bar{a}v.t \quad v \text{ in } V_a$		Direct. The tuple v bound to ā is "output" and t expresses the subsequent behavior.
Internal Event -tao-t		Direct. -tao- specifies unobserved action and subsequent behavior is expressed by tree t.
Composition: t | u	i) for each branch of CT  a branch 	Direct. Composition of behaviors in CCS is directly expressed in the CT representation using "|".

Figure 3.8: Mapping CCS to CT

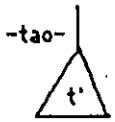
ii) for each branch of CT



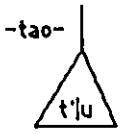
a branch



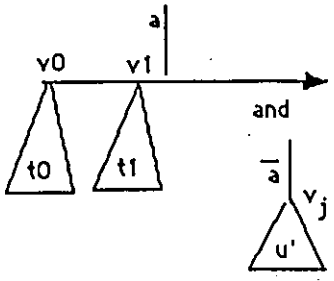
iii) for each branch of CT



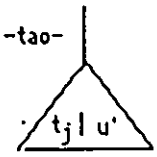
a branch



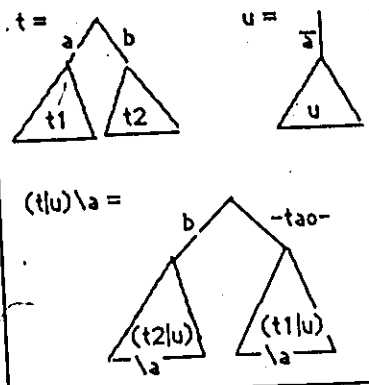
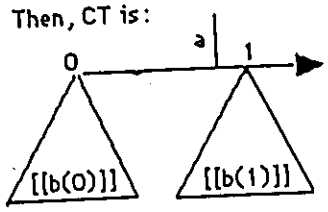
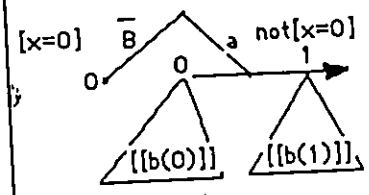
iv) for each pair of branches of CT



a branch



Similarly for branches of u.

Restriction: t \ a	\a: CT_L ---> $CT_{L-(a,\bar{a})}$ 	Direct. Deny all a and $\bar{a}$ experiments so that t\a is formed by pruning away all branches and subbranches labelled by a or $\bar{a}$.
Relabelling: B[S]	[S]: CT_L ---> CT_M	Abstracted. Label set L maps to label set M in CT.
Identifier: b(E1,...,En)	Example. Let b(y) = ay.b(y) Then, CT is:  [[]] = CT with (E1,...,En) taking on tuple value (0,1,...)	Direct. If the behavior identifier b is guardedly well-defined, then a unique CT is defined for the behavior identifier and tuple value for E1,...,En.
Conditional: If E then B else B'	Ex. If x=0 then $\bar{B}$x.NIL else ay.b(y) 	Abstracted. Represent in CT as a choice with necessary propagation of variables and labels including the predicate and choice of the label.

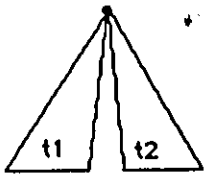
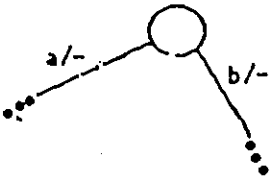
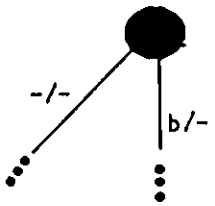
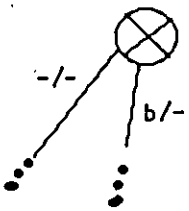
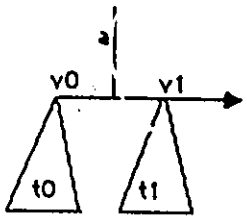
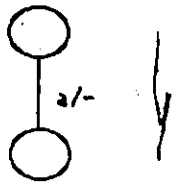
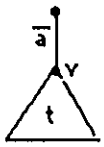
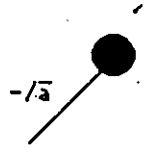
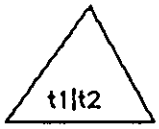
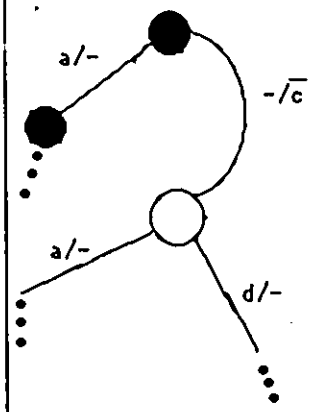
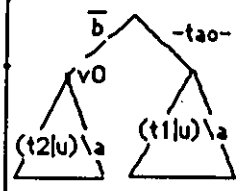
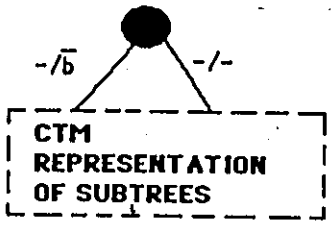
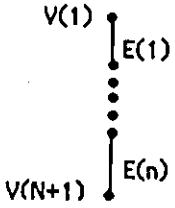
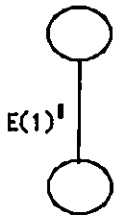
CT REPRESENTATION	CTM REPRESENTATION	MAPPING
NIL	e=(P,-,-)	A nil event in CT is represented in CTM as a null event.
 CHOICE	i) Let $t1=ax_1,\ldots,x_n\ldots$ $t2=bx_1,\ldots,x_n\ldots$ Then,  Assuming nil predicates.	i) If choice in CT is between two or more different binding events, then create a deterministic vertex in CTM with emanating edges labelled: bound-variable/-.
	ii) Let t1=-tao-... $t2=bx_1,\ldots,x_n\ldots$ Then,  Assuming nil predicates	ii) If choice in CT is between internal events, identical binding events, or qualifying events, create a nondeterministic vertex in CTM with emanating edges labelled appropriately.
	iii) Let t1=[P1]-tao- $t2=bx_1,\ldots,x_n$ Then, 	iii) If possible nondeterministic choice depends on the truth value of a predicate, create a grey vertex with appropriate label set.

Figure 3.9: Mapping CT to CTM

BINDING EVENT 		Represent the binding event as an edge with label: bound_variable/-
QUALIFYING EVENT 	 Assuming nil predicates	As part of a choice, a qualifying event causes the choice vertex to be nondeterministic in CTM. The choice edge is labelled: -/qualified_variable
INTERNAL EVENT 	 Assuming nil predicates	As part of a choice, a -tao- event causes the choice vertex to be non-deterministic. The choice edge is labelled -/-.
COMPOSITION Let t1=ax1,...,xn. by1,...,yn... t2=$\overline{c}$v.dz1,...,zn... 		Parallel composition expressed using "|" in CT is linearized in CTM. Choice between 2 or more parallel events is represented as a deterministic vertex only if all choices specify unique inputs and there is no predicate influence. A nondeterministic vertex specifies the choice and edges are labelled appropriately if there is a null input or 2 or more identical input events without predicate influence. Otherwise, a grey vertex specifies the choice.

RESTRICTION  	 	Restricted labels in CT have their corresponding branches and subtrees pruned in CTM.
LINEAR SEQUENCE  	 	Represent a linear sequence as a singe edge between V(1) and V(N+1)

3.4 Tree Model Coverage of LOTOS

We wish to show that the compressed tree model(CTM) covers the Language of Temporal Ordering Specifications(LOTOS)[ISO9]. To aid the readers understanding, we develop the coverage in two steps:

1. In Section 3.4.3, we show a mapping from a subset of LOTOS called Basic LOTOS [ISO9] to CTM. This mapping will serve as a first approximation to LOTOS coverage (i.e., concepts developed for coverage of Basic LOTOS hold also for LOTOS). Each Basic LOTOS construct is mapped to a CTM construct.

2. In Section 3.4.4 we show a mapping from LOTOS to CTM. Each LOTOS construct is mapped to a CTM construct.

Section 3.4.1 gives a general description of LOTOS. Section 3.4.2 gives assumptions made in showing CTM coverage of (Basic)LOTOS. Section 3.4.5 gives an example mapping from a LOTOS specification to its corresponding CTM.

3.4.1 General Description of LOTOS

In LOTOS, distributed systems are described in terms of processes. A system is a process that may consist of any number of interacting subprocesses. These subprocesses may in turn be refined, and so on, such that a LOTOS specification is a hierarchy of processes. A process is described in terms of its ability to communicate with its environment via interactions. Processes are thought of as black boxes and can communicate with their environment only through what are called gates. Gates can be thought of as windows through which a process can perform interactions. The atomic form of interaction is an event: events specify synchronized communication that may exist between two processes that can both perform that event through a rendezvous at a particular gate.

The format of a process definition in LOTOS is:

process{process_identifier}{parameter_part}
{behavior_expression}
endproc

where:

- process_identifier is the name by which the process can be referenced.
- parameter_part gives:
  - the list of interaction points(i.e., gates).
  - a formal parameter list that can be used to parameterize the process.
  - a list of output parameter types(i.e., sorts) defining the **functionality** of the process.
- behavior_expression is the LOTOS expression defining the observable behavior of the process.

3.4.2 Assumptions

LOTOS specifications can be divided into two parts: a behavior description part and a data description part[Stee86]. We show a mapping for only the behavior description part of a LOTOS specification. The data description part, given in terms of ACT ONE[ACT 85] algebraic specifications, are left outside the scope of this thesis.

The functionality of a LOTOS process is defined to be the product of the domains of the values passed at successful termination of that process. A process definition is **well defined** if the functionality of the behavior expression of that process definition is equal to the functionality of the process identifier of that process definition. We assume that the process definitions of the LOTOS specification from which we derive a CTM obey the functionality constraints of **well defined** process definitions.

Composition of parallel behaviors can be defined in terms of three operators: "|||", "||", and "| $[a_1, \ldots, a_n]$ |". We assume that any composition of multiple parallel behaviors is given in terms of only one of these operators. Thus,

- $b_1 \,||\, b_2 \,||\, \ldots \,||\, b_n$ follows our assumption,
- $b_1 \,|||\, b_2 \,||\, \ldots \,||\, b_n$ does not follow our assumption.

For all mappings performed, we assume the existence of an environment which may influence process functionality through synchronizing on given events. We also assume that the environment is not able to influence functionality through

a multiple–rendezvous (i.e., only two processes are able to rendezvous at a given time).

3.4.3 Mapping Basic LOTOS to CTM

Basic LOTOS[ISO9] has the following features:

- The events for behavior expressions for all processes are limited to the gate names of that process as well as the event **exit**, the internal event **i**, and the event **stop**.
- Events are based on a finite alphabet of possible events.
- The behavior expression of a process can invoke other processes or recursively invoke its own process.
- There is no parameterization of processes.
- Process functionality is **not** defined in terms of output **sorts**(see [ISO9]).
- All other features given in Section 3.4.1 not in contradiction with those given above.

We treat the events of Basic LOTOS as we treated the labels of synchronization trees(ST's) of Section 3.3.2: each event is treated as an input. To show a mapping from Basic LOTOS to CTM, we map each Basic LOTOS construct to a CTM construct. Refer to Figure 3.10 on page 83 for the following:

Inaction

Inaction specified by the event **stop** or the internal event **i** is represented in CTM as a null event e=(P,–,–).

Action Prefix

The ";" construct produces a new behavior expression out of an existing one by prefixing it with an event name. The ";" implies sequentiality of events. Sequences of events are represented in CTM as abstractions into edges. Let $e_1; e_2; \ldots; e_n$ be a sequence of events in a Basic LOTOS specification. This sequence is represented in CTM as $e_1' = (v_1', v_{n+1}')$, where:

- v_1' is the initial vertex of e_1'

- v'_{n+1} is the terminal vertex of e'_1
- $e'_1 = e_1, e_2, \ldots, e_n$ where $e_i = (P_i, I_i, O_i)$ [5] stands for an event.

Choice

The "[]" operator specifies choice between behavior expressions. Choice is represented in CTM by deterministic, nondeterministic, and grey C–vertices. Let

$$B_1[]B_2[]B_3[]\ldots[]B_n$$

be the choice behaviors at a given point in a Basic LOTOS specification. This choice is represented in CTM as a C–vertex with a Choice_set=$(e_1, e_2, \ldots, e_n)$ where e_i is:

1. the first event of the choice behavior B_i, i=1,...,n, if B_i does not express choice of first event; or
2. the set of choice events of B_i, i=1,...,n, if B_i expresses choice of first event.

Apply the criteria of Section 2.2 to determine the type of the C–vertex.

Composition

There are three types of parallel composition of behavior which can be specified in Basic LOTOS. We consider each in turn.

1. The "|||" operator specifies the parallel composition of two or more Basic LOTOS behaviors such that there is arbitrary interleaving of events specified in these behaviors. Behavior specified using "|||" must be linearized in CTM. Let
   - c=$c_1, \ldots, c_k$ be the possible next events for parallel behaviors $P_1, \ldots, P_k$, each specified using "|||"

     (i.e., $P_1 \,|||\, P_2 \,|||\, \ldots \,|||\, P_k$),

     of a process P at a given point in a Basic LOTOS specification.
   - $p_1, \ldots, p_r$ be possible next events for the environment of process P.
   - i,j be indices in $1, \ldots, r$.
   - m,n,s be indices in $1, \ldots, k$.

[5] Handling of variable declarations (i.e., "?") and value declarations (i.e., "!") is discussed in Section 3.4.4.

(a) **If**

- $p_i = c_m$,
- $p_j = c_n$,
- $m \neq n$,

then create a C–vertex in CTM with outgoing edges:

- $e_1, \ldots, e_s$
- s = number of synchronizing sequences.

In this case, there is more than one possible sequence synchronizing with the environment and this is represented in CTM as a choice. Employ the criteria of Section 2.2 to determine the type of the C–vertex.

(b) **If**

- $p_i = c_m$,
- $\neg(p_j = c_n)$,
- $((i \neq j) \vee (m \neq n))$

then map this single choice into CTM as part of a sequence mapping. In this case, there is one sequence synchronizing with the environment.

(c) If $\neg(p_i = c_m)$, then create a terminal vertex in CTM. In this case, there are **no** sequences synchronizing with the environment.

2. The "||" operator specifies the parallel composition of two or more Basic LOTOS behaviors such that parallel behaviors **must** synchronize with respect to all events. Two behaviors synchronize with respect to an event if they both offer this event. Behavior specified using "||" must be linearized in CTM. **Let**

   - c=$c_1, \ldots, c_k$ be the possible next events for parallel behaviors $P_1, \ldots, P_k$, each specified using "||"

     (i.e., $P_1 \,||\, P_2 \,||\, \ldots \,||\, P_k$),

     at a given point in a Basic LOTOS specification.
   - m,n,q,r,s,t be indices in $1, \ldots, k$.

   (a) **If**

- $c_m = c_n$,
- m≠n; and
- $c_q = c_r$,
- $q \neq r$,
- $\neg((q = m \wedge r = n) \vee (q = n \wedge r = m))$,

then create a C–vertex in CTM with outgoing edges:

- $c_1, \ldots, c_s$;
- $c_t = c_m \vee c_t = c_q$,
- t=1, ..., s = number of unique synchronizing sequences

labelled $c_1/-, \ldots, c_s/-$ for all c_m, c_q as above. In this case, there is more than one possible sequence synchronizing between parallel processes and this is represented in CTM as a choice. Employ the criteria of Section 2.2 to determine the type of the C–vertex.

(b) **If**

- $c_m = c_n$
- $m \neq n$
- $\neg(c_q = c_r)$
- $\neg((q = m \wedge r = n) \vee (q = n \wedge r = m))$

then map this single choice into CTM as part of a sequence mapping. In this case, there is only one possible sequence synchronizing between parallel processes.

(c) **If**

- $\neg(c_m = c_n)$,
- $m \neq n$

then create a terminal vertex in CTM. In this case, there are no sequences synchronizing between parallel processes.

Thus, all possible synchronizing sequences are represented in CTM. Only synchronizing behavior sequences specified in this type of parallel composition are mapped to CTM: an explicit pruning occurs since non–synchronizing event sequences are not mapped.

3. The | $[a_1, \ldots, a_n]$ | operator specifies the parallel composition of two or more Basic LOTOS behaviors such that parallel behaviors **must** synchronize with respect to events $a_1, \ldots, a_n$: call these **dependent events**. Synchronization on any other events between parallel behaviors is not necessary: call these **independent events**. Behavior specified using "| $[a_1, \ldots, a_n]$ |" must be linearized in CTM.

   Let

   - c=$c_1, \ldots, c_k$ be the possible next events for parallel behaviors $P_1, \ldots, P_k$, each specified using "| $[a_1, \ldots, a_n]$ |"

     (i.e., P_1 | $[a_1, \ldots, a_n]$ | P_2 | $[a_1, \ldots, a_n]$ | ... | $[a_1, \ldots, a_n]$ | P_k),

     at a given point in a Basic LOTOS specification,
   - $p_1, \ldots, p_r$ be possible next events offered by the environment.
   - a,b be indices in $1, \ldots, r$
   - i,j,m,n,q,s,t be indices in $1, \ldots, k$.

   (a) If

   - $c_i, c_j, i \neq$j, are independent events,
   - $c_i = p_a$,
   - $c_j = p_b$,

   then create a C–vertex in CTM with outgoing edges:

   - $c_1, \ldots, c_w \bigcup c_1, \ldots, c_t$
   - w = number of unique independent events synchronizing with the environment
   - t = number of unique dependent events synchronizing with one another:
     - $c_m = c_n$
     - c_m, c_n are dependent events
     - $c_q = c_s$
     - c_q, c_s are dependent events
     - $\neg((c_m = c_q \wedge c_n = c_s) \vee (c_m = c_s \wedge c_n = c_q))$

In this case, at least two independent sequences are synchronizing with the environment in the parallel composition. Employ the criteria of Section 2.2 to determine the type of the C–vertex.

(b) If

- $c_m = c_n$,
- c_m, c_n are dependent events; and
- $c_q = c_s$,
- c_q, c_s are dependent events; and
- $\neg((c_m = c_q \bigwedge c_n = c_s) \bigvee (c_m = c_s \bigwedge c_n = c_q))$

then we create a C–vertex in CTM with outgoing edges:

- $c_1, \ldots, c_w \bigcup c_1, \ldots, c_t$
- w = number of unique independent events synchronizing with the environment.
- t = number of unique dependent events synchronizing with one another:
  - $c_m = c_n$,
  - $\neg((c_m = c_q \bigwedge c_n = c_s) \bigvee (c_m = c_s \bigwedge c_n = c_q))$

In this case, at least two unique sequences are synchronizing with one another in the parallel composition. Employ the criteria of Section 2.2 to determine the type of the C–vertex.

(c) If

- $\neg(p_i = c_m)$, c_m an independent event,
- $\neg(c_q = c_s)$, c_q, c_s dependent events,

then create a terminal vertex in CTM. In this case, there are no synchronizing sequences either between processes or between a process and the environment.

Hiding

The "hide $a_1, \ldots, a_n$ in B" construct specifies behavior in which events $a_1, \ldots, a_n$ are not observable by an observer external to behavior B. The hide construct is used in the context of a parallel composition. When the events of the parallel

composition are linearized, which is necessary when mapping to CTM, those events specified to be hidden are represented in CTM choice and sequence mappings by the event e=(P,–,–). Such events can be either dependent or independent. Independent events are effectively forced to synchronize within the parallel composition since they can not synchronize with the environment (i.e., they are hidden). Since the hidden event is internal to behavior B, no input or output is observable external to B in terms of this event. CTM provides the mechanism (i.e., type E_4 edges) through which such internal events are specified.

Enabling

The "B1≫B2" construct specifies sequential composition of behavior in which process B2 may begin its functioning upon the successful termination of process B1. B1 is defined to terminate successfully if it executes the event **exit**. CTM uses choice and sequence mappings to specify B1. Subsequent to this, choice and sequence mappings are performed to map B2 to CTM: each terminal vertex of the mapping for B1 reached via the event **exit** has appended to it the CTM mapping for B2. The event **exit** itself is mapped to CTM as the event e=(P,exit,-) appended to each terminal vertex of the subtree representing B_1. Thus, **exit** is treated as an input.

Disabling

The "B1[>B2" construct specifies behavior in which process B2 may at any time interrupt the functioning of B1. If this occurs, B2 begins its functioning. We treat the disable construct just as we would a parallel construct. Each event in B1 has as a choice the root of the CTM for B2 (i.e., the entire CTM mapping of B2 is appended as a choice for all events in B1). Employ the criteria of Section 2.2 to determine the type of the resulting c–vertices. If B1 terminates successfully (i.e., reaches a terminal vertex in its mapping), then subsequent functioning is specified by other Basic LOTOS constructs and B2 is no longer specified as a choice to every event. There is no compression of linear sequences of B1 since each event of B1 has an alternative. Thus, B1 is composed entirely of C–vertices and edges representing single events.

