Cause-Effect Validation of Requirements for Distributed Systems

By

Khénaïdoo Nursimulu

THESIS

Submitted to the School of Graduate Studies and Research
in Partial Fulfillment of the Requirements for the

Degree of

Master in Computer Science

Department of Computer Science - University of Ottawa
(Ottawa-Carleton Institute for Computer Science)

ISBN 0-315-96011-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and gratitude to my thesis supervisor Professor Robert L. Probert for his support, encouragement, understanding and fruitful discussions throughout my thesis research. I would again thank Professor Probert for his generous financial support all throughout my studies. I would also like to thank Professor Hassan Ural and Professor Jean-Pierre Corriveau for their helpful suggestions which by no doubt have improved the quality of this thesis.

My deepest respect goes to my parents, brothers and sisters who gave me the moral support and encouragement to further my education in Canada.

Finally, I gratefully acknowledge financial support by the Telecommunications and Research Institute of Ontario (TRIO) and the Canadian Institute of Telecommunications Research (CITR).

ABSTRACT

In this thesis we propose a requirement validation technique based on the Cause-Effect Graphing (CEG) technique. Originally, the CEG technique, proposed by Elmendorf [Elme73] as a black-box testing technique, was used to derive test cases from an informal specification to test the corresponding implementation. The test cases obtained from this method is correct only if the requirements are correct in the first place. Therefore, a technique is required to validate the requirements themselves. The key contribution of this thesis is a framework and methodology for employing the Cause-Effect Graphing approach as a technique for validation of specifications especially in the context of distributed systems.

This technique has also been extended to apply to **distributed software.** The extension has given rise to the Extended Cause-Effect Graphing (ECEG) technique where we use **scenarios** as an effective means of checking network behaviors against customer expectations. Our ECEG approach has been validated by case studies of telephony features. In these case studies, we give some examples of instances of undesirable scenarios, such as feature interactions, which were derived directly from customer specifications, thus demonstrating that major problems in requirements definitions can be detected and corrected even before software design begins.

This validation approach is based on heuristics which in turn are based on a limited survey of the practices in the application domain. A key benefit of this approach is that scenarios can be derived **automatically** from the CEG based on path sensitization, an idea borrowed from the world of Integrated Circuits verification methods. A tool prototype has also been developed on the Macintosh and Sun Workstation environments.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

Chapter 1

Introduction

1.1 Background

Today it is generally accepted that verification and validation activities should be performed at each phase of the software life-cycle. It is also widely recognized (but not necessarily widely adopted in practice) that the development of reliable software and systems should proceed from precise, formalized specifications of the system requirements. In that context, *system validation* appears to be of great importance already at the early stage of the software life-cycle. Its main objective is to reveal specification, design, and implementation errors in the system for the purpose of improving system quality. The validation activity is composed of two separate activities, namely, *verification* and *testing*. During verification the system is validated using formal and logical means while testing consists of executing the system in a controlled environment over a finite set of tests. The testing activity can further be decomposed into two major categories which relate either to black-box testing or white-box testing. Test cases in black box testing are derived from the system specification and they test for the features and external behaviors of the system. In white-box testing, the test cases are derived from the internal logic of the design and implementation of the system.

One of the most difficult and error-prone phases of software engineering is the phase in which customer requirements are captured and documented. Studies [Hall91, Glas81] have shown that incomplete, inconsistent, and erroneous specifications of system requirements are the cause of "persistent" errors, errors which escape verification and validation during design and development, and which produce costly problems in the field. Various authors, [Parn72, BaGo79, LiZi77, LiBe79, Meye85, TuMa87], have proposed specification criteria in order to improve the specification phase. Among the criteria two that are pertinent to the specification process are *completeness* (i.e capturing all the user requirements) and *minimality* (i.e capturing nothing but the user requirements). Many Formal Description Techniques (FDTs) support model checking, a means of testing for these

properties on a mathematical basis (e.g [BoMi90]). However, very few pragmatic guidelines exist for deriving effective, precise specifications of reliable software, and for subjecting the specifications to systematic testing.

The process wherein a formal description is constructed from the requirements of a given system is not always documented and is usually "informal" in nature [Geha82]. This phase is very important in the sense that the sooner defects are spotted in the system under development the less-costly and the easier it will be to correct [Broo87]. User involvement is required in this phase to promote understanding and validation of the specified system [Haml88]. The test data generated during the validation phase should be presented in a simple and attractive fashion to the user as he/she is more aware about the intended objectives of the future system than the computer specialist.

1.2 Objective and Contributions of this Thesis

In this thesis we propose a requirement validation technique based on the Cause-Effect Graphing technique. Originally, this black-box testing technique, proposed by Elmendorf [Elme73], was used to derive test cases from an informal specification to test the corresponding implementation. The test cases obtained from this method will be correct only if the requirements are correct in the first place. Therefore, a technique is required to validate the requirements itself.

Our thesis contribution is twofold: first we present a framework and methodology for employing the Cause-Effect Graphing approach as a technique for validation of specifications especially in the context of distributed systems. Our second contribution presents some drawbacks [Wals83, AdGr83, Arab86, Woit92] in Myers' Cause-Effect Graphing technique as well as possible resolutions to them.

This technique has also been extended to apply to **distributed software** which can be modelled by SDL (Specification Description Language), a CCITT Recommended language for describing distributed (communications) systems, and validated by case studies of telephony features. From these studies, we observed that communications systems require particular CEG constructs; for example, some effects may need to be reused as causes,

leading to compound examples (called *scenarios* †) of key interactions among subscribers and network devices. These **key scenarios**†† can be used as an effective means of checking network behaviors against customer expectations. We give some examples of instances of undesirable scenarios, such as feature interactions, which were derived directly from customer specifications, thus demonstrating that major problems in requirements definitions can be detected and corrected even before software design begins.

This validation approach is based on heuristics which in turn are based on a limited survey of the practices in the application domain. A key benefit of this approach is that key scenarios can be derived **automatically** from the CEG based on path sensitization, an idea borrowed from the world of Integrated Circuits verification methods. A tool prototype has been developed on the Macintosh and Sun Workstation environments.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 describes briefly Myers Cause-Effect Graphing technique, its drawbacks, as well as possible resolution of these drawbacks. Chapter 3 shows how the CEG technique can be adapted for telecommunications requirements. In this chapter an analysis of natural language specifications of telecommunications requirements is also given. Chapter 4 shows how telecommunications features and services can be represented by using an extension of the CEG technique, while chapter 5 presents a methodology for validating telecommunications features and services whose representations are given as in chapter 4. The conclusion of this thesis along with areas for further study follow in chapter 6. Appendix A illustrates the automatic derivation of a decision table from an ECEG as well as the prototype tools which have been developed to promote both the CEG and ECEG techniques.

---

† A *scenario* is a meaningful sequence of causes and effects used for testing purposes

†† Key scenarios denote a subset of scenarios more liable to find errors in the implementation of the system

Chapter 2

Myers Cause-Effect Graphing (CEG) Technique : A Review and Analysis

Cause-Effect Graphing is basically a hardware testing technique adopted for software testing by Elmendorf [Elme73]. Since then, several researchers have further developed this method [Elme74, Elme75, Myer76, Myer79, Howd80, Adgr83, Wals83, Prob85, Arab86]. A Cause-Effect Graph is a formal representation into which a natural language specification of a system or subsystem's behavior is translated. The graph is actually a Boolean logic circuit where a simple graphical notation is used.

In software testing, Cause-Effect Graphing (CEG) is classified as a black-box testing technique (no explicit reference to code or design) where only the external behavior of the implemented system is tested. In this technique a cause-effect graph is traced to build a decision table which is later on converted into test cases. Myers proposed an algorithm based on the CEG technique to obtain test cases that he claims will select in a systematic way a ***high-yield*** † set of test cases [Myer79]. This technique has a beneficial side effect in pointing out points of incompleteness and ambiguity in the specification [Myer79]. Finally, this method is the only black-box testing technique that handles the effects of combinations of causes while other black-box testing techniques isolate the individual contribution of one cause at a time.

The chapter is organized as follows. Section 2.1 reviews how Myers uses the CEG technique to derive test cases from natural language specifications. Section 2.2 gives an analysis of Myers' approach. Finally, possible solutions to drawbacks in Myers' approach are presented in Section 2.3, and assessed in Section 2.4.

† High-yield set of test cases denote a set of test cases good enough to detect major errors found in the implementation.

2.1 Overview of Myers technique

2.1.1 Test case derivation process

Myers uses the CEG technique to derive test cases from natural language specifications. The following illustrates his test derivation process :

1.Large specifications should be broken down into "workable" parts. This is necessary because CEGs become very large and difficult to manage when used on large specifications. The size of the workable part is determined by the nature of the specification and user convenience. For instance, when testing a time-sharing system, a "workable part" might be the specification of an individual command. Even if such decomposition means a loss of global representation, it appears to be the only method which promotes user understanding [Myer79].

2. The causes and effects in the workable parts of the specification are identified. A ***cause*** is a distinct input condition or an equivalence class of input conditions which can be controllable. A cause can be interpreted as an entity which brings about an internal change in the system. An ***effect*** is an output condition or a system transformation which is observable. An effect can be a state or message resulting from a combination of causes. Causes and effects are identified by reading the specification word by word and underlining words or phrases that describe causes and effects. Each cause and effect is assigned a unique identifier (formed by alphanumeric characters, for example).

3. The semantic content of the specification is analyzed and transformed into a Boolean graph linking causes and effects by means of ***and*** nodes, ***or*** nodes, and ***not*** edges. This is the Cause-Effect Graph.

4. The graph is annotated with constraints describing dependencies between causes and/or effects because of syntactic or environmental constraints.

5. By methodically tracing (manually or automatically) the CEG from effects to causes, the CEG in converted into a limited-entry decision table. Each column in the table represents a test case.

6. The columns in the decision table are converted (manually) into test cases.

More details of each of these steps will be given in the following sections.

2.1.2 Cause-Effect Graph Basic Notation

The notation used in the CEG technique to relate causes to effects is shown in Figure 2.1. The Cause-Effect Graph is read from left to right. The graph consists of nodes representing causes, effects and intermediate nodes (intermediate nodes are used to link causes to effects when the graph becomes too complicated or when we cannot link causes to effects directly). Causes are represented in the graph by the leftmost nodes, i.e nodes that do not have nodes on their left connected to them. Conversely, the rightmost nodes, i.e nodes that do not have nodes on their right connected to them, represent the effects. Nodes that are connected both on their left and right to other nodes are intermediate nodes.

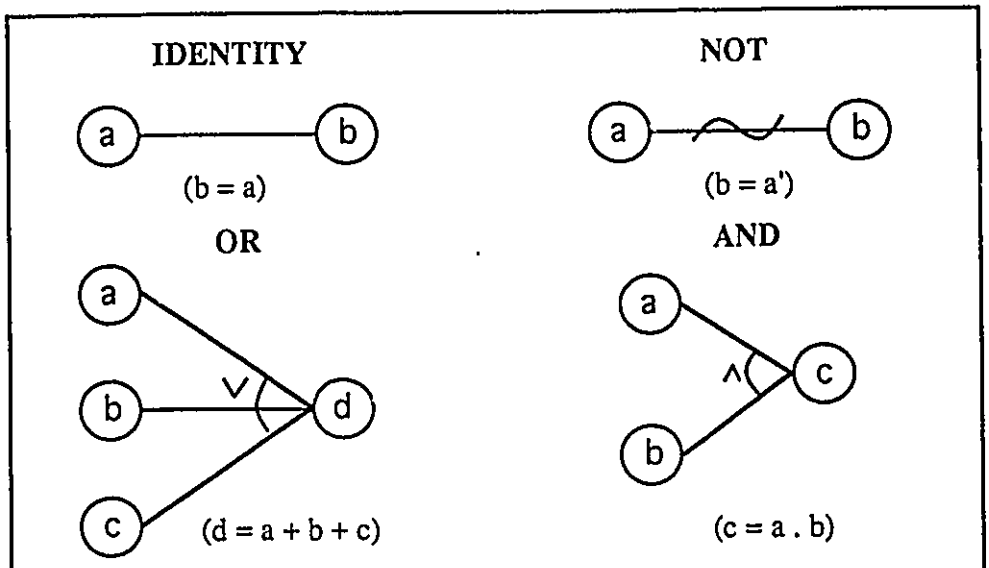


Figure 2.1 Cause-Effect Graph symbols relating causes to effects

Four Boolean functions (***Identity, Not, Or*** and ***And***) determine how nodes are linked in the CEG. The *identity* function states that if node *a* (Figure 2.1) is present (usually denoted by the value 1) then node *b* is present: else node *b* is absent (usually denoted by the value 0). The *not* function states that if node *a* is 1 then node *b* is 0; else node *b* is 1. The *or* function states that if node *a* or *b* or *c* is 1 then node *d* is 1; else node *d* is 0. The *and* functions states that if both nodes *a* and *b* are 1 then node *c* is 1; else node *c* is 0. The first

two functions have only one input while the last two functions are allowed to have any number of inputs. In Boolean notation "1" (present) is equivalent to "true" while "0" (absent) is equivalent to "false".

As an illustration of the above notation consider the following informal specification. "*A student can either be enrolled at University of Ottawa, at the University of Carleton, or at the University of Québec in Hull. If he/she obtains grade A+ in Mathematics and he/she is at University of Ottawa or at University of Quebec in Hull, then the computer system at the university will allow him/her to take the skating course in winter. If he/she obtains grade A+ and he/she is at University of Carleton then the system will allow him/her to take the skiing course in winter. If his/her grade is less than A+ in any university the system will order him/her to take the skate polishing course in winter.*"

From the above specification the following causes are obtained and identified by a unique identification number:

1. Student is enrolled at university of Ottawa
2. Student is enrolled at university of Quebec in Hull
3. Student is enrolled at university of Carleton
4. Student obtains A+ in Mathematics

Note that there is no cause representing the absence of an event as this can be represented using the *not* function. Moreover, each cause represents an atomic event. Similarly, the following effects are obtained and identified by a unique identification number (they should be different from the cause identification number too):

100. Student may take the skating course in winter
101. Student may take the skiing course in winter
102. Student has to take skate polishing course in winter

Note that the effects are all observable because after taking into account the cause conditions the system will display the effect on the computer screen for the student to know the possible courses in which he/she may or has to register. Figure 2.2 shows the CEG obtained from the lists of causes and effects. Intermediate nodes 10 and 11 have been added to the graph to facilitate the construction of the CEG. The CEG in Figure 2.2, although a correct representation of the specification does contain one impossible combination of causes

- causes 1, 2 and 3 cannot to be set to 1 (present) simultaneously. This situation is represented on the CEG by using ***constraints***, explained in the next section.

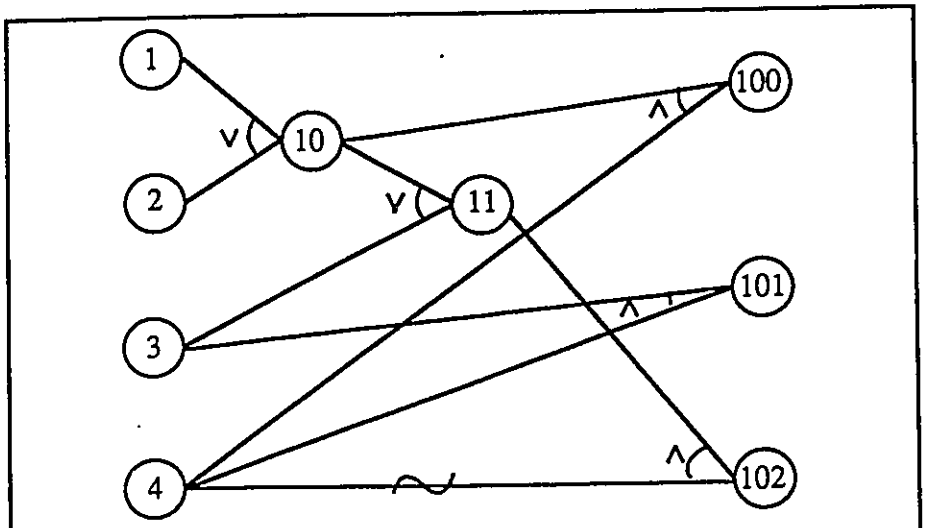


Figure 2.2 Example of a Cause-Effect Graph

2.1.3 The constraints

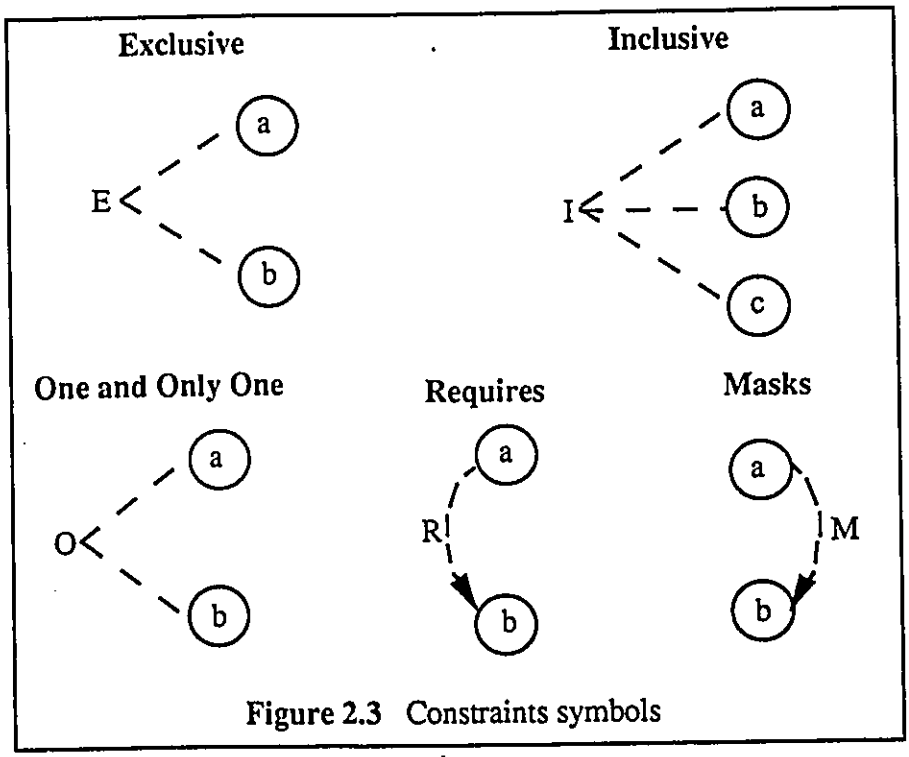


Figure 2.3 Constraints symbols

In most specifications, certain combinations of causes are impossible in practice because of syntactic or environmental considerations (e.g a student cannot obtain two grades

for the same course). To account for these, the notation in Figure 2.3 is used. The first four constraints (E, I, O, R) apply to causes while the M constraint applies to effects. The *E* constraint states that it must always be true that at most one of causes *a* and*b* can be 1 (*a* and *b* cannot be 1 simultaneously). The *I* constraint states that at least one of causes *a*, *b*, and *c* must be 1 (*a*, *b*, and *c* cannot be 0 simultaneously). The *O* constraint states that one, and only one of causes *a* and *b* must be one (*a* and *b* cannot be both simultaneously 1 or 0). The *R* constraint states that, for cause *a* to be 1, cause *b* must be 1 (i.e it is impossible for *a* to be 1 and *b* to be 0). The *M* constraint (applicable to effects only) states that if effect *a* is 1, effect *b* is forced to 0 (i.e the presence of one effect may mask the presence of another effect. This usually happens when more than one effect is present simultaneously).

In the previous example it was stated that a student cannot be enrolled at more than one university at a time. Therefore, the E constraint is applied to causes 1, 2, and 3. The resulting CEG is given in Figure 2.4.

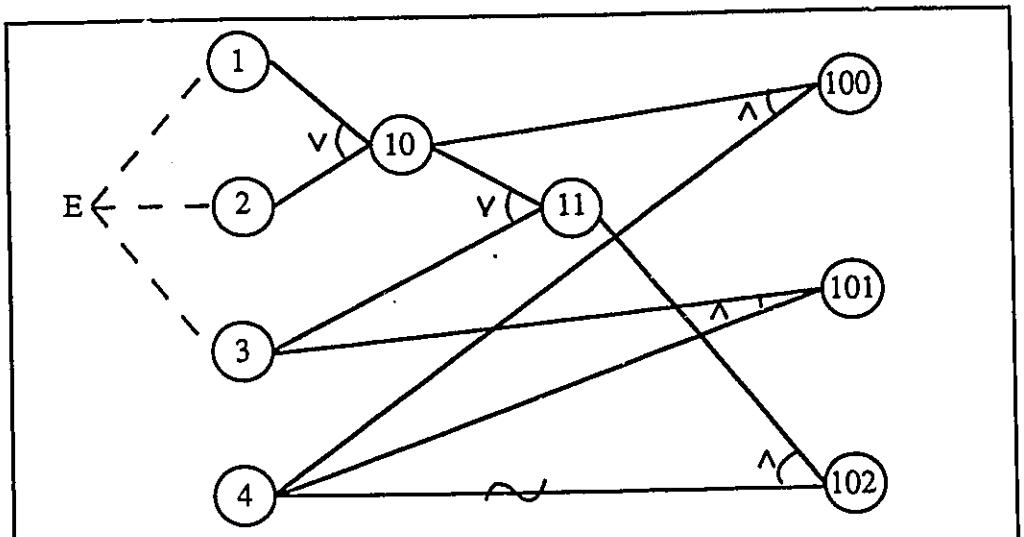


Figure 2.4 Cause-Effect Graph with the Exclusive constraint

2.1.4 Deriving a Decision Table from a Cause-Effect Graph

The next step after having built the Cause-Effect Graph is to derive a decision table from the graph. A decision table consists of two parts : the conditions and the actions. In each column of the decision table, the first rows represent the conditions to be satisfied in order to obtain the actions in the last rows. The presence of an action or a satisfied condition is represented by "1" and the absence of an action or an unsatisfied condition is represented

by "0". In the decision table obtained from the CEG, each column represents the conditions to be satisfied for a specific effect to occur. Since the number of columns in a decision table correspond to the number of derived test cases and our aim is obtain the fewest test cases possible we should, therefore, have a well-designed procedure to obtain a minimal-entry decision table. The procedure is as follows :

1. Select an effect to be in the present (1) state.
2.Tracing back through the graph, find a set of combinations of causes (subject to the constraints) that will set this effect to 1 (this step is expanded below).
3. Create a column in the decision table for each combination of causes (This step is given in detail on page 11)
4. For each combination, determine the states of all other effects and place these in each column.

In performing step two, three considerations are

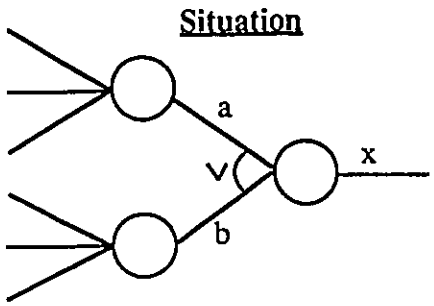
1. When tracing back through an *or* node whose output should be 1, never set more than one input to the *or* to 1 simultaneously. This is called *path sensitizing*. Its objective is to avoid not detecting certain errors because of one cause masking another cause.

2. When tracing back through an *and* node whose output should be 0, all combinations of inputs leading to a 0 output must, of course, be enumerated. However, if one is exploring the situation where one input is 0 and one or more of the others are 1, it is not necessary to enumerate all conditions under which the other inputs can be 1.

3. When tracing back through an *and* node whose output should be 0 only one condition where *all* inputs are 0 need be enumerated. (If the *and* is in the middle of the graph such that its inputs come from other intermediate nodes, there may be an excessively large number of situations under which all of its inputs are 0) [Myer79]

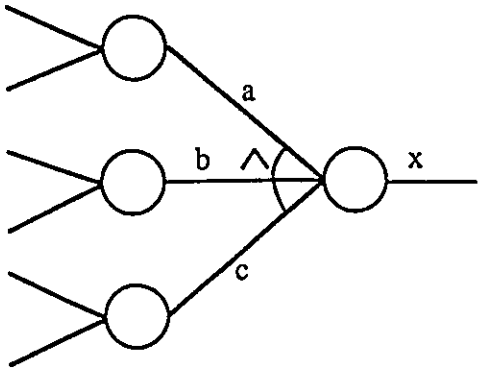
Figure 2.5 summarizes the above considerations. According to Myers these considerations have an important purpose : to lessen the combinatorics of the graph. They eliminate situations that tend to be ***low-yield*** † test cases. If low-yield test cases are not

† Low-yield test cases denote test cases least probable to detect major errors in the implementation.



States

- If x is to be 1, do not bother with the situation where a = b = 1 (consideration 1).
- If x is to be 0, enumerate all situations where a = b = 0.



- If x is to be 1, enumerate all situations where a = b = c = 1.
- If x is to be 0, include only one situation where a = b = c = 0 (consideration 3). For the states 001, 010, 011, 100, 101, and 110 of a, b, and c, include only one situation each (consideration 2).

Figure 2.5 Considerations used when tracing a Cause-Effect graph

eliminated, a large CEG will produce an astronomical number of test cases. If the number of test cases is too large to be practical, one will select some subset, but there is no guarantee

		1	2	3	4	5	6
Cause	1	1	0	0	0	0	1
	2	0	1	0	0	1	0
	3	0	0	1	1	0	0
	4	1	1	1	0	0	0
Effect	100	1	1	0	0	0	0
	101	0	0	1	0	0	0
	102	0	0	0	1	1	1

Table 2.1 Decision Table from above example

that the low-yield test cases will be the ones eliminated. Hence it is better to eliminate them during the analysis of the graph. Table 2.1 gives the decision table obtained from the example above by following the above procedure.

2.1.5 Deriving Test Cases from a Decision Table

Each column in the decision table corresponds to a test case. The following steps have to be undertaken for each column in the decision table:

1. Consider the causes to be present and those that should be absent. Look for appropriate data that corresponds to the presence and absence of the given causes.
2. Determine the actual expected output from the input data. Actual expected outputs are the actual observable appearances of the required presence/absence of effects.
3. Give a unique identifier to the test case.

Test case no	Input	Expected output
1.	Student at Ottawa university **and** he/she obtains A+ in maths	Student may take skating course
2.	Student at Quebec university in Hull **and** he/she obtains A+ in maths	Student may take skating course
3.	Student at Carleton university **and** he/she obtains A+ in maths	Student may take skiing course
4.	Student at Carleton university **and** he/she does not obtain A+ in maths	Student has to take skates polishing course
5.	Student at Quebec university in Hull **and** he/she does not obtain A+ in maths	Student has to take skates polishing course
6.	Student at Ottawa university **and** he/she does not obtain A+ in maths	Student has to take skates polishing course

Figure 2.6 Test case requirements

When choosing the test data, the user may use black-box testing techniques like ***boundary-interior*** † technique to obtain still higher high-yield test cases . Figure 2.6 shows the test cases that are obtained from the decision table of Table 2.1.

Afterwards, the test cases (consisting of the test case number, the input and expected output) are placed as records in a file. The implementation of the system under test is augmented with a test harness (test driver) which will take the input from the test case file and submit it to the system. The output of the system is taken by the handler which compares it with the expected output. If the obtained result is the same as the expected output then the test is passed else the user is notified with this failure.

2.2 Analysis of Myers Approach to Cause-Effect Graphing

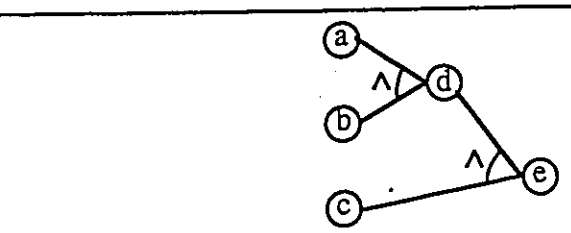
The process of deriving test cases from a natural language specification is composed of several steps described by Myers and presented in the previous section. Several authors, [Woit92, Wals83, Arab86, AdGr83], have pointed out inconsistencies in the CEG technique. Some of these claims are well-founded while others are not when the scope of this technique is considered. Two common arguments against the CEG technique are the difficulty in identifying "causes" and "effects" because of their vague description, and the proneness to error of creating a CEG because it involves examining the "semantics" of a natural language specification. We agree with these claims because analyzing natural language specifications is not an easy task. However, we do believe that someone familiar with the nature of the system being specified will more likely spot "causes" and "effects" in a natural language specification as well as its "semantics". Furthermore, since a CEG is built directly from an informal specification, this representation is likely to be closer to the customer's connotations, and therefore it would be much more easier to collaborate with the user in order to spot ambiguities and incompleteness in the requirements than Formal Description Techniques which usually are at a lower level and thus, further away from the customer. Moreover, deriving test scenarios to test the specification itself, as will be seen in chapter 3, makes this method less error-prone and hence, more attractive.

---

† Boundary-interior testing technique is a black-box testing technique where test data are selected such that these data are found just within, on, and just beyond the borders of the input.

Observation 1:

Many authors also argue that Myers process of deriving a decision table from a CEG is inconsistent. We believe in the correctness of this claim in the sense that this phase is described by Myers in an imprecise, difficult, and contradictory fashion. Myers starts by stating three considerations (pp. 65 in [Myer79]) that should be followed when deriving a decision table from a CEG, and then restates these considerations in a summary (pp. 67 in [Myer79]). The two statements are contradictory. In consideration 2 Myers states that when tracing back through an *and* node whose output should be *0*, all combinations of inputs leading to a *0* output must, of course, be enumerated, except in the case where when one is exploring the situation where one input is *0* and one or more of the others are *1*, it is not necessary to enumerate all conditions under which the other inputs can be *1*. In the summary, where a similar situation occurs Myers states that we should enumerate only one situation under which one of the inputs is *0*, hence contradicting the previous statement. As an illustration consider the CEG given in Figure 2.7(a). When deriving the decision table for



(a) An Example CEG

		1	2	3	4	5
Cause	a	0	1	0	0	1
	b	0	1	0	1	0
	c	0	0	1	1	1
Effect	e	0	0	0	0	0

(b) When applying first statement

		1	2	3
Cause	a	0	1	0
	b	0	1	0
	c	0	0	1
Effect	e	0	0	0

(c) When applying second statement

Figure 2.7 Contradiction in Myers' rules

the CEG using Myers' consideration 2, when cause *c* is set to *1* all the combinations of causes *a* and *b* leading to output *0* at node *d* should be considered. The decision table in Figure 2.7(b) is thus obtained. When using the rules given in Myers' summary, when cause *c* is set to *1* only one situation where output *d* is set to *0* should be considered, thus giving rise to the decision table in Figure 2.7(c). Two different decision tables are obtained when

applying Myers' statements. Such ambiguity will surely mislead the user, thus complicating the whole process.

Myers claims that his considerations will eliminate situations that tend to be low-yield test cases. However, his considerations allow the user to choose input combinations without any proper guidance (for example, consideration 3 states that when tracing back through an "and" node whose output should be 0, only one condition where all inputs are zero need be enumerated). Clearly, the user may not have chosen the most "high-yield" combinations. Hence, Myers' claim is not always true.

Observation 2:

When analyzing the rules given by Myers to derive a decision table it was observed that if a CEG is written differently, i.e the syntax is different while the semantics is the same, a different decision table could be obtained. This contradicts the requirement that equivalent semantics be tested equivalently. For instance, suppose we want to construct a CEG representing an effect e given by

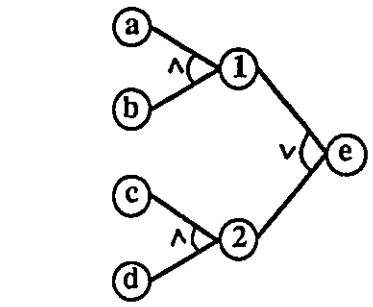
e = (a and b) or (c and d) where a,b,c, and d are causes

This effect could be represented by the CEG given in Figure 2.8(a). Now, one of the other way of writing effect e, using De Morgan's law, is as follows.

e = not(not(e)) by the Involution axiom of Boolean logic
= not(not((a and b) or (c and d)))
= not(not(a and b) and not(c and d)) by De Morgan's Law
= not((not(a) or not(b)) and (not(c) or not(d)))
= not((not(a) and not(c)) or (not(a) and not(d)) or (not(b) and not(c)) or (not(b) and not(d)))
= not(not(a) and not(c)) and not(not(a) and not(d)) and not(not(b) and not(c)) and not(not(b) and not(d))
= (a or c) and (a or d) and (b or c) and (b or d)

If the new formula of e is used the CEG given in Figure 2.8(b) will be obtained. When Myers considerations to derive decision tables for the two CEGs the decision tables given in Figure 2.8(c) and Figure 2.8(d) are obtained. As it can be noted, since the two decision tables are different they will give rise to two different sets of test cases. Therefore,

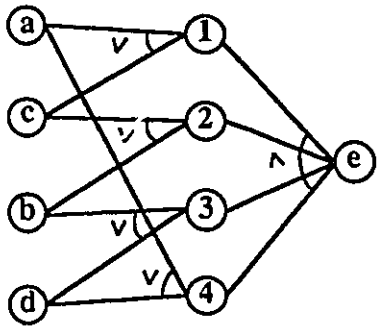
with Myers considerations, the set of test cases obtained from a CEG will depend on how effect e is formulated. This observation means that Myers considerations are not logically consistent.



(a) CEG of e = (a and b) or (c and d)

	1	2	3	4	5	6
Causes						
a	1	1	1	0	1	0
b	1	1	1	0	0	1
c	0	1	0	1	1	1
d	0	0	1	1	1	1
Effects						
e	1	1	1	1	1	1

(c) Decision Table from CEG in (a)



(b) CEG of e = (a or c) and (a or d) and (b or c) and (b or d)

	1	2
Causes		
a	1	0
b	1	0
c	0	1
d	0	1
Effects		
e	1	1

(d) Decision Table from CEG in (b)

Figure 2.8 Logical Inconsistency of Myers' rules

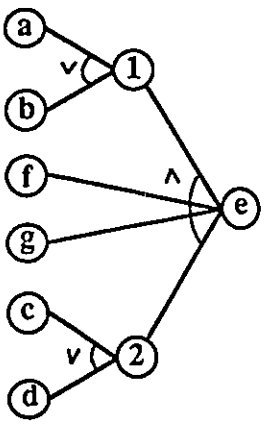
Observation 3:

When building a decision table, most of the time, some entries are left blank meaning that they are irrelevant for the presence/absence of a given effect. Myers propose to fill these *don't care* causes before deriving the test cases . However, there is no obvious pattern to the way in which Myers assigns states to *don't care*. Again leaving this exercise to the user's discretion may lead to low-yield test cases.

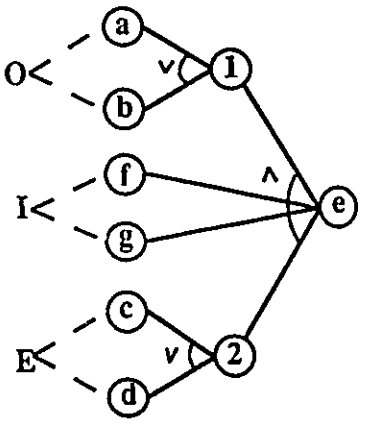
Observation 4:

One important aspect we believe is missing in Myers description of the CEG technique is that nothing is said about the conformance of the CEG to its corresponding

natural language specifications. This lack of verification may lead to faulty and incomplete test cases. For example, when deriving a decision table no considerations can be applied to detect any constraint omission in the CEG. This example is illustrated in Figure 2.9. It can be observed that the same decision table is obtained whether the One-and-only-one, Inclusive, or Exclusive constraints are omitted or not. This may result in misleading the designer at the design phase if he/she uses the CEG to obtain a good insight into the specification.



(a) Example of a CEG with missing constraints



(b) Added constraints to CEG in (a)

	1	2	3	4
Causes				
a	1	1	0	0
b	0	0	1	1
c	1	0	1	0
d	0	1	0	1
f	1	1	1	1
g	1	1	1	1
Effects				
e	1	1	1	1

(c) Same decision table for both CEGs in (a) and (b)

Figure 2.9 Superfluous Constraints Analysis

Observation 5 :

One of the other aspects that Myers does not consider when deriving test cases is the forcing of effects to be *absent*. He concentrates only on their presence. Setting an effect explicitly to *absent* helps in determining if that effect is always present or always absent.

Even if some ambiguities and incompleteness have been spotted in Myers approach, the concepts of the CEG approach used in specifying the functional behavior of a system make it attractive from a human usability perspective. The CEG technique is an advance over informal, ad-hoc specifications of systems. It is more methodical even though subjective in the first stage of constructing the CEG, and therefore more uniform, repeatable, and reliable. It is semi-formally based on a graphical form of propositional logic which gives the user some degree of confidence in the specification power of the graph [AdGr83]. The following section will provide possible solutions to the above drawbacks in Myers approach.

2.3 Possible Solutions to Above Drawbacks in Myers Technique

In the previous section it was observed that Myers approach towards the CEG technique consists of several drawbacks. This section presents possible solutions to these drawbacks. Different methods used to derive test cases from a CEG will be considered. Since all of the methods proposed make use of the Path Sensitization technique [Arms66], we will start by presenting this technique. In the next section, we will compare and assess these methods.

2.3.1 Path Sensitization Technique

Definition 1 { Cause Set}

A ***Cause Set of an effect e*** is a the set of causes whose presence or absence determine the presence or absence of effect e.

Figure 2.10 shows a Boolean logic network with a certain number of inputs and outputs. As it can be observed presence or absence of effect e_1 (in bold) is determined by presence or absence of inputs x_1, x_2, x_5, and x_6 (shown in bold). Therefore we will say that the Cause Set of effect e_1 is the set $\{x_1, x_2, x_5, x_6\}$. Similarly, the Cause Set of effect e_2 is the set $\{x_1,x_3,x_4,x_6\}$.

Definition 2 {Sensitization}

Let C be the cause set of an effect e. By ***sensitizing effect e to a particular cause c_j*** (element of C) we mean that we will have to set all the causes in C, except c_j, to specific values (each cause will either be set to present or absent) so that presence or absence of effect e will depend entirely on presence or absence of cause c_j.

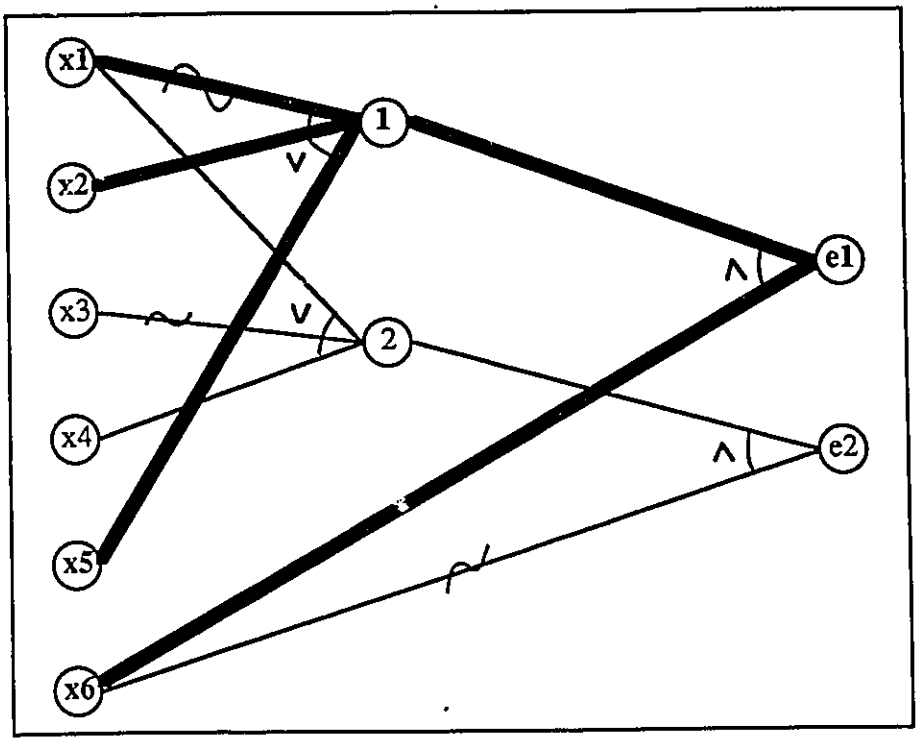


Figure 2.10 Example of a CEG

Consider again Figure 2.10. Suppose we want to sensitize effect e_1 to cause x_2. To do so causes x_1, x_5, and x_6 should be set to specific values so that e_1 will be completely dependant on x_2. To know which values to give to x_1, x_5 and x_6 each intermediate node (found on the path between x_2 and e_1) will have to be sensitized to the cause x_2. Thus, for node 1 to depend entirely on cause x_2, causes x_1 and x_5 have to be set to 1 (meaning present) and 0 (meaning absent) respectively. With this setting the value of node 1 to be equal to the value of x_2. Now, sensitizing effect e_1 to x_2 has been reduced to sensitizing e_1 to node 1. For effect e_1 to depend entirely on node 1 node x_6 has to be set to 1 (present). Hence, by setting $x_1 = x_6 = 1$ and $x_5 = 0$ effect e_1 = value of node 1 = value of x_2. In other words, with this setting, if cause x_2 is present then effect e_1 will be present else effect e_1 will be absent. Figure 2.11 shows the basic rules for path sensitization.

Definition 3 { Sensitized path }

When sensitizing an effect e to a particular cause c, the path leading from cause c to effect e in the Boolean logic network is called a ***sensitized path.***

The bold lines from cause x2 to effect e1 in Figure 2.10 mark a sensitized path.

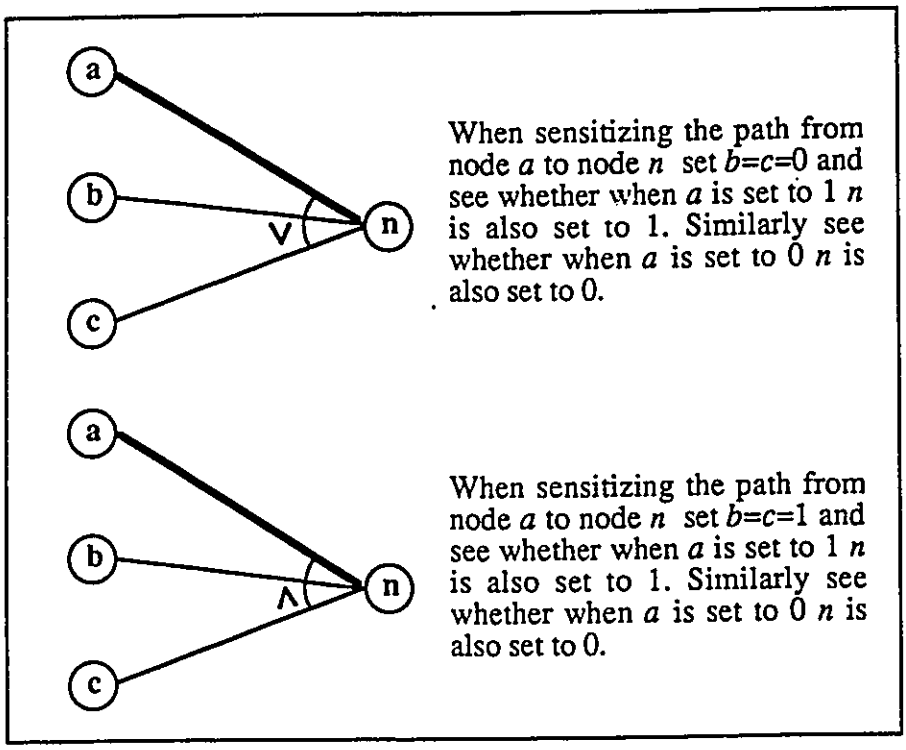


Figure 2.11 Rules of path sensitization

Definition 4 { Path Sensitization Technique }

Path Sensitization Technique is a testing technique where the specification of the system is represented using a Boolean-logic network (i.e a CEG) and combinations of causes are derived from the network by sensitizing each effect of the system to each cause in its corresponding cause set.

The combinations of causes obtained can be used to derive actual test data, when testing the implementation of the system, or act as test scenarios that can be used to validate the specification itself. Moreover, these combinations of causes can be used to see if the CEG conforms to the specification. When deriving combinations of causes using the Path Sensitization Technique each cause is checked individually to see if it is stuck_at_0 or stuck_at_1 which are defined below.

Let c_i be a cause in Cause Set C of effect e. After using the Path Sensitization technique to sensitize effect e to c_i all causes in C have been set to specific values (presence or absence) and it is determined if (value of e = value of c_i) or (value of e = not value of c_i). At this point it can be determined if cause c_i is stuck_at_0 or is stuck_at_1.

Definition 5 { Stuck_at_0 , Stuck_at_1}

i) Suppose (value of e = value of c_i). This means that if cause c_i is present then effect e is present, else effect e is absent. Now, if cause c_i is set to absent and presence of e is observed then cause c_i (or effect e) is said to be ***stuck_at_1***. If cause c_i is set to present and absence of effect e is observed then cause c_i (or effect e) is said to be ***stuck_at_0***.

ii) Suppose now that (value of e = not value of c_i) . If cause c_i is set to present and presence of effect e is observed then cause c_i (or effect e) is said to be ***stuck_at_0***. If cause c_i is set to absent and absence of effect e is observed then cause c_i (or effect e) is said to be ***stuck_at_1***.

The methods based on the path sensitization technique presented in the next section are based on the Equivalent Form [Wals83], Arabatzis Test Case Design [Arab86], and finally, our own Basic Path Sensitization Technique. The methods are illustrated by application to the Naur Text Reformatter problem, a well-known problem in the testing research literature [Naur69], and summarized in Figure 2.12(a). A CEG based on this specification is given in Figure 2.12(b).

Given an input text having the following properties :
1: It is a stream of characters, where the characters are classified as break and BL (blank), NL (new line indicator), or ET (end-of-text indicator).
2: The final character in the text is ET.
3: A word is a nonempty sequence of non-break characters.
4: A break is a sequence of one or more break characters.