3.4.4 Mapping LOTOS to CTM

LOTOS has the following features in addition to those existing in BASIC LOTOS:

- The ability to specify value communication in terms of structured interactions. Events in LOTOS are specified in terms of value (output in CTM) declarations and variable (input in CTM) declarations.
- The ability to specify behavior that depends upon conditions on values (i.e., guards).
- Parameterization of processes.
- Process functionality defined in terms of output sorts.
- Generalization of choice constructs in terms of values of variables and gate identifiers.
- Generalization of parallel constructs in terms of values of gate identifiers.

To show a mapping from LOTOS to CTM, we map each LOTOS construct to a CTM construct. Refer to Figure 3.11 on page 86 for the following:

Inaction

The mapping of inaction in LOTOS is identical to the mapping of inaction for Basic LOTOS.

Action Prefix

The mapping of action prefix in LOTOS is identical to the mapping of action prefix for Basic LOTOS.

Structured Event Offers

A structured event offer consists of a label (i.e., gate name) identifying the point of interaction and a finite, non-empty list of attributes. These attributes are either value (output in CTM) or variable (input in CTM) declarations:

- **value declarations** are of the form:

  g!v

  where g is the gate name and v is the value offered.
- **variable declarations** are of the form:

  g?v:sort

  where g is the gate name and v is the variable specified of a given sort.

Let

g?input_list!output_list

be a structured event offer in a process A where:

- **?input_list** is a finite list of variable declarations,
- **!output_list** is a finite list of value declarations,
- there is any arbitrary interleaving of value and variable declarations.

We first map such structured events for the case where A **is not** used in a parallel operation. In this case, the structured event offer is mapped to an event e=(P,I,O) where:

- I = ($\bigcup_1^n x_i \mid x_i \in$ input_list) where x_i is the variable specified for the ith variable declaration and n is the number of variable declarations in the structured event, i=1,...,n.
- O = ($\bigcup_1^r y_i \mid y_i \in$ output_list) where y_i is the value offered for the ith value declaration and r is the number of value declarations in the structured event, i=1,...,r.

We now map such structured events for the case where A **is** used in a parallel composition. Without loss of generality, assume that process A is in parallel composition with process B. In this case, the structured event offers in both processes are mapped to CTM events e=(P,I,O) for each possible synchronizing pair of parallel events (see the mapping for composition to determine the possible synchronizing pairs). The mapping of the members of input_list and output_list to the I and O of e=(P,I,O) is dependent upon the type of synchronization that occurs for each variable and value declaration. The mapping is as follows:

1. For each value declaration **g!E1** in process **A**, and each matching value declaration **g!E2** in a process **B**, where E1=E2, we construct an e with an output **A.E1** and an output **B.E2**, i.e., e=(P,–,(A.E1,B.E2)).
2. For each value declaration **g!E** in process **A**, and each matching variable declaration **g?x:sort** in a process **B**, where sort(E)=sort, we construct an e with an input **B.x** and an output **A.E**, i.e., e=(P,B.x,A.E).

3. For each variable declaration **g?x:sort1** in process **A** and each matching variable delcaration **g?y:sort2** in a process **B**, where sort1=sort2, we construct an e with an input **A.x** and an input **B.y**, i.e., e=(P,(A.x,B.y),–).

We note that each I and O in e is prefixed with the process identifier to which it pertains. This will become important for testing in order to identify the process to which the input or output applies.

Generalized Choice

The "[]" operator is generalized such that it is possible to specify alternative behavior between any number of behavior expressions. There are two forms:

1. For summation on gates,

   choice g in $[g_1, \ldots, g_n]$ []A

   specifies that behavior A is parameterized in terms of gate g. Thus,

   $$A[g_1][]A[g_2][] \ldots []A[g_n]$$

   is an equivalent behavior expression. In CTM, a summation on gates is mapped to a C–vertex with the CTM mapping for behavior specified in A instantiated in terms of choice and sequence mappings for each variant gate value $(g_1, \ldots, g_n)$. Such enumeration of choice behaviors is necessary in CTM since each choice can be statically differentiated from information in S (i.e., the LOTOS specification). The summation on gates is simply a shorthand notation for specifying the range of **static** choices. Apply the criteria of Section 2.2 to determine the type of the C–vertex.

2. For summation on values,

   choice x:t [] B

   specifies that behavior B is parameterized in terms of the value of x, which is of sort t. Such parameterization of subsequent behavior depend on context information and, hence, is **not** statically modelled in CTM. Context information is assumed to flow from the root of CTM. B is itself mapped to CTM.

Generalized Parallelism

The parallel operators "|||", "||", and "| $[a_1, \ldots, a_n]$ |" can be used to compose behavior expressions parameterized in terms of gate names. Thus,

$$\text{par g in } [g_1, \ldots, g_n] \text{ par\_op A}$$

where

- par_op is one of "|||", "||", or "| $[a_1, \ldots, a_n]$ |"
- A is a behavior expression

is equivalent to its expansion:

$$\text{A}[g_1] \text{ par\_op A}[g_2] \text{ par\_op } \ldots \text{ par\_op A}[g_n].$$

In order to map generalized parallelism to CTM, we first expand the generalized parallel composition to its equivalent (non-parameterized) form. From this form, we can map the composition to choice and sequence mappings as explained in the composition section.

Composition

In LOTOS, the mapping of the parallel constructs is identical to the mapping for Basic LOTOS with the exception that synchronization is now defined in terms of structured event offers instead of a finite alphabet of events. Thus, to define a mapping of composition in LOTOS to CTM, we need only to:

1. Define when two structured event offers are able to synchronize.
2. Apply the mapping for composition in Basic LOTOS which was presented in Section 3.4.3
3. Apply the mapping for structured event offers to events of CTM which was presented in Section 3.4.4.

To perform the mapping, we define the following:

Relative Position of an attribute of a structured event offer is the number of attributes between it and the gate identifier of the structured event plus one.

Matching Occurence between two attributes at the same **relative position** of two structured events to be either:

1. Two variable declarations with the same sorts.
2. Two value declarations with the same values.
3. A value declaration whose value is a member of the sort of the corresponding variable declaration.

Using the above definitions, we can enumerate the criteria for synchronization of structured event offers. Two structured event offers can synchronize if:

1. Each has the same number of attributes.
2. Each attribute in one event offer has a matching attribute in the other event offer.

Hiding

The mapping of hiding in LOTOS is identical to the mapping of hiding for Basic LOTOS.

Guarded Expressions

Any behavior expression may be **preceded** by a predicate. Such a behavior expression is said to be guarded. If the predicate holds, the behavior described by the behavior expression is possible, otherwise, it is not possible. Each predicate of a behavior expression is mapped to either:

1. P of the event e=(P,I,O); where e is the CTM mapping of the first event of the guarded behavior expression and the behavior expression expresses no choice of first event.
2. P_i of each event $e_i = (P_i, I_i, O_i)$, where e_i is the CTM mapping of the first event of each alternative expressed in the behavior expression; i=1, . . . ,number of alternatives.

Let

$[x > 5]\rightarrow$sap!x;. . .

denote a LOTOS guarded behavior expression. We map this to the event:

e=(P,I,O)=([$x > 5$],–,sap.x).

Disabling

The mapping of disabling in LOTOS is identical to the mapping of disabling for Basic LOTOS.

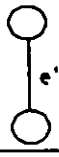
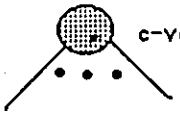
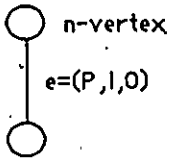
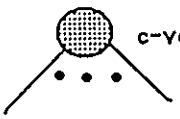
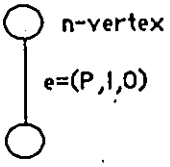
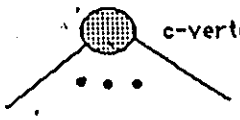
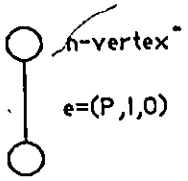
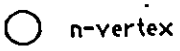
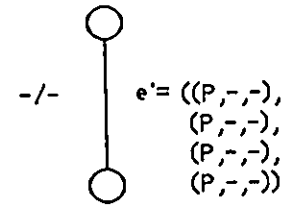
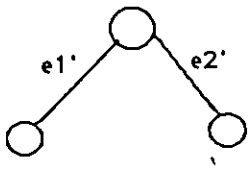
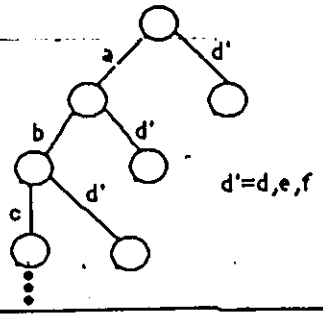
BASIC LOTOS BEHAVIOR	CTM REPRESENTATION	MAPPING
INACTION: stop internal event i	e=(P,-,-)	Inaction is represented in CTM by a null event.
ACTION PREFIX: a;b;c		Sequences of events are abstracted into edges.
CHOICE: A[]B[]C	 choice_set = (A,B,C)	A choice is represented in CTM as a deterministic, nondeterministic, or grey vertex.
Composition 1. P1|||P2|||...|||Pk	i)  ii)  iii)	i) If two or more events can synchronize with the environment, then represent this as a choice. ii) If only one sequence can synchronize with the environment, represent this as a single event. iii) If no sequences can synchronize with the environment, represent this with a terminal vertex. since further functioning does not occur.
2. P1||P2||...Pn	i)  ii) 	i) If two or more sequences in the parallel composition synchronize with one another, represent this as a choice. ii) If only one sequence in the parallel composition synchronizes, represent this as a single sequence.

Figure 3.10: Mapping Basic LOTOS to CTM

BASIC LOTOS BEHAVIOR	CTM REPRESENTATION	MAPPING
	iii) 	iii) If no sequences can synchronize, represent this with a terminal vertex since further functioning does not occur.
3. P1|a1,...,an| P2|a1,...,an| . . Pk|a1,...,an| —	i)  ii)  iii) 	i) If at least two dependent sequences or at least two independent sequences are able to synchronize, represent this as a choice. ii) If only one set of dependent sequences or independent sequences are able to synchronize, represent this as a single event. iii) If no sequences are able to synchronize, represent this with a terminal vertex since further functioning does not occur.
HIDING hide a,b,c in B where B = a;b;c;d;stop|| a;b;c;d;stop		Hidden events are mapped to E4 edges in CTM. The hidden events become e=(P,-,-).
Enabling B1>>B2 where: B1 := a;b;c ;exit []d;e;f;exit B2 := g;h;i ;stop	 e1'=abcghi e2'=defghi	Append to the terminal vertices of the mapping for B1 the CTM mapping for B2.

BASIC LOTOS BEHAVIOR	CTM REPRESENTATION	MAPPING
DISABLING B1[>B2 where: B1=a;b;c;stop B2=d;e;f;exit		Append as choice to every event in B1 the CTM for B2. If B1 terminates successfully via an exit, then subsequent mapping is performed and B2 is no longer appended as a choice.

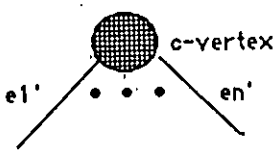
LOTOS BEHAVIOR	CTM REPRESENTATION	MAPPING
STRUCTURED EVENTS i) Matching Value Declarations: g!E1 g!E2 E1=E2	i) e=(P,-,(A.E1,B.E2))	i) For matching value declarations in a process A and a process B, we append outputs A.E1 and A.E2 to the output list of the event for each such matching attribute pair.
ii) Matching Value and Variable Declaration: g!E g?x:sort	ii) e=(P,B.x,A.E)	ii) For a matching value declaration and variable declaration in processes A and B respectively, we append an output A.E and an input B.x to the event for each such matching attribute pair.
iii) Matching Variable Declarations g?x:sort g?y:sort	iii) e=(P,(A.x,B.y)-)	iii) For matching variable declarations in processes A and B, we append inputs A.x and B.y for each such matching attribute pair.
SUMMATION ON GATES choice g in [g1,...gn] [] A		Behavior parameterized in terms of gate values is enumerated in CTM as explicit choice of each behavior mapped with instantiated gate values.

Figure 3.11: Mapping LOTOS to CTM

LOTOS BEHAVIOR	CTM REPRESENTATION	MAPPING
GENERALIZED PARALLELISM par g in [g1,...,gn] par_op A	A[g1] par_op A[g2] par_op ... A[gn]	We abstract the generalized parallelism construct before mapping to CTM. Use choice and sequence mappings in CTM as for composition.
COMPOSITION Let E1=g?var_set !val_set E2=g?var_set !val_set	i) If var_set and val_set of E1 and E2 are identical, the events can synchronize. ii) Map synchronizing events as for composition in Basic LOTOS. iii) Map structured interactions as defined previously.	
GUARDED EXPRESSIONS [guard] -> B	e=([guard],(mapping for B))	Guards to behavior expressions are mapped as predicates to the first event(s) of the guarded behavior expression B.

3.4.5 Example Mapping

We give an example mapping from a LOTOS specification of a X.25 multiplexer/demultiplexer. The specification is given in Figure 3.12. From the specification, we note the following:

1. Functioning of the multiplexer/demultiplexer is defined in terms of the parallel composition of a multiplexer process and a demultiplexer process.
2. The specification does not contain any EXIT or STOP operators. Thus, infinite functioning is specified.
3. Functioning of the process is defined for the packet level[P] and link level[L] gates.
4. **lcn** is the logical channel number on which data are received.
5. **ud** is the user data received.
6. **Data_pdu** is the protocol data unit into which data is multiplexed or from which data is demultiplexed.

The corresponding CTM is given in Figure 3.13. In order to fit the CTM into the space provided, we use bold labels to denote continuation of the CTM: vertices with identical bold labels correspond to each other.

```
process X.25_Mux_Demux[P,L]:noexit :=
   X.25_Mux[P,L]|||X.25_Demux[P,L]
   where
   process X.25_Mux[P,L]:noexit :=
      P!Req?lcn:lcn_id_sort?ud:user_data_sort;
      L!Req!Data_pdu;
      X.25_Mux[P,L]
   endproc
   process X.25_Demux[P,L]:noexit :=
      [lcn ≠ null AND ud ≠ null] L!Ind!Data_pdu;
      P!ind!lcn!ud;
      X.25_Demux[P,L]
   endproc
endproc
```

Figure 3.12: Example Multiplexer/De-multiplexer

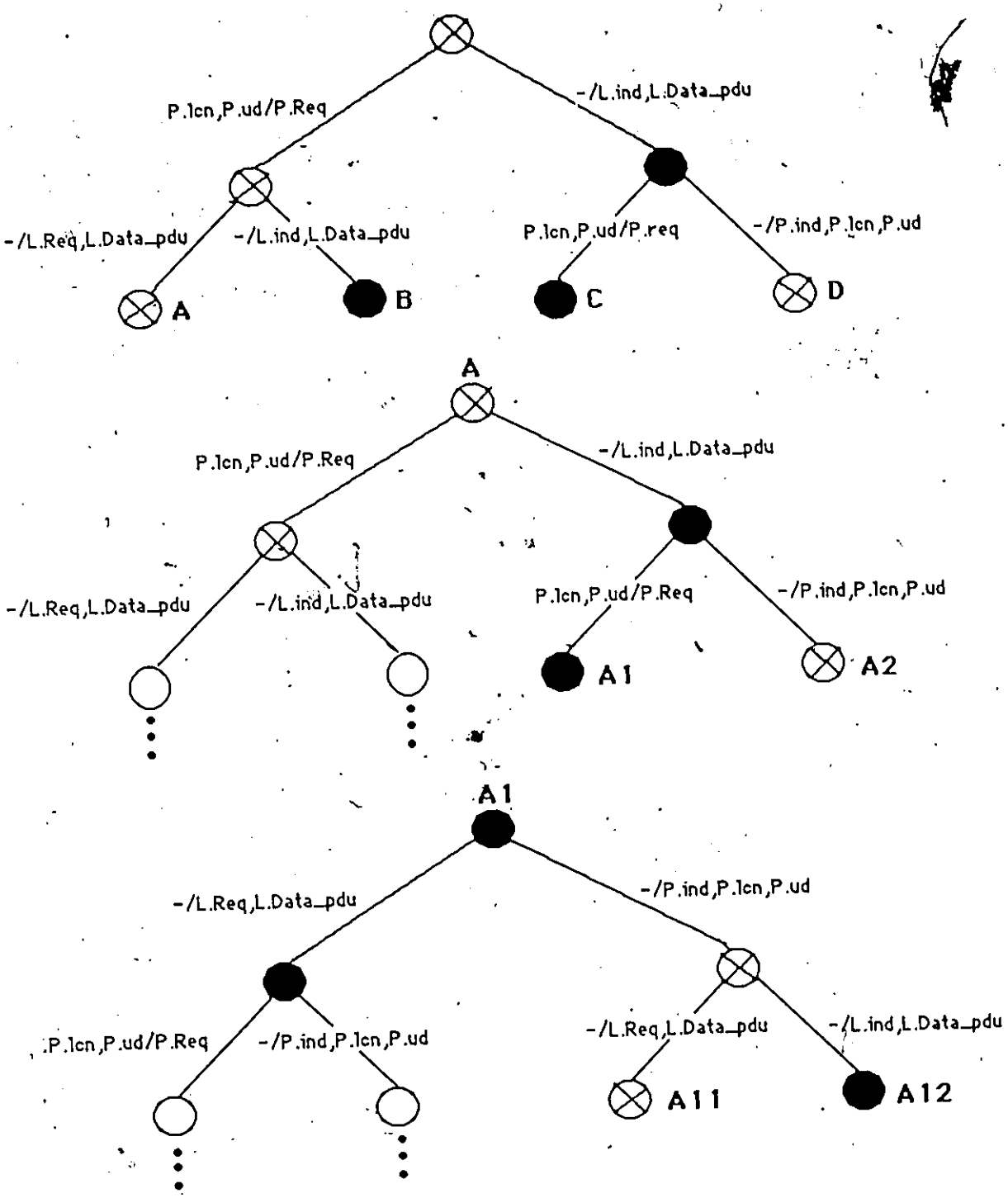
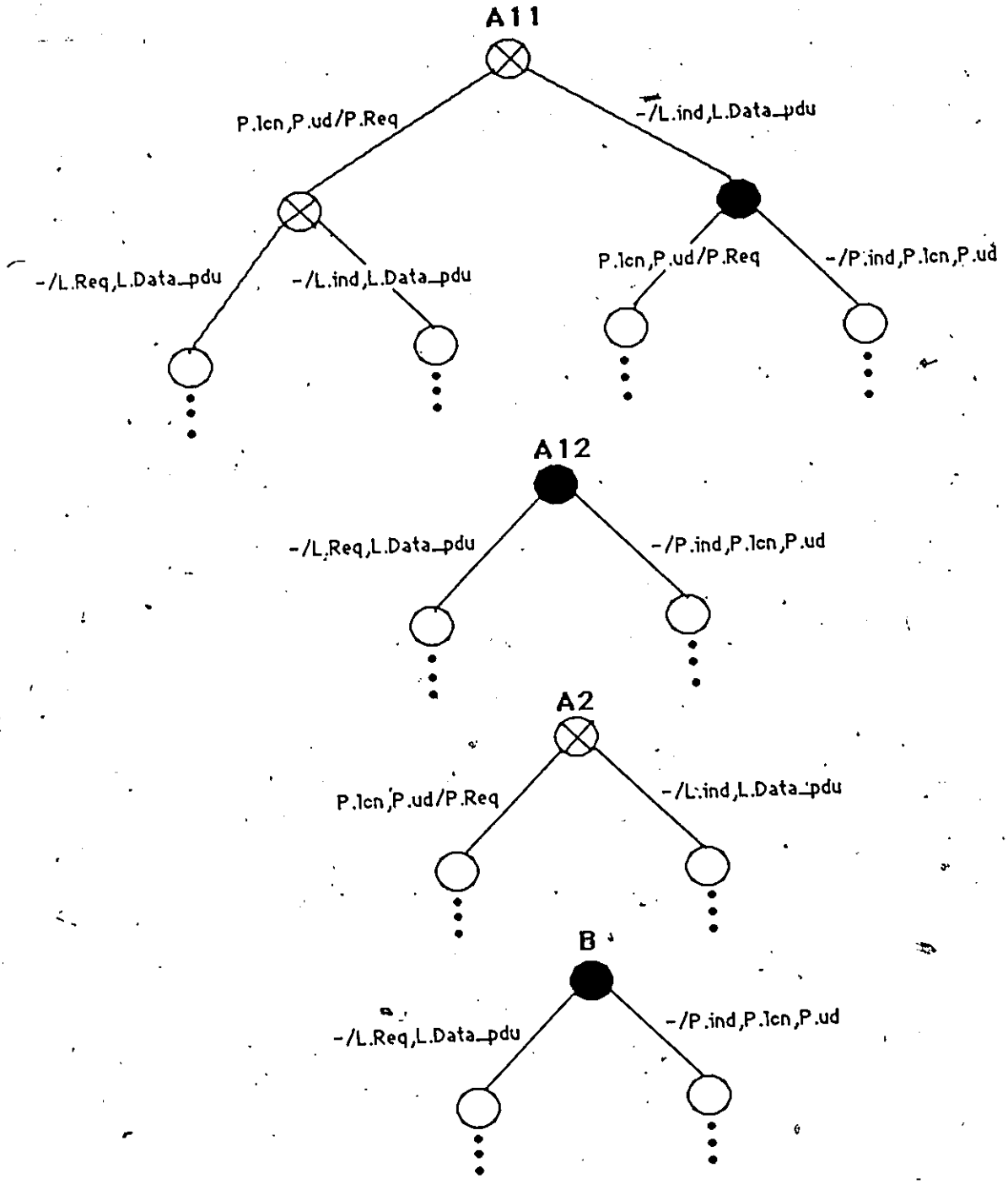
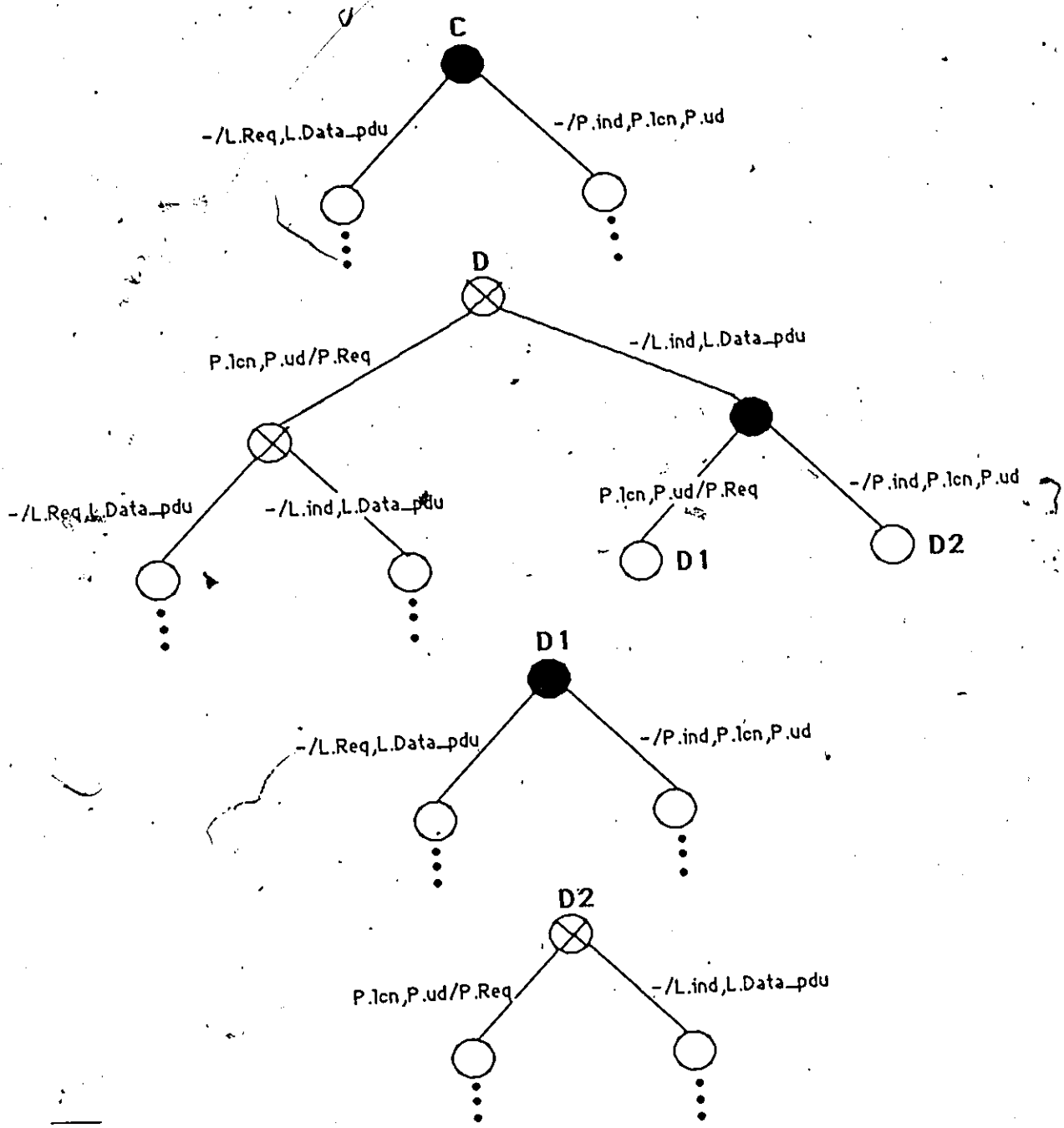