The program's output should be the same sequence of words as in the input with the following properties:
1: A new line should start only between words and at the beginning of the output text, if any;
2: A break in the input is reduced to a single break character in the output;
3: As many words as possible should be placed on each line (i.e., between successive NL characters);
4: No line may contain more than MAXPOS characters (words and BL's);
5: An oversize word (i.e., a word containing more than MAXPOS characters) should cause an error exit from the program (i.e., a variable Alarm should have the value TRUE);

To understand the internal program logic, the following definition of program variables is required:
MAXPOS - maximum number of characters allowed per line.
FILL - the number of characters printed on the line so far.
BUFFER - an internal work area for storing a word.
BUFPOS - the number of characters in the work area buffer. This is set to zero after the word in BUFFER is printed.

Figure 2.12(a) Specification of the Naur Text Reformatter Problem (TRP)[GoGe75]

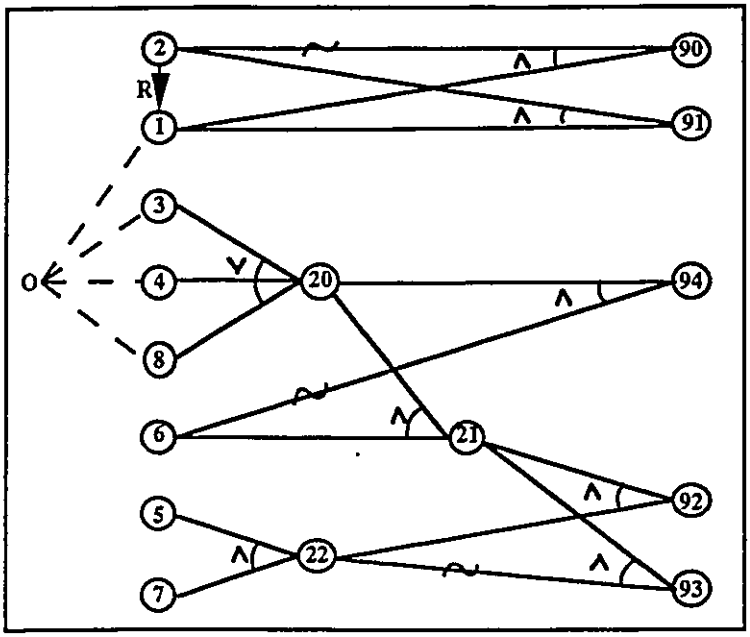


Figure 2.12(b) CEG for the Text Reformatter Problem [Naur69]

2.3.2 Equivalent Normal Form

Armstrong has proposed a method for deriving a nearly minimal set of fault-detection tests, using the Equivalent Normal Form (abbreviated ENF), of the circuits [Arms66]. The ENF of a circuit represents the dependence of the output on the inputs and the states of the internal wires. However, some information about the type of fault that may be present on each wire is omitted. The ENF of a circuit is obtained by expressing the output of each gate as a function of its inputs and preserving the identity of each gate by a suitable subscript [Arms66, FrMe71]. Thus, if the inputs to an AND gate are labeled a and b, and the output is labeled c, we represent this by $c = (a.b)_c$. Applying this procedure to all gates in the circuit, an expression (usually in a factored form) of the output in terms of the input variables is obtained. This factored expression is then expanded into the ***sum-of-products*** † form in the usual manner except that a pair of parentheses is removed, the subscript associated with it is concatenated to the subscripts of the variables within the pair of parentheses. Each subscripted input variable (e.g *not 5[93]*) in the ENF is referred to as a literal. If an input variable appears in the ENF with different subscripts, they are considered to be distinct literals. Literals connected by ANDs are called product terms.

An appearance of a literal in the ENF is usually tested for stuck-at-1 by assigning the value of 0 to it, 1's to all other literals in the term and suitable values to additional variables

† Sum-of-products represent product terms join together by "or" or "+" operations.

so as to cause all other terms in the ENF to be 0. An appearance of a literal is tested for stuck-at-0 by assigning the value of 1 to all literals in the term containing it and making all other terms equal to 0.

The Equivalent Normal Form was applied to software testing by representing the program as a collection to hardware gates as is done in the CEG approach [Wals83]. The ENF is developed by expressing the output conditions or intermediates states as a function of the inputs and at the same time preserving the identity of each node. Figure 2.13 shows the ENF of the effects derived from the CEG given in Figure 2.12(b) (this graph was obtained from [Wals83]. It has been slightly modified to include all the constraints and one missing "and" relation which we believe is a typing mistake).

The next step is to test each literal in an ENF for stuck-at-0 and stuck-at-1 using the same principal as in circuit testing presented above. After applying these procedures we obtain the set of cause combinations given in Figure 2.14 [Wals83]. For instance, when deriving the input conditions for ENF(93) the following is obtained:

- Literal 1 (* (3[20,21,93] and 6[21,93] and not 5[22,93] *)
  - set 3[20,21,93] = 1, 6[21,93] = 1, 5[22,93] = 0, 7[22,93] = 1, 4[20,21,93] = 0, 8[20,21,93] = 0

- Literal 2 (* (4[20,21,93] and 6[21,93] and not 5[22,93] *)
  - set 4[20,21,93] = 1, 6[21,93] = 1, 5[22,93] = 0, 7[22,93] = 1, 3[20,21,93] = 0, 8[20,21,93] = 0

- Literal 3 (* (8[20,21,93] and 6[21,93] and not 5[22,93] *)
  - set 8[20,21,93] = 1, 6[21,93] = 1, 5[22,93] = 0, 7[22,93] = 1, 3[20,21,93] = 0, 4[20,21,93] = 0

- Literal 4 (* (3[20,21,93] and 6[21,93] and not 7[22,93] *)
  - set 3[20,21,93] = 1, 6[21,93] = 1, 5[22,93] = 1, 7[22,93] = 0, 4[20,21,93] = 0, 8[20,21,93] = 0

- Literal 5 (* (4[20,21,93] and 6[21,93] and not 7[22,93] *)
  - set 4[20,21,93] = 1, 6[21,93] = 1, 5[22,93] = 1, 7[22,93] = 0, 3[20,21,93] = 0, 8[20,21,93] = 0

- Literal 6 (* (8[20,21,93] and 6[21,93] and not 7[22,93] *)
  - set 8[20,21,93] = 1, 6[21,93] = 1, 5[22,93] = 1, 7[22,93] = 0, 3[20,21,93] = 0, 4[20,21,93] = 0

```
ENF(90) = (1 and not 2)[90]
        = 1[90] and not 2[90]

ENF(91) = ( 1 and 2) [91]
        = 1[91] and 2[91]

ENF(92) = (21 and 22) [92]
        = ((20 and 6)[21] and (5 and 7)[22])[92]
        = (((3 or 4 or 8)[20] and 6)[21] and (5 and 7)[22])[92]
        = (3[20,21,92] and 6[21,92] and 5[22,92]  and 7[22,92]) or
          (4[20,21,92] and 6[21,92] and 5[22,92]  and 7[22,92]) or
          (8[20,21,92] and 6[21,92] and 5[22,92]  and 7[22,92])

ENF(93) = (21 and not 22) [93]
        = ((20 and 6)[21] and not (5 and 7)[22])[93]
        = (((3 or 4 or 8)[20] and 6)[21] and (not 5 or not 7)[22])[93]
        = (3[20,21,93] and 6[21,93] and not 5[22,93]  or
          (4[20,21,93] and 6[21,93] and not 5[22,93]  or
          (8[20,21,93] and 6[21,93] and not 5[22,93]  or
          (3[20,21,93] and 6[21,93] and not 7[22,93]  or
          (4[20,21,93] and 6[21,93] and not 7[22,93]  or
          (8[20,21,93] and 6[21,93] and not 7[22,93]

ENF(94) = (20 and not 6)[94]
        = ((3 or 4 or 8)[20] and not 6)[94]
        = (3[20,94] and not 6[94]) or
          (4[20,94] and not 6[94]) or
          (8[20,94] and not 6[94])
```

Figure 2.13 ENF obtained from the CEG in Figure 2.12(b)

	Input Conditions							
	1	2	3	4	5	6	7	8
ENF 91	1	1	-	-	-	-	-	-
90	1	0	-	-	-	-	-	-
94	-	-	1	-	-	0	-	-
94	-	-	-	1	-	0	-	-
94	-	-	-	-	-	0	-	1
92	-	-	0	0	1	1	1	1
92	-	-	0	1	1	1	1	0
92	-	-	1	0	1	1	1	0
93	-	-	1	0	0	1	1	0
93	-	-	1	0	1	1	0	0
93	-	-	0	1	0	1	1	0
93	-	-	0	1	1	1	0	0
93	-	-	0	0	0	1	1	1
93	-	-	0	0	1	1	0	1

Figure 2.14 ENF Conditions for Text Reformatter Problem

2.3.3 Arabatzis Test Design Based on Path Sensitization

In his thesis [Arab86], Arabatzis proposed a new method to test case design approach based on path sensitization. His approach consists of two steps. The first is the construction of a standard Cause-Effect Combinational Network (CCN) representation of the system under test. The standard CCN is equivalent to a CEG. The second step is to use path sensitization techniques to generate a set of test requirements.

His second step consists of deriving for each effect its corresponding canonical product-of-sum (PoS) form. Later on, he uses this new form to build the sum-of-product form (SoP). The second step is algorithmic and therefore can be automated. Figure 2.15 shows the PoS and SoP representation for each effect derived from the CEG given in Figure 2.12(b).

Note that in Figure 2.15 ".", "+", and " ' " have the same meaning as "and", "or", and "not" respectively as used in the CEG technique. Just as in the case of Equivalent Normal Form, the stuck-at-0 and stuck-at-1 test procedures are applied to the PoS and SoP representation to derive test requirements. Each character (e.g. 6', 3) is called a literal. Literals connected by ANDs are called product-literals, while terms connected by ORs are called sum-literals. An appearance of a product-literal in a SoP representation is usually tested for stuck-at-0 by assigning the value of 1 to it, 0's to all other product-literals in the SoP. An appearance of a sum-literal in a PoS is tested for stuck-at-1 by assigning the value of 0 to it, 1' for all other sum-literals in the PoS.

When applying the above procedure to obtain the appropriate input conditions from Figure 2.15 Figure 2.16 is obtained. For instance, when deriving the input conditions for effect 93 the following is obtained:

- **PoS(93)** = 6.(3+4+8).(5'+7')
  - to test if cause 6 is stuck-at-1 => set 6 = 0, 3 = 1, 4 = 0, 8 = 0, 5 = 0, 7 = 1
  - to test if sum-literal (3+4+8) is stuck-at-1 => set 6 = 1, 3 = 0, 4 = 0, 8 = 0, 5 = 0, 7 = 1
  - to test if sum-literal (5'+7') is stuck-at-1 => set 6 = 1, 3 = 0, 4 = 0, 8 = 1, 5 = 1, 7 = 1
- **SoP(93)** = 6.3.5' + 6.3.7' + 6.4.5' + 6.4.7' + 6.8.5' + 6.8.7'

- to test if product-literal 6.3.5' is stuck-at-0 => set 6 = 1, 3 = 1, 5 = 0, 4 = 0, 8 = 0, 7 = 1
- to test if product-literal 6.3.7' is stuck-at-0 => set 6 = 1, 3 = 1, 7 = 0, 4 = 0, 8 = 0, 5 = 1
- to test if product-literal 6.4.5' is stuck-at-0 => set 6 = 1, 4 = 1, 5 = 0, 3 = 0, 8 = 0, 7 = 1
- to test if product-literal 6.4.7' is stuck-at-0 => set 6 = 1, 4 = 1, 7 = 0, 3 = 0, 8 = 0, 5 = 1
- to test if product-literal 6.8.5' is stuck-at-0 => set 6 = 1, 8 = 1, 5 = 0, 3 = 0, 4 = 0, 7 = 1
- to test if product-literal 6.8.7' is stuck-at-0 => set 6 = 1, 8 = 1, 7 = 0, 3 = 0, 4 = 0, 5 = 1

Effect	PoS representation	SoP representation
90	= 1.2'	= (1.2')
91	= 1.2	= (1.2)
92	= (21.22) = ((6.20).(5.7)) = ((6.(3+4+8)).(5.7)) = 5.6.7.(3+4+8)	= 5.6.7.(3+4+8) = (6.3.5.7) + (6.4.5.7) + (6.8.5.7)
93	= (21.22') = ((6.20).(5.7)') = ((6.(3+4+8)).(5'+7')) = 6.(3+4+8).(5'+7')	= 6.(3+4+8).(5'+7') = 6.3.5' + 6.3.7' + 6.4.5' + 6.4.7' + 6.8.5' + 6.8.7'
94	= (6'.20) = 6'.(3+4+8)	= 6'.(3+4+8) = 6'.3 + 6'.4 + 6'.8

Figure 2.15 PoS and SoP representation for the Text Reformatter Problem

In Figure 2.16 test requirements number 1 to 14 correspond to test for stuck-at-0 while the remaining requirements (number 15 to 22) correspond to test for stuck-at-1. The input entries in bold represent the forcing conditions while the effect entries in bold correspond to the target effect.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Causes																						
1	1	1	-	-	-	-	-	-	-	-	-	-	-	-	**0**	**0**	-	-	-	-	-	-
2	0	1	-	-	-	-	-	-	-	-	-	-	-	-	0	**1**	-	-	-	-	-	-
3	-	-	**1**	0	0	**1**	**1**	0	0	0	0	**1**	1	0	-	-	**1**	**0**	**1**	**0**	**0**	1
4	-	-	0	**1**	0	0	0	**1**	**1**	0	0	0	1	0	-	-	0	**0**	0	**0**	**0**	0
5	-	-	**1**	**1**	**1**	**0**	1	**0**	1	**0**	1	-	-	-	-	-	1	1	**0**	0	-	-
6	-	-	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	-	-	**0**	1	**0**	1	0	**1**
7	-	-	**1**	**1**	**1**	1	**0**	1	**0**	1	**0**	-	-	-	-	-	1	1	1	1	-	-
8	-	-	0	0	**1**	0	0	0	0	**1**	**1**	0	0	**1**	-	-	0	**0**	0	**0**	**0**	0
Effects																						
90	**1**	0	-	-	-	-	-	-	-	-	-	-	-	-	**0**	0	-	-	-	-	-	-
91	0	**1**	-	-	-	-	-	-	-	-	-	-	-	-	0	**0**	-	-	-	-	-	-
92	-	-	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	-	-	**0**	**0**	0	0	0	0
93	-	-	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	-	-	0	0	**0**	**0**	0	0
94	-	-	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	-	-	1	0	1	0	**0**	**0**

Figure 2.16 Test requirements for the Text Reformatter Problem

2.3.4 A New Basic Path Sensitization Technique

In this section a more intuitive approach to derive a decision table from a CEG using the path sensitization technique is presented. The basic sensitization rules given in Figure 2.11 are used. The path from each cause, say c, to an effect, say e, is traced to see how c is related to e (i.e. either presence of c = presence of e, or presence of c = absence of e) and at the same time setting specific values to all other causes necessary to determine effect e. For instance, to sensitize the path from cause x1 to effect e2 in Figure 2.10, first start at node x1 and assign cause x1 the value y (it will either be 0 or 1). Then go to the next node from x1, namely intermediate node 2. To sensitize the path from node x1 to node 2, set x3 = 1 and x4 = 0. With these values the output of node 2 will be y. To sensitize the path from node 2 to node e2, set x6 = 0. The output of node e2 will be y. Thus, by having x3 = 1, and x4 = x6 = 0, (presence of cause x1 = presence of effect e2) and similarly (absence of cause x1 = absence of effect e2). By setting cause x1 to 0 (absent) and then to 1 (present) with the appropriate values for x3, x4, and x6 it can be determined if effect e2 is sensitive to cause x1. Section 2.3.4.1 presents an algorithm use to sensitize the effects to the causes in a given CEG.

The benefit of this technique resides in the fact that the algorithm given is simple, easy to implement, and it works for every CEGs. Besides, when applied to the same CEG, this technique gives practically the same result as Arabatzis technique which is more complex and

does not apply for all CEGs especially for CEGs where there exist an effect related to other nodes by the "or" relation [Arab86].

2.3.4.1 Basic Path Sensitization Technique (PST) Algorithm

Step 1
Let g be a CEG.
For each effect e_i in g **do**
Backtrack g from e_i to build the Cause Set of e_i.
Name this Cause Set C_i.
At the same time build a set N_i to indicate all the intermediate nodes found on the path from e_i to its causes.
endfor

Step 2
For each cause c_i in Cause Set C_j (*of effect e_j*) **do**
(* we are sensitizing e_j to cause c_i *)
Let the value of c_i be x (*denote either presence or absence *)
Repeat
Repeat
Go to the forward node n (* at first it is directly linked to c_i , then it will be linked to the previous n*)
If n ∉ Nj then return to starting node (* from where we go to node n *) **endif**
until n ∈ Nj
If relation at n is "and" **then**
Set all incoming arcs to n (except the one from c_i) to 1
(* We should therefore backtrack each one of these arcs to set values to the appropriate causes responsible for the arcs being set to 1. In this process two special cases may occur : i) several combinations of the causes may set the arcs to 1, ii) a given cause may be set to different values through separate routes *)
Else (* relation at node n is "or" *)
Set all incoming arcs to n (except the one from c_i) to 0
(* We should therefore backtrack each one of these arcs to set values to the appropriate causes responsible for the arcs being set to 0. The same special cases noted above can be found here *)
endif
If the sensitized link to node n is set to "not" **then**
If the input to n on this link before applying "not" is x **then** the output of n will be x'
Else If the input is x' **then** the output of n will be x
endif
Else
If the input on this sensitized link is x **then** the output of n will be x
Else If the input is x' **then** the output of n will be x'
endif
endif
until n is effect e_j
If the output e_j is set to x **then**
value of e_j = value of cause c_i
Else

value of e_j = not value of cause c_i
endif
Create two combinations of causes, one with cause c_i set to 0 (absent) and another one with c_i set to 1 (present). (* Note that each combination of causes will be represent by a single column in the resulting Decision Table *)
endfor
Step 3
Repeat Step 2 for each effect e_j in g
Remove any duplicate column in the decision table (DT)
Apply the constraints in the graph to each column in the DT to see if it is a valid case

2.3.4.2 Analysis of the above Algorithm

Usually, the testing process is more cost-effective if the system under test is exercised with the fewest number of test cases, especially in the case of large systems. It would be interesting to do the worst-case analysis of the above algorithm. Suppose that we are deriving test cases for an effect *e* that has *n* causes in its cause set. For each cause in the cause set of e we should derive two test cases to sensitize this effect to that specific cause (one for presence and one for absence of the effect). Hence, the maximum number of test cases for effect e will be 2n. Since during sensitization two causes can yield the same test case, hence the total number of test cases derived for that effect is equal or less than 2n. Obviously, The number of test cases for all effects will depend on the number of effects as well as the number of causes in the cause set of each effect.

The algorithm given is a direct implementation of the Path Sensitization Technique, i.e the automatic procedure is the same as the manual procedure. It can be observed that when backtracking from a node, say n, to set appropriate values to the causes responsible for an input, say i, being set to a given value , x, two special cases may occur. The two cases are as follows :

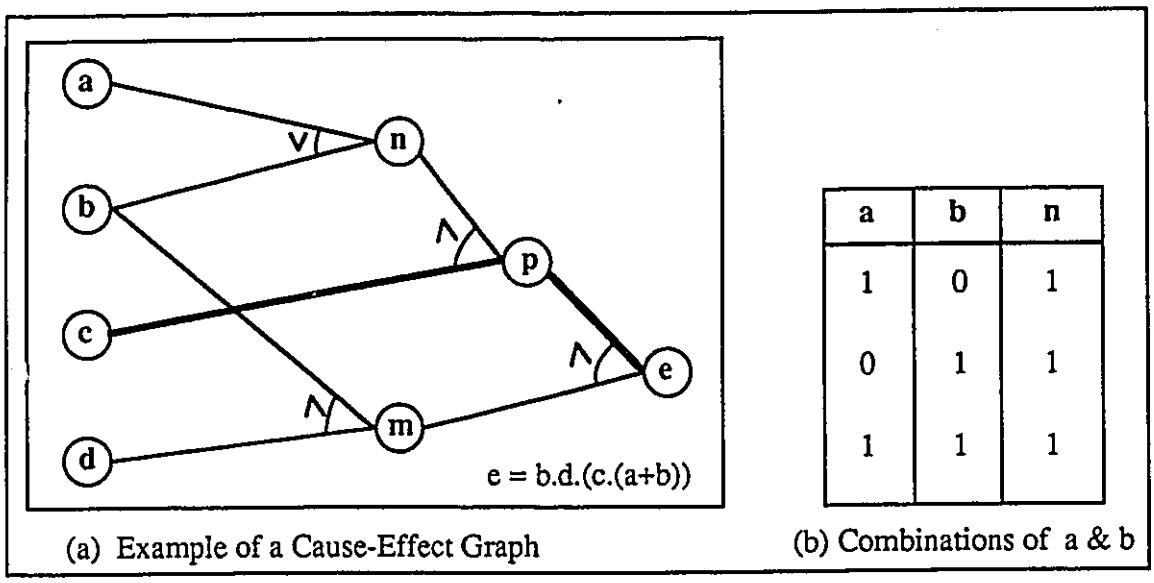
a) A given cause may be set to different values through separate backtracking routes from node n

b) Several combinations of causes may set input i to x

To illustrate these special cases consider Figure 2.17 below. Suppose effect e is sensitized to cause c (shown in bold in the CEG in Figure 2.17(a)).

a) Using the sensitization rule, at node p the input from node n has to be set to 1(present). Now, when setting node n to 1 three different combinations (Figure 2.17(b)) for causes a and b are possible. This illustrates case b above. Now, suppose at this phase the

combination a = 1 and b = 0 is chosen. When proceeding to effect e, it can be noticed that node m has to be set to 1. To do so both causes b and d should be set to 1. At this point it can be observed that cause b was already set to 0. This illustrate case a) above.



(a) Example of a Cause-Effect Graph

a	b	n
1	0	1
0	1	1
1	1	1

(b) Combinations of a & b

Figure 2.17 Example of sensitizing an effect to a cause

Consider in greater detail case a) where a given cause may be set to different values through separate routes from a given node n. When this case occurs an appropriate combination of causes that will satisfy all considerations can be selected. For example in Figure 2.17 we can select a = 0, b = 1. However, this requires that one or more combinations can satisfy the considerations. **When this cannot be done, either there is a mistake in the specification or in the CEG*** . Figure 2.18 illustrates this case.

Figure 2.18 is a slight modification of Figure 2.17 except that the relation at node n is changed to *and* and a *not* has been placed on the link between node b and m. First the input to node p from node n is set to 1. Then causes a and b should both be set to 1 for node n to be 1 . Now when proceeding to node e the input from node m has to be set to 1. To set node m to 1 cause d should be set to 1 and cause b to 0. This goes in direct conflict with the previous settings where a=b=1. At the same time the previous settings cannot be changed because there is only one possible combination. **Claim :** There is an error in either the CEG or the specification.

---

* a catalogue of such symptoms of specific errors or logical inconsistencies is given in Chapter 5.

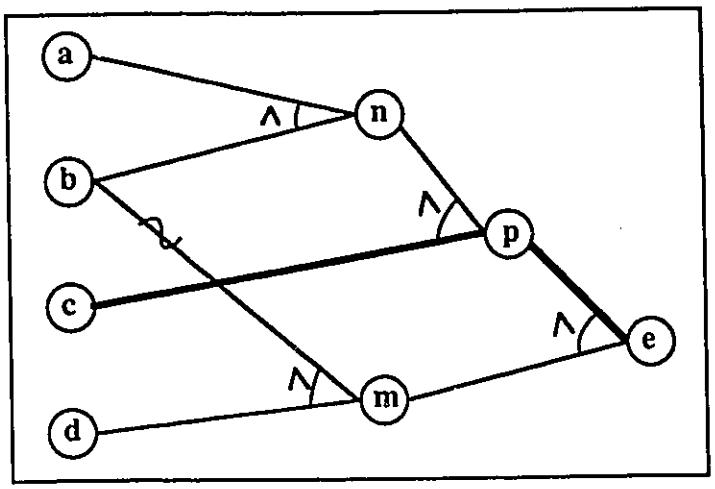


Figure 2.18 Deduction of Error in CEG or specification

b) Now, consider case b) above. When several cause combinations are possible any one of them can be selected and if case a) also occurs then we can choose another combination. When the contribution of a particular cause is sensitized to an effect , say e, it would seem logical to say that taking any valid combination of the other causes in the Cause Set of e would not affect the result. However, since our objective is to derive combinations of causes that detect most of the faults found in the specification or the implementation one should, therefore, choose the combination(s) of causes that seem will most probably satisfy this objective. Choosing the appropriate combination(s) of causes would require knowledge of the nature of the system under specification, i.e the combination(s) chosen for a telecommunications system might be different from the combination(s) chosen for a database management system. The best way would be to show all the possible combinations to someone having some experience in the nature of the system and request that he/she choose. However, the number of possible cases may be too large. Thus, guidance for the tester in his/her choice is required. Two possible solutions dealing with this case are shown next.

Solution 1 of b) :

As a first solution suppose effect e is sensitized to cause c. With the algorithm above two combinations of causes were derived: one for effect e to be absent and one for it to be present. Consider the case where e is absent. If several combinations of causes (except c) in the Cause Set of e are possible, then it seems logical to choose a combination where the maximum of the causes are set to 1 (present). This follows from the fact that usually when some (atomic, positive) causes are present some effects are observed. Hence our choice is to test for the most unusual case and hence the **importance of the contribution of cause**

c has increased. By this we mean that if most of the causes in the cause set of e are present and by setting cause c to *absent* effect e is absent too, then it can be said that absence of cause c has overridden the presence of all the other causes. Hence the importance of the contribution of cause c is said to be increased. Similar when effect e is present, choose a combination of causes where the maximum number of causes is absent. This follows from the fact that usually absence of causes leads to absence of effects. Hence our choice is another unusual case. To test more intensively, more than one combination of causes can be chosen following the same concept. This way of selecting combinations of causes brings an additional **high-yield guideline** to the Basic Path Sensitization Technique, i.e **when sensitizing an effect to a cause the combinations of causes that will increase the importance of the contribution of that cause are chosen.**

Solution 2 of b):

As another solution, instead of creating only two combinations of causes when sensitizing each cause, more cause combinations can be created if case b occurs as above. In

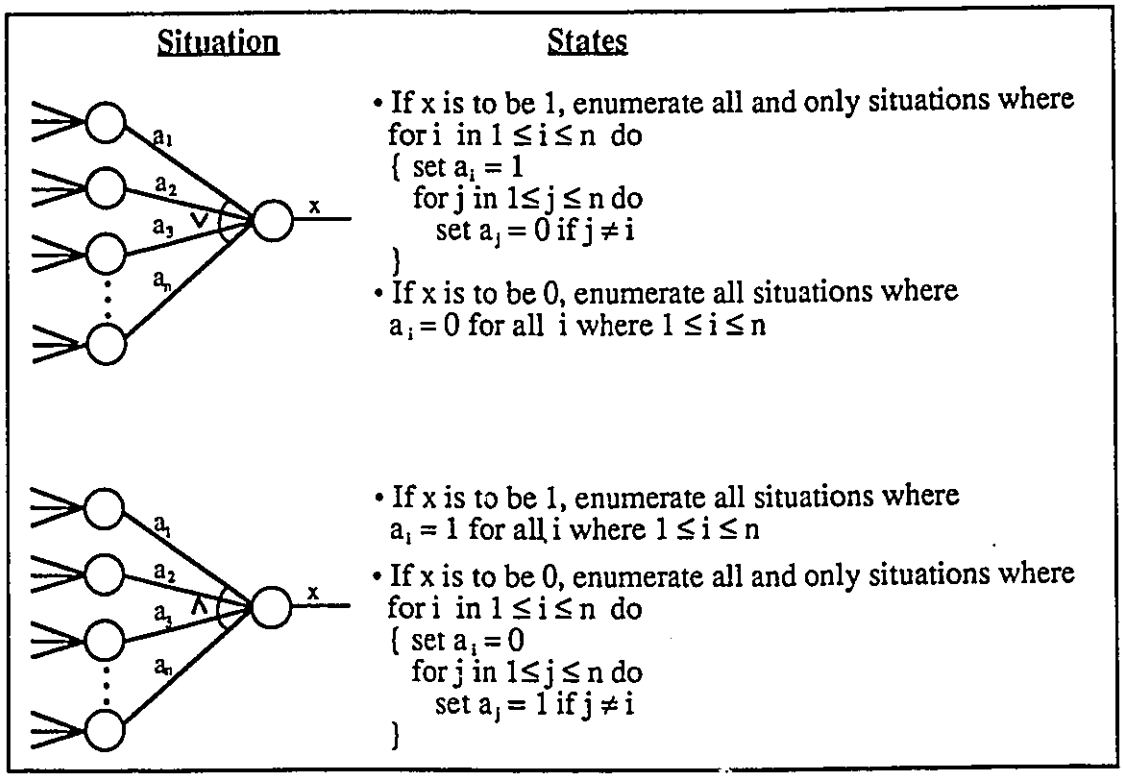


Figure 2.19 Considerations used when tracing a Cause-Effect graph

that case, when backtracking the CEG to set appropriate values to some causes, the rules in Figure 2.19 can be used to determine all the possible cause combinations. As can be

observed, the rules given in Figure 2.19 are very similar to the considerations of Myers. The only difference is that in this case these rules are secondary to our main objective, namely to sensitize an effect to a particular cause, while in the case of Myers they were used as its primary objective,namely deriving cause combinations. Also, we believe that the rules given are more appropriate for sensitization concepts and they are expressed in a clearer fashion than Myers' rules.

As it can be observed above, Path Sensitization Technique of test generation isolates the individual contribution of one cause at a time. Due to the individual focus of the technique, it is not always possible to test for any missing constraints among the causes, particularly among causes that feed into an *or* node. For example, consider Figure 2.20(a) and (b) below. Assume that Figure 2.20(a) shows a correctly derived CEG while Figure 2.20(b) shows an incorrectly derived one. We assume both graphs were obtained from the same specification.

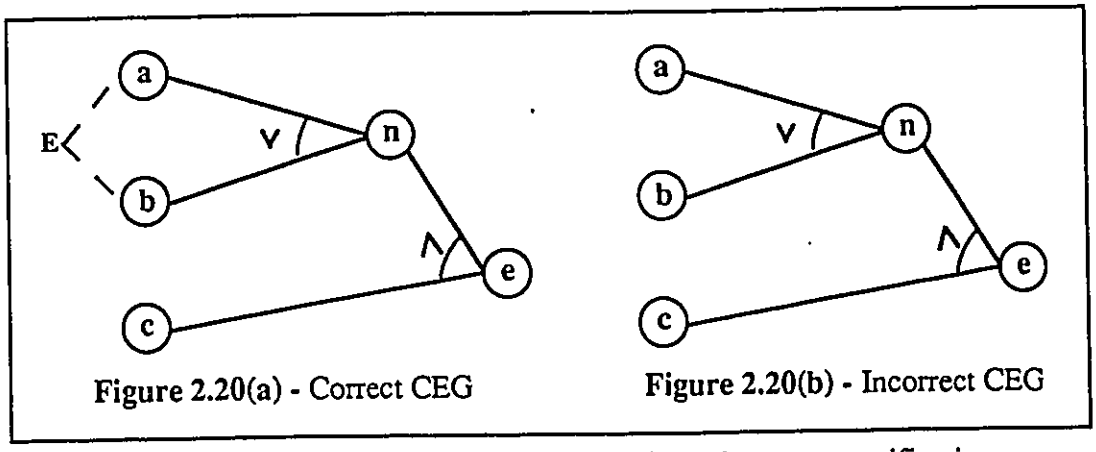


Figure 2.20(a) - Correct CEG

Figure 2.20(b) - Incorrect CEG

Figure 2.20 Two different CEGs derived from the same specification

Table 2.2 gives the combinations of causes derived from both CEGs in Figure 2.20 using any of the above Path Sensitization Techniques. Thus, detecting the missing exclusive constraint in the CEG given in Figure 2.20(b) is not possible.

a	b	c	e
0	0	1	0
1	0	1	1
0	1	1	1
1	0	0	0

Table 2.2 Combinations of causes from Figure 2.20

When analyzing all the possible constraints that can be annotated to a CEG the following observations can be made.
Heuristics for Detection of Missing Constraints:

- The *exclusive constraint* says that at most one out of a group of causes may be set to 1 at a given time. Therefore, if we derive a cause combination where we make most of the causes in that group present we can detect from domain-specific knowledge that the exclusive constraint is missing.

- The *one-and-one-only constraint* says that one and only one of a group of causes may be set to 1 at a given time. Therefore, if we derive a cause combination where we make most of the causes in that group present we can detect if the one-and-one-only constraint is missing. Furthermore, if we derive a cause combination where all the causes in that given group are absent then we can again detect omission of that constraint.

- The *inclusive constraint* says that at least one out of a group of causes should be present. Hence, if we make all of the causes in that group absent we will detect omission of that constraint.

- The *requires constraint* says that for one cause to be set to 1 another cause should also be set to 1. We believe that most Path Sensitization Techniques will likely detect this kind of omission as it deals with causes on an individual basis. In addition, using our high-yield guidelines should enable detection of this type of omission.

- The *masks constraint* says that when an effect is set to 1 it will mask the output of another effect. Again, we believe that most Basic Path Sensitization Techniques will

generate combinations of causes and from the cause combinations we can say for each effect made present whether or not another effect will be observed.

From these general observations we can deduce that omissions of some constraints can be detected by using the Path Sensitization Technique. Our strategy is to add two extra combinations of causes (one with the maximum causes in the Cause Set of effect e absent and the other one with the maximum causes in the Cause Set of effect e present) to the set of combinations of causes obtained. For instance, in the case of the CEG in Figure 2.20, we would have derived two combinations of causes where a=b=c=0 and a=b=c=1. This last combination of causes would have detected omission of the exclusive constraint. Setting most of the causes in a given Cause Set to 1 has a positive side-effect in the sense that this combination of causes may capture any interference between causes*. In any case this interference will only be captured if two or more interfering causes are both set to 1. Hence,

```
For each effect, e_i, in a CEG g do
   For each cause c_j in the Cause Set C of effect e_i do
      ->Apply the algorithm given to sensitize effect e_i to cause c_j
        (* When backtracking a given node either use solution 1 or solution 2 given
        above *)
      ->For each possible combinations create two cause combinations one with c_j
        set to 1 and another one with c_j set to 0
      ->Create a column in the decision table for each cause combination
      ->Use the values set to the causes in C and the constraints present in the CEG
        to determine the values of the other causes in the cause effect graph
      ->Use the values set to the causes in C in order to determine the values of the
        other effects in g for these two combinations
      ->Remove any redundant column
   endfor
   ->If possible, create two additional cause combinations where in one most of the
     causes in C are absent and in the other most of them are present.
   ->Create a column in the decision table for each cause combination
   ->Use the values set to the causes in C and the constraints present in the CEG to
     determine the values of the other causes in the cause effect graph
   ->Use the values set to the causes in C in order to determine the values of the
     other effects in g for these two combinations
   ->Remove any redundant column
endfor
->Derive test scenarios from the decision table obtained
```

Figure 2.21 Test Scenario derivation using the Basic Path Sensitization Technique

* This means that when a cause is present it may interfere with the ability of another cause to be present. This interference may or may not be intended.

by adding a combination of causes where most of the causes are present we could detect most of the interference. However, there is still another case where by setting a particular cause to 1 it may hide the interference of two or more causes. But, we believe that even if this case occurs the customer will detect major cause interferences since the specification will later be validated using test scenarios.

Since our goal is to validate the CEG against the specification and also to validate the specification against the objectives of the customer test scenarios will be derived (section 3.3 in chapter 3) from the decision table obtained by the Basic Path Sensitization technique. Later, test cases may also be derived from the decision table. *"Don't-cares"* can then be assigned specific values and these assignments should be recorded and consistently used whenever a fault is detected. Figure 2.21 summarizes the process of test scenario derivation based on the Basic Path Sensitization Technique.

Figure 2.22 and Figure 2.23 below show the decision tables obtained when applying solution 1 and solution 2 to the CEG in Figure 2.12 respectively. In both figures, the causes in bold denote the causes in the Cause Set of the effect under consideration, while a cause both in bold and underlined denotes the sensitized causes (when in one column there are more than one sensitized cause this means that each one of the sensitized cause produces the same column). An effect in bold denotes the target effect.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Causes																				
1	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0
2	**0**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0
3	0	0	**1**	0	**0**	**1**	0	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	0	**0**	**0**	**0**	**1**
4	0	0	**0**	**1**	0	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	0	**0**	**0**	**0**	**0**
5	-	-	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	-	-	-	**1**	**1**	-	**1**	**1**	-	-
6	-	-	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	-	**1**	**1**	**0**	**1**
7	-	-	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	-	-	-	**1**	**0**	-	**1**	**0**	-	-
8	0	0	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	0	**0**	**0**	**0**	**0**
Effects																				
90	**1**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0
91	**0**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	**1**	**1**	**1**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	**0**	0	0	0
93	0	0	**0**	0	0	**1**	**1**	**1**	**1**	**1**	0	0	0	0	**0**	0	0	**0**	0	0
94	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	1	0	0	0	**0**	**0**

Figure 2.22 Decision Table of the Text Reformatter Problem when using Solution 1

2.4 Comparison and Evaluation of the Above Path Sensitization Techniques

In section 2.3.3 we have presented several techniques based on Path Sensitization in order to eliminate the drawbacks inherent in Myers approach to CEG technique. The techniques presented start from a CEG and then build a decision table representing the test cases. We have concentrated our efforts on the derivation of the decision table since this is where most of the drawbacks in Myers' approach occur.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Causes																		
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0
4	0	0	0	1	0	0	0	1	1	0	0	0	0	1	0	0	1	0
5	-	-	1	1	1	1	0	1	0	1	0	0	-	-	-	1	1	1
6	-	-	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
7	-	-	1	1	1	0	1	0	1	0	1	1	-	-	-	1	1	1
8	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	0	0	1
Effects																		
90	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
91	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0
94	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Causes																				
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0
4	1	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0
5	0	0	1	1	1	1	1	1	1	0	0	0	-	1	1	0	-	-	-	-
6	1	1	1	1	1	1	0	0	0	0	0	0	-	1	1	1	0	1	1	1
7	1	1	1	1	0	0	0	0	0	1	1	1	-	1	0	1	-	-	-	-
8	0	1	0	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1
Effects																				
90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Figure 2.23 Decision Table of the Text Reformatter Problem when using Solution 2

Among the drawbacks noted in Myers approach were a) ambiguous rules are given to derive a decision table, b) only presence of effects are considered, c) the CEG is not validated against the specification, d) no method is presented to assign *don't cares* in the decision table, and e) two representations of the same CEG may yield different decision tables. Figure 2.24 below gives a brief summary of how each of the above techniques presented addresses these drawbacks in Myers' approach.

In Figure 2.24 the symbol "P/A" means that both presence and absence of effects are tested while "Complexity" means the degree of difficulty that will be met by a tester to derive test cases using these methods. It can be noticed that the Basic path sensitization technique is the only one that validates a CEG against its specification and records any *don't care* assignments in the decision table whenever test cases are derived. All the methods shown above can be automated. Moreover, since they are all based on path sensitization we can say that different representations of the same CEG will produce more or less the same decision table from any of the methods. Some justification for this is that since each effect is sensitized to each cause, whatever the representation of a CEG the Cause Set of each effect will be the same and therefore the decision table should be quite similar for all representations.

	Ambiguous rules	Effect	Validating CEG w.r.t spec	Don't care assignment	Complexity
Equivalent Normal Form	No	P/A	No	No	Small
Arabatzis Approach	No	P/A	No	No	Large
Basic path sensitization Solution 1	No	P/A	Yes	Yes	Small
Solution 2	No	P/A	Yes	Yes	Medium

Figure 2.24 Comparison of various Path Sensitization Techniques

When each one of the above methods have been applied to the Text Reformatter Problem we may observe the following :

- The Equivalent Normal Form produces the same decision table as Myers' CEG technique. However, the ENF technique does not show whether other effects may be observed when we are expecting presence or absence of a particular effect and at the same time it does not consider the case where constraints are present in the CEG. The main advantage of this technique is that is it simple and shows exactly which path is sensitized.

- Arabatzis' method produces more test cases than the ENF approach and explicitly mentions which one is the target effect and the determining conditions. It also shows the state of the other effects whenever we are sensitizing a particular effect. However, this method does not consider the case where constraints are present in the CEG. Though quite complex to implement, this method produces a good and well documented test suite. One drawback in Arabatzis method is that his algorithm to derive the PoS form of an effect works only in the case where an "AND" relation exists between that effect and the other nodes (the algorithm will not work in the case of an "OR" relation.

- The Basic Path Sensitization technique is the only one that addresses all the drawbacks in Myers technique. In this technique two solutions were proposed. In solution 1, a decision table quite similar to Arabatzis' is produced while in solution 2 a much bigger decision table is obtained. The advantage of this technique is that it is simple, easy to implement, can be applied for all CEGs, and gives practically the same results as Arabatzis when applied to the same CEG. Furthermore, in solution 1 we have an upperbound on the number of cause combinations, thus on the number of test cases, given as follows:

  - *If a CEG has m effects and each effect e_i ($1 \leq i \leq m$) has n_i causes in its cause set then the upper bound to the number of cause combinations that can be derived from the CEG is given by* $2+(2\sum_{i=1}^{m} n_i)$.

We can say that solution 1 gives a decision table that can be used to detect common mistakes and omissions while solution 2 gives a decision table that will test most of the faults that may be present in the CEG, in the specification, or later on, in the implementation itself.

As a conclusion to this chapter, we observe that several methods exist to derive a decision table from a CEG. Each method has its advantage and disadvantage. Through experience in specific systems, a tester can determine which method is most appropriate to do a better job. For instance, if a system does not require intensive testing then the ENF approach may be used. If the system requires high reliability then solution 2 in the Basic path sensitization method will be most appropriate, while if the system requires a good level of reliability, either solution 1 of the Basic path sensitization or Arabatzis method can be used.

Chapter 3

Extending Cause-Effect Graphing Technique for Telecommunications Systems

In this chapter, we show how to extend the CEG approach discussed in the previous chapter to take into account the key features of interactive systems, in particular, the ordering of causes and events over time. A key application discussed here is to telecommunications systems.

Today, there is a great need for reliable communications system software. Most communications systems are made up of thousands, sometimes millions lines of code. Implementing and maintaining such systems are very costly. Therefore, these systems, when implemented, should respond exactly and correctly to the customer's needs. One way to help achieve this objective is to start from a valid, correct, unambiguous, and complete specification of what the user wants in the first place.

Telecommunications specifications are very specific by nature in the sense that they represent real-time systems which consist of a set of functionalities where some of the functionalities may happen sequentially, others concurrently, and still others may occur only for a given period of time. These specifications tend to be both large and complex, consisting of several services and features. For a long time, in the telecommunications industry, services and features were specified on an informal basis, i.e in natural language. Owing to the fact that natural language is ambiguous by nature and cannot be manipulated mathematically many researchers have searched for ways to represent formally telecommunications features and services within a formal system where mathematical logic and proofs can be applied. In doing so they have come up with ***Formal Description Techniques*** (FDTs). The use of formal description technique makes it possible to analyze and simulate alternative system solutions. Formal description techniques offer a well defined

set of concepts for the user of the techniques, improving his/her capability to produce a solution to a problem and to reason about the solution.

Among the formal description techniques that are used today to represent telecommunications systems are ***SDL*** (Specification and Description Language)[BeHo89], ***ESTELLE*** [BuDe87], and ***LOTOS*** (Language Of Temporal Ordering Specifications) [BoBr87]. Whatever the FDT chosen to represent a feature or a service we always start from its requirements which usually is informal. Therefore, in order to capture all the functionalities and concepts in the informal requirement we need to understand and analyze natural language representation of telecommunications features and services.

This chapter is organized as follows. The next section gives a brief description and analysis of natural language telecommunications requirements and at the same time we show how SDL is used to represent some of the constructs found in these requirements. Section 3.2 shows how the Cause-Effect Graphing technique extended to represent telecommunications requirements and at the same time we show how some SDL constructs for processes can be mapped onto the ECEG. It should be noted that SDL is a minor point of motivation here because our objective is only to show that just as SDL constructs for processes capture the basic concepts in telecommunications requirements so can the Extended Cause-Effect Graphing technique. Section 3.3 shows how to derive telecommunications scenarios from a decision table obtained through the ***Extended Cause-Effect Graphing*** (ECEG) approach.

3.1 Analysis of Telecommunications Requirements from Natural Language Descriptions.

Informal specifications, i.e specifications written in natural-language (English for instance), are analyzed in this section. The objective is to derive the general structures of telecommunications system requirements and to show how SDL captures these natural language constructs. The specification language SDL was chosen among the FDTs because it is a high-level language which seems to be the closest to informal specifications.

3.1.1 General Structure of Informal Telecommunications Requirements

Consider any informal telecommunications system specification. This specification is usually written in English and is thus made up of sentences. In general, each sentence adds

meaning and at the same time clarifies the specification. The number of sentences in a given specification depends on the nature of the specification and on the needs of the customer. Groups of sentences are used to specify a given functionality in the future system. It can be thought that the more you have sentences describing a given functionality the more that function becomes clear and complete. However, this is not always the case. This is because some sentences may contradict others and thus introduce ambiguity into the specification. Moreover, some sentences can be redundant and confer no additional meaning. Therefore a good informal specification should contain only useful, clear, concise and unambiguous sentences.

Depending on the nature of the system under specification sentences in a specification can be classified into well-defined ***classes*** where each class conveys a specific meaning. It often happens that a sentence contains several distinct information (conditions, actions, events). In that case each information can be isolated from the sentence and classified into an appropriate class. For instance, sentences in informal telecommunications system specifications can be classified as follows (note that in this case a sentence is considered as a group of words (subject, verb, prepositional phrase) that is grammatically correct, that can stands on its own and that conveys a meaning):

Class 1	Sentences which define the communication environment. For example these sentences may give the number and the names of the communicating entities.
Class 2	Sentences denoting message transmission.
Class 3	Sentences denoting message reception.
Class 4	Sentences which describe the internal change when an entity receives or sends a message.
Class 5	Sentences that establish chronology of events.
Class 6	Sentences that introduce constraints on the system.
Class 7	Sentences that give conditions to be satisfied for some events to occur.
Class 8	Sentences that introduce examples in the specification.