Figure 3.13: CTM for Multiplexer/De-multiplexer


A11
P.Icn,P.ud/P.Req
-/L.ind,L.Data_pdu
-/L.Req,L.Data_pdu
-/L.ind,L.Data_pdu
P.Icn,P.ud/P.Req
-/P.ind,P.Icn,P.ud
A12
-/L.Req,L.Data_pdu
-/P.ind,P.Icn,P.ud
A2
P.Icn,P.ud/P.Req
-/L.ind,L.Data_pdu
B
-/L.Req,L.Data_pdu
-/P.ind,P.Icn,P.ud


C
-/L.Req,L.Data_pdu
-/P.ind,P.lcn,P.ud
D
P.lcn,P.ud/P.Req
-/L.ind,L.Data_pdu
-/L.Req,L.Data_pdu
-/L.ind,L.Data_pdu
P.lcn,P.ud/P.Req
-/P.ind,P.lcn,P.ud
D1
D2
D1
-/L.Req,L.Data_pdu
-/P.ind,P.lcn,P.ud
D2
P.lcn,P.ud/P.Req
-/L.ind,L.Data_pdu

4 THE CONFORMANCE TESTING PROBLEM

In the previous chapters, we presented a framework, in terms of a tree model, which serves as a basis for characterization of those issues specific to the conformance testing problem. We also provided mappings from various formal description techniques through which to derive skeleton test sequences that will be employed for conformance testing. In this chapter, we first refine the concepts of a specification and its corresponding implementation. We then present heuristics to complete our previously obtained skeleton test sequences to obtain test sequences for testing valid and defensive behavior of an implementation. A characterization of the set of conforming implementations of a given specification in terms of a set of tests is then given, followed by suggestions of solution strategies for selectively applying a particular subset of tests to a given implementation.

4.1 Definition of S_i and I_i

Given S, consider the following:

- IMP = $\{I_1, I_2, \ldots, I_n\}$ is a set of conforming and nonconforming implementations ($I_i, 1 \leq i \leq n$) of S.
- S_i is the implementors view of S.
- I_i is an implementation conforming to S_i.

S_i has the following two properties:

1. S_i is deterministic since the implementation I_i has to resolve any nondeterminism in the original specification S.
2. S_i differs from S_j, where $i \neq j$, in the way it resolves at least one nondeterministic choice specified in S.

Thus, each implementation in IMP differs from all other implementations in this set by at least one nondeterministic choice it has resolved. To clarify, we define the following:

Invariant choice construct implementation is such that order of presentation of each choice $c_1, \ldots, c_n$ **can not** have influence on I_i in terms of externally observable sequences of events, $n \geq 2$.

Variant choice construct implementation is such that order of presentation of each choice $c_1, \ldots, c_n$ can have influence on I_i in terms of externally observable sequences of events, $n \geq 1$. (Note that an impementation can refuse to implement given alternative(s). Hence, the choice construct can be decomposed to a single choice (i.e., n=1)).

Then,

- Alternatives of deterministic choices originally specified in S are preserved in S_i and are **implemented invariantly** as choice constructs in I_i(eg;case, if, etc.).

- Alternatives of nondeterministic choices originally specified in S may or may not be preserved in S_i and are **implemented variantly** as choice constructs in I_i.

Thus, it is the nondeterministic choice(s) specified in S which are implemented variantly that cause each $I_i \in$IMP to differ in its view of S (i.e., S_i).

We note that when discussing each $I_i \in$IMP, there are many implementation issues which are outside the scope of the specification S. Each I_i can be thought of as a view of the corresponding specification(S_i) plus its handling of specific implementation concerns IC = $\{ic_1, \ldots, ic_n\}$ where $ic_j, 1 \leq j \leq n$, denotes a specific implementation concern and **n** is the number of implementation concerns for I_i. An **implementation concern** is any aspect of an implementation which is not addressed in S. For example, the method of segmentation and blocking, the set of encoding and decoding rules, and the method of buffer management of a given implementation I_i may all be concerns of an implementation which are not addressed in S. Therefore,

$$I_i = S_i \cup IC$$

where:

- ic_1 = method of segmentation and blocking
- ic_2 = set of encoding and decoding rules
- ic_3 = method of buffer management

- ic_4 = operating system interface
- etc.

Thus, when speaking of an implementation's conformance to S, although implementation concerns are of consequence it is assumed that solely resolution of nondeterministic choices specified in S (which are implemented variantly) differentiate each $I_i \in$IMP.

4.2 Conformance Testing Problem

In order to define the conformance testing problem, we first define the following:

Mandatory Capability of a Specification: Any capability specified in S which must be provided by any $I_i \in$ IMP.

Conditional Capability of a Specification: Any capability specified in S which must be provided by an $I_i \in$ IMP only when the conditions for inclusion of this capability as outlined in the corresponding standard apply.

Optional Capability of a Specification: Any capability specified in S which can be selectively provided in $I_i \in$ IMP so long as any requirements on which the option depends or which depend on the option as specified in the corresponding standard are observed.

Valid Derivation of a Specification: A specification S_i which expresses a subset of the functionality expressed by the original specification S is a valid derivation if it is composed of the following:

1. the functionality of all **mandatory capabilities** stated in S.
2. the functionality of all **conditional capabilities** stated in S only if the conditions to be met set out in the corresponding standard(s) apply.
3. the functionality of all **optional capabilities** stated in S only if conditions on which these options depend are observed.

Conforming Implementation of a Specification: An implementation which is shown to satisfy capabilities and options consistent with the capabilities and

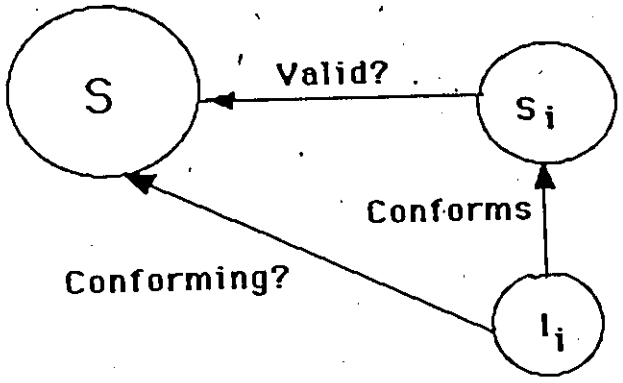


Figure 4.1: Conformance Testing Problem

options of the specification stated to be supported in the Implementation Conformance Statement(ICS)[Rayn87][6].

Now, we can define **conformance testing** as selecting a set of test sequences T_i which help to determine whether:

- S_i is a valid derivation of S or, alternatively;
- I_i is a conforming implementation of S. Here, we assume that I_i is the implementation conforming to S_i.

We note that T_i is deterministic in that it tests the behavior of the deterministic implementation, and nothing else. Figure 4.1 illustrates the conformance testing problem.

Our primary objective is to characterize a set of conforming implementations by a set of tests. However, we are, in general, not given the source code for these implementations. The only resource we have from which we derive test sequences is the given S (which is represented by CTM). Thus, we wish to use CTM to derive test sequences.

4.3 Possible Test Selection Strategies

Let a set of implementations of a given S be denoted by IMP. We characterize the set IMP' $\subseteq$IMP of implementations conforming to S by a set of tests T such that

[6]See Section 4.5.3 for discussion on ICS

any implementation I_i that passes tests in T is in IMP'. I_i passes tests in T if each test purpose[7] defined for T is fulfilled (i.e., the required testing for each test purpose is properly performed) and the observed behavior for I_i does not differ from its expected behavior. Since CTM represents possibly infinite interaction sequences, it must be made finite in order to obtain a finite number of finite length test sequences. Testing is by nature finite: we can not hope to conduct an infinitely long test because we will never get to the point of analyzing the test result. A finite test is composed of a finite number of finite test sequences. We thus define a testing tree (TT)[Chow78] to be a finite CTM of S. TT contains only those tests which are intended to test the reactions of each $I_i \in$IMP to valid input. We wish to augment TT to obtain tests for reactions of each I_i to erroneous input. The aim here is to achieve a reasonable level of confidence that I_i is robust[Myer79]. Therefore, we define T to be TT plus test sequences to determine reactions of implementations to erroneous input.

T has the following properties:

- T is an augmented TT which tests for valid and defensive behavior of any $I_i \in$ IMP.
- T characterizes the set of conforming implementations IMP' $\subseteq$IMP.
- T is nondeterministic because it specifies tests for all possible behaviors of our set of conforming implementations IMP', which, when these behaviors are taken as a whole, are nondeterministic.
- T is also nondeterministic in that only a subset of all sequences in T is applied to any I_i.
- T is an adaptive set of test sequences: it adapts itself to a given I_i to yield a subset of test sequences relevant to testing the conformance of I_i to S.
- T can be thought of as a generic[ISO7] or abstract[ISO7] set of test sequences for IMP[8]: T does not specify tests for any particuar $I_i \in$ IMP, it specifies tests for all $I_i \in$ IMP.

[7]See Section 4.5

[8]See Section 4.5 for discussion on generic and abstract test sequences.

- T is not directly obtainable from S because S may not, in general, specify defensive behavior.

We wish to obtain a deterministic set of test sequences specific to a given $I_i \in$ IMP. To do so, we define T_i to be a pruned T specific to a particular I_i. T_i is the set of test sequences that are adaptive for a given I_i because of existing grey choices in I_i: for T_i, grey choices resolved as deterministic choices taken by I_i are not known to be such by the tester and hence, an adaptive test method must be applied.

In an **adaptive test method**, the implementation being tested can exhibit any one of a variety of valid behaviors. These valid behaviors are the consequence of variant choice construct implementation(s) in the implementation. In testing an arbitrary implementation, we are unsure of which variants of choice construct implementation were chosen. For example, consider the simple transport entity example of Section 2.4.2. The specifier deliberately did not specify exactly which buffer (Buf_3 or Buf_5) must be checked first by an implementation. Thus, the choice between checking of Buf_3 or Buf_5 first is left to be resolved by the implementor as a variant choice construct implementation. Since checking Buf_3 first and checking Buf_5 first are both viable options, adaptation to responses of I_i is still necessary for the tester. In this case, two possible valid sequences of behavior (i.e., responses) are possible for the implementation. The tester must be capable of adapting itself to responses exhibited by the implementation by applying appropriate next stimuli to enable continuation of the test.

An adaptive test method employs **adaptive test sequences** which are in a tree form. Using the terminology developed for CTM, any member of the choice_set of a given vertex in a given adaptive test sequence is a viable alternative for application to the implementation. When a given alternative is applied to the implementation, the subtree reached through the edge denoting this alternative is now the remaining portion of the adaptive test sequence which can be applied to the implementation. Thus, an adaptive test sequence is able to respond to the various alternatives which can be taken by the implementation.

There are two ways of selecting adaptive test sequences to implementation apply to I_i, each of which requires some knowledge of how nondeterministic choices are resolved by I_i[9].

[9]This knowledge may be obtained from the Implementation Conformance Statement(ICS) which is submitted by the implementor of a particular implementation of S.

1. Apply T to I_i. This reduces to applying T_i because there is an implicit pruning of T when applied to I_i. This requires an on-line selection of test sequences (see Section 4.8).

2. Derive T_i from T and apply T_i to I_i. This requires an off-line selection of test sequences (see Section 4.9). A special case of this second alternative is to derive T_i from TT and apply T_i to I_i. This is also an off-line selection of test sequences. However, this method **does not** test for defensive behavior.

Figure 4.2 illustrates selective application of test sequences to I_i. These methods of selection correspond to possible solution strategies for the conformance testing problem. We are left with the question of which is better to apply that is addressed in Sections 4.8 and 4.9. Of course, we are also left with the question of how to derive T from TT and T_i from T. Section 4.4.3 gives an algorithm to obtain TT

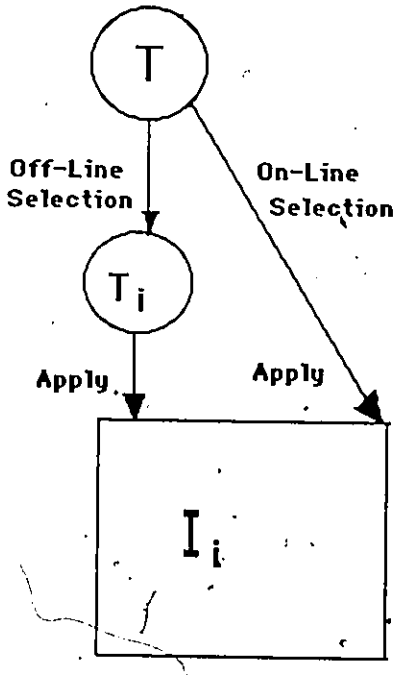


Figure 4.2: Application of Conformance Testing

from CTM. Section 4.5 gives a discussion on obtaining T from TT. Section 4.6 gives a discussion on obtaining T_i from T.

4.4 Obtaining TT from CTM

We wish to obtain a finite version of CTM which we call testing tree(TT). To do so, we must first define a manner in which to label the CTM derived from the FDT's considered in Section 3. We then use these labels to derive TT. Section 4.4.1 explains why labels are necessarry for deriving TT. Section 4.4.2 defines labellings for the various FDT's. Section 4.4.3 gives a simple algorithm to derive TT from a labelled CTM.

4.4.1 Need for Labelling CTM

CTM is composed of event sequences and choice of event sequences. During mapping system behavior expressed in FDT's to event sequences in CTM, no consideration was given to recognition of a previously encountered event sequence. Although S is, in general, able to specify infinite behavior, it is syntactically finite. Repetition of sequences of events in CTM should be recognized in order to derive the corresponding TT. In order to facilitate the recognition of repetitive sequences, we need to label CTM.

4.4.2 Labelling of CTM

We now define methods through which we label the CTM representing a FSM, ESTELLE, LOTOS, or CCS based specification such that these labels can be used by the algorithm of Section 4.4.3 to produce a testing tree. In defining the labels, we note that the labels for FSM's and ESTELLE are very similar, as are the labels for LOTOS and CCS. This is not surprising since these pairs of FDT's are based on common models (i.e., state based[Rev 83] and event based[Miln80], respectively).

Labelling of CTM For FSM

Since a FSM has a finite number of states, the only way in which we can specify infinite behavior is through a loop(i.e., a closed path). Thus, the key to labelling CTM in mapping from FSM's is to ensure that any information which enables us to detect a loop in the FSM is also mapped (via these labels) to CTM.

A loop in a FSM consists of a linear sequence of transitions $t_1, \ldots, t_{n-1}$ between a pair of states S_1 and S_n such that $S_1 = S_n$. It is clear that we detect a loop in the FSM upon encountering a previously visited state (i.e., S_1). Thus, the information needed to detect a loop is previously encountered state identifiers. In order to

map this necessary information to CTM, we label vertices in CTM with the state identifiers of the FSM whose transition definitions are mapped to events of CTM as explained in Section 3.1.1. These labels can then be used by the algorithm of Section 4.4.3 to obtain a testing tree.

We employ the following labelling while performing the mapping from FSM's to CTM defined in Section 3.1.1:

1. The initial state of the FSM mapped to CTM labels the root vertex of CTM.

2. Any other state in the FSM with choice of next transition corresponds to a C–vertex in CTM whose label is the state identifier that it stands for.

Labelling of CTM For ESTELLE

As with FSM's, since a NFS has a finite number of states, the only way in which we can specify infinite behavior is through a loop (i.e., closed path). In mapping from a NFS to CTM as explained in Section 3.2.3, an identical labelling procedure is used as for the mapping from FSM's to CTM.

Labelling of CTM for LOTOS

Since a LOTOS specification is syntactically finite [ISO9], it can only specify infinite behavior through instantiation (a call) of an existing process (i.e., a recursive call). Thus, it is necessary in CTM to convey information to recognize a recursive instantiation of a given process. A LOTOS specification does not contain explicit state names, it is composed of behavior subexpressions. It is apparent, though, that in order to produce TT from CTM, recognition of previously encountered behavior subexpressions in performing a mapping from LOTOS to CTM is necessary but not sufficient. Consider the following Basic LOTOS behavior expression B:

```
B :=
    a;P1
    []
    b;a;P1;c
```

where **P1** denotes any LOTOS behavior subexpression. Now, when B has been mapped to CTM, the first occurrence of P1 might be considered as re–occuring when the second occurrence of P1 is encountered in CTM (assuming CTM is traversed post–order). However, subsequent behavior to the second occurrence of P1 (i.e., c)

differs from subsequent behavior to the first occurrence of P1 (i.e., no subsequent behavior). The key issue is recognition of re-occuring behavior subexpressions which syntactically terminate in the LOTOS specification. For the example above, the first occurrence of the behavior expression containing P1 syntactically terminates with P1. However, the second occurrence of the behavior subexpression containing P1 **does not** terminate with P1. Hence, these two behavior subexpressions containing P1 (or a;P1) are not identical.

In order to recognize previously encountered behavior subexpressions, we:

1. augment the LOTOS specification with identifiers, and
2. define the current LOTOS behavior subexpression being mapped in terms of these identifiers.

In order to augment a LOTOS specification with identifiers, we first define what a unique occurrence of a LOTOS behavior subexpression is at a certain point while the LOTOS specification is mapped into a CTM.

A LOTOS behavior subexpression is **unique** if either:

1. There is no point previous in the LOTOS specification at which this behavior subexpression occurs. For example, let **B=a;P1;c** be a LOTOS behavior subexpression. If the only previous behavior specified in the LOTOS specification is e;f;g[] (i.e., the LOTOS behavior expression to this point is e;f;g []B), then B is unique since there is no previous identical behavior subexpression specified.
2. There is a point previous in the LOTOS specification at which this behavior subexpression occurs but subsequent behavior to the LOTOS behavior subexpression in question is specified. For example, let:

   $$B'$$=a;P1 []

   be a LOTOS behavior subexpression previous to that specified by B as defined above. Since B has subsequent behavior which differs from B′ (i.e., c), B is unique.

The problem of recognizing an identical previous behavior subexpression is a problem concerned with parsing of the LOTOS specification. Also, note that we do not check equivalence of LOTOS behavior subexpressions [10].

Now, only the first occurrence of a unique LOTOS behavior subexpression is augmented with a **unique set of identifiers** (i.e., the process invocation and LOTOS operator identifiers in this behavior). All subsequent occurrences of this behavior subexpression are augmented with the same set of identifiers given to the first occurrence.

The LOTOS specification is augmented with identifiers by the following augmentation rules:

1. After each **choice operator**, insert an identifier $\{C_i\}$, $1 \leq i \leq$ number of choice operators in the LOTOS specification.

2. After each **action prefix operator** which is immediately followed by an action or EXIT or STOP, insert an identifier $\{A_i\}$, $1 \leq i \leq$ number of such action prefix operators in the LOTOS specification.

3. After each **parallel composition operator** or **disable operator**, insert an identifier $\{P_i\}$, $1 \leq i \leq$ (number of parallel operators + number of disable operators) in the LOTOS specification.

4. After each **process invocation** insert an identifier $\{I_i\}$, $1 \leq i \leq$ number of process invocations in the LOTOS specification.

We note that there is no need to insert identifiers after the sequential composition operator $\gg$. This is due to the fact that the arguments of this in–fix operator are process identifiers and these process identifiers are themselves given identifiers (I_i).

Definition: An **id_set** is a subset of the identifiers used to augment a LOTOS specification.

The vertices of CTM are labelled with the members of an **id_set** which is initially empty. As the LOTOS specification is mapped to CTM, the current id_set becomes the label of the current CTM vertex being mapped. The **current id_set** is maintained as follows:

[10] This is a topic for future research and may eventually be incorporated into the University of Ottawa's LOTOS interpreter.

1: Each identifier has a corresponding index whose initial value is zero. Let index(id) denote the current index value of identifier id. An identifier's index is incremented each time its corresponding behavior or operator becomes active and is decremented each time its corresponding behavior or operator becomes inactive. A given behavior or operator becomes inactive when it no longer meets the criteria for being active as given in the following points 2, 3, 4, and 5.

2. A choice operator is **active** if we reach a point in the LOTOS specification at which the choice operator is present but we have not yet taken one of the choices offered. For example, let

$$\text{a[]}\{C_3\}\text{b}$$

denote the current behavior subexpression in the LOTOS specification which is being mapped to CTM. Then, the choice is active and index($\{C_3\}$) = index($\{C_3\}$) + 1. If alternative a or b is chosen, then index($\{C_3\}$) = index($\{C_3\}$) - 1.

3. An action prefix operator is **active** if the action immediately preceding the corresponding action prefix operator has just been performed. For example, let

$$\text{a;}\{A_{12}\}\text{b;}\{A_{13}\}\text{;c}$$

denote the current behavior subexpression in a LOTOS specification at which action **a** has just been performed. Then, the first action prefix operator is active and index($\{A_{12}\}$) = index($\{A_{12}\}$) + 1. If behavior **b** is performed, then index($\{A_{12}\}$) = index($\{A_{12}\}$) - 1, index($\{A_{13}\}$) = index($\{A_{13}\}$) + 1.

4. A parallel operator is **active** if behavior specified by the parallel operator has begun its functioning and neither of its component processes specified to be running in parallel has terminated its functioning through reaching either a **STOP** or an **EXIT** operator. For example, let

$$\text{P1}\{I_6\}|||\{P_8\}\text{P4}\{I_7\}$$

denote the current behavior subexpression in a LOTOS specification in which behavior P1 and P4 are running in parallel. Consider only identifier $\{P_8\}$. Index($\{P_8\}$) = Index($\{P_8\}$) + 1 when the parallel operator is reached. If process P1 or process P2 reaches an EXIT or STOP, then index($\{P_8\}$) = index($\{P_8\}$) - 1.

5. A process invocation behavior is **active** if we have reached a behavior subexpression in the LOTOS specification at which this process has been offered but has not yet begun its functioning. For example, let

   a;b;P1$\{I_7\}$

   denote a behavior subexpression in a LOTOS specification at which process P1 has just been offered. Then, index($\{I_7\}$) = index($\{I_7\}$) + 1. If P1 begins its functioning, then index($\{I_7\}$) = index($\{I_7\}$) - 1.

6. An identifier is a member of the current id_set if its corresponding index is non-zero. For example, if index($\{I_7\}$) = 1, index($\{P_1\}$) = 2, and all other identifier indices are zero, then id_set = $\{I_7, P_1\}$.

By labelling CTM in such a manner, we implicitly associate states and state transitions with the LOTOS specification: the current id_set is the state, the possible next events are the transitions, and subsequent id_sets are the possible next states. We can thus identify in CTM our current state and, by applying the algorithm of Section 4.4.3, obtain TT from CTM.

Example LOTOS Labelling

In order to exemplify labelling of CTM during its derivation, we give a stepwise derivation of a CTM for the Basic LOTOS specification given in Figure 4.3. This Basic LOTOS specification has been augmented with identifiers. We note in particular that the second occurrence of the behavior

P1||P2;stop (i.e., in line 8)

is given the same identifiers ($\{I_4\}$,$\{P_1\}$,$\{I_5\}$) as its first occurrence (i.e., in line 6). This is in accordance with the augmentation rules given previously. As well, the second occurrence of P1 (i.e., in line 6) is given a different identifier ($\{I_3\}$) than the

```
1    Process B[a,b,c,e] :=
2        (a;P1{I1};{A1}  stop
3            []{C1}
4        b;{A2}c;{A3} ;e;(P2{I2};{A1}  stop
5                            []{C2}
6                            a;P1{I3}≫P1{I4}||{P1}P2{I5};{A1} stop)
7            []{C3}
8        c;P1{I4} ||{P1} P2{I5} ;{A1} stop)
9    where
10       process P1[b,e]
11           (e
12           []{C4}
13           b); P2{I6}
14       endproc
15       process P2[a]
16           a;{A4}  exit
17       endproc
18   endproc
```

Figure 4.3: Example Augmented LOTOS Specification

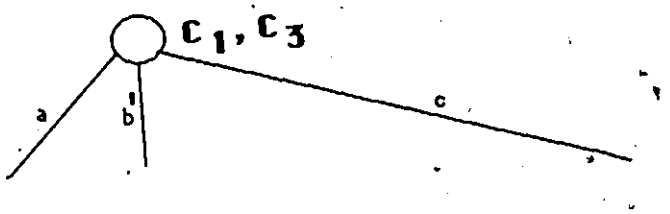


Figure 4.4: Behavior B is invoked

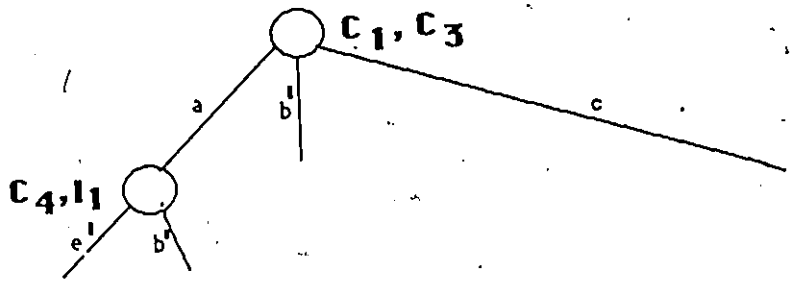


Figure 4.5: Alternative a is Chosen

first occurrence ($\{I_1\}$) of P1 (i.e., in line 2) since subsequent behaviors of these two occurrences differ (i.e., ;stop versus P1 ||P2 ;stop).

When process B is invoked, choice is offered between events a, b, and c. The active choice operators are identified $\{C_1\}$ and $\{C_3\}$. Figure 4.4 illustrates CTM to this point.

Now, if alternative a is chosen, $\{C_1\}$ and $\{C_3\}$ become inactive. The process invocation identifier $\{I_1\}$ becomes active, as does the first choice offered in process P1, $\{C_4\}$. Figure 4.5 illustrates CTM to this point.

If either alternative e or alternative b from the vertex labelled C_4, I_1 is chosen, behavior specified by P2 with identifier $\{I_6\}$ becomes active. This behavior reaches an exit through a linear sequence (e'=e;a, b'=b;a) which, in turn, reaches a stop after returning from the original call to P1. Hence, terminal vertices are reached. Figure 4.6 illustrates CTM to this point.

Returning now to the original choice specified at the root vertex of CTM, if alternative b is chosen, $\{C_1\}$ and $\{C_3\}$ become inactive. A linear sequence of events (which is abstracted as b' = b;c;e) takes us to the next choice offered ($\{C_2\}$) which becomes active along with the process invocation identifier for process P2, $\{I_2\}$. Figure 4.7 illustrates CTM to this point.

Note that there is a nondeterministic choice offered at ($\{C_2\}$, $\{I_2\}$). If we take the first of these (which is an invocation of P2$\{I_2\}$), a linear sequence (of the

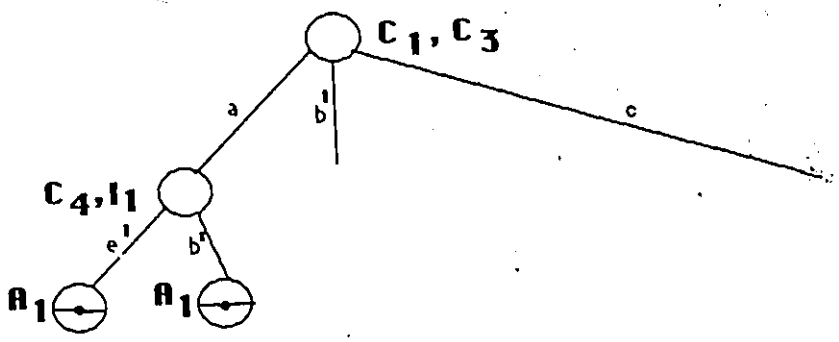


Figure 4.6: Alternative e or b is Chosen

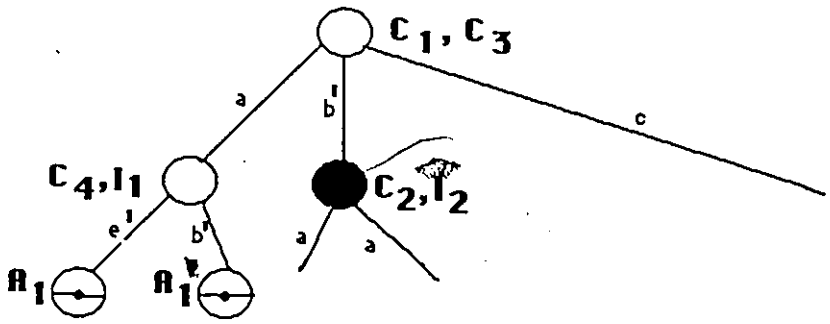


Figure 4.7: Alternative b is Chosen

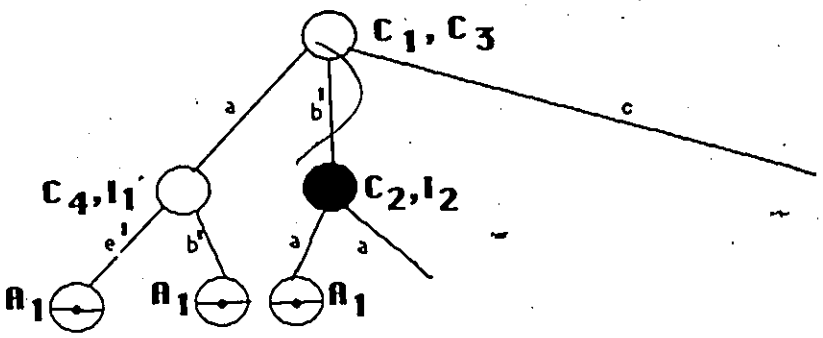


Figure 4.8: First Alternative a is Chosen

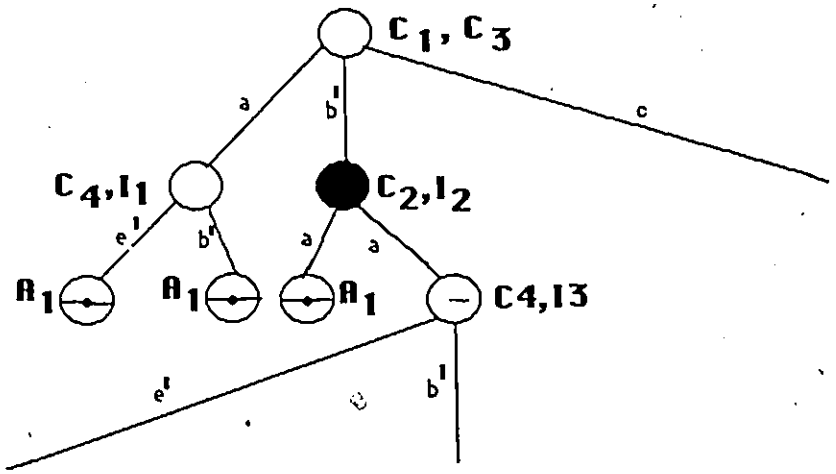


Figure 4.9: Second Alternative a is Chosen

single event a) brings us to a stop action and hence, a terminal vertex. Figure 4.8 illustrates CTM to this point.

If the second alternative offered at $\{C_2\}$, $\{I_2\}$ is taken (which is also an event a), then an invocation to P1$\{I_3\}$ with its choice $\{C_4\}$ is performed. Figure 4.9 illustrates CTM to this point.

If either alternative offered at the vertex now labelled C_4, I_3 is taken, this brings us through a linear sequence (e' = e;a, b' = b;a) to a point where parallel behaviors P1 and P2 are offered with their corresponding identifiers $\{I_4\}$, $\{P_1\}$, and $\{I_5\}$ becoming active. The choice offered in P1 also has its identifier $\{C_4\}$ become active. Figure 4.10 illustrates CTM to this point.

Now, if behavior P1 is invoked from either vertex labelled $\{I_4\}$, $\{P_1\}$, $\{I_5\}$, $\{C_4\}$,

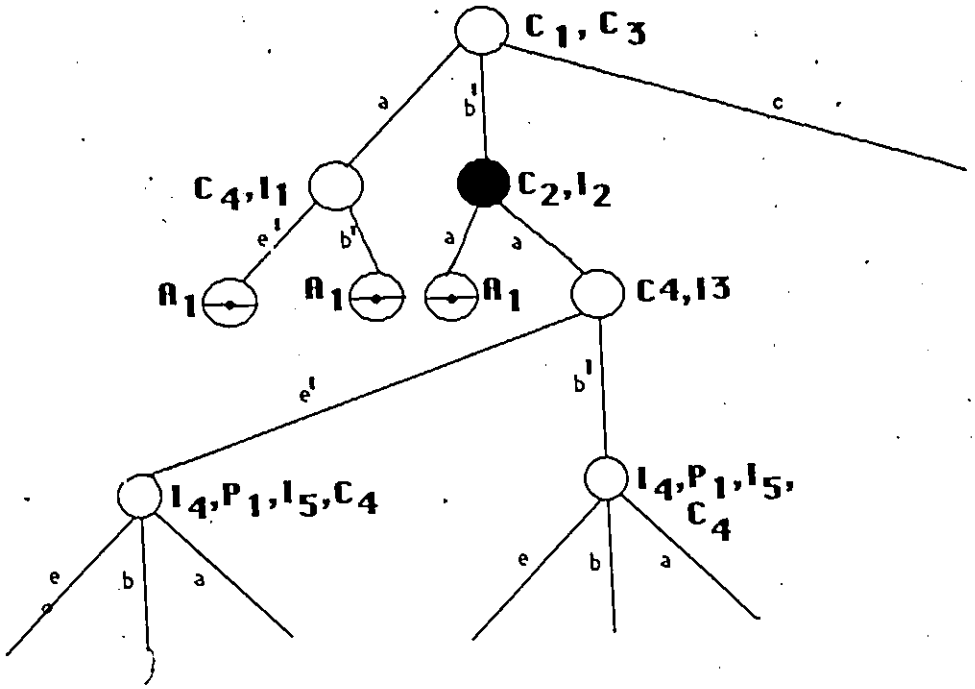


Figure 4.10: Alternative e or b is Chosen

by taking either alternative e or b, this brings us to a second invocation of process P2$\{I_6\}$ which is running in parallel with the first invocation of process P2$\{I_5\}$. The parallel operator ($\{P_1\}$) is still active since process P1 did not reach a stop or an exit. $\{I_4\}$ and $\{C_4\}$ become inactive since process P1 had begun its functioning. $\{I_6\}$ becomes active due to the call of process P2 from process P1. $\{I_5\}$ is still active since the original invocation of process P2 has not yet begun its functioning. Figure 4.11 illustrates CTM to this point.

The alternatives now specified for vertices labelled P_1, I_5, I_6 specify choice between identical event a for the two invocations of process P2 now running in parallel (i.e., $\{I_5\}$ and $\{I_6\}$). If either alternative is taken, then the parallel operator ($\{P_1\}$) becomes inactive (since one of the processes running in parallel reaches an exit) and the only remaining event is an a in the remaining process. When this event is performed, subsequent functioning to the original call of parallel processes P1$\{I_7\}$ and P2$\{I_5\}$ is enabled, which is a stop$\{A_1\}$. The linear sequence of events is abstracted as a'=a;a. Figure 4.12 illustrates CTM to this point.