The classification of sentences helps in analyzing an informal specification to spot causes and effects (section 3.2). For instance, sentences that fall in classes 1 or 4 usually show the presence of *state causes* or *state effects* respectively. Sentences in class 2 or 7 are more likely to be *event causes* while those in class 3 will usually be *event effects*. Sentences in class 5 introduce the semantics required to represent order of events in the ECEG. Sentences in class 6 give the constraints to be annotated on the ECEG while those in class 8 can actually be used as *user-directed* test scenarios.

3.1.2 Common mistakes in informal specifications

Since natural language is by nature ambiguous as different persons will understand the same sentence differently, one is bound to make mistakes, have omissions and create ambiguities when analyzing an informal specification.

One of the major defects present in informal specification is incompleteness. Incompleteness means that the functionality requested by the customer is not fully defined. Therefore, with such a specification, at the design phase the designer will have to make certain assumptions that may or may not correspond to the customer's wish. Some examples of incompleteness in telecommunications systems specifications are as follows :

- When a communicating entity sends a message m, nothing is said about reception of that message
- The ordering of message transmission is not mentioned.
- Nothing is said about the number of times a message m can be sent before a positive acknowledgement is received.
- Transmission errors are ignored.

Another major defect in natural language specifications is ambiguity. This type of defect is very annoying and often may be dangerous. This arises when the functionality of the system is not properly described i.e. contains sentences that have double-meaning.

For a long time, many of these defects were detected during the design phase or the implementation phase of the software life-cycle. The later an error is found, the more costly and the more difficult it is to correct it. Therefore, we need a technique that will help the

specifier in making correct and complete specifications. Today FDTs allow validation of specifications before the design phase. We believe that the same objective can be reached when using the ECEG approach.

3.1.3 The Specification Description Language (SDL)

SDL is a standard language for the specification and description of systems. It has been developed and standardized by CCITT (International Telegraph and Telephone Consultive Committee). The development of SDL started in 1972 with its latest version in 1992. SDL has been designed to specify and describe the functional behavior of telecommunications systems. Today SDL is mainly known within the telecommunications field, but it has a broader application area.

SDL covers different levels of abstraction, from a broad overview down to detailed design level. It has not been intended to be an implementation language. Automatic translation of a SDL specification to a programming language is possible.

The behavior of a system in SDL is constituted by the combined behavior of a number of *processes* in the system. A process is a finite state machine, that works autonomously and concurrently with other processes. The cooperation between processes is performed *asynchronously* by discrete messages, called *signals*. A process must always be contained in a *block* , which is the main structuring concept. A system contains one or more blocks, interconnected with each other and with the boundary of the system by *channels* . A channel is a means of conveying signals. A block can be partitioned into (sub)-blocks and channels, similarly to the partitioning of the system. Partitioned blocks do not contain any processes while unpartitioned ones contain only processes. Since our aim is to show how SDL captures natural language constructs, we will therefore limit our analysis of SDL to processes only (process communication, procedure, dynamic creation and termination of processes, and internal loops will not be addressed). We will use graphical SDL to illustrate the constructs used in SDL to capture natural language constructs.

In SDL there are five basic constructs for the description of a process : *start, state, input, output* and *nextstate*. Figure 3.1 shows the SDL/GR (Graphical SDL) representation of these constructs. The start is necessary to show the start of a process behavior. State is used to represent the present state of the system, and nextstate represents the next state of the

system after an action is done upon it. Input denotes the action made upon the system while output denotes the reaction of the system to the input.

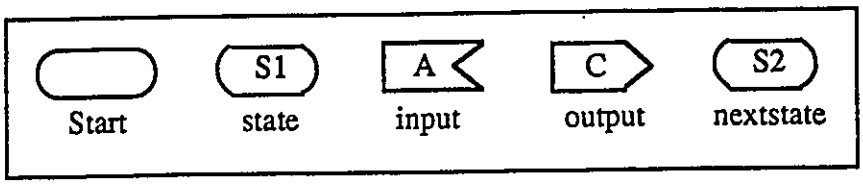


Figure 3.1 Basic constructs for the description of a process

A process may locally use and manipulate data stored in *variables*. During a transition a process uses its local variables using the *task* construct. A task construct is always an assignment in SDL. Figure 3.2 gives an example of variables and their use.

Sometimes it is necessary to specify different actions in a process depending on some conditions. This is done in SDL by using the *decision* construct. Different actions could be taken depending on the value of some variables. In the *decision symbol* a question is asked. The question may have various answers, but when interpreting the decision, one and only one answer should be correct. By using the ELSE answer we can guarantee that one answer is always correct. Figure 3.3 gives an example of the decision construct. In that example if the value of variable v is less than 10 we will go to state s2, if v is between 10 and 20 then we will go to state s3, else if v is greater than 20 then we will go to state s4.

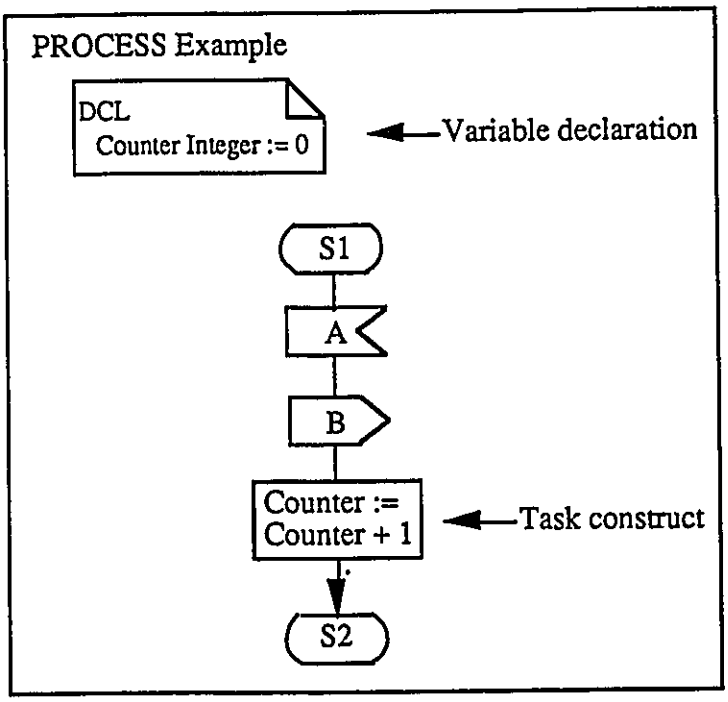


Figure 3.2 Manipulation of variable in SDL

We have shown the concepts of input and output in connection with a transition. They were only used to initiate a transition. However, they can also be used to carry data from one process to another. In this case, the output construct includes one or more data values. In the corresponding input construct, variables are specified which stored the received data values. The types of the values sent in the output must be compatible to the types of the variables in the corresponding input construct. In Figure 3.4 the values *false* and *10* are sent from process p1 to process p2 using signal A. Process p2 owns variables

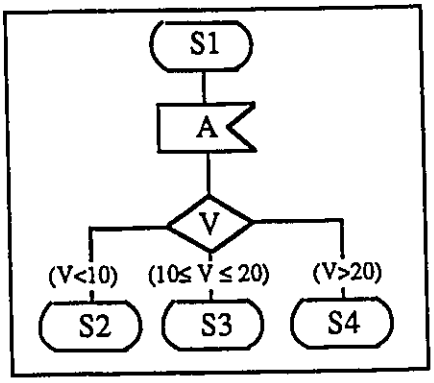


Figure 3.3 Use of the decision construct

v1 and v2 of type Boolean and Integer, respectively. When signal A is consumed by process p2, the value *false* is assigned to v1 and *10* is assigned to v2.

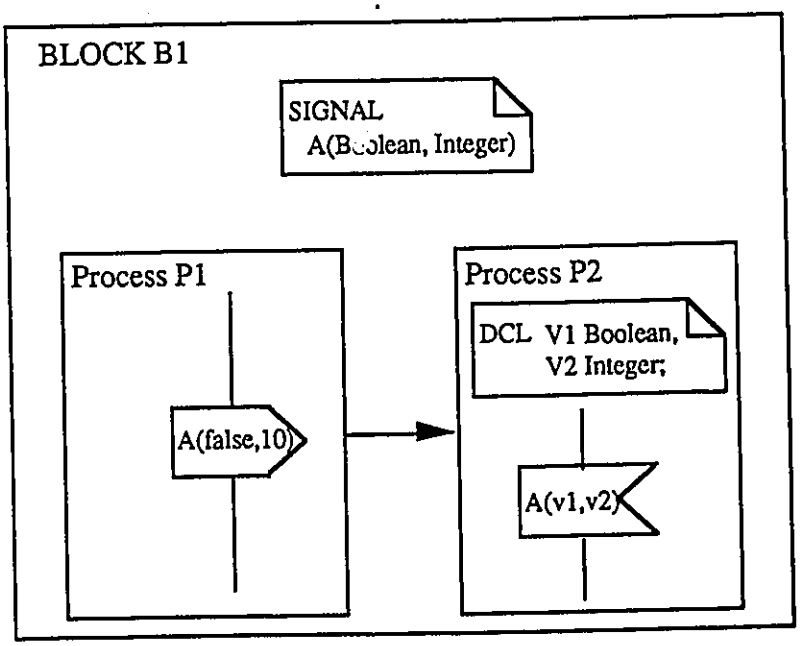


Figure 3.4 Signal with data

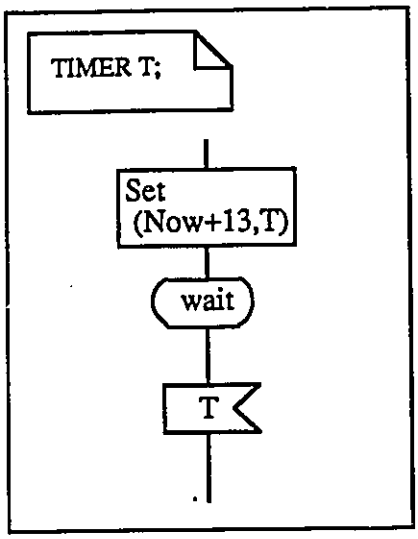


Figure 3.5 Description and use of a timer

Usually, in some telecommunications systems certain time constraints must be defined. In SDL the *timer* construct is usually used for this. The timer is an object, owned by a process, that is able to generate a timer signal and put this signal into the ***input queue*** † of the process. Figure 3.5 gives an example showing this construct.

The constructs in SDL shown above are limited to a subset of constructs in processes. We believe that these constructs will capture most of the constructs found in telecommunications requirements. In the next section we will show how the Extended Cause-Effect Graphing technique can be used to represent these constructs.

3.2 Extended Cause-Effect graphing Technique (ECEG)

Simplistically, a telecommunication system may be viewed as a black box that reacts to external stimuli (events) by producing at times some outputs (responses). At any given moment the system may either be in a stable state (i.e it can stay in that state indefinitely) or in an unstable state (i.e it will have to go to a stable state within some predefined amount of time). When an external event happens the system may change state and at the same time may produce some outputs. The objective of the extended cause-effect graphing technique is to represent a system (or part of it) in a graphical way, and then validate it by taking the necessary information from its corresponding Natural-Language specification. Since this approach is in essence a black box approach (no explicit reference to code or design), we will

† each process has an input FIFO queue where input events are queued

look for *external events*, *possible system states*, and *possible observable outputs* from the system specification. Later, this information will be converted into an Extended Cause-Effect Graph before proceeding to the validation phase.

A Cause-Effect Graph consists of a set of nodes, called *causes* and *effects*, a set of *arcs* with Boolean operators (*and, or*) between the nodes representing the relation between causes and effects, and a set of constraints representing the environmental (realism) constraints on the system. We extend this usual definition to yield Extended Cause-Effect Graphs for telecommunications services.

3.2.1 Extended Cause-Effect Graph Constructs for Telecommunications Services

We now will redefine causes, effects, and constraints as follows:

Causes

Causes implicit in a Natural-Language Specification of a telecommunications system may fall into one of the following classes:

- Event - represents input to the system
- State - represents the state of the system
- Condition - represents predicate of the system that is neither an event nor a state

In order to indicate the class of a cause the following notation should be used when an identification number is given to a cause :

1. Precede the identification number of a cause with "ce." when that cause represents an event.
2. Precede the identification number of a cause with "cs." when that cause represents a stable state. Replace s by s' if the state is unstable.
3. Precede the identification number of a cause with "cc." when that cause is a condition.
4. An effect can become a subsequent cause (this will arise in Example 1 below). This is a common aspect of telecommunications software specifications. Its identification number is not affected.
5. Once the causes are placed into classes then we can initialize the identification number of each class to "class_type.1", and then number them consequently.

Example 1

Consider the following Three-Way Calling specification. *"With Three-Way Calling, if A is engaged in a call with B, and wishes to add C to the call, then A depresses the switchhook button momentarily (an action called "flashhook") to put B on hold; the switching node responds to the flashhook by sending A a dialtone; when A receives the dialtone, A dials C's number; after C answers, A performs a second flashhook to remove B from hold and begin the Three-Way Call. After the TWC has been established, A can remove C from the call with a flashhook, or if either B or C hangs up, a standard two-party call continues between A and the remaining party. Whenever A hangs up the call is terminated."* [Came91]

Table 3.1 shows the causes that can be derived from the above specification (note that we have indicated by a star causes that are effects too. Effects will be defined below).

Effects

As for the case of causes, we can also classify the effects observed from telecommunications systems. However, causes need to be atomic while effects should contain all outputs observed for that particular case. Hence, we are more tempted to place the observable outputs and any system state change into the same effect, thus decreasing the number of nodes in the corresponding cause-effect graph. But, in telecommunications systems, effects are quite often reused as causes which in turn require an atomic nature. Moreover, the system may reach the same state for different effects. Hence, we should make a compromise between the size of a cause-effect graph and the reusability of effects. The following seems to be the most appropriate compromise based on numerous case studies performed on telecommunications requirements. Start by classifying effects into the following classes:

- Event - represents an observable output of the system
- State - represents a state reached by the system
- Event and State - represents both output and state reached

Next perform the following operation for each effect e_i in the "Event and State" class.

IF ((e_i is reused) and (e_i contains a state present in other effects too)) **THEN**
{ build two effects e_{ie} and e_{is} where e_{ie} will represent the event in e_i and
e_{is} will represent the state in e_i

}

ELSE

{ do not make any modification for that effect }

Causes	Identification Number
*A is engaged in a call with B	cs.1
A executes flashhook	ce.1
A dials C's number	ce.2
C answers phone	ce.3
B hangs up	ce.4
C hangs up	ce.5
A hangs up	ce.6

Table 3.1 Example of causes and corresponding identification numbers

In order to identify the class of a given effect the following notation should be used when an effect identification number is assigned:

1. Precede the identification number of an effect with "ee." when that effect represents an event.
2. Precede the identification number of an effect with "es." when that effect represents a stable state. Replace s by s' when the state is unstable.
3. Precede the identification number of an effect with "ees." when that effect represent both an event and a state.
4. Whenever a cause becomes an effect do not modify its identification number.
5. Once the effects are placed into classes then we can start the identification number of each class with "class_type.1".

Example 2

Consider the same Natural-Language Specification given in Example 1. Then Table 3.2 shows the effects and their identification number that can be obtained from the specification (here we have included the effects that are causes too - we have indicated them by a star in the table).

Constraints

In general constraints are used to show certain combination of causes and/or effects that are impossible because of syntactic or environmental constraints. The ECEG approach allows presence of five constraints : *Requires, Exclusive, One-and-only-one, Inclusive,* and *Mask.* The first four apply to causes while the last one applies to effects. For example, in example 1 we can observe easily that cause ce.3 (C answers phone) requires the presence of cause ce.2 (A dials C's number).

These constraints can also be used to represent invariant in telecommunications systems. Below we show an example of this. In the next section we will show how the Requires constraint may be used to represent ordering of events.

Effects	Identification Number
*B goes on hold. Switching node sends dialtone to A.	ees.1
*A receives dialtone	ee.1
C removed from call.	ee.2
*Connection between A and C established	es.1
*B removed from hold. TWC established	es.2
B removed from call. Std. two-party call for A and C	ees.2
Call is terminated	ee.2

Table 3.2 Example of Effects and corresponding identification number

Example 3

We know that in telephony systems one of the invariant is that a subscriber A cannot both be engaged in a call and be onhook. Hence, we will have to show this invariant on the ECEG. This can be shown as follows:

One-and-one-only (A engaged in a call with B, A is onhook)

However, it should be noted that whenever some invariant is this simple and easily spotted we don't need to make it explicit and overload the ECEG with it. In cases where the invariant is not obvious, we should show it on the ECEG.

Event Sequencing

Usually in telecommunications systems events are partially ordered, i.e for a given event to occur some other events have to occur first and so on. To illustrate this aspect we

will give an example of a finite-state machine and an equivalent cause-effect graph where the sequence of events is not made explicit.

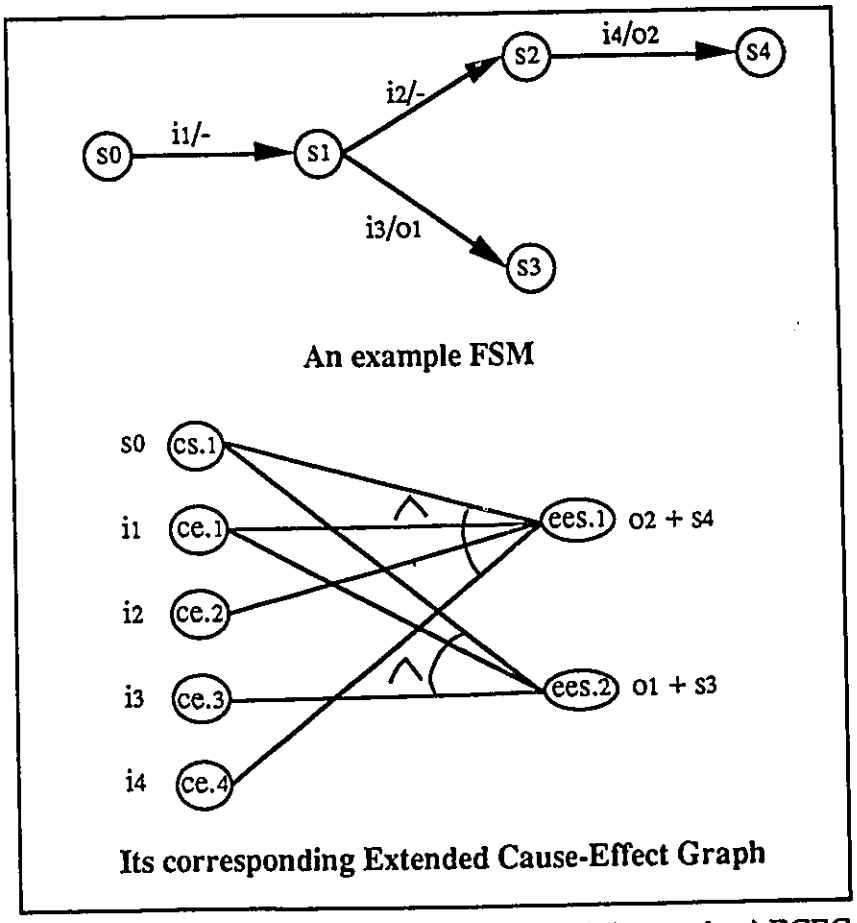


Figure 3.6 Example of FSM and its related (incomplete) ECEG

Example 4

From Figure 3.6 we can notice that from the ECEG we cannot deduce the order of occurrence of events. In order to represent the order of events in ECEGs we have developed

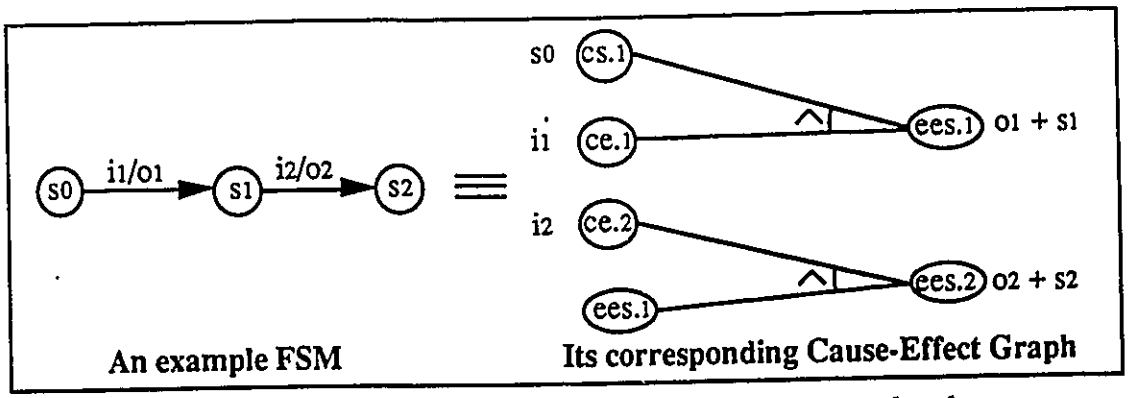


Figure 3.7 Example of reuse of effects method to represent ordered events

two methods which complement each other. One of the methods is *reuse of effects* while the other one is *use of the Requires constraint..*

In the reuse of effects method, we recognize when a combination of causes produces an intermediate event and subsequently the intermediate event with combination of some other causes produces another effect. Thus, the intermediate event is first represented as an effect and then used as a cause. By sequencing intermediate events, we can therefore represent the order of events. This method is illustrated in Figure 3.7.

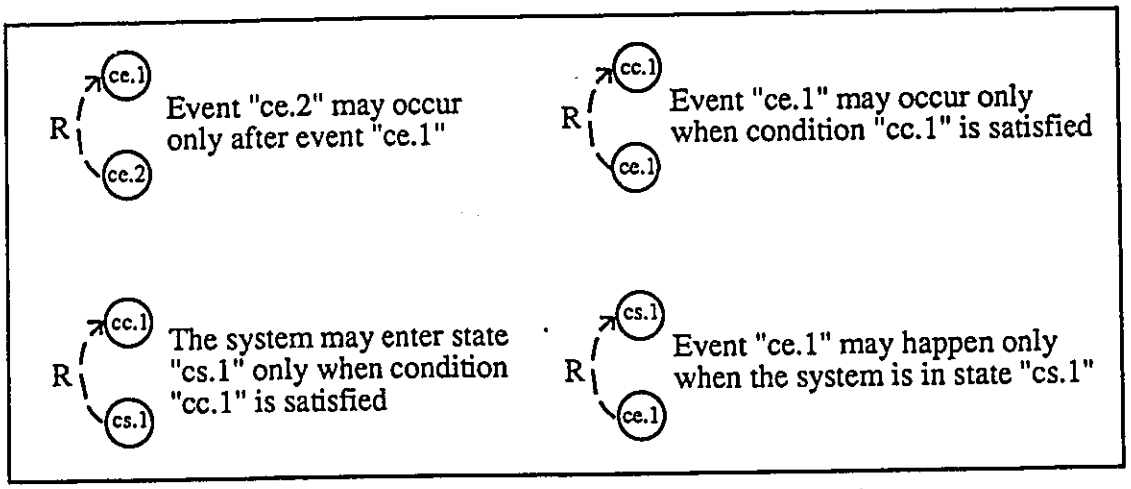


Figure 3.8 Introduction of the Requires constraint

The reuse of effects method works as long as the intermediate events are observable. Consider the case where they are not observable (e.g in Figure 2.3 in Chapter 2 if when applying cause ce.1 and ce.2 no observable effects are produced). In this case we will use the second method, i.e. introduce corresponding Requires constraints into the Cause-Effect Graph. The idea is simple and is well illustrated in Figure 3.8.