Returning to the vertices labelled I_4, P_1, I_5, C_4, if alternative a is chosen, process P2 begins its functioning and $\{I_5\}$ becomes inactive. Since process P2 reaches an

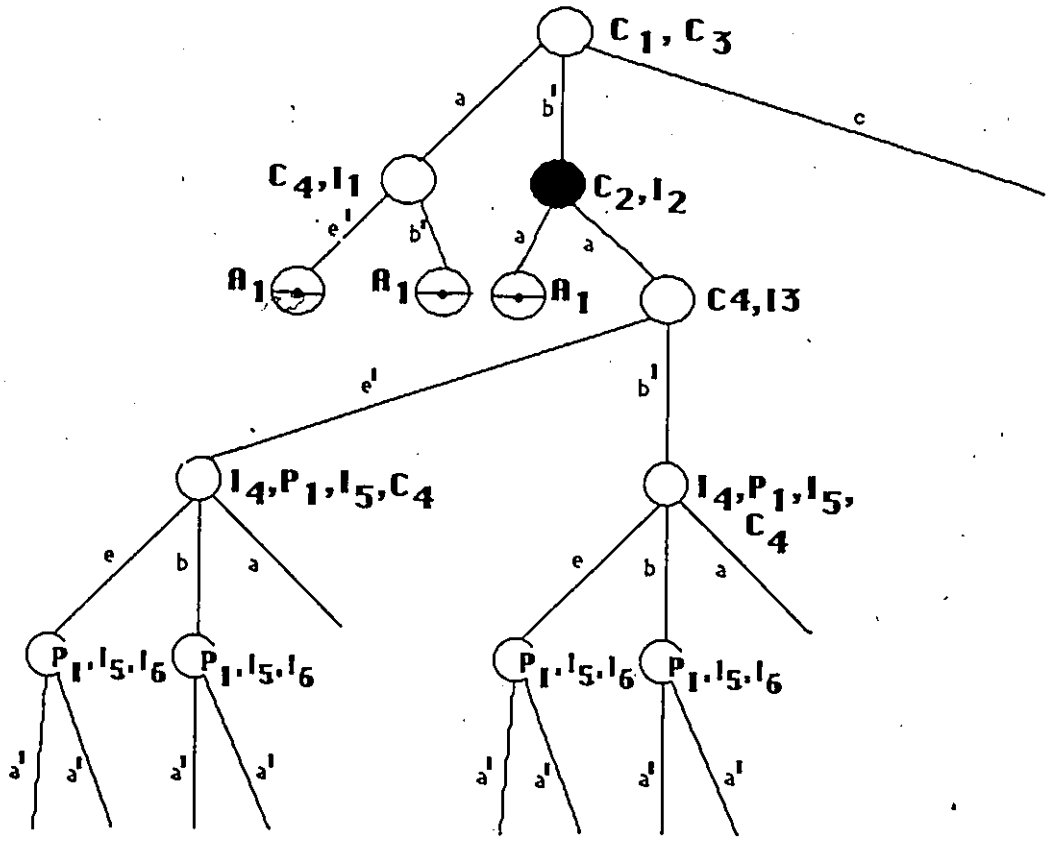


Figure 4.11: Invoke Process P1

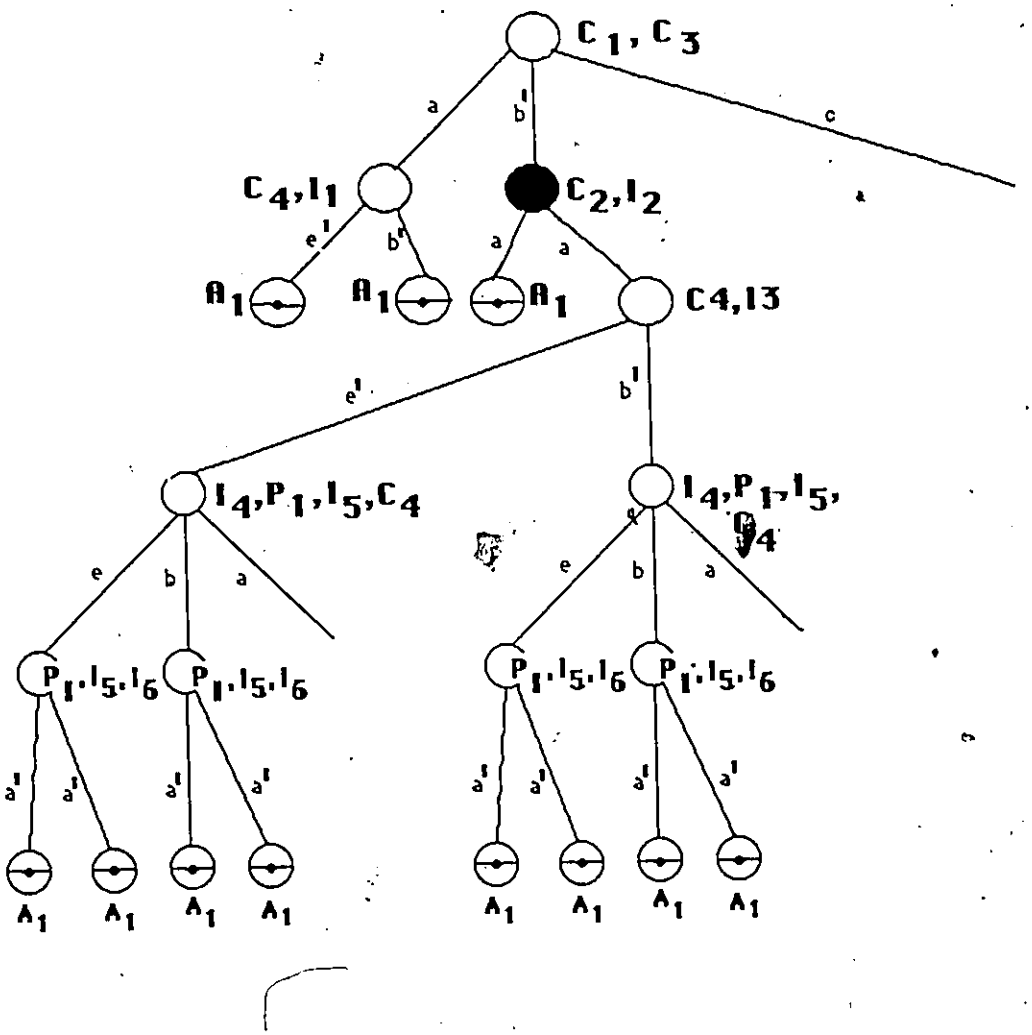


Figure 4.12: Linear Sequence of Event a is Performed

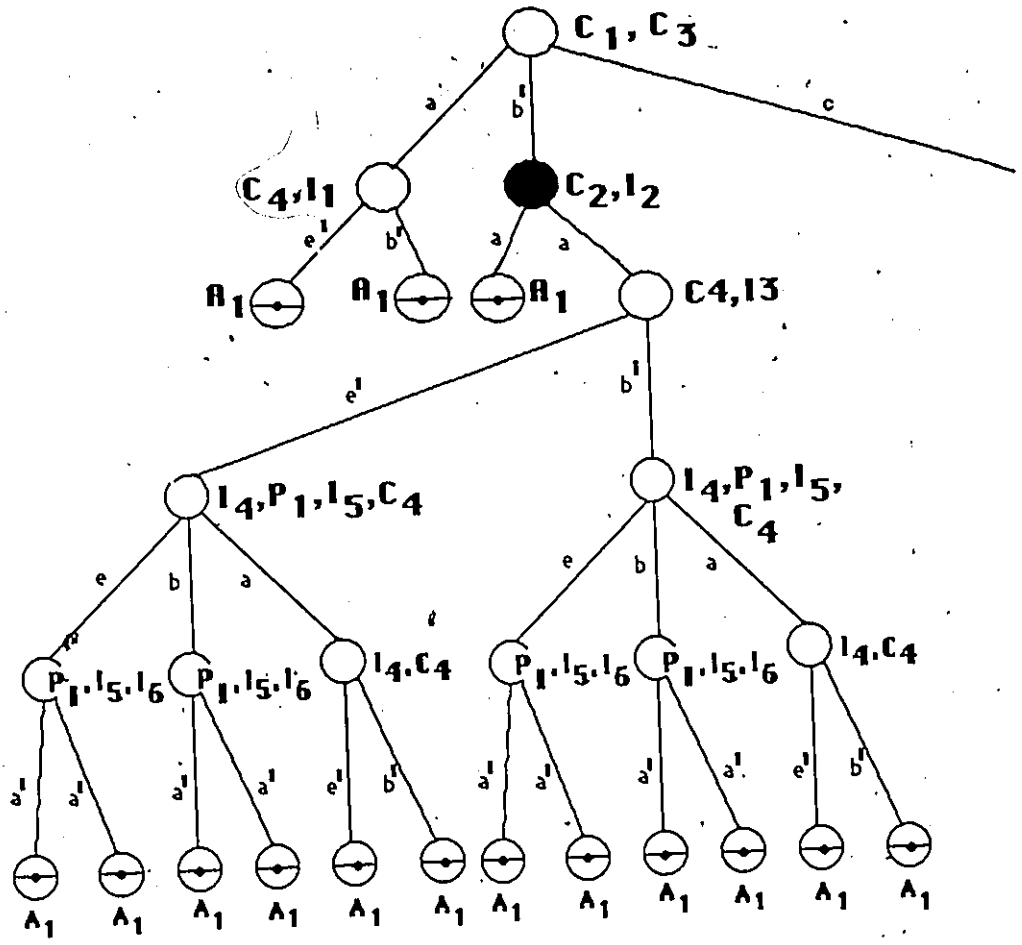


Figure 4.13: Remaining Behavior in Parallel Processes is Performed

exit, the parallel operator $\{P_1\}$ also becomes inactive. The only behavior remaining is specified by process P1$\{I_4\}$ with its active choice, $\{C_4\}$. If either behavior e or b is chosen in process P1, a linear sequence of events through a call to process P2 brings us to a stop (i.e., e'=e;a, b'=b;a). Figure 4.13 illustrates CTM to this point.

Returning now to the root of CTM, if alternative c is chosen, parallel behavior between P1 and P2 is invoked with its corresponding identifiers I_4, P_1, I_5, C_4 becoming active. The same sequence of events from this vertex labelled I_4, P_1, I_5, C_4 are possible as for the previous vertices with this same label. Figure 4.14 illustrates the resulting CTM.

Labelling of CTM for CCS

As with the mapping from LOTOS, it is necessary to label the derived CTM such

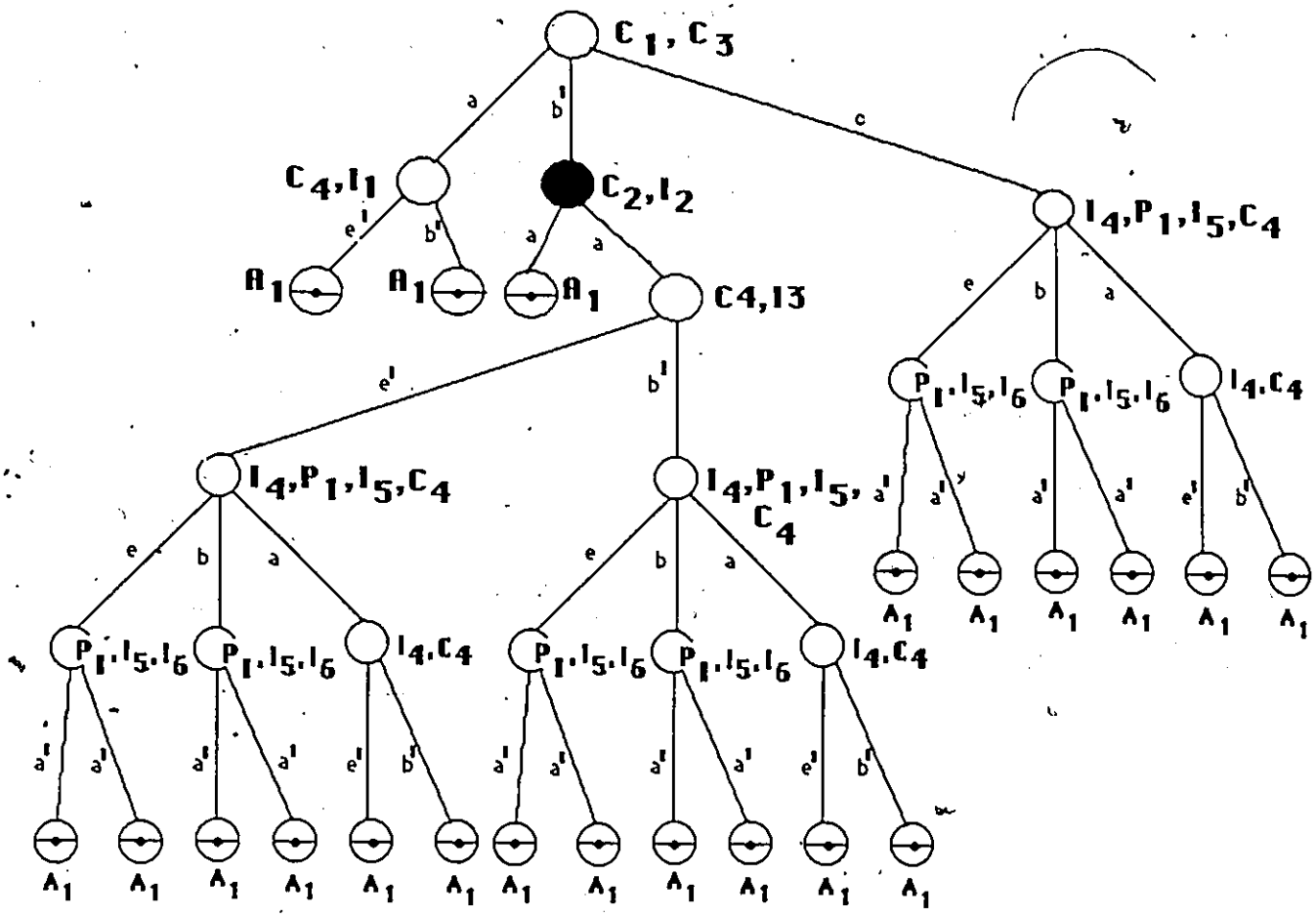


Figure 4.14: The Labelled CTM

that it is possible to recognize previously encountered behavior subexpressions in performing a mapping from a CCS specification. To do so, we:

1. augment the CCS specification with identifiers, and
2. define the current CCS behavior subexpression being mapped in terms of these identifiers.

The CCS specification is augmented with identifiers as follows:

1. After each summation operator or "If" of a conditional construct, insert an identifier $\{S_i\}$, $1 \leq i \leq$ (number of summation operators + conditional constructs) in the CCS specification. For example, let B $+\{S_2\}$ B$'$ denote the current point in the CCS specification which is being mapped to CTM. Then, the summation is active and index($\{S_2\}$) = index($\{S_2\}$) + 1. If behavior B or B$'$ is chosen, then index($\{S_2\}$) = index($\{S_2\}$) − 1.

2. After each action construct, which is immediately followed by an action, insert an identifier $\{A_i\}$, $1 \leq i \leq$ number of such action constructs in the CCS specification. For example, let

$$\alpha x_1, \ldots, x_n \{A_4\} \beta y_1, \ldots, y_n$$

denote a point in a CCS specification at which action $\alpha x_1, \ldots, x_n$ has just been performed. Then, the action is active and index($\{A_4\}$) = index($\{(A_4\}$) + 1. If action $\beta y_1, \ldots, y_n$ is performed, then index($\{A_4\}$) = index($\{A_4\}$) − 1.

3. After each compostion operator, insert an identifier $\{P_i\}$, $1 \leq i \leq$ number of composition operators in the CCS specification. For example, let B| $\{P_1\}$ B' denote a point in a CCS specification at which behavior B and B' are running in parallel. Then, index($\{P_1\}$) = index($\{P_1\}$) + 1. If behavior B or B' reaches a NIL action, then index($\{P_1\}$) = index($\{P_1\}$) − 1.

4. After each identifier construct, insert an identifier $\{I_i\}$, $1 \leq i \leq$ number of identifier constructs in the CCS specification. For example, let b$(E_1, \ldots, E_n)\{I_6\}$ denote a point in a CCS specification at which behavior b has just been invoked. Then, index($\{I_6\}$) = index($\{I_6\}$) + 1. If b begins its functioning, then index($\{I_6\}$) = index($\{I_6\}$) − 1.

In performing the mapping from CCS to CTM, an id_set is maintained exacty as for the mapping from LOTOS to CTM:

1. A CCS summation or conditional construct is treated like a LOTOS choice construct is treated.
2. A CCS action construct is treated like a LOTOS action prefix construct is treated.
3. A CCS composition operator is treated like a LOTOS composition operator is treated with the exception that a CCS behavior terminates when it reaches a NIL action.
4. A CCS identifier construct is treated like a LOTOS process invocation is treated.

Again, as in labelling CTM during mapping LOTOS specifications, we use the current id_set to label the vertices of CTM.

4.4.3 Algorithm to Obtain TT from CTM

Given a labelled CTM, we perform a post–order traversal of the tree and keep track of labels which have been encountered. If a label has been previously encountered, then we use the subsequent prune procedure to prune this subtree from CTM. The resultant tree is TT. Note that the following properties of CTM are preserved in TT:

1. Each edge in TT has the same format as an edge in CTM.
2. The type of each vertex in TT is the same as the corresponding vertex in CTM.

Procedure Prune(root);

In order to create a finite CTM, it is necessary
to prune the subtrees rooted at vertices with previously traversed labels. An outline of a recursive procedure to perform this function
is given by the following. For the vertex that is the root of the subtree to be pruned:

Traverse each subtree rooted at each child of the vertex in

post-order, starting from the first child on the left.
Delete the vertex.

```
VAR
   temp:vptr;

Begin
   If (root≠nil) then
   Begin
      prune(root↑.son);
      temp:=root;
      root:=root↑.sibling;
      release(temp);
      Repeat
         prune(root);
         temp:=root;
         root:=root↑.sibling;
         release(temp);
      Until (root=nil);
   End;
End;
```

4.5 Obtaining T from TT

In TT, only one of the vertices carrying the same label has a subsequent subtree(s). Each edge in TT stands for a linear sequence of events, $e_1, \ldots, e_n$, where each e_i, $1 \leq i \leq n$, is equal to (P,I,O). Each root to terminal vertex path in TT implies a test sequence. Note in TT that vertex types of CTM are preserved. That is, we can identify points of deterministic, nondeterministic, and grey choice in TT. Note also that the information contained in CTM and preserved in TT during the transformation from CTM to TT should be used during the construction of test cases by a test designer.

The test sequences implied by TT test for only valid behavior of a given I_i and do not express actual values for interaction identifier parameters (i.e., $x_1, \ldots, x_n$ in $ii(x_1, \ldots, x_n)$). We wish to obtain a set of test sequences which test for both valid and defensive behavior of any I_i, and which are executable (i.e., express values for

interaction identifier parameters).

We define T to be a forest of adaptive test sequences (i.e., trees) which test for both valid and defensive behavior of any I_i. T is the union of two sets of adaptive test sequences T_A and T_B.

We let T_A be adaptive test sequences which are syntactically valid in TT and which express parameter variations of interaction identifiers of TT. In performing tests for conformance, T_A specifies adaptive test cases for testing only valid behavior of I_i. The set of test sequences specified by T_A measure the levels of reliability (i.e., ability of an implementation to perform a required function under stated conditions for a stated period of time) and accuracy (i.e., freedom from error) of I_i. We are, however, also interested in measuring the robustness of I_i: the extent to which I_i can continue to operate despite the introduction of invalid inputs. It is certain that I_i may be subjected to invalid inputs occasionally due to unforeseen circumstances. If I_i is unable to operate when subjected to these invalid inputs, then its level of robustness is low and I_i may not be considered a usefull piece of software (i.e., has low quality). Thus, we wish to augment T_A with tests for invalid behavior, which we call T_B. Hence, we define T to be a union of T_A and T_B ($T = T_A \bigcup T_B$) which tests for reactions of each $I_i \in$IMP to erroneous as well as valid input.

In general, the test sequences contained in T are organized in a hierarchal fashion so as to aid in the testing process. An adaptive test sequence in this hierarchy consists of the following components:

Preamble Subtree: A tree denoting the set of events necessary to bring the implementation to be tested to a state (i.e., a context of the implementation defined in terms of instantiated free variables) required for the test as given in the test purpose [Rayn87]. (Note that this is normally simply a linear sequence of events.)

Test Body Subtree: A tree denoting the set of events necessary to fulfill the test purpose [Rayn87].

Postamble Subtree: A tree denoting the set of events necessary to bring the implementation to be tested to a state required after execution of the test-sequence [Rayn87]. (Note that this also is normally a linear sequence of events.)

In what follows, the sets of adaptive test sequences used to construct T are given in terms of these three components. Sections 4.5.1 and 4.5.2 clarify the definitions

of T_A and T_B and discuss how they are obtained. Section 4.5.3 gives a heuristic to obtain executable T from TT. Section 4.5.4 gives an example of obtaining an executable T for the OSI protocols[ISO1].

4.5.1 Test Sequences For Testing Valid Behavior (T_A)

In each S, a set of capabilities to be provided by an implementation is given. In performing tests for conformance of a given I_i to S, these capabilities are mapped to a set of test purposes such that test sequences fulfilling these test purposes will exercise each capability claimed to be provided. These test purposes can be thought of as functions of the capabilities specified in S.
Let

- C = $\{c_1, \ldots, c_m\}$ be the set of capabilities specified in S [11] and
- P = $\{vtp_1, \ldots, vtp_n\}$ be the set of test purposes for valid behavior specified in S.

Then, define a relation $\alpha(c_i)$ to be:

$$\alpha(c_i) \longrightarrow \{vtp_1, \ldots, vtp_r\}$$

where:

- $c_i, 1 \leq i \leq m$, is the ith capability of C
- vtp_j, $1 \leq j \leq r$, is the jth test purpose $\in$ P implied by capability $c_i \in$ C
- $r \leq n$
- $\bigcup_1^m \alpha(c_i) =$ P

We use P to develop T_A which tests for valid behavior of a given implementation.

For example, let C = (error checking, trace, recovery).
Then,

- α(error checking) $\longrightarrow$

  1. vtp_1 = submit valid input to ensure it is not taken as being erroneous.

[11] The manner in which these capabilities are explicitly specified in S is an open question which is under study by ISO.

2. vtp_2 = submit invalid input to ensure it is taken as being erroneous.

- α(trace) $\longrightarrow$

  1. vtp_3 = disable trace and ensure that trace is not performed.

  2. vtp_4 = enable trace and ensure trace is performed correctly.

- α(recovery) $\longrightarrow$ vtp_5 = submit invalid input followed by valid input to ensure recovery is functioning properly.

Now, we make use of both P and TT to construct T_A. If our aim is to construct T from TT as being composed of generic test cases (see Section 4.5.3), then the test designer may incorporate information contained in TT about the deterministic, nondeterministic, and grey choices into these test cases. Such information can then be used during:

1. The derivation of executable test cases.

2. The selection of a particular subset of T as T_i (see Section 4.6).

Since each vtp_i, $1 \leq i \leq n$, in P is intended to exercise a specific capability (or part of a specific capability) of I_i, this in turn means that a subset of test sequences specified in TT is used to fulfill a given vtp_i. This follows immediately from the fact that TT specifies all possible legal interaction sequences (in a tree form) of any I_i, and one or more of these interaction sequences (i.e., a subset) fulfills a given vtp_i. Hence, each test purpose implies a set of subtrees of TT which will be arranged in such a manner so as to exercise the corresponding capability or part of the corresponding capability specified in S. Let:

- T_root_{ij} denote the root of the jth subtree of TT implied by test purpose vtp_i

- ($\text{T\_leaf}_{ij1}, \ldots, \text{T\_leaf}_{ijr}$) denote the r leaves of the jth subtree implied by test purpose vtp_i.

Then, define a relation T_trees(vtp_i) to be:

$$\text{T\_trees}(vtp_i) \longrightarrow \begin{bmatrix} T\_root_{i1}, & (T\_leaf_{i11}, \ldots, T\_leaf_{i1r}) \\ \vdots & \vdots \\ T\_root_{is}, & (T\_leaf_{is1}, \ldots, T\_leaf_{isr}) \end{bmatrix}$$

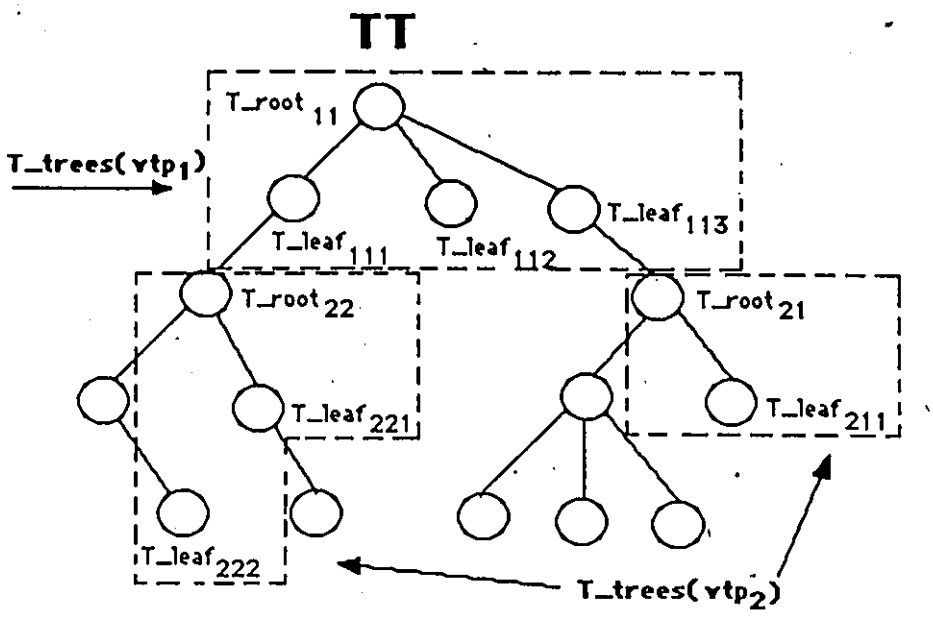


Figure 4.15: Illustration of T_trees Relation

where s is the number of subtrees of TT implied by test purpose vtp_i. Figure 4.15 illustrates the T_trees relation.

It is the union of all subtrees of TT, which we call Sub_t_set, implied by members of P through relation T_trees which are used to construct tests for valid behavior in T (i.e., T_A):

$$\text{Sub\_t\_set} = \bigcup_{i=1}^{n} \text{T\_trees}(vtp_i)$$

Sub_t_set has the following two properties:

1. Each element of Sub_t_set is composed of preamble, test body, and postamble subtrees.

2. Sub_t_set specifies all valid behavior in TT (i.e., each occurrence of an event sequence in TT is represented in Sub_t_set).

Each test purpose vtp_i is used to construct a test case tc_i, $1 \leq i \leq n$, of T_A which is itself a tree since we use adaptive testing. The members of T_trees(vtp_i) are attached together in the following manner to create tc_i:
Let:

1. root(tc_i) $\in$ T_trees(vtp_i) be the root of the tree defining test case tc_i. The root of the test case tree is one of the roots of TT implied by test purpose vtp_i and this root is normally a preamble subtree.

2. Append(T_leaf$_{ijk}$,T_root$_{lm}$) be a function which attaches to the leaf T_leaf$_{ijk}$ the tree with root T_root$_{lm}$:

   - i is the index corresponding to vtp_i, $1 \leq i \leq n$
   - j is the jth T_root implied by vtp_i
   - k is the kth leaf of T_root$_{ij}$
   - T_leaf$_{ijk}$ is either a C–vertex or terminal N–vertex (i.e., we do not append subtrees within abstracted linear sequences)[12].

Thus, Append simply attaches to leaves of the test case tree being constructed the trees of TT implied by vtp_i.

To exemplify, consider Figure 4.16. Here, the test case tc_1 has root T_root$_{11}$. The Append function is applied three times to complete the test case:

1. Append(T_leaf$_{111}$,T_root$_{12}$)
2. Append(T_leaf$_{112}$,T_root$_{13}$)
3. Append(T_leaf$_{113}$,T_root$_{14}$)

Now, to create a test case, we continuously append to leaves of the tree being constructed members of T_trees(vtp_i) (i.e., test body and postamble subtrees) until tc_i is completed. The manner in which this is done is a function of the test purpose. Essentially, subtrees of TT are organized and appended together in such a fashion so as to specify test cases which fulfill their corresponding test purposes. The set of trees yielded by producing tc_i for each vtp_i, i=1,..., number of test purposes, is a forest of adaptive test sequences. We call this forest T_A.

For example, let vtp_1 be *check the recovery procedure.* The set of subtrees of TT implied by this test purpose is T_trees(vtp_1) =

- T_root$_{11}$, (T_leaf$_{111}$)

[12]Generally, a linear sequence of events can be considered as atomic when testing for valid behavior.

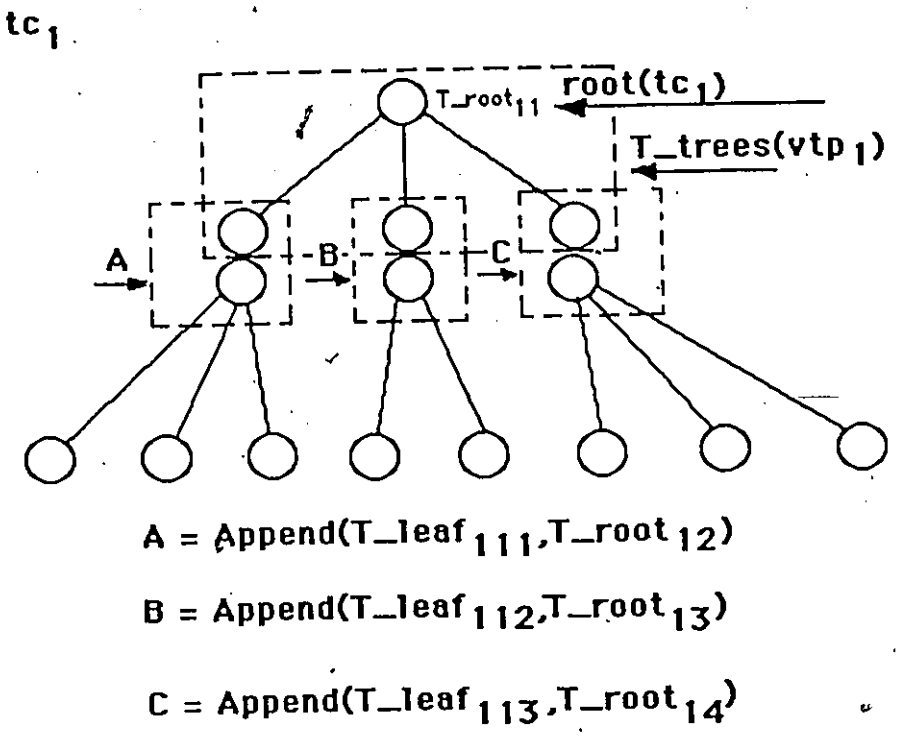


Figure 4.16: Illustration of Root and Append

- T_root_{12}, (T_leaf_{121}, T_leaf_{122}, T_leaf_{123})
- T_root_{13}, (T_leaf_{131})

where:

- T_root_{11} = initialize logon
- T_leaf_{111} = ready to receive data
- T_root_{12} = receive data
- T_leaf_{121} = erroneous data stream end
- T_leaf_{122} = valid data stream end
- T_leaf_{123} = erroneous data stream followed by valid data stream end
- T_root_{13} = begin recovery
- T_root_{131} = ready to receive data

This set of subtrees of TT implied by vtp_1 is constructed together in such a fashion so as to fulfill vtp_1 by defining a root for tc_1 and applying the append function continuously. Thus, the test case tc_1 is:

- root(TC_1) = T_root_{11}
- Append(T_leaf_{111}, T_root_{12})
- Append(T_leaf_{121}, T_root_{13})
- Append(T_leaf_{122}, T_root_{13})
- Append(T_leaf_{123}, T_root_{13})

Figure 4.17 illustrates this example.

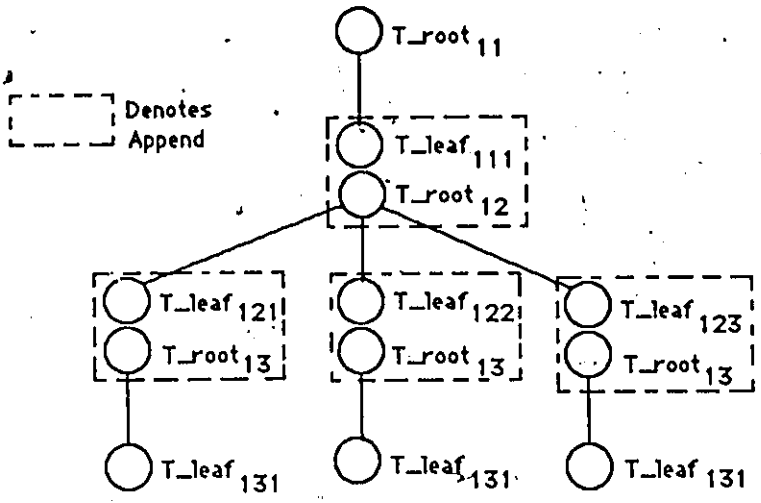


Figure 4.17: Example Test Case

4.5.2 Test Sequences For Testing Defensive Behavior (T_B)

We recall that test sequences in T_B test for defensive behavior in I_i. To test for such defensive behavior (i.e., robustness), we must consider all aspects of each I_i which can be tested. T_B includes tests for the following:

1. Valid sequences containing some invalid interactions ii$(x_1, \ldots, x_n)$. These are sequences for one invalid value for each parameter field of each input interaction identifier.

2. Invalid sequences of valid or invalid interactions ii$(x_1, \ldots, x_n)$. For example, one out of sequence input interaction identifier for all critical interaction sequences.

Let:

- C = $\{c_1, \ldots, c_m\}$ be as defined previously.
- D = $\{dtp_1, \ldots, dtp_n\}$ be the set of test purposes for testing defensive behavior of any I_i claiming to conform to S.

Then, define relation $\beta(c_i)$ to be:

$$\beta(c_i) \longrightarrow \{dtp_1, \ldots, dtp_r\}$$

where:

- c_i, $1 \leq i \leq m$ is the ith capability of C.
- dtp_j,$1 \leq j \leq r$ is the jth defensive test purpose $\in$ D implied by capability $c_i \in$ C.
- $r \leq n$
- $\bigcup_{i=1}^{n} \beta(c_i) = \mathrm{D}$

We use D to construct T_B which tests for defensive behavior of a given implementation.

The exact manner in which D is used to construct T_B differs from the manner in which P was used to construct T_A. On one hand, since T_B specifies tests for valid sequences of valid interactions with invalid values for parameter fields, T_B contains sequences which **are** subtrees of TT (i.e., such sequences are syntactically valid in S). On the other hand, since T_B specifies tests for invalid sequences of valid interactions, T_B also contains sequences which **are not** subtrees of TT (i.e., such sequences are syntactically invalid in S). Thus, we define the relations Def_Trees1(dtp$_i$) and Def_Trees2(dtp$_i$) to imply testing trees which test for defensive behavior of an implementation which **are** subtrees of TT and which **are not** subtrees of TT, respectively.

Let:

- DT_root$_{ij}$ denote the jth subtree of TT implied by defensive test purpose dtp$_i$.
- DT_leaf1$_{ij1}$,...,DT_leaf$_{ijr}$ denote the r leaves of the jth subtree implied by defensive test purpose dtp$_i$.

Then, define relation Def_trees1(dtp$_i$) to be:

$$\text{Def\_trees1}(dtp_i) \longrightarrow \begin{bmatrix} DT\_root1_{i1}, & (DT\_leaf1_{i11}, \ldots, DT\_leaf1_{i1r}) \\ \vdots & \vdots \\ DT\_root1_{is}, & (DT\_leaf_{is1}, \ldots, DT\_leaf_{isr}) \end{bmatrix}$$

where s is the number of subtrees of TT implied by defensive test purpose dtp$_i$. The members of Def_trees1 are constructed together in such a fashion so as to create a new test case in the same manner as for members of T_trees. The exact manner in which this is done is a function of the defensive test purpose.

For example, let dtp$_1$ be *input an illegal quality of service parameter value to ensure it is refused by the implementation.* The set of subtrees of TT implied by this defensive test purpose is Def_trees1(dtp_1) =

- DT_root1_{11}, (DT_leaf1_{111})
- DT_root1_{12}, (DT_leaf1_{121}, DT_leaf1_{122})
- DT_root1_{13}, (DT_leaf1_{131})

where:

- DT_root1_{11} = idle state
- DT_leaf1_{111} = negotiate quality of service
- DT_root1_{12} = receive quality of service parameter
- DT_leaf1_{121} = send quality of service not supported message
- DT_leaf1_{122} = send illegal quality of service message
- DT_root1_{13} = send abort
- DT_leaf1_{131} = idle state

This set of subtrees of TT is constructed together in the same fashion as were the subtrees implied by each vtp_i in P. Each dtp_i in D has a corresponding defensive test case DTC_i. Thus, the defensive test case DTC_1 is:

- root(DTC_1) = DT_root1_{11}
- Append(DT_leaf1_{111}, DT_root1_{12})
- Append(DT_leaf1_{121}, DT_root1_{13})
- Append(DT_leaf1_{122}, DT_root1_{13})

Now, define relation Def_trees2(dtp_i) to be:

$$\text{Def\_trees2}(dtp_i) \longrightarrow \begin{bmatrix} DT\_root2_{i1}, & (DT\_leaf2_{i11}, \ldots, DT\_leaf2_{i1r}) \\ \vdots & \vdots \\ DT\_root2_{is}, & (DT\_leaf2_{is1}, \ldots, DT\_leaf2_{isr}) \end{bmatrix}$$

where s is the number of trees necessary to fulfill dtp_i. Note that trees necessary to fulfill a dtp_i are not in TT. The manner in which each tree implied by Def_trees2 is derived is largely a heuristic procedure. A preamble subtree can always be derived from TT. Then, to each leaf of the preamble subtree, we append a subtree that starts

with an edge which corresponds to an unspecified input interaction and continues with an appropriate expected response that will terminate the test case. Here, by an unspecified input interaction we mean a syntactically valid input interaction that is not specified to be in the choice_set at that vertex (leaf of the preamble subtree) by S.

We define T_B to be all trees implied by the Def_trees1 and Def_tree2 relations:

$$T_B = \text{Def\_trees1}(\text{dtp}_i) \bigcup_{i=1}^{m} \text{Def\_trees2}(\text{dtp}_i)$$

The set of defensive tests in T_B contribute to the forest T in one of two ways:

1. A testing tree in T_B can be appended to an existing testing tree in T_A.
2. A testing tree in T_B can be a stand alone tree in the forest T.

Again, the manner in which each testing tree of T_B augments T_A to form T is a function of each specific defensive test purpose. We note that it may be possible to define a relation between the set of test purposes $\bigcup_i vtp_i$ and the set of defensive test purposes $\bigcup_j dtp_j$. Perhaps each capability of S implies some members in each of these sets. This is a subject for further study. Section 4.5.4 gives an example of obtaining T_A and T_B to produce T for the specific case of OSI communication protocols [ISO1]. We now present a heuristic to obtain T from TT.

4.5.3 A Heuristic to Obtain T from TT

In order to present a heuristic for obtaining T from TT, we clarify what an abstract versus generic set of test sequences is (as was alluded to in Section 4.3). In general, there is information needed by a tester beyond the set of test sequences to be applied in order to carry out a test for conformance of a given implementation to its specification. This information includes:

Abstract Test Method: This specifies the points closest to the implementation at which control and observation that will be exercised during testing are specified.

Particulars of the test architecture: These are any information necessary to conduct the tests such as tester module, responder module, etc.

Implementation Conformance Statement(ICS): This is the set of capabilities claimed to be supported by a particular implementation.

The set of test cases specified in T is considered **generic** if none of the above information is known. In this case, T is thought of as being applicable to any implementation of a given S, in any architecture that will allow the use of control and observation points nearest to the implementation (i.e., those points that are at the lower and upper layer interface of the IUT). If the abstract test method is known, then the set of test cases specified in T is considered **abstract**. Here, T specifies tests in terms of those interaction sequences that can be controlled and observed in a given test method where not all information necessary for testing is known. If the particulars of the test architecture are known, then the set of tests specified in T are considered to be executable. This is equivalent to our first solution strategy outlined in Section 4.3 and will be further discussed in Section 4.8. If all of the above information is known, then T can be pruned so as to obtain T_i which is the subset of T that is applicable to a particular I_i. T_i is also an executable set of tests. This is equivalent to our second solution strategy outlined in Section 4.3 and will be further discussed in Sections 4.6 and 4.9. A conformance test suite (CTS) [Rayn87] is either T or T_i.

Now, in order to obtain executable T, we must first know the particulars of the test architecture. Then, knowing this, test sequences of T are obtained by employing the following heuristic:

1. Varying parameters for each value we are interested in to obtain T_A. TT is used as the source for T_A.
   Let:

   - C = $\{c_1, c_2, \ldots, c_m\}$ be the set of capabilities specified in S.
   - P = $\{vtp_1, vtp_2, \ldots, vtp_n\}$ be the set of test purposes for valid behavior implied by C.
   - $P_{c_i} = vtp_1, \ldots, vtp_r$ be the test purposes to test for the capability c_i.

   Then, for each $vtp_j \in P_{c_i}$ and for each $c_i \in C, 1 \leq i \leq m, 1 \leq j \leq r, T_A$ contains a valid behavior test case tc_j. Let $\gamma(tc_j) = \{y_1, \ldots, y_s\}$ be the set of parameters to be varied in tc_j. Then, each $tc_j \in T_A$ should be replicated as many times as is necessary to accomodate parameter variations for γ.