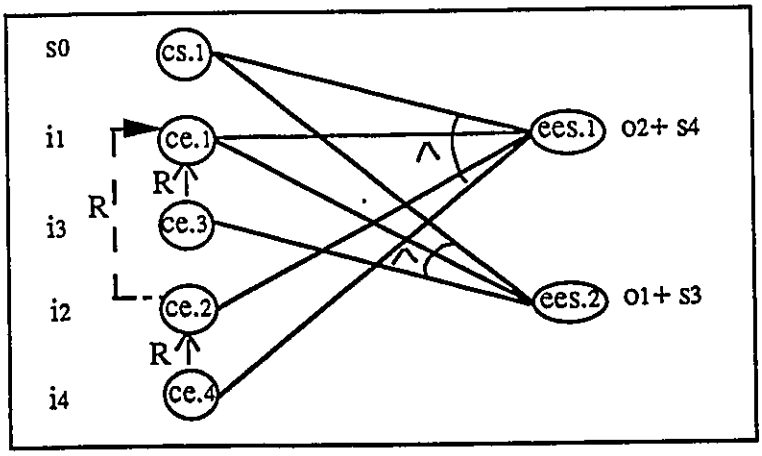
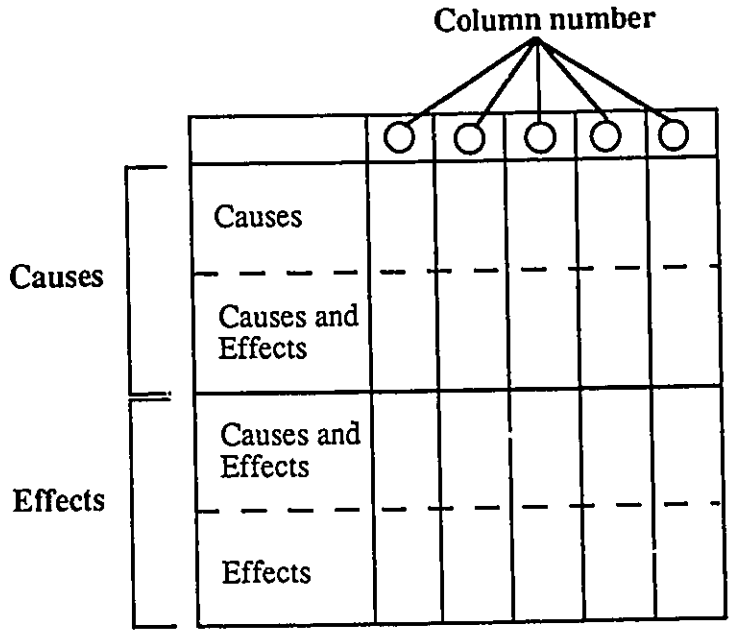


Figure 3.9 The complete ECEG from FSM in Figure 3.7

Note that this type of requires constraint is different than the existing (CEG) method usual which is used more to relate conditions rather than events . In the CEG case whenever a cause (condition) requires another cause, both causes should be present at the same time. In our new notation regarding events they may not be present at the same time. Figure 3.9 gives an example of using the requires constraint to represent ordering of events.

3.2.2 Modification to the Decision Table in the ECEG Technique

The decision table is modified so that causes which are also effects can be easily represented. Below we give the structure that the new decision table should have.



As can be observed, the causes that are also effects appear twice in the decision table: once considered as causes and once considered as effects. Note however that even if the causes that are also effects appear twice in the decision table, this does not imply that both occurrences will have the same value. To know which value should be assigned to each the following notation should be used:

Let C_{ce} represent a "cause and effect " in the cause part of the decision table.
Let E_{ce} represent the same "cause and effect " which appears in the effect part of the decision table.

IF (C_{ce} is set to 1) **THEN**

{ **IF** (presence of cause C_{ce} produce effect E_{ce}) **THEN**

set E_{ce} to 1

ELSE

set E_{ce} to 0

}

ELSE IF (E_{ce} is set to 1) **THEN**

{ **IF** (cause C_{ce} is required for the presence of E_{ce}) **THEN**

set C_{ce} to 1

ELSE

set C_{ce} to 0

}

Avoiding Incompleteness in the Decision Table

When the entries of a decision table are being assigned most of the time what can be observed is that there are some of the entries whose values are not important for the presence of some effects. This in itself is not a mistake, but if properly analyzed may lead to some faults in the natural language specification. But first of all we should try to fill most of the entries by simple deductions. The following procedure may be found useful:

1. Use all the constraints in the ECEG to fill most of the missing entries.
2. Usually, the whole system should be in a single state at a given moment of time. Therefore, whenever one cause (effect) state entry is set to 1 then set all other cause (effect) state entries to 0.
3. If after applying 1 and 2 there are still missing entries (these would most probably be cause events and effects too), then for each column where there is one or more missing entries we should argue if these particular cause events could occur concurrently with those already present. If yes then set these entries to 1, deduce the effects and see if no mistake arise. Else set the missing entries to 0.

Note the following invariant in any decision table :

IF all cause entries (causes + causes and effects) are set either to 0 or 1 **THEN**

All the effect entries (effects + causes and effects) are set either to 0 or 1

The same methods given in chapter 2 to derive a decision table from its corresponding cause-effect graph can also be used in the case of extended cause-effect graph without any major modifications.

3.2.3 Rules for constructing and interpreting a Telecommunications ECEG

In order to properly represent telecommunications systems using the Extended Cause-Effect Graphing Technique some rules have to be observed in the construction and analysis of an ECEG. Below we give some of the rules that have to be observed.

1. Always start with a cause node (of state type) that represents the starting system state.

2. Since we are mostly interested in representing the functionality of telecommunications features, we will not consider all possible situations whereby this given feature could be activated. We will therefore disregard all the past histories of the system before that feature is activated. Moreover, we assume for now that only the events mentioned in the specification actually happen and nothing else during operation of that feature. Later, when we tackle the feature interaction problem, we will consider the *possible contexts, possible external events, and histories.*

3. Cause events happens during a small time period (i.e. are more or less spontaneous), while cause states and conditions may remain at a certain value for a longer period of time depending when events will occur to change their values.

4. When a cause event,ce, is set to 1 (present), if that cause contributes to the presence of a given effect is determined, after presence of that effect the cause event (ce) will be reset to 0 (absent).

5. From 4 and 3, if the presence of a cause event, ce, brings about the presence of an effect, e_1, which is reused along with ce to produce another effect, e_2, then cause ce has to occur twice, once to produce e_1, and another one to produce e_2.

6. We assume that the delay between the causes and the effects is practically zero. Obviously in some cases, especially when tackling the feature interaction problem (its definition is given in Chapter 4), we will have to consider all the possible cases where delay may become important.

The rules are general enough to take into consideration most of telecommunications systems. In chapter 4 we will modify and enhance these rules in order to apply them specifically to telephony systems.

3.2.4 Mapping SDL constructs onto Extended Cause-Effect Graphing Technique

In this section we will show how the SDL constructs shown in section 3.1.3 can be represented using the ECEG constructs. It should be noted that SDL is a minor point of motivation here because our objective is only to show that just as the SDL constructs shown in section 3.1.3 capture the basic concepts in telecommunications requirements so can the Extended Cause-Effect Graphing technique, hence supporting the generality of the ECEG technique. Figure 3.13 illustrates this mapping.

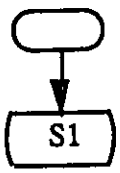
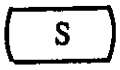
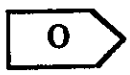
SDL representation	Corresponding ECEG representation
1. The start 	 (Represented by a starting cause state)
2. State representation 	 Either a cause or an effect state (or both)
3. Input 	 (Cause event)
4. Output 	 (Effect event)

Figure 3.10(a) Mapping SDL onto ECEG

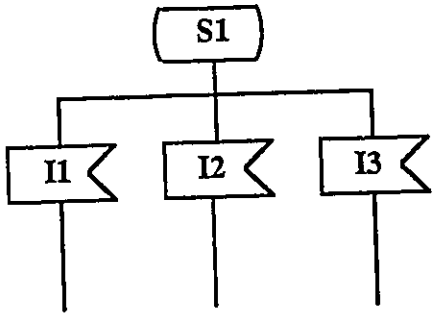
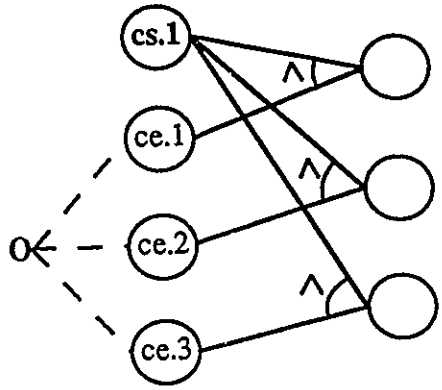
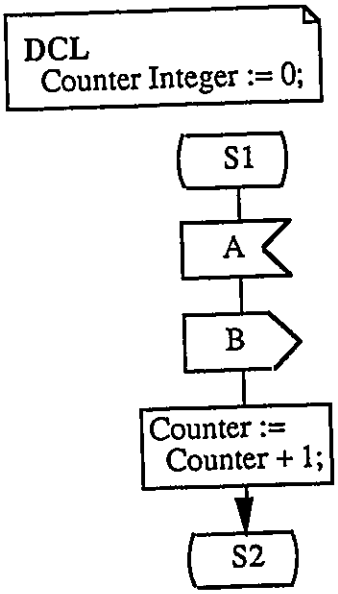
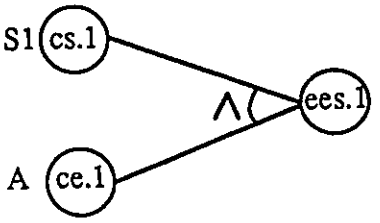
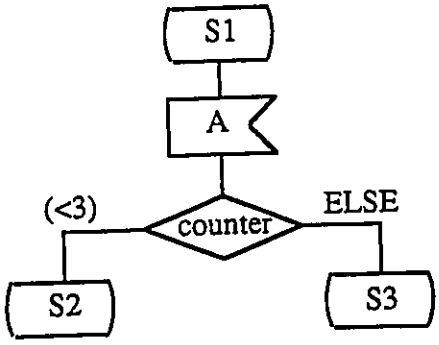
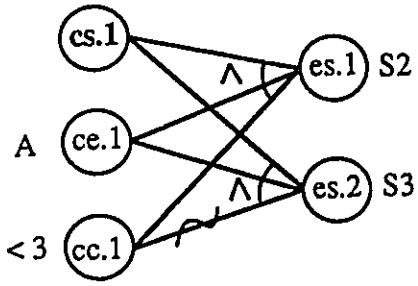
SDL representation	Corresponding ECEG representation
5. Choice of actions 	
6. Local variables 	Var Counter : integer := 0;  B is output, increment counter is incremented by 1 , new state is S2
7. Decisions 	

Figure 3.10(b) Mapping SDL onto ECEG

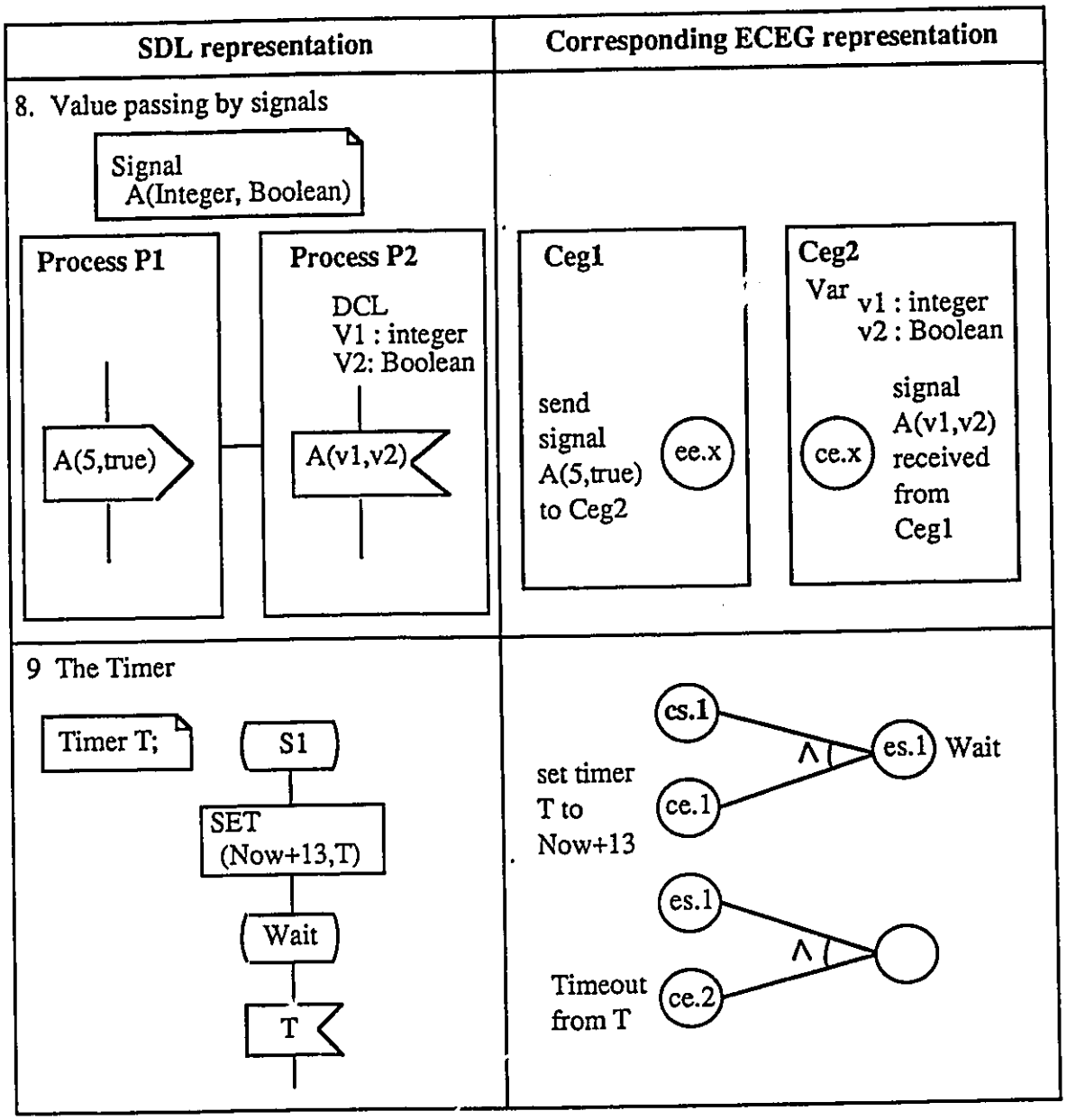


Figure 3.10(c) Mapping SDL onto ECEG

Example

We will use the specification of the system *Daemongame* [BeHo89] to illustrate the concept of mapping SDL onto ECEG. The analysis and representation of this system in SDL is found in many papers on SDL. It is therefore, appropriate to use this system to illustrate the representation power of ECEG. The Natural Language specification is as follows:

"This system is a game having any number of players. The players belong to the environment of the system. A "daemon" in the environment of the system sends Bump signals randomly to the system. A player has to guess whether the number of the Bump signals received by the system is odd or even. The guess is made by sending a Probe signal to the system. The system replies by the signal Win if the number of the received Bump signal is odd, otherwise by the signal Lose. The system keeps track of the score of each player. A player can ask for the current value of his score by the signal Result, which is answered by the system with the signal Score. Before a player can start playing, he must log in. This is accomplished by the signal Newgame. A player logs out by the signal Endgame." [BeHo89]

From the specification above, we can derive the causes and effects and they are shown in Table 3.3. These types of causes and effects are easily identified since most of them are about sending and receiving of messages It can be noticed that effect "New game started" has been reused because only after a new game has started that the player can send bump signal to the system. We have used the cause "System in ready state" to indicate the start of the whole process.

Causes	Effects
cs.1 System in ready state	ee.1 System increments the number of received Bump signals by 1
ce.1 System receives Bump signal	ee.2 New game started *
ce.2 System receives Newgame signal	ee.3 System sends the Win signal and update the score of player
ce.3 System receives Probe signal	ee.4 System sends the Lose signal and updates the score of player
cc.1 The number of Bump signal received is odd	ee.5 Systems outputs signal Score
ce.4 System receives Result signal	ee.6 Game ends
ce.5 System receives Endgame signal	

Table 3.3 Causes and effects for the system Daemongame

From the causes and effects given in Table 3.3 and the informal specification we can derive the ECEG given in Figure 3.11. It can be noted that this is a valid representation as each of the SDL constructs (in the SDL graphical representation in [BeHo89]) is represented clearly by the ECEG.

3.3 Automated Derivation of Decision Table from ECEG

In Chapter 2 (Section 2.3.4.1), we give an algorithm to derive a Decision Table from a CEG. In order to make this algorithm applicable in the ECEG context, provisions have to be made to accommodate reuse of causes and effects as well. Since the aim of the Basic PST is to sensitize an effect to a cause, the algorithm itself does not make the difference between a normal effect and a reused one. This case is similar for the causes too. Hence, only a few

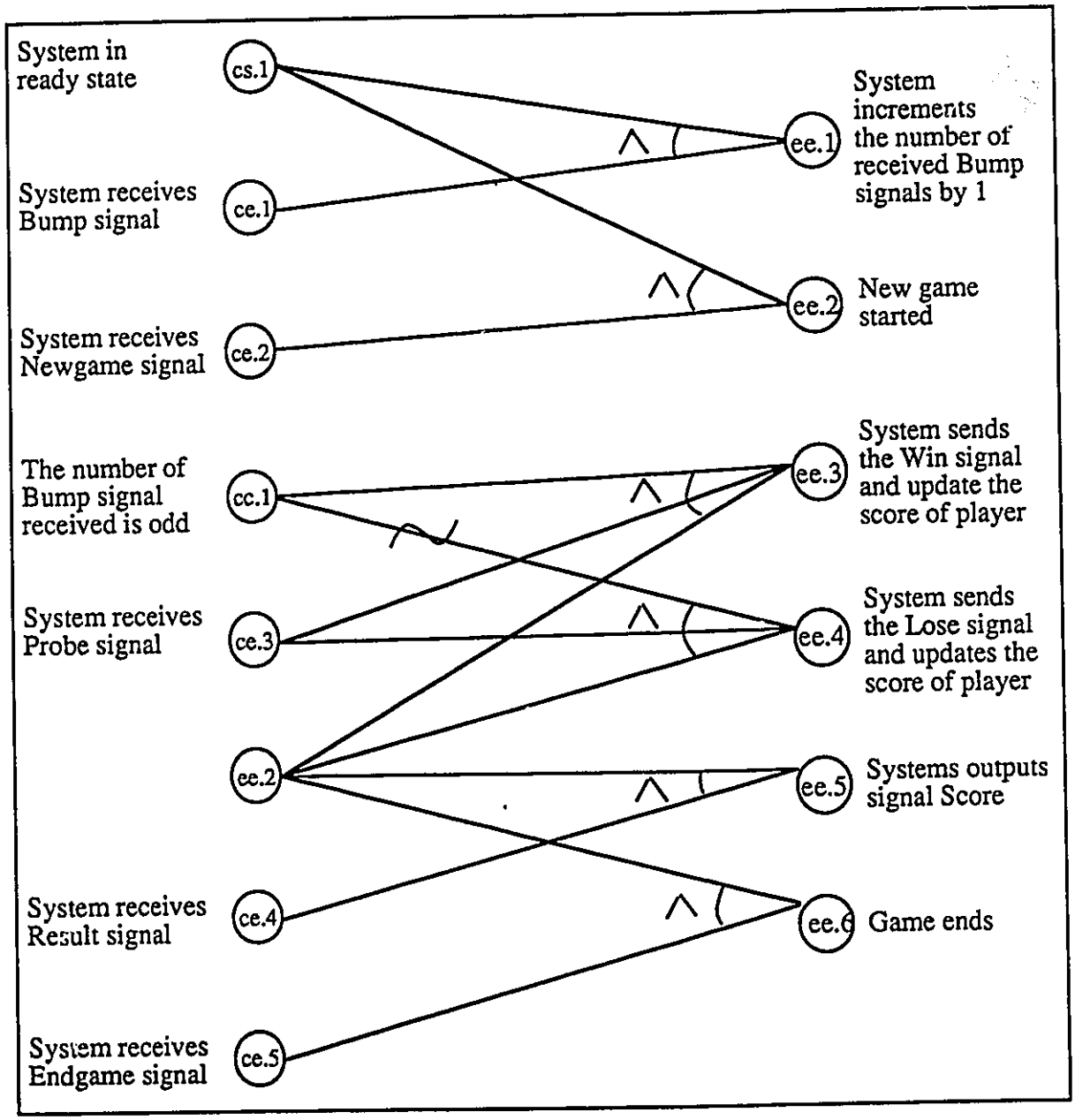


Figure 3.11 ECEG representation for system Daemongame

more steps are required for the Basic PST algorithm to make it suitable to derive a decision table from an ECEG.

The added step is as follows :

Step 0

- Create two nodes for each reused effect. One node will establish that effect as an effect, while the other one will designate that effect as a cause. Give different ids to the two nodes created (it is advisable just to prefix the cause identifier by a star.
- Create two nodes for each reused cause. One node will establish that cause as a cause, while the other one will designate that cause as an effect. Give different ids to the two nodes created (it is advisable just to prefix the effect identifier by a star.

Step 3

- When applying constraints to each column in the decision table, the *Requires* constraint should be interpreted differently from its usual CEG sense. In ECEG semantics the *Requires* constraint just introduces order of events. When checking for impossible combination of causes, this constraint will therefore be interpreted differently.

By separating each reused cause and effect into distinct nodes, the ECEG has been transformed into a CEG. The Basic PST can thus be applied to this transformed ECEG with proper interpretation of the *Requires* constraint as this constraint has different meanings in the CEG and the ECEG context.

3.4 Deriving User-Directed Scenarios

In chapter 2 we have shown several methods to derive a decision table from its corresponding cause-effect graph. After the decision table has been derived we can derive test cases each one corresponding to a column in the decision table. The test cases derived are then applied to the implementation of the system. This method is very useful if the specification conforms to what the client really wants, i.e the specification satisfies most of the user requests. But what happens when this is not the case?. One way of validating the system is through *user-directed scenarios*. User-directed scenarios are obtained from the decision table. A user-directed scenario consists of the causes that should be present and

those that should be absent as well as the effect produced or not (obviously instead of using the identifier of causes/effects we should use the English text of causes/effects). The scenarios are called user-directed because we should validate them against the objectives of the customer and therefore they should be quite easy to read, understand and unambiguous.