2. Constructing test sequences for T_B by giving:

   (a) Valid sequences containing some invalid interactions ii$(x_1, \ldots, x_n)$. These are: sequences for one invalid value for each parameter field of each input interaction identifier in the sequence. Let:

   - C be as above
   - D = $\{dtp_1, \ldots, dtp_n\}$ be the set of defensive test purposes implied by C.
   - $D_{c_i} = dtp_1, \ldots, dtp_r$ be the defensive test purposes to test for the defensive behavior implied by capability C_i.

   For each $dtp_j \in D_{c_i}$ and for each $c_i \in C, 1 \leq i \leq m, 1 \leq j \leq r, T_B$ contains a defensive behavior test case tc_j. If tc_j contains valid sequences containing some invalid interactions ii$(x_1, \ldots, x_n)$, then tc_j should be replicated as many times as necessary to accomodate:

   i. Invalid values for each parameter $x_i \in x_1, \ldots, x_n$ of each input interaction identifier ii.

   (b) Invalid sequences of valid or invalid interactions ii$(x_1, \ldots, x_n)$. These are sequence containing one out of sequence interaction identifier for all critical interaction sequences.

Variation of parameter values are of utmost importance when constructing T. We note that parameters are of two types:

enumerated : a set of values

- all legal PDU parameters
- possible values of bitwise parameters
- scalar types in Pascal
- sets

continuous : a range of values

- integer parameters
- real parameters

Depending on the type of parameter being considered, various methods can by applied to obtain a set of parameter values which will establish the highest possible confidence that I_i performs as specified [SaBo87]. These methods include boundary value analysis, equivalence partitioning [Myer79], etc. Further, the Implementation Conformance Statement (ICS) can also be used to obtain such a set of parameter values. Information contained in the ICS is used to establish parameter values which have the greatest probability of detecting errors in I_i. Of course, a complete and informative ICS is necessary to apply such a method and is a topic of current research [ISO7].

4.5.4 Example of Obtaining T from TT

We obtain a T for a particular layer N communications protocol in the reference model of OSI[ISO1] as follows:

We must consider all capabilities of the system to be tested. Each of these capabilities is expressed by S in terms of the following:

1. Support of Service primitives(SP's)[ISO1].
2. Support of Protocol data units(PDU's)[ISO1].
3. Support of Timers.
4. Ability to use the underlying service[ISO1].

We distinguish two types of tests and state how they are realized in T:

Type 1: Testing for parameter variations of a given S. Since T_A assigns values to $x_1, \ldots, x_n$ of each interaction ii($x_1, \ldots, x_n$), we augment T_A with test sequences for each actual parameter value we are interested in. This may result in duplicate test sequences, each containing a different value for certain parameter fields. Test sequences for this type of test are defined by the T_trees relation. Since T_B may specify tests with valid sequences of interactions with invalid parameter values, we augment T_B with each invalid parameter value we are interested in. Test sequences for this type of test are defined by the Def_trees1 relation since these sequences are syntactically valid.

Type 2: Testing for exceptional cases in the control flow of a given S (i.e., occurrences of SP's and PDU's). T_B specifies illegal sequences of observable SP's and PDU's. Test sequences for this type of test are defined by the Def_tree2 relation since these sequences are syntactically invalid.

We apply the following heuristic to implement type 1 and type2 testing in order to augment T_A and T_B. The relation which will be applied during augmentation of T_A and T_B (i.e., T_trees, Def_trees1, or Def_trees2) is also given in parentheses. We augment T_A to include tests for the following:

- support of all variations of SP types (T_trees).
- default values of parameter fields of each SP (T_trees).
- boundary values and one midrange value for integer parameters of SP's (T_trees).
- common values in bitwise parameters of SP's (T_trees).
- all critical pairs and one normal value pair of interdependent parameter pairs of SP's (T_trees).
- support of all variations of PDU types (T_trees).
- default values for each PDU parameter (T_trees).
- boundary values and one midrange value for integer parameters of PDU's (T_trees).
- common values in bitwise parameters of PDU's (T_trees).
- all critical pairs and one normal value pair of interdependent parameter pairs of PDU's (T_trees).
- exercising of all defined values of timers for N-entity (T_trees).

We construct T_B to include in T tests for the following:

- at least one invalid SP type (Def_trees2).
- one out of sequence SP for all critical interaction sequences (Def_trees2).
- one invalid value for each parameter field of each SP (if possible) (Def_trees1).
- one out of sequence PDU for all critical interaction sequences(Def_trees2).
- at least one invalid PDU type (Def_trees2).
- one invalid value for each parameter field of each PDU (if possible) (Def_trees1).

- recovery from invalid indications from (N-1)-service (Def_trees2).

The resulting T is a testing tree with:

1. actual input values.
2. expected output values.
3. test sequences for reactions of each $I_i \in$IMP to valid and erroneous input.

We note that the above heuristic can be applied to multi-connection tests by augmenting T_A and T_B with tests for all possible connection endpoint identifiers [SaBo87]. Of course, the pragmatics of testing multiple connections can be prohibitive in doing so. Some heuristics on constructing test sequences to test defensive behaviors are given in [SaBo87].

4.6 Obtaining T_i from T

We wish to obtain a set of test sequences from T which are specific to a given $I_i \in$ IMP which we call T_i. To do so, we make use of the Implementation Conformance Statement(ICS) which is provided by the implementor of each I_i. Each ICS describes the characteristics of I_i which are to be tested by describing capabilities and options supported by I_i for which conformance is claimed. Such characteristics can be classified as providing claims on:

1. Mandatory features and capabilities.
2. Optional features and capabilities.
3. Conditional features and capabilities.
4. How some instances of nondeterministic choice are resolved.
5. Ranges of parameter values supported.

These claims provide precise knowledge of how the choices are implemented in I_i. We use such knowledge to both prune various trees of T in order to obtain T_i, and to assign verdicts to test cases of T_i (i.e., pass, fail, inconclusive).

Let $C' \subseteq C$ be the set of capabilities claimed to be supported in the ICS. Then $\bigcup_{i=1}^{m} \alpha(c_i) = P$ and $\bigcup_{i=1}^{m} \beta(c_i) = D$ are the sets of test purposes and defensive test

purposes, respectively, implied by this set of capabilities; m = number of capabilities claimed to be supported in the ICS.

We recall that P and D imply sets of trees in T. To express the pruning of T, let

$$\text{prune\_set} = \bigcup_i^n \text{T\_trees}(TP_i) - \bigcup_i^m \text{T\_trees}(TP_i)$$

be those trees which are implied by S but which are not implied by S_i. We recall that the function Append(T_leaf,T_root) appends to T_leaf a tree rooted at T_root. Now, let **Append_set** give the set of trees which were appended together using function Append at a given point in T. If any tree in Append_set $\in$ prune_set, then we remove this tree from T. Thus, if an alternative expresses behavior which is not claimed to be supported in the ICS, then this alternative is removed (i.e., pruned) from T.

We now show how specific claims of the ICS are used to prune T. The information contained in test cases comprising T on deterministic, nondeterministic, and grey choices can be used together with the claims in the ICS to obtain T_i. Each claim is classified in terms of the type(s) of claim(s) it provides (1,2,3,4, or 5) as above. For the following, pruning of an alternative means removing this edge and its subsequent subtree from T. The ICS makes claims on:

Supported Functions(1,2,3) If an alternative specifies behavior for a function that is not supported, then prune the alternative. Any mandatory feature which is not claimed to be supported will not be tested. Such omission of a mandatory feature is called **allowed nonconformance**[ISO7]. Usually, the static conformance review[ISO7] will reject the implementation for further testing if some mandatory capabilities or some capabilities that are required by supported capabilities are not implemented. For example, let C = (error checking, trace, recovery) be the set of capabilities specified in S [13]. Let C' = (error checking, recovery) be the set of capabilities supported in I_i. Let $\text{T\_root}_5 \in \text{Append\_set}(\text{T\_root}_2)$ be an alternative in T such that T_root_5 = begin trace. Since $\text{T\_root}_5 \in$ prune_set, $T_i = \text{T} - \text{T\_root}_5$. We prune all such alternatives which specify behavior for non_supported functions. The result

[13]Of course, any implementation will have many capabilities. For sake of clarity, we use very simplified capability sets.

is T_i. Assuming that I_i passes the set of tests now specified in T_i, there are three possible scenarios for assigning a verdict of conformance of I_i to S:

1. If trace is a mandatory capability, then I_i is a non-conforming implementation of S.
2. If trace is a conditional capability, then I_i is a conforming implementation of S if the set of conditions for exclusion of trace hold. For example, if trace can not be present when error checking is present, then I_i is a conforming implementation of S.
3. If trace is an optional capability, then I_i is a conforming implementation of S.

Supported Parameters(1,2,3,5) If an alternative specifies behavior involving a parameter which is **not** supported, then prune the alternative. For example, let C = (support of CRC–16 parameter, support of enquiry priority parameter) be the set of capabilities specified in S. Let C′ = (support of CRC–16 parameter) be the set of capabilities supported in I_i. Let $T\_root_{10} \in$ append(T_root_6) be an alternative in T such that T_root_{10} = check protocol data unit contents. If choice_set of T_root_{10} = {(–,CRC–16,–), (–,priority,–)}, then T = T – (–,priority,–). Hence, an alternative which specifies behavior for the unsupported parameter is pruned from T along with its subsequent subtree. Again, there are three possible scenarios for assigning a verdict of conformance of I_i to S:

1. If support of the enquiry priority parameter is mandatory, then I_i is a nonconforming implementation of S.
2. If support of the enquiry priority parameter is a conditional capability then I_i is a conforming implementation of S if, for example, the enquiry priority parameter can not be included when CRC–16 is enabled.
3. If support of the enquiry priority parameter is optional, then I_i is a conforming implementation of S.

Choices made by implementor for particular procedures(4) Prune the nondeterministic choice alternatives not claimed to be supported.

For example, let C = (support of window 1024, support of window 2048) be the set of capabilities specified in S. Let C' = (support of window 1024) be the set of capabilities supported in I_i. Let $T\_root_7 \in Append\_set(T\_root_6)$ be an alternative in T such that T_root_7 = input to window. If choice_set of T_root_7 is {(–,in_1024,–), (–,in_2048,–)}, then T = T – (–,in_2048,–). Hence, an alternative which specifies behavior for the unsupported procedure is pruned from T along with its subsequent subtree. Since support of a window is a mandatory capability, I_i is a conforming implementation of S if it passes the remaining tests specified in T_i.

Section 4.6.1 gives an example of obtaining T_i for the specific case of OSI communication protocols[ISO1].

We now briefly elaborate on two issues: resolution of nondeterministic choice, and influence of a particular test architecture and IXIT (Implementation eXtra Information for Testing) upon obtaining T_i.

As stated previously, each I_i can be characterized in terms of its mandatory, optional, and conditional capabilities. We can categorize the possible types of nondeterminism of a given S in these capability types as follows:

Mandatory Feature Nondeterminism: It is related to a feature which **must** be provided by I_i and which is implemented as a variant choice construct. In this case, nondeterminism is a function of the manner in which the feature is implemented, not a function of **which** features are implemented. The number of choices specified in S is preserved in I_i. Consider the simple transport entity example of Section 2.4.2. We have the possibilities of either outputing from Buf_3 or Buf_5 when both buffers are non-empty. Nondeterminism is due to the manner in which this outputting from the buffers is implemented. Normally, the ICS will not provide a claim on how such nondeterministic choices are resolved.

Optional Feature Nondeterminism: It is related to a feature which can be provided in I_i. In this case, nondeterminism is a function of whether or not the option is chosen to be implemented. The number of choices specified in S may or may not be preserved in I_i. Consider, for example, a laser printer which can optionally support French characters. Nondeterminism here is due

to which options are implemented (i.e., support or not support French characters). Normally, the ICS provides claims on how such nondeterministic choices are resolved. We note here that inclusion of a given option in I_i can introduce nondeterminism between the option and existing functions in I_i. Such nondeterminism is resolved as a variant choice construct implementation and, normally, the ICS does not provide claims on how this nondeterminism is resolved.

Conditional Feature Nondeterminism: It is related to a feature which must be provided in I_i only if the conditions for doing so hold. In this case, nondeterminism is, once again, a function of whether or not the option is implemented. The number of choices specified in S may or may not be preserved in I_i. Consider a laser printer which supports French characters only if the host computer has a particular French software package. Nondeterminism here is due to which features are implemented. ICS normally provides information on how such nondeterministic choices are resolved.

To this point, the set of test cases in T_i are still not executable. In order to make them executable, we must establish a variety of options for testing. These options are given in terms of the particulars of the test architecture and parameters necessary to testing via the IXIT [14].

Use of an abstract test method does not limit the freedom of the tester to implement test cases specified in T_i in any way[Rayn87]. It is the particulars of the test architecture which resolve these possibilities. When a tester knows the points of control and observation at which to interface I_i (i.e., addresses and ports), he is able to conduct the test assuming knowledge of any other parameters necessary for testing is obtained from the IXIT. The IXIT contains such information as is necessary for the test operator to run the conformance test suite specified in T_i and is reserved for alternatives and matters relevant to the testing environment. This information can be:

- addressing information,
- information in the ICS which needs to be precised (eg. timer values),

[14] Use of the IXIT is also necessary for those sequences specified in T in order for those sequences specified in T to be executable.

- information to determine which capabilities are testable and untestable. Consider, for example, an implementation of a layer N protocol in the OSI reference model. Certain features of these protocols are not currently observable and, hence, are not testable. Such features can be:

  Pragmatically unobservable: Have a prohibitively high cost associated with their testing [Matt86].

  Theoretically unobservable: Not directly testable by definition [Matt86].

  Note that an untestable capability will imply a further pruning of T_i,

- etc.

We say that the IXIT parameterizes T_i such that T_i is now a selected parameterized executable test suite [ISO11].

4.6.1 Obtaining T_i for OSI Communications Protocols

For OSI, the Protocol Implementation Conformance Statement(PICS) makes specific claims for the implementation to be tested. We now show how specific claims of the PICS are used to prune a T for an OSI implementation to be tested. Again, each claim is classified in terms of the type(s) of claim(s) it provides(1,2,3,4, or 5) as given in Section 4.6. For the following, pruning of an alternative means removing this edge and its subsequent subtree from T. The PICS makes claims on:

Supported Functions(1,2,3) If an alternative specifies behavior for a function that is not supported, then prune the alternative.

Supported PDU's(1,2,3) If an alternative specifies behavior involving an unsupported PDU, then prune the alternative.

Initiator/Responder Capability(1,2,3,4) If an alternative specifies behavior for initiator/responder capability when such a capability is not claimed to be supported, then prune the alternative.

Supported Parameters(1,2,3,4,5) If an alternative specifies behavior involving a parameter which is **not** supported, then prune the alternative.

Negotiation Options Implemented(2,3,4) If an alternative specifies behavior involving a negotiation option which is not claimed to be supported, then prune the alternative.

Timer Values(1,2,3,5) If an alternative specifies behavior involving an unsupported timer value, then prune the alternative.

Choices made by implementor for particular procedures(4) Prune the nondeterministic choice alternatives not claimed to be supported.

Actual Values of Parameters that are supported(5) This requires adjusting alternatives to accomodate the values chosen by the particular implementation.

When we apply the above set of pruning rules, we obtain T_i. We note that since T was augmented to test for defensive behavior of any $I_i \in$ IMP, we do not prune any alternatives which specifically test for defensive behavior of I_i. Hence, for example, if T was augmented with a test for an illegal PDU, we do not prune this alternative when obtaining T_i unless that alternative has been claimed not to be supported.

4.7 Characterization of Conforming Implementations

As a summary of the material presented thus far in Section 4, we now characterize a set of conforming implementations IMP$'$ by a set of tests.
Let:

- IMP be a set of conforming and nonconforming implementations of S.
- IMP$' \subseteq$ IMP be a set of conforming implementations of S.
- C be the set of capabilities of S.
- $I_i \in$ IMP be an implementation of S. I_i claims conformance to $C' \subseteq$ C of S.
- P $\cup$D, P=$vtp_1, \ldots, vtp_r$, $D = dtp_1, \ldots, vtp_s$, be the set of test purposes and defensive test purposes corresponding to all $C_i \in$ C.
- $\{tc_1, \ldots, tc_r\}$ be the set of test cases fulfilling test purposes specified in P.

- $\{dtc_1, \ldots, dtc_s\}$ be the set of defensive test cases fulfilling defensive test purposes specified in D.
- $T = \{tc_1, \ldots, tc_r\} \bigcup \{dtc_1, \ldots, dtc_s\}$
- $T_i \subseteq T$ specify tests for C'

Then, $I_i \in IMP'$ if:

- No $c_j \in C - C'$ is a mandatory capability, and
- No $c_j \in C'$ violates conditions for inclusion of c_i in I_i,

and either

- I_i passes relevant tests in T,

or

- I_i passes tests in T_i.

Thus, any implementation which passes a set of tests which tests for all relevant valid and defensive behavior of a valid derivation of S is a conforming implementation of S. We now discuss the on_line and off_line solution strategies to the conformance testing problem.

4.8. On_Line Selection

In order to investigate the relative merits of the on-line selection strategy to the conformance testing problem, we discuss how on_line selection relates to several issues involved in performing conformance testing of a given I_i to its S. The intent here is to give some direction for a subject which is of current theoretical interest [ISO11].

We recall that the IXIT is used to parameterize an abstract CTS such that this CTS becomes executable. It is important to note than when applying either of the following on-line selection conformance testing strategies that we will discuss, the parameter settings given in the IXIT **do not change**. Although we may select and apply a subset of test sequences given in T via analysis during the test run of hitherto obtained results, we can not change an IXIT parameter if, for example, we decide that changing this IXIT parameter will reduce the time taken to execute

the test. Such redefinition of an IXIT parameter can influence assignment of test verdicts in such a manner that comparability of test results is compromised. Consider conducting a conformance test for a multiplexer which can handle from 8 to 16 channels. Let an IXIT parameter denoting the maximum number of supported channels be set to 16 at the beginning of the test run. If during the test run we discover that I_i seems to support only 8 channels, we may be tempted to reset the supported channels IXIT parameter to 8 if, for example, this allows the test driver to gather data from only 8 channels (and hence speed up the test run). In general, this can not be done because changing of the IXIT parameter can adversely influence comparability of results already obtained with results yet to be obtained. Hence, one identified possible shortcoming of the on–line selection strategy is temptation to change IXIT parameters based on results obtained and current context of the test run. Such a temptation occurs because obtained results are used to dynamically (i.e., on_line) select test sequences to apply to I_i in the following first strategy, obtained results may be used to dynamically select test sequences to apply to I_i in the following second strategy, and it may not be immediately apparent why these obtained results can not also be used to dynamically reset an IXIT parameter.

4.8.1 On_line Strategy 1

The literature [ISO11] defines a **capability list** to be the list of capabilities of an I_i as determined through execution of capability tests. We do not distinguish each test case TC_i in CTS in terms of it being a capability test, behavior test, conformance resolution test, etc.[ISO7]. Instead, each test case is intended to test a given $C_i \in$ C.

Let C'' $\subseteq$ C be the list of capabilities of I_i as determined through application of test cases in CTS. During an on–line selection CTS run, C'' is initialized to capabilities in S (i.e., C) and is updated as results of the test run are obtained. If the application of a given TC_i demonstrates that a $c_i \in$ C is not provided by I_i, then that c_i is deleted from C''. No other form of updating C'' is allowed. It is the current C'' which influences dynamic (i.e., on–line) selection of test cases to be applied to I_i: only those test cases in T intended to test capabilities in C'' are consequently applied.

There are several issues which should be addressed in prospectively using the above on–line selection strategy:

1. Should the capability list (i.e., C″) be used instead of the ICS? The above strategy may yield a capability list which differs from the ICS by being a list of observed capabilities, rather than claimed capabilities. Basically, the problem here is in assigning a verdict of conformance. What should be used in passing a verdict: ICS or C″? Should ICS be updated to C″?

2. If C″ and ICS differ, is retesting necessary if the set of capabilities in C″ conform to S? If functionality expressed in the ICS $\subseteq C''$, then there is no problem. Otherwise, in general, retesting is necessary if the ICS is not allowed to be updated by the implementor to C″. The reason for this is clear: we can not claim a set of capabilities which were tested and found to be absent. Retesting is not necessary if the ICS is allowed to be updated by the implementor to C″.

3. Does this on–line selection of test cases adversely influence comparability of results? Consider the following scenario. Assume two testers are given the same I_i. Tester **A** uses this first on–line test sequence selection strategy (i.e., does **not** use ICS), and tester **B** uses an off–line test sequence selection (i.e., uses ICS). Now, if tester **A** claims I_i is a conforming implementation and tester **B** claims the opposite, how can it be determined who is correct? The problem is that identical test sequences are not applied by testers **A** and **B**.