In the example given in Table 2.1 of chapter 2, if we take column 1 in the decision table the following will be a user-directed scenario :

> *Student can take the skating course :- student enrolls at university of Ottawa and he obtains an A+ in Mathematics.*

In the case of the ECEG technique we have developed two closely related user-directed scenarios : *simple scenarios* and *concatenated scenarios.* Simple scenarios are those scenarios which consist of an effect along with the causes responsible for the presence or absence of that effect. The causes that make up simple scenarios are not reused and they contain no effects that have been reused. In the ECEG given in Figure 3.14 an example of a simple scenario would be as follows:

> *A new game is started (ee.2) :- the system is in the ready state (cs.1)* **and** *the system receives a Newgame signal (ce.2).*

A concatenated scenario is just like a simple scenario except that it consists also of effects that have been reused. In fact, instead of writing the effects we will write the name of the simple scenario which brings about the presence or absence of that effect in question. For example, if we name the above scenario as scenario # 1 then an example of a concatenated scenario still based on Figure 3.14 will be as follows:

> *System sends the Win signal and update the score of the player :- Scenario # 1* **and** *the system receives a bump signal* **and** *the number of bump signals received is odd.*

The scenarios are given to the client who should approve or disapprove of each one. Consider the case where the user disapprove of a given scenario. At that point we can note five potential sources for this problem: the scenario may have been incorrectly derived from the decision table, the decision table may have incorrectly been obtained from the ECEG, the ECEG may be incorrect with respect to the specification, the specification may be incorrect,

or a combination of any one of the above. Since deriving a decision table and then the test scenarios are algorithmic steps we will ignore these sources of errors, assuming that the corresponding algorithms have thoroughly been tested and are correct. To know the source of the error, the ECEG should first be checked with respect to the specification to detect any omission or wrong interpretation. If no error is found after doing this, then we can say that the specification is incorrect or it has wrongly been interpreted by the tester. At that point any change in the specification should be made in collaboration with the customer.

In this chapter, the ECEG technique for telecommunications systems has been introduced as well as how this approach could be used to represent some of the constructs in SDL. It should be noted that SDL is a minor point of motivation here because our objective is only to show that just as the SDL constructs shown in section 3.1.3 capture the basic concepts in telecommunications requirements so can the Extended Cause-Effect Graphing technique, hence showing the generality of this approach.

In addition, since the ECEG testing technique starts from a natural-language specification, this technique will, therefore, represent all and only the information that can be gathered from the requirements and from the basic concepts of a telecommunications system. For instance, if the history of a system is not present in the requirement, then our technique will not represent it while if it is present then it will be represented. Moreover, since our technique is a black-box testing technique we will not represent internal events of a system. Only observable events will therefore be represented. The next chapter illustrates the ability of ECEG to represent telecommunications features and services.

Chapter 4

Representation of Features and Services by Extended Cause-Effect Graphs (ECEG)

In this chapter, we show how ECEGs are applied to very general distributed systems, namely to telephony systems.

Several distinct definitions for the terms feature and service are present in the literature [e.g Dwor89, Boum93, Grif93, Zave93, Inou92, Kuis93, Rana92, Came91, Cam93]. In some cases, the word "service" is used to mean "feature"; in other cases, "service" describes features together with other capabilities [Grif93]. Since in this chapter and the next one the terms features and services will be quite often used, it is necessary to give our definitions of these terms in order to clarify our intentions to the reader.

Definition 4.1 - Feature

A *feature*, f, provides one functional component of a service, S. S provides the environment necessary for f to operate as the latter cannot function by itself.

In the Extended Cause-Effect Graph (ECEG) representation, each feature will be represented by one ECEG.

Definition 4.2 - Service

A *service*, S, consists of a set of (possibly only one) features and the necessary overhead required to provide an environment where each feature in that set can be activated without any need for additional components. Note that "service" is often used to refer to a single feature and a necessary operational environment. ♦

In the ECEG representation, a service will be represented by a set of ECEGs where each ECEG will represent a specific feature. In principle, the whole service can be represented by only one ECEG but such a graph will be large and difficult to manage.

Definition 4.3 - Feature (Service) Interaction

A **feature interaction** between one feature, f_1, and another one, f_2, occurs when f_1, and f_2 function as expected when they are not present in the same service but they do not operate in the same way when present in the same service.

This section will show how informal specifications can be represented using the ECEG technique.

4.1 Representation of Telecommunications Features and Services

Telecommunications specifications are very specific by nature in the sense that they represent real-time systems which consist of a set of functionalities where some of the functionalities may happen sequentially, others concurrently, and still others may occur only for a given period of time. These specifications tend to be both large and complex. They usually consist of several services and features.

Such large specifications should be broken down into "workable" parts. This is necessary because Extended Cause-Effect Graphs become very large and difficult to manage when used on large specifications. The size of the workable part is determined by the nature of the specification and user convenience. Since services in telecommunications systems consist of a certain set of features, a "workable part" may be the specification of either a service if it is quite small or even an individual feature. Since our objective is to be able to validate the given informal specification, it is advisable to break down the large specification into individual features. Even if division means a loss of global representation, it appears to be the only method which promotes user understanding.

Later on, after each individual feature has been analyzed some or all of these features will be brought together in groups of two or more to see if the behavior of any one of the features is not affected by the presence of the other features, i.e to detect any feature interaction [Dwor89]. To be able to perform this last step, an algorithm is given in order to

compose together two ECEGs representing separate features into a single one which expresses all and only the behaviors of the features represented.

4.2 ECEG Representation of Individual Telecom Features

When the specification of each feature has been isolated from the original monolithic specification, then we should build an ECEG representation of that feature. To do so, we need first to identify the causes, effects, constraints, and relations between the causes and the effects. However, this step will be more efficient if some guidelines specific for telephony systems are followed.

In order to properly represent telephony features using the ECEG technique some guidelines have to be observed in the construction and analysis of an ECEG. The guidelines are quite general as their objective is mainly to make explicit what are implicit in a telephony ECEG. These guidelines follow directly from those given in chapter 3 except that in those presented here are more specific to telephony systems. The guidelines are classified into three sections: rules for the construction phase, rules for the interpretation phase and explicit assumptions found in an ECEG.

Construction Phase

1. Before spotting causes, effects, and constraints from a given telephony specification, identify each subscriber by a letter (A, B, C, ..) if this identification has not been done in the specification. This will help to clarify the specification.

2. Always start with a cause node (of state type) that represents the starting system state. Usually, this state is designated by cs.1

3. Disregard all the past histories of the system before the system is activated.

4. Every time a subscriber performs a new action upon the telecommunications system create a cause event (ce._) to represent that action.

5. Every time a subscriber arrives to a new observable state create an effect state (es._) (it may be an "event and state" effect (ees._) if an event is also produced) to represent that effect.

6. Every observable response by the system should correspond to an effect event (ee._).

7. Since our interest is in representing features from the customer's point of view the global system state will be represented by the actual state of each subscriber. The internal system state will therefore be ignored.

Interpreting Phase

1. Whenever a given effect state, es, is reached this state will represent the global system state. However, usually, only the state reached by a given subscriber is mentioned. The following gives the general interpretation rules used to determine the global system state (these rules are illustrated through examples):

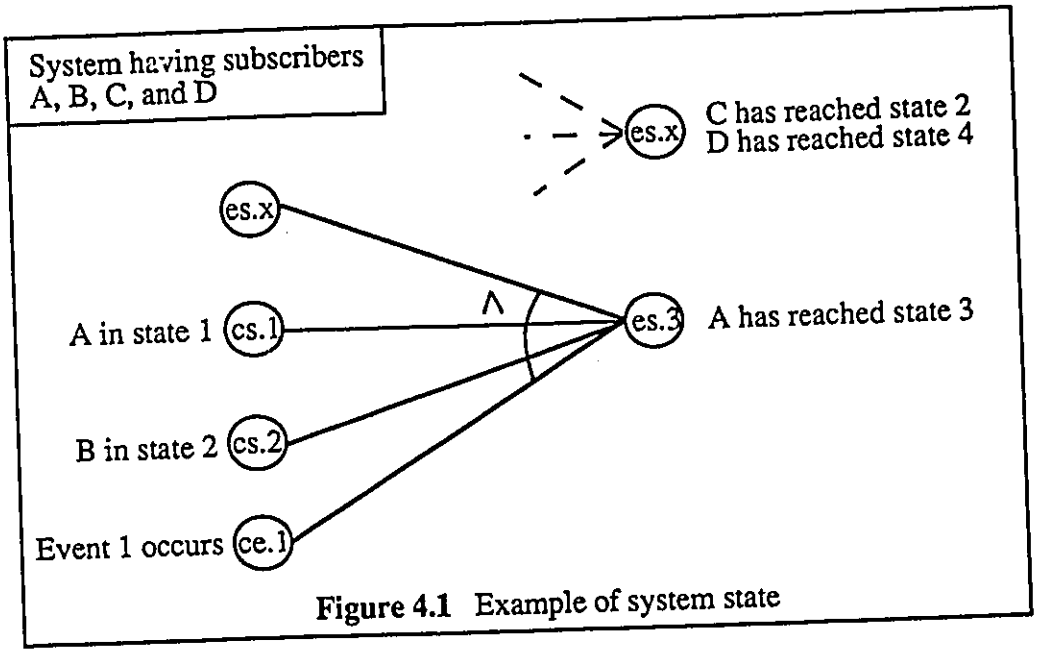


Figure 4.1 Example of system state

- Consider the example given in Figure 4.1. When effect es.3 is observed since only state of subscriber A is mentioned the state of the other subscribers will be unchanged. In our example, therefore, when effect es.3 is observed the global

system state will be <A in state 3, B in state 2, C in state 2, D in state 4>. Note how we backtrack to obtain the states of all subscribers.

- Consider now the example in Figure 4.2. When A is in state 1, B in state 2, es.x set to 1, and event 1 occurs both es.3 and es.4 will be observed (suppose that no effect has priority other the other one). In that case since both A and B have changed state the new system state will be <A in state 3, B in state 5, C in state 2, D in state 4>.

- Consider the example in Figure 4.3. When effect ee.1 is observed no change of state is observed. In that case the global system state has not changed, i.e has remained as <A in state 1, B in state 2, C in state 2, D in state 4>.

2. When a cause event,ce, is set to 1 (present), if that cause contributes to the presence of a given effect, after presence of that effect is determined the cause event (ce) will be reset to 0 (absent).

3. From rule 2, if the presence of a cause event, ce, brings about the presence of an effect , e_1, which is reused along with ce to produce another effect, e_2, then cause ce has to occur twice, once to produce e_1, and the second time to produce e_2.

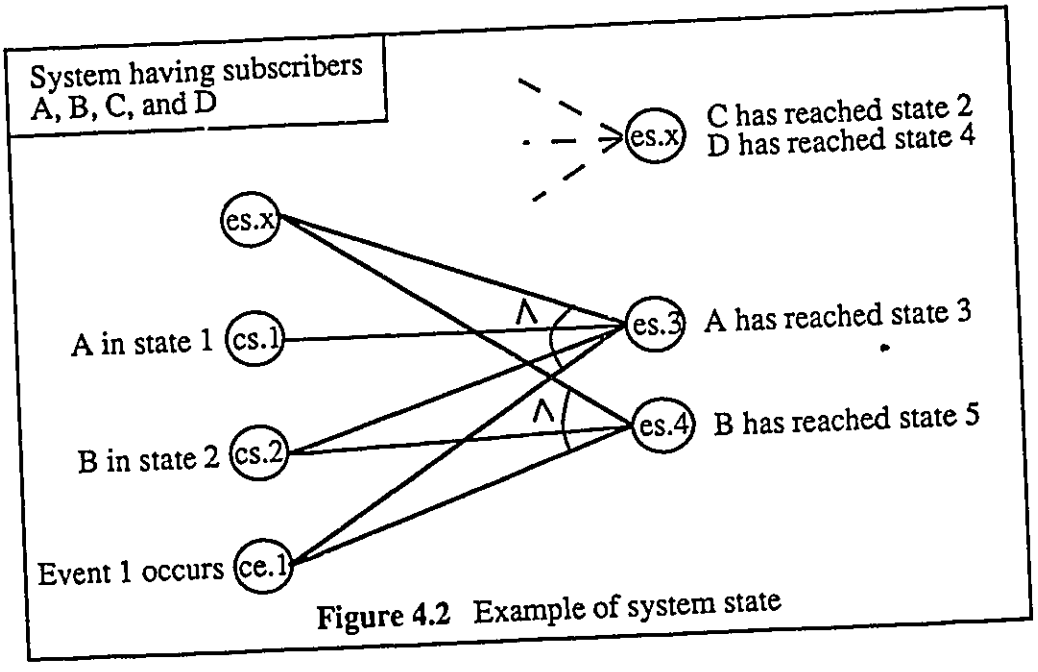


Figure 4.2 Example of system state

Explicit Assumption

1. At the starting state (cs.1) if the state of all subscribers participating in the functioning of a given feature are not mentioned explicitly (i.e they will participate in that feature at a later stage) then we will assume for now that their states at the start are unimportant.

2. Cause events happen during a small time period (i.e. are more or less spontaneous).

3. Cause states and conditions may remain at a certain value for a long period of time depending when events will occur to change their values.

4. We assume that the delay between the causes and the effects is practically zero except in the case where effects are reused.

5. Assume for now that only the events mentioned in the specification actually happen and no other events occur during operation of that feature

Later on, when the feature interaction problem is tackled, the possible contexts, possible external events, possible cases where delay may become important, and histories will be considered.

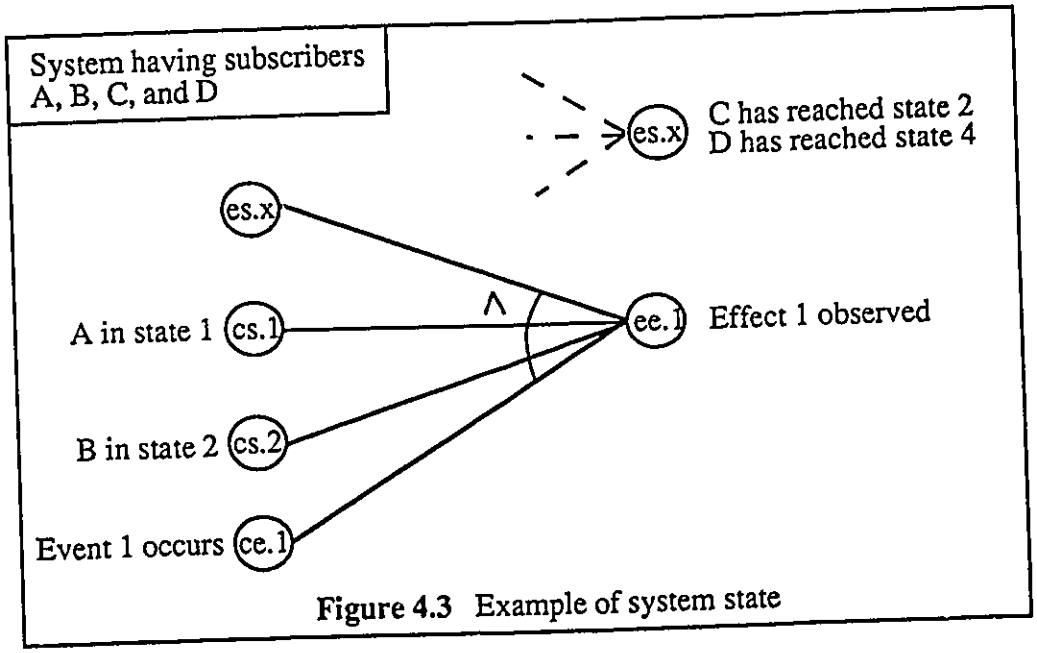


Figure 4.3 Example of system state

4.3 Examples: Telephony Features

The examples given in this section are well-known features from telephony systems. These examples are Three-Way Calling and Call Waiting. For each of these examples their English language specifications are given, and it is shown how the causes and effects are derived and how their respective ECEGs are built. The specifications are taken from [Came91].

4.3.1 Three-way Calling (TWC)

The natural language specification is given as follows : " *With Three-Way Calling, if A is engaged in a call with B, and wishes to add C to the call, then A depresses the switchhook button momentarily (an action called "flashhook") to put B on hold; the switching node responds to the flashhook by sending A a dialtone; when A receives the dialtone, A dials C's number; after C answers, A performs a second flashhook to remove B from hold and begin the Three-Way Call. After the TWC has been established, A can remove C from the call with a flashhook, or if either B or C hangs up, a standard two-party call continues between A and the remaining party. Whenever A hangs up the call is terminated*" [Came91].

The causes and effects derived from the English Language specification are given in Table 4.1 below. A star at the end of a cause or an effect statement means that this cause or effect will be reused as an effect or a cause respectively. At the start, the state of the system is "A is engaged in a call with B". We therefore make this statement a state cause (cs.1). When A depresses the switchhook an action is made and the result of that action is that B goes on hold and the switching node sends A a dialtone. An event cause ("A execute flashhook", (ce.1)) and an effect that contains both an event and a state reached ("B goes on hold (state) are derived. Switching node sends dialtone to A (event)", (ees.1)). Note the timing constraint imposed by the specification : "when A receives the dialtone, A dials C's number". In order to represent this timing constraint effect ees.1 is reused as a cause and an effect "A receives dialtone" (ee.1) created. "A dials C's number" will become an event cause (ce.2), but since A should first obtain a dialtone therefore effect ee.1 will be turned into a cause too.

Causes	Effects
cs.1 A engaged in a call with B *	ees.1 B goes on hold. Switching node sends dialtone to A *
ce.1 A executes flashhook.	ee.1 A receives dialtone *
ce.2 A dials C's number	es.1 Connection between A and C established *
ce.3 C answers phone	es.2 B removed from hold. TWC established *
ce.4 B hangs up	ees.2 B removed from call. Standard two-party call established btwn. A and C.
ce.5 C hangs up	es.4 Call is terminated.
ce.6 A hangs up	ee.2 C removed from call

Table 4.1 Causes and Effects of Three-Way Calling

In the specification it further says that " after C answers, A performs a second flashhook to remove B from hold and begin the Three-Way Calling". This expresses a sequence of events and also the fact that C answers means that a connection is made between A and C. Following the same principle as above, "C answers phone" is made into an event cause (ce.3), "Connection between A and C established" is made into a state effect (es.1) and then reused as a cause , while "B removed from hold and Three-way Calling established" turned into a state effect (es.2).

The specification continues by taking effect es.2 (" After TWC has been established") as the present state and shows various events that may happen afterwards (thus effect es.2 will act as a cause too). "A can remove C from the call with a flashhook" will be turned into an event effect "C removed from call" (ee.2). Note that the flashhook made by subscriber A need not be turned into a cause since a corresponding cause already exists. Cause events "B hangs up" (ce.4) and "C hangs up" (ce.5) are made up from "if either B or C hangs up". The results of these causes events (ce.4 or ce.5) will be " a standard two-party call continues between A and the remaining party". Two effects "C removed from call. Standard two-party call between A and B" and "B removed from call. Standard two-party call between A and C" (es.3) are therefore derived. The first effect consists of two effects ("C removed from call" and "Standard two-party call between A and B") which are already present. The last sentence "Whenever A hangs up the call is terminated" gives rise to an event cause "A hangs up"

(ce.6) and a state effect "Call is terminated" (es.4). Figure 4.4 shows the ECEG representation of TWC.

Note how sequence of events have been shown on the graph. Effects ees.1, ee.1, es.1 and es.2 have been reused. The requires constraint has been added during the phase where subscriber A calls C and the latter answers the phone in order to show the logical order of events. The English statement of the causes and effects have been annotated on the ECEG in order to increase readability of the graph. In an automated tool this will be done upon the user's request.

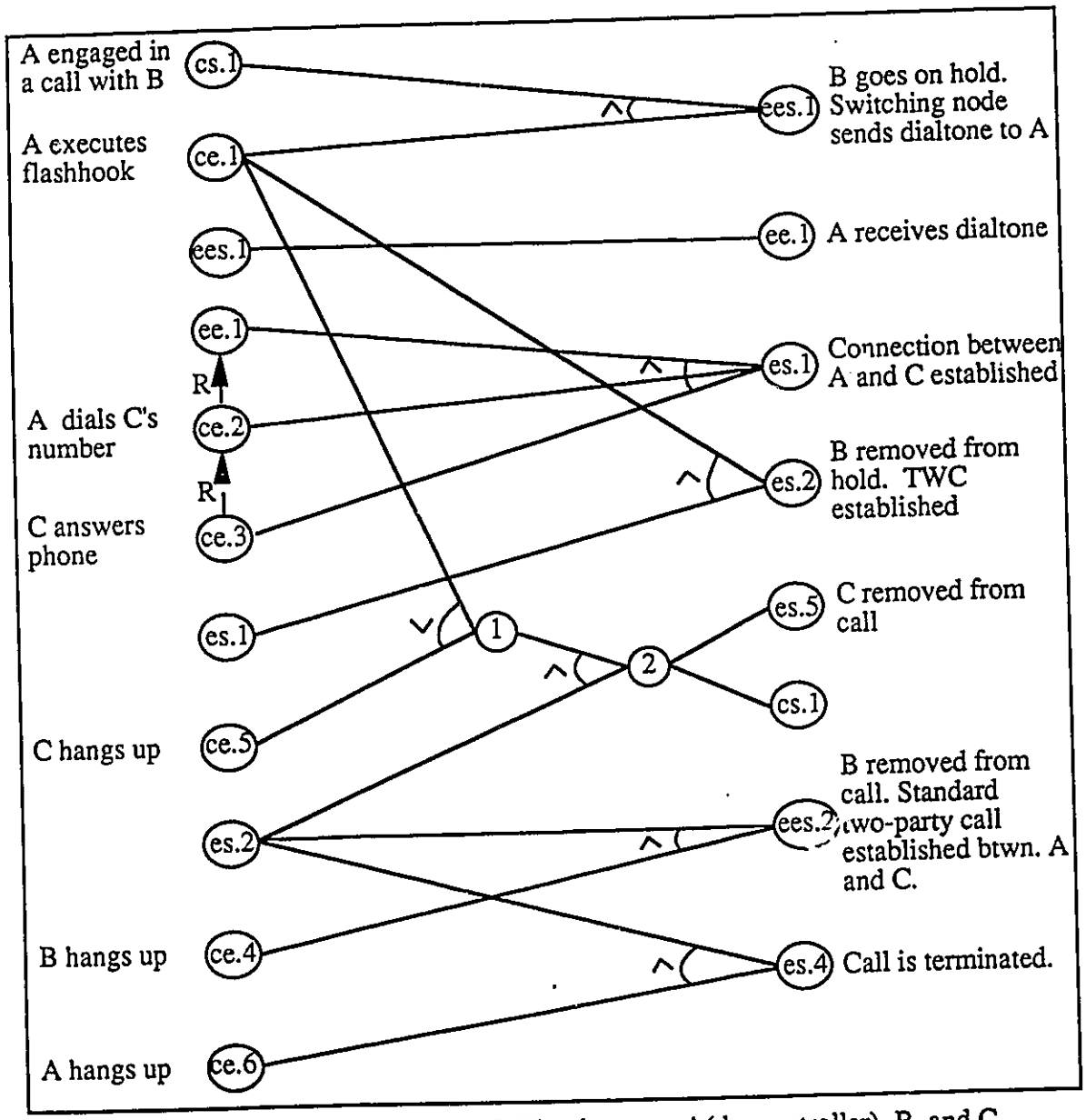


Figure 4.4 ECEG of Three-Way Calling between A(the controller), B, and C

4.3.2 Call Waiting (CW)

The natural language specification is given as follows : " *With Call-Waiting, if A is engaged in a call with B, and C calls A, a signal tone will be heard on A's line, and audible ringing will be heard on C's line. A can then place B on hold and answer C's call with a single flashhook. A can switch back and forth between the two calls via flashhooks, until one or the other of the calls is terminated. Should party A inadvertently hang up while either B or C is on hold, A will be automatically rung back, and upon answering is connected to the held party*" [Came91].