4. In different applications of the same CTS on a I_i, does this on–line selection guarantee a repeatable test? To obtain comparability of results (i.e., the same conforming/nonconforming verdict), identical test sequences **must** be applied. It is conceivable that such may not be the case when employing this on–line conformance testing strategy, particularly on different architectures constructed to provide a chosen abstract test method[ISO7].

5. This on–line selection testing strategy will not identify non–claimed capabilities at the earliest possible point in testing. Consider, for example, a capability of S which may at any point in testing express its functionality. Even though this capability was not supported in I_i, the tester would have to be capable of responding to it at any point during testing. Such readiness to respond may be realized in terms of a poll, for example. This polling introduces unnecessary overhead in performing the test since the capability was not claimed to

be present.

6. In employing this on_line selection testing strategy, it is necessary to keep track of C''. This is an overhead of employing this method.

7. This on_line selection strategy will identify extra capabilities of I_i which may be present but which were not to be tested since they were not claimed to be supported in the ICS. Not only can this introduce problems in the test reporting procedure (i.e., proprietary information), but also, these extra capabilities were tested and, hence, effort was expended (and wasted) to do so.

4.8.2 On_line Strategy 2

It is conceivable that the ICS can be used during the execution of a test to select test sequences to be applied to I_i in an on_line manner. To do so, the ICS would have to be represented in a form which is processable by a testing tool. This testing tool would process the ICS along with T to obtain each TC_i, one at a time, to be applied to I_i. At present there is, in general, no set proforma for an ICS[ISO7]. Assuming, however, that an ICS proforma becomes available[15], there are several issues to address in prospectively using this second on_line selection strategy:

1. If the ICS is used as an input to an on_line test sequence selection tool, this tool should have some ability to scrutinize the ICS to check for static conformance requirements [ISO7], inconsistencies, incompleteness, etc. The ability to automatically scrutinize the ICS is a potentially appealing aspect of applying this strategy. Not only is the ICS made use of, but also, it is made use of in an automated fashion. However, the manner in which scrutinization and representation of the ICS is accomplished is a subject for further research.

2. Conceivably, this on_line selection strategy enables an automated assignment of verdicts to test case results. Since the ICS information, which is used in result analysis, is in a machine processable form, results of a given test case application can be automatically compared to ICS information and a verdict assigned. There are two issues to consider here:

[15] A reasonable assumption since the ISO project 97.21.23 is expending much effort in developing such an ICS proforma specific to OSI.

(a) If behavior exhibited by I_i differs from ICS but still conforms to S, is retesting necessary? Can the ICS be updated to reflect the actual behavior of I_i? This is a problem similar to the first on_line selection strategy in which C'' differed from the ICS yet still conformed to S.

(b) If behavior exhibited by I_i conforms to ICS but not to S, can testing continue? This is a potential problem with this second on_line selection strategy because such inconsistencies between the ICS and S are normally determined prior to on_line testing in a static conformance review [ISO7].

3. Normally, an ICS can not be updated during execution of a CTS. However, since the ICS is in a machine processable form, it is possible to do so using this strategy. There are two questions here. First, should such updating of the ICS be allowed? Obviously, an ICS which truly represents expressed functionality of I_i is desirable. Second, when is updating of the ICS allowed? If, for example, an optional capability was claimed to be supported but was not supported, then this type of updating may be allowed. If, though, a mandatory capability is claimed and not supported, such updating **should not** be allowed. Implications for assignment of verdicts of conformance of I_i to S may make such updating of the ICS undesirable.

4. It is possible using this selection strategy to automate the entire static conformance review process. Obviously, a processable ICS may enable development of tools for analysis of static conformance requirements. This is a potential merit of this on_line selection strategy and is an area of current research.

5. A necessary overhead of this strategy is processability and storage of the ICS. As well, an immediately apparent problem is representation in a machine processable form of conditions upon which conditional capabilities depend. Such conditions can be virtually anything. If these conditions can not, in general, be represented in a machine processable form, then the potential benefit of automated verdict assignment can not, in general, be performed. It is of course possible that test operator input may be necessary in such cases. This is also a subject for further research.

6. Since this method makes use of the ICS, repeatabilty of tests and comparability of test results are likely attainable. Of course, updating of the ICS

during test execution can adversely affect comparability and repeatability of test results. With this method, though, since the set of test sequences to be applied to I_i are not known before test execution, analysis of test results may be more difficult since there is some question of which test sequences are to be applied. These issues are subjects for further research.

7. Unlike the first on_line selection strategy, this strategy **will not** identify non-claimed capabilities of I_i. This has positive implications for test reporting procedures.

8. Use of this method may allow application of many test sequences before discovery of an error which could have been caught in a static conformance review. This potential problem is avoided if the static conformance review process is automated. At present, though, such automation is not available and minimization of test sequences to be applied is not optimal when using this method.

Discussion on the above strategies highlights some important considerations in using the on_line selection strategy. The questionable use of the ICS (i.e., lack of use of the ICS) in the first strategy leads to some serious questions of applicability of the method. Potential automation is the greatest possible benefit in the second on_line selection strategy. The entire concept of dynamic (i.e., on_line selection) testing is a topic of active research[ISO11]. It is hoped that the points mentioned above will contribute to this research.

4.9 Off_Line Selection

We now investigate the relative merits of the off_line selection strategy to the conformance testing problem. Current conformance testing procedures employ off_line selection [ISO11][ISO12][Matt86]. We will review some of these procedures and comment on why they are employed.

In [ISO12], two basic requirements are placed on the tester and implementor before a test for conformance of a given I_i can begin:

1. The implementor **must** supply an ICS prior to applying for a conformance assessment of I_i.

2. Enough information must be exchanged between the tester and implementor to guarantee an accurate conformance assessment of I_i (i.e., the IXIT must be produced).

It is the implementors sole responsibility to submit an ICS which clearly specifies claimed capabilities of I_i. The IXIT, though, is the result of collaboration between the tester and the implementor. This is logical since the implementor is requesting testing of an I_i with specific capabilities while both the implementor and tester need to establish particulars to enable testing (i.e., implementation system characteristics, test architecture characteristics and requirements, etc.).

It should be clear that an IXIT is always necessary to enable testing, whether an on_line or off_line selection strategy is employed. Unlike the above on_line selection strategies, though, the tester will not be tempted to change any IXIT parameters using the off_line strategy. The reason for this is that off_line selection implies a batch mode of testing: all test sequences to be applied are selected before running the test and, hence, minimal test operator interference is necessary. Of course, the test operator will normally monitor progress of the test and halt the test, if necessary.

The ICS is used in off_line selection to select test sequences to be applied to I_i. Its functions prior to on_line testing are to:

1. permit a static conformance review of the implementation.

2. enable selection of test cases from CTS for execution [ISO12].

A static conformance review is a paper analysis [ISO12] of the ICS which ensures any static requirements necessary to enable functioning of I_i are met. Such static requirements can be, for example, ranges of values supported for various parameters, default settings for various parameters, constants, interdependencies between capabilities, etc. This use of the ICS is necessary to enable testing and is a merit of the off_line strategy when compared to strategy 1 for on_line selection. [16]

The basis of the off_line selection strategy is selection of test cases prior to test execution. The ICS is used to obtain this set of test cases (i.e., T_i). As claimed in Section 4.5, test cases are functions of test purposes which are, in turn, functions

[16] It is not immediately apparent how the on_line solution strategy 1 employs the ICS for such reviews.

of capabilities of I_i. The ICS defines a subset of the capabilities of I_i (i.e., C') and, hence, a subset of test cases in CTS. One can forsee many benefits to this selection strategy:

1. The set of test sequences to be applied to I_i is minimized at the earliest possible time. Hence, the test driver need not store any superfluous information intended to help select a subset of test sequences on_line.

2. The information in the ICS is made use of and scrutinized prior to testing. This both optimizes the use of information available and enforces a review of the ICS to check for inconsistencies, omissions, etc.

3. There is no question prior to testing of which set of test sequences are to be applied to I_i. This has positive implications for analysis of test results, repeatability of tests, etc.

4. The tests can be run with minimum interference from the test operator. Essentially, tests can be conducted in batch mode.

5. Effects of minimizing test sequences to be applied at the earliest point are multiplied when retesting becomes necessary.

6. Only that set of capabilities claimed to be supported will be tested. Thus, unlike on_line selection, there is no problem with test reporting procedures and there is no unnecessary testing performed.

It should be clear that use of the ICS prior to test execution is desirable to achieve an analyzable, repeatable, and minimal test. Practice has shown, however, that even use of the ICS may not always be sufficient to conduct the test efficiently. In[Matt86], experiences of operating a real conformance test center show that not only is static testing (i.e., static conformance reviews and off_line selection) an important function of running such a center, but prioritization of test cases to achieve the most significant coverage is an economic necessity. Hence, even the subset of test cases implied through use of the ICS often needs to be pruned.

The off_line selection strategy seems to be the current state of the art of conformance testing practice. It is feasible, however, to combine ideas from both off_line and on_line solution strategies. Such a hybrid method may optimize various phases

of the conformance testing process. This remains to be seen as experience in the field progresses.

5 CONCLUSIONS AND FUTURE WORK

This thesis has presented a framework in terms of a compressed tree model (CTM) which serves as a basis through which characterization of those issues specific to the process of conformance testing is presented. CTM models the system behavior expressed by a specification S in a compact form and emphasizes points of by abstracting linear sequences of events expressed in S in terms of a compressed tree.

Given a specification S using Finite State Machines, ESTELLE, CCS, or LOTOS, we define mappings to a corresponding CTM representation. This CTM is then used in deriving skeletons for test sequences which characterize the set of conforming implementations of S. Heuristics for augmenting this set of test sequences are presented to obtain a set of test sequences T. Test sequences in T can be classified as testing valid (T_A) or defensive (T_B) behavior of an implementation I_i. This characterization of the test sequence derivation process is a contribution towards the development of the theory of conformance testing.

We define an implementation of a specification I_i as the implementors view of the specification he/she has implemented, S_i. We then characterize the conformance testing problem as showing that either I_i is a valid implementation of S or, alternatively, showing that S_i is a valid derivation of S, assuming I_i is a conforming implementation of S_i. We further use the concepts developed both in the mappings and in the characterization of the conformance testing problem to characterize the conforming set of implementations of a given specification. This is the first attempt at showing such characterizations.

Given our characterization of the conformance testing problem, we then show that there are two possible approaches to solving the conformance testing problem:

1. We can apply a generic set of test sequences T to I_i or, alternatively,

2. we can apply a specific set of test sequences T_i derived from T to I_i. A special case of this second alternative is to derive T_i from TT and apply T_i to I_i. This case, however, does not specify tests for defensive behavior of I_i. This characterization of solution strategies to the conformance testing problem and our discussion of these on_line and off_line solution strategies are also contributions towards the development of the theory of conformance testing.

Throughout the thesis, algorithms and procedures are presented for solving various problems studied during thesis research. It is hoped that these procedures will contribute to development of tools for both further research and development of testing and specification tools.

This thesis has also identified several topics for further research in the general area of conformance testing. These include:

- The development of test sequence derivation tools which employ the mapping techniques defined in Section 3. These tools would aid in the development of conformance test suites for implementations whose specifications are available in the FSM, ESTELLE, CCS, or LOTOS formalism. Much effort is being thrust into advancement of development and use of FDT's. The University of Ottawa has taken a leadership role through development of its LOTOS interpreter [Obai86]. Any work going on in FDT's is related to the work of this thesis in that it is hoped that concepts developed for particular FDT's used in this thesis are applicable not only to these FDT's, but also to other FDT's. Test sequence derivation from FDT system specifications is a concern of this thesis which incorporates the work of FDT specialists.

- A solution to the problem of recognizing repetitive behavior subexpressions in LOTOS specifications would aid in the derivation of test sequences from LOTOS based specifications. Work on this problem is under way at the University of Ottawa.

- Algorithms to vary parameter values in T_A and T_B and to obtain test sequences for T_B such that coverage is maximized are problems of crucial importance. Work on the parameter variation problem is widespread and on-going.

- Algorithms to select test sequences from T specific to a given implementation are important for pragmatic reasons. Work on this selection problem is also widespread and on-going.

- A test designer should make use of grey choice points within a specification such that a test architecture can be constructed which allows for execution of adaptive test cases. Specifically, information in TT can be made use of in components of the test system. The exact manner in which such information can be used is a subject for further research.

- The exact manner in which capabilities imply subtrees of T transitively through test purposes and a formalization of the **manner** through which these test trees are attached together to construct test cases are subjects for further research.

- The recognition of capabilities in a specification is vital for obtaining an adequate set of test purposes (and defensive test purposes). Some initial work on a representation scheme for system specifications such that system capabilities are easily recognized is ongoing at the University of Ottawa.

- A formal definition of a relation through which a capability can imply both test purposes and defensive test purposes, and a formal definition of the exact relation between test purposes and defensive test purposes may aid in the development of efficient solutions to the test sequence derivation and selection problems.

- Development of a proforma ICS is a priority in ISO TC97 SC21 project 97.21.23. Availability of such an ICS has implications for development of efficient solutions to the test sequence selection, result analysis, and static conformance testing problems. (Semi-) automated solutions to these problems are feasible if such an ICS proforma becomes available.

- Pragmatic considerations in conducting tests involving multiple connections in a distributed system are important to enable efficient testing and are subject to further study.

In our discussion of the on_line versus off_line test sequence selection strategies for the conformance testing problem, topics for future research were also identified. These include:

- The possible use of a capability list instead of an ICS has both possible benefits and problems. Specifically, the question of how use of a capability list influences comparability of test results, repeatability of tests, and necessity for re–testing are topics for further research.

- Given a machine processable ICS, automated scrutinization for static conformance requirements is a possibility for furthering the art of conformance

testing. At present, this seems to be a very promising area for further research.

- Assuming that a machine processable ICS is available, the question of when updating of the ICS should be permitted, if at all, is a problem of interest. If such updating is allowed, a set of rules for doing so should be formulated. As well, a mechanism through which to represent any type of conditional capability in such an ICS will aid in the selection, execution, and analysis processes.

- A method through which to prioritize test cases will undoubtedly increase the efficiency of testing. This, of course, is a standard testing problem and is a topic of on-going research.

- It is conceivable that a hybrid between the on-line and off-line selection strategies can be employed. This is a problem of on-going research as experience in the field progresses.

The state of the art of conformance testing is rapidly advancing. At present, a clear framework which formalizes those issues specific to conformance testing and characterizes conforming implementations is unavailable. It is hoped the tree model presented in this paper, the subsequent characterization of the conformance testing problem in terms of this model, and the characterization of a conforming implementation will provide such a framework.

References

[ACT 85] Jan de Meer, Hahn–Meitner–Institue Berline GmbH, Klaus Peter Hasler, Tutorial on ACT ONE, Draft Copy, Technische Universitaet Berlin, January 1985.

[Chow78] Tsun S. Chow, "Testing Software Design Modeled by Finite-State Machines", IEEE Transactions on Software Engineering, SE-4, No. 3, 1978, pp. 178–187.

[Good 75] John B. Goodenough and Susan L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Trans. on Software Engineering, SE–1, No 2, 1975, pp. 156-173.

[Howd80] W.E. Howden,"Functional program testing", IEEE Transactions on Software Engineering, Volume SE–6, No. 2, 1980, pp. 162–169.

[ISO1] International Standards Organization, Open Systems Interconnection – Basic Reference Model, IS 7498, 1983.

[ISO2] International Standards Organization, Open Systems Interconnection - Illustration of a method to derive testsequences from a formal description, ISO/ TC 97/SC 21/ WG1 Project 97.21.23 Expert Contribution, Netherlands, Sept. 1985.

[ISO3] International Standards Organization, On the use of LOTOS for the specification of OSI test suites, ISO/ TC 97/ SC 21/ WG 1 Project 97.21.23, Expert Contribution, UNIPREA, September 1985.

[ISO4] International Standards Organization, Process Specifications, their implementations, and their tests. ISO/ TC 97/SC 21/WG 1 Project 97.21.23, Expert Contribution, NN1 and UNI/UNIPREA, March 1986.

[ISO5] International Standards Organization, Transport Protocol Specification, ISO/ TC 97/ SC 16 N 1169, DP 8073, June 1982.

[ISO6] International Standards Organization, Transport Service Definition, ISO/ TC 97/ SC 16 N 1162, DP 8072, June 1982.

[ISO7] International Standards Organization, Working Draft for OSI Conformance Testing Methodology and Framework, ISO/ TC 97/ SC 21 N 410, September 1986.

[ISO8] International Standards Organization, Draft Proposal of Estelle, A FDT based on extended state transition model, ISO/ TC 97/ SC21 DP 9074, September 1985.

[ISO9] International Standards Organization, Open Systems - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, ISO/ TC 97/ SC 21 DP 8807, July, 1986.

[ISO10] International Standards Organization, Working Draft on Testing and FDT's, ISO/ TC 97/ SC 21 WG 1 N1515, Sept. 1986.

[ISO11] International Standards Organization. Working Draft of OSI Conformance Testing Methodology and Framework, Part 5: Test Laboratory Operations, ISO/ TC 97/ SC 21/ N 1526 REV, June, 1987.

[ISO12] International Standards Organization, Working Draft of OSI Conformance Testing Methodology and Framework, Part 4: Requirements on Suppliers and Clients of Test Laboratories, ISO/ TC 97/ SC 21/ N 1526 REV, June, 1987.

[Kanu86] B. Kanungo, L. Lamont, R.L. Probert, H. Ural. A Useful Finite State Machine Representation for X.25 Conformance Testing. Proceedings of the 6th IFIP/WG6.1 Int. Workshop on Specification, Verification, and Testing. Montreal, June 1986.

[Linn86] J.P. Favreau and R.J. Linn Jr., "Automatic Generation of Test Scenario Skeletons from Protocol Specifications Written in Estelle", Proceeding of the 6th IFIP/WG6.1 Int. Workshop on Specification, Verification, and Testing, Montreal, June 1986, pp. 135–150.

[Matt86] Robert S. Matthews, K.H. Muralidhar, Michael K. Schumacher, "Conformance Testing: Operational Aspects, Tools, and Experiences", Proceedings of the 6th IFIP/WG6.1 Int. Workshop on Specification, Verification, and Testig, Montreal, June 1986, pp. 191–202.

[Miln80] Milner, R. "A Calculus of Communicating Systems", Lecture notes in Computer Science, Springer- Verlag, Berlin 1980.

[Myer79] Glenford J. Myers, The Art of Software Testing, John Wiley and Sons, Inc, 1979.

[Piat80] T.F. Piatkowski, "Remarks on the feasibility of validating and testing ADCCP implmentations," Proceedings of Trends and Applications Symposium(NBS), Garthersburg, MD. 1980.

[Obai86] J.P. Briand, M.C. Fehri, L. Logrippo, and A. Obaid, "Structure and Use of a LOTOS Interpreter", Department of Computer Science, University of Ottawa Technical Report, TR-86-09, September 1986.

[Rayn87] D. Rayner, OSI Conformance Testing, Tutorial, Proc. of 7th IFIP/WG6.1 Int. Workshop on Specification, Verification, and Testing, Zurich, June 1987.

[Rev 83] G.E.Revesz. "Introduction to Formal Languages". McGraw-Hill, 1983.

[SaBo84] B. Sarikaya and G. v. Bochmann, "Synchronization and specification issues in protocol testing", IEEE Transactions on Communications, Volume 32, No. 4, 1984, pp. 389–395.

[SaBo85] B. Sarikaya and G.v. Bachmann, "Obtaining Normal Form Specifications for Protocols", Proceedings of COMNET 85, Budapest Hungary. Oct 1985, pp. 6.133 – 6.149.

[SaBo87] Behcet Sarikaya, Gregor V. Bochmann, and E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Transactions on Software Engineering, Volume SE–13, No. 5, May 1987. pp. 518–531.

[Sopp86] Margaret Soppe, "A tool for user guided test suite derivation from formal specifications", Master of Computer Science Thesis, Twente University of Technology, The Netherlands, 1986.

[Stee86] Chris Steenbergen, Conformance Testing of OSI Systems, Master of Computer Science Thesis, Twente University of Technology, The Netherlands, 1986.

[Ural87] Hasan Ural, "A test derivation method for protocol conformance testing", Proc. of 7th IFIP/WG6.1 Int. Workshop on Specification, Verification, and Testing, Zurich, June 1987, pp. 347–358.

[Wal83] P.J.Walsh, "An Analysis of Test Case Selection", Phoenix Second International Conference on Computers and Communications, 1983, pp. 437–441.

[Wey 80] Elaine J. Weyuker and Thomas J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Trans. on Software Engineering, Vol. SE–6, No.3, 1980, pp.236-246.

[Zimm80] "OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection", IEEE Transactions on Communication, Vol Com–28, No. 4, 1980, pp. 425–432.