The causes and effects derived from the English Language specification are given in Table 4.2 below. A star at the end of a cause or an effect statement means that this cause or effect will be reused as an effect or a cause respectively. At the start, the state of the system is "A is engaged in a call with B". This statement is therefore made into a cause state

Causes	Effects
cs.1 A engaged in a call with B *	ee.1 Signal tone will be heard on A's line, and audible ringing will be heard on C's line *
ce.1 C calls A	es.1 B goes on hold. Connection between A and C established *
ce.2 A performs flashhook	es.2 C goes on hold. Connection between A and B established *
ce.3 B hangs up	ee.2 A is rung back due to B *
ce.4 C hangs up	ee.3 A is rung back due to C *
ce.5 A hangs up	es.3 B is disconnected
ce.6 A answers phone	es.4 A is connected to C
	es.5 C is disconnected

Table 4.2 Causes and Effects of Call Waiting

(cs.1). When C calls A an action is made and the result of that action is that a signal tone will be heard on A's line, and audible ringing will be heard on C's line. One cause event "C calls A" (ce.1) and one effect event "signal tone will be heard on A's line, and audible ringing will be heard on C's line" (ee.1) are thus derived. Effect ee.1 will be reused as a cause in order to show the order of events. When A performs a flashhook to answer C's call the following cause event can be deduced"A performs flashhook" (ce.2), the effect state "B goes

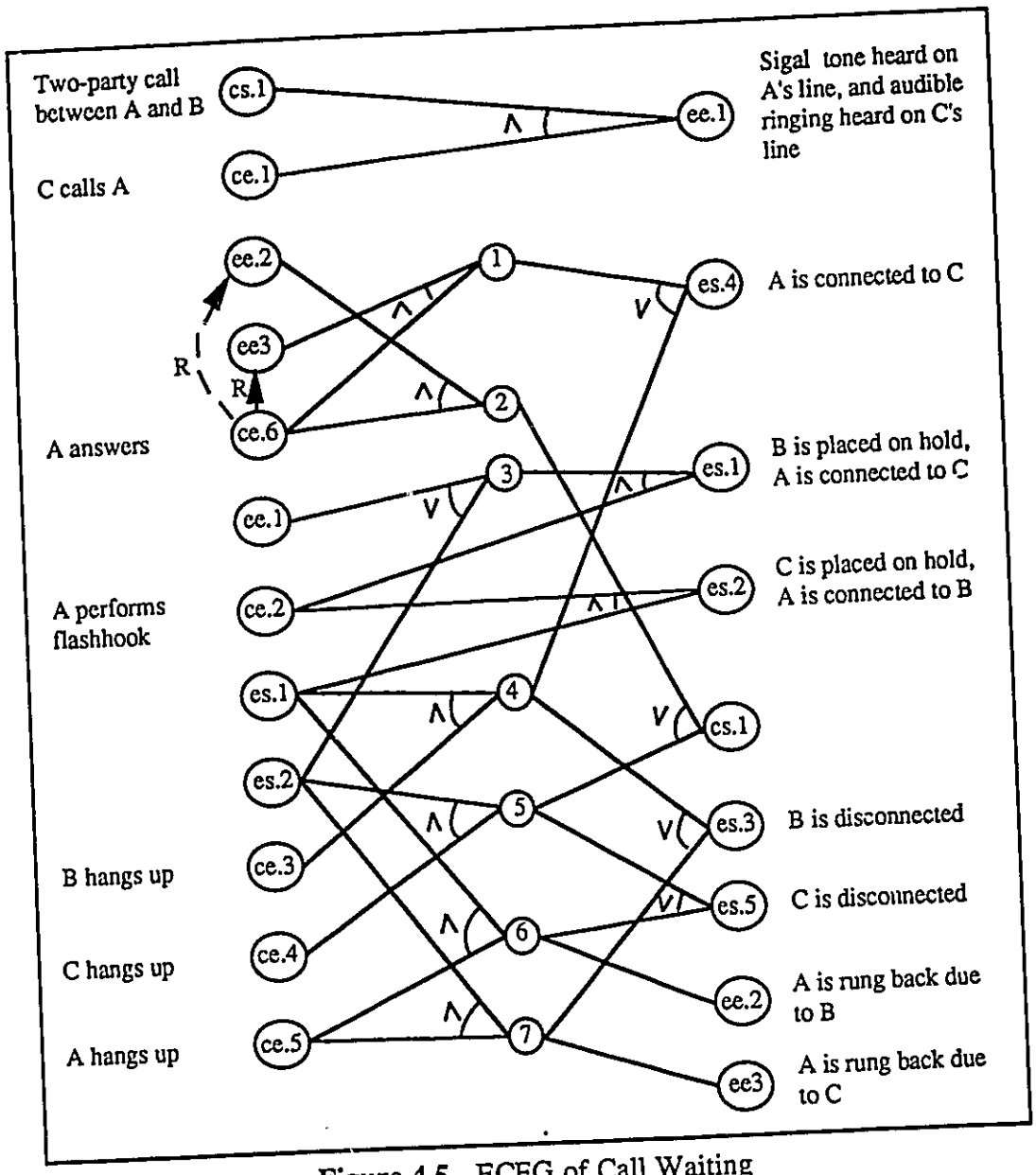


Figure 4.5 ECEG of Call Waiting

on hold. A is connected to C" (es.1). By switching back and forth between B and C another effect state will be observed "C goes on hold. A is connected to B" (es.2). Both effects es.1 and es.2 will be reused as causes because they will act alternatively as the present state before some events. The phrase "until one or the other call is terminated" gives rise to cause events "B hangs up" (ce.3) and "C hangs up" (ce.4) as well as effects states "B is disconnected" (es.3), "A is connected to C" (es.4) and "C is disconnected" (es.5). Note that in the case of effect es.4 "A is connected to B" should usually have been added but since a corresponding cause already exist therefore this phrase is omitted in es.4. Usually effects es.3 and es.4 should be placed together, but since effect es.4 will be observed later on without effect es.3

therefore they are separated. From the last sentence in the specification the following cause events "A hangs up" (ce.5) and "A answers phone" (ce.6), and also the effect events "A is rung back due to B" (ee.2), "A is rung back due to C" (ee.3) can be deduced. The other effects have already been derived.

Figure 4.5 shows the ECEG derived. Note how sequence of events have been shown on the graph. Effects ee.1, ee.2, ee.3, es.1 and es.2 have been reused. The requires constraint has been added during the phase where subscriber A answers the phone. In that phase it is required that A should be rung back first. The English statement of the causes and effects have been annotated on the ECEG in order to increase readability of the graph.

4.4 Feature Interaction Analysis by Combining ECEG's

It was mentioned before that a service contains a set of features. In our technique each feature is represented by an ECEG. Each feature is validated individually. In addition, it would be useful to see if all features can coexist and operate successfully when present in the same service. In the ECEG technique, this is done by combining together ECEG representation of each feature into a single ECEG in order to spot any feature interactions. However, only pairs of ECEGs will be combined to facilitate analysis and validation.

The idea behind the combination algorithm is to derive one ECEG, g, from two others , g_1 and g_2, (representing features f_1 and f_2 respectively) where the following properties will be held in g :

• Causes in g	=	Causes in g_1	$\cup$	Causes in g_2
• Effects in g	=	Effects in g_1	$\cup$	Effects in g_2
• Constraints in g	=	Constraints in g_1	$\cup$	Constraints in g_2
• Relations in g	=	Relations in g_1	$\cup$	Relations in g_2

From the above properties, g will therefore represent both g_1 and g_2 and hence features f_1 and f_2. As it will be observed in the next section, it may be possible that some causes in g_1 are also present in g_2 (i.e they have the same semantics) and the same may be possible with the effects, constraints, and relations. No duplication is made in g. Below an algorithm for combining two ECEGs into one only is presented.

Combination Algorithm

Given two ECEGs, $ECEG_1$ and $ECEG_2$, our objective is to combine these two ECEGs into a single one, namely, $ECEG_0$. The combination procedure is as follows :

- Assign either $ECEG_1$ or $ECEG_1$ to $ECEG_0$. It is preferable to assign the larger ECEG to $ECEG_0$ in order to decrease the amount of work. Larger means a larger number of nodes or a larger set of relations. Precede all the identifications of causes and effects in $ECEG_0$ with "1." if $ECEG_1$ has been assigned to $ECEG_0$ or with "2." if $ECEG_2$ has been assigned to $ECEG_0$. Following the same principle precede any constraint in $ECEG_0$ with either "1." or "2.".

(* For the purpose of simplicity, we will assume in what follows that $ECEG_1$ is the one that is assigned to $ECEG_0$.
*)

- For each cause, c_i, in $ECEG_2$ compare it with all causes and effects in $ECEG_0$ **do**
  If a cause, c_j, having the same meaning as c_i is present in $ECEG_0$ **then**
    Precede the identification of cause c_j with "2"
  else if a cause, c_j, a subset of cause c_i is present in $ECEG_0$ **then**
    Break down c_i into two separate causes c_j and c_k ($c_i = c_j \cup c_k$)
    Precede the identification of cause c_j with "21"
    Precede the identification of cause c_k with "1"
  else if a cause, c_j, containing cause c_i is present in $ECEG_0$ **then**
    Break down c_j into two separate effects c_i and c_k ($c_j = c_i \cup c_k$)
    Precede the identification of effect c_i with "21"
    Precede the identification of effect c_k with "2"
  else if an effect, e_j, having the same meaning as c_i is present in $ECEG_0$ **then**
    Precede the identification of effect e_j with "2"
    Create a cause node with the same identification as the new identification of e_j
  else if no cause or effect having the same meaning as c_i is present in $ECEG_0$ **then**
    Precede the identification of c_i with "2"
    Add c_i to $ECEG_0$
  endif
endfor

- For each effect, e_i, in $ECEG_2$ compare it with all causes and effects in $ECEG_0$ **do**
  If an effect, e_j, having the same meaning as e_i is present in $ECEG_0$ **then**
    Precede the identification of effect e_j with "2"
  else if an effect, e_j, a subset of effect e_i is present in $ECEG_0$ **then**
    Break down e_i into two separate effects e_j and e_k ($e_i = e_j \cup e_k$)
    Precede the identification of effect e_j with "21"
    Precede the identification of effect e_k with "1"
  else if an effect, e_j, containing effect e_i is present in $ECEG_0$ **then**
    Break down e_j into two separate effects e_i and e_k ($e_j = e_i \cup e_k$)
    Precede the identification of effect e_i with "21"
    Precede the identification of effect e_k with "2"
  else if a cause, c_j, having the same meaning as e_i is present in $ECEG_0$ **then**
    Precede the identification of cause c_j with "2"
    Create an effect node with the same identification as the new identification of c_j
  else if no cause or effect having the same meaning as e_i is present in $ECEG_0$ **then**
    Precede the identification of e_i with "2"
    Add e_i to $ECEG_0$
  endif
endfor

- For each constraint, t_i, in $ECEG_2$ compare it with all constraints in $ECEG_0$ **do**
  If a constraint, t_j, similar to t_i and between the same causes or effects as t_i is present in $ECEG_0$ **then**
    Precede the identification of constraint t_j with "2"
  else
    Precede the identification of constraint t_i with "2"
    Add t_i to ECEG0
  endif
endfor

- Relate each effect in $ECEG_0$ with identification number preceded with either "2" or "21" to its set of causes just as in $ECEG_2$. If the relation is already present in $ECEG_0$ then do not copy this relation from $ECEG_2$ to $ECEG_0$. Use additional intermediate nodes when appropriate to show different paths which can bring about the presence of an effect.

(* Note that in this process the list of intermediate nodes from $ECEG_2$ is added to $ECEG_0$ only if they are not already present in $ECEG_0$. The intermediate nodes too should be given unique identification numbers. Two intermediate nodes, n_i and n_j, will have the same identification number if they have the same list of nodes, L_n, pointing to them, the same relation relating the nodes in L_n (or/and) , and n_i and n_j will both be set to 0 or 1 for all combinations of the nodes in L_n.
*)

- Analyze the new list of causes and effects and add any new constraints if required.

- The new ECEG, $ECEG_0$, is thus obtained.

It can easily be noted that if $ECEG_0$ would be the identical whether $ECEG_1$ or $ECEG_2$ was assigned to $ECEG_0$ at the beginning of the procedure. Moreover, it can be observed that $ECEG_0$ will represent both behaviors from $ECEG_1$ and $ECEG_2$. This algorithm could be easily automated except that the user has to identify the causes and effects in $ECEG_1$ and $ECEG_2$ that have the same semantics at the beginning and then this information fed to the computer for automation. After combination, the user should still analyze the graph to add any new constraints. In the following an application of this algorithm is illustrated.

4.5 Examples

In this section we will illustrate how the combination algorithm works. We will first start by showing how we can represent the combined functionality of Three-Way Calling and Call Waiting features on a single handset. Then the functionality of two features found on remote handset will be represented. Three-Way Calling feature will be used as an illustration for this latter case.

4.5.1 Features on a single handset

Here we will show how the ECEG representation of Three-Way Calling and Call Waiting features can be combined into a single ECEG representing both features.

Since the ECEG of Call Waiting, say $ECEG_1$, seems to be more complex that the ECEG of Three-Way Calling, say $ECEG_2$, then we will first assign $ECEG_1$ to $ECEG_0$. At the same time each identification of the nodes in $ECEG_0$ is preceded with "1.". All the causes, intermediate nodes, effects, and constraints found in the Three-way calling Graph but

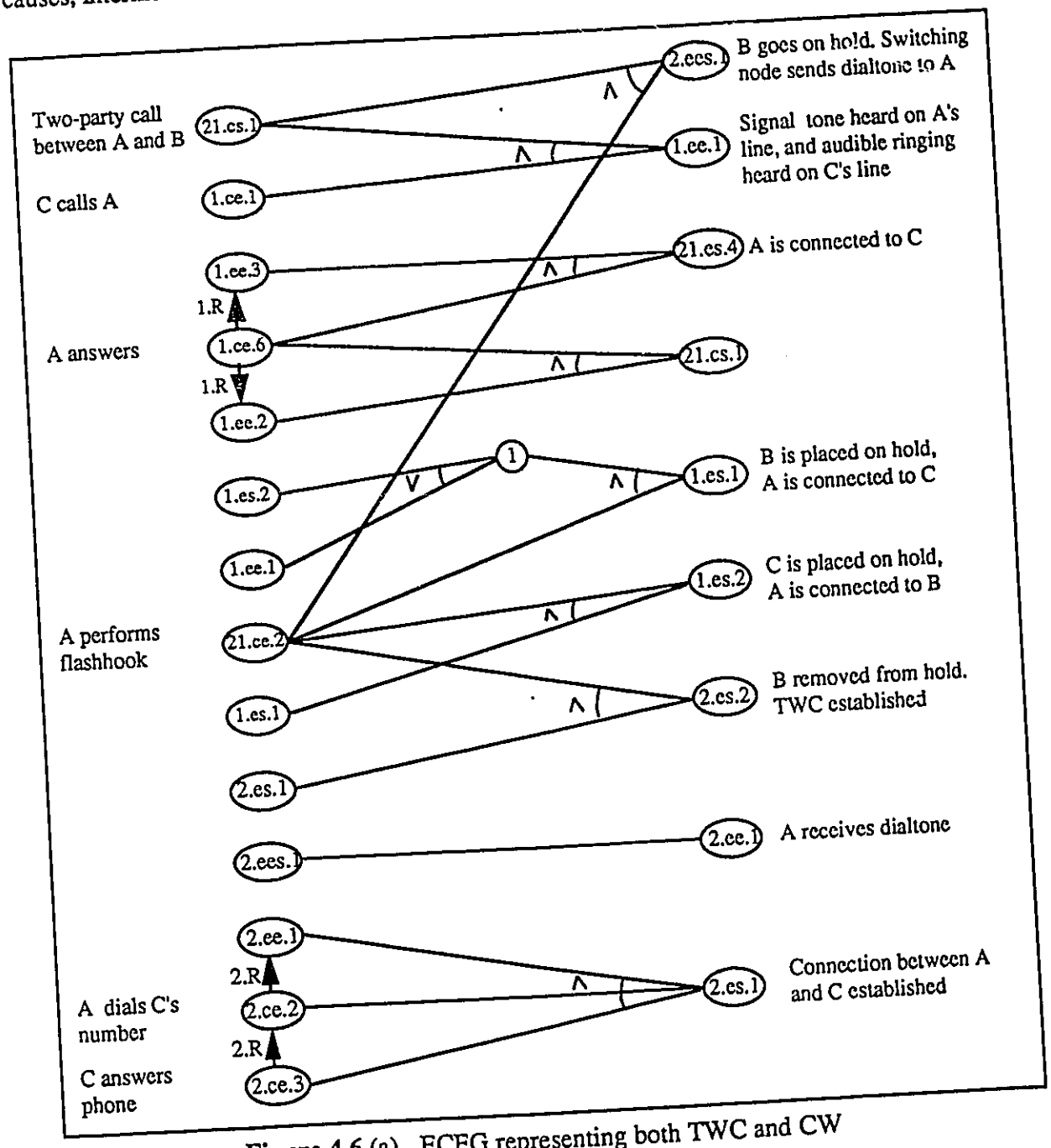


Figure 4.6 (a) ECEG representing both TWC and CW

not found on the Call waiting graph are added to $ECEG_0$ and at the same time appropriate nodes are renamed. The added causes are ce.2 (A dials C's number) and ce.3 (C answers phone), while causes cs.1 (Two-party call between A and B), ce.1 (A executes flashhook), ce.6 (A hangs up), ce.4 (B hangs up), and ce.5 (C hangs up) in $ECEG_2$ are already present in $ECEG_0$, namely through cs.1, ce.2, ce.5, ce.3, and ce.4 respectively.

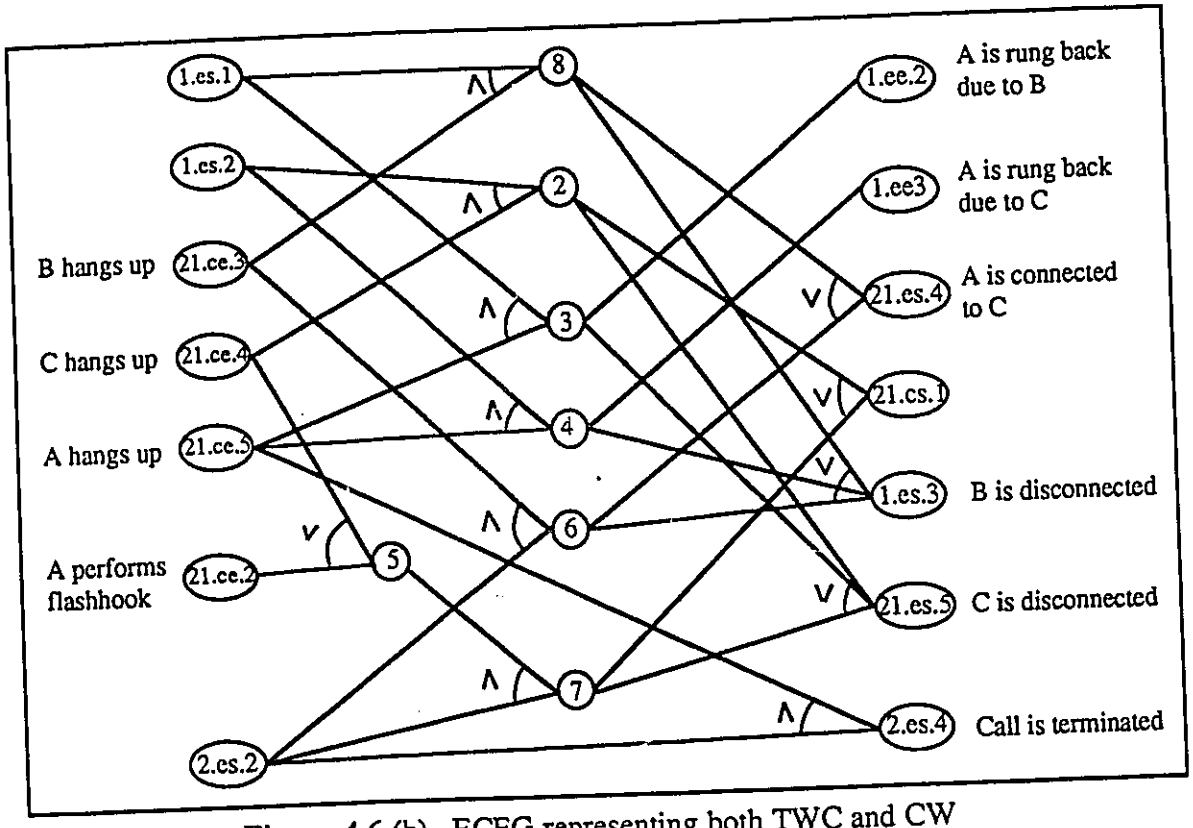


Figure 4.6 (b) ECEG representing both TWC and CW

The added effects are ees.1(B goes on hold. Switching node sends dialtone to A), es.1 (Connection between A and C established), ee.1 (A receives dialtone), es.2 (B removed from hold. TWC established), and es.4 (Call is terminated), while effect es.5 (C removed from call) in $ECEG_2$ is already present in $ECEG_0$ through es.5. Note the special case with effect ees.2 (B removed from call. Standard two-party call between A and C) in $ECEG_2$. This effect is made up of two separate effects, es.3 (B is disconnected) and es.4 (A is connected to C) already present in $ECEG_0$. Thus effect ees.2 is not added to $ECEG_0$

All the relations and constraints found in the Three-Way calling graph are added to the new combined $ECEG_0$. In this case in order to show all the relations in the TWC graph and the CW graph correctly some intermediate nodes (1-8) have been introduced. The

intermediate nodes do not correspond to the other intermediate nodes present in either $ECEG_1$ or $ECEG_2$. Figure 4.6 shows the $ECEG_0$ obtained. In this case the procedure has taken some time when done manually, but since automatization is possible then the process will be very easy and rapid.

4.5.2 Features on distributed access handsets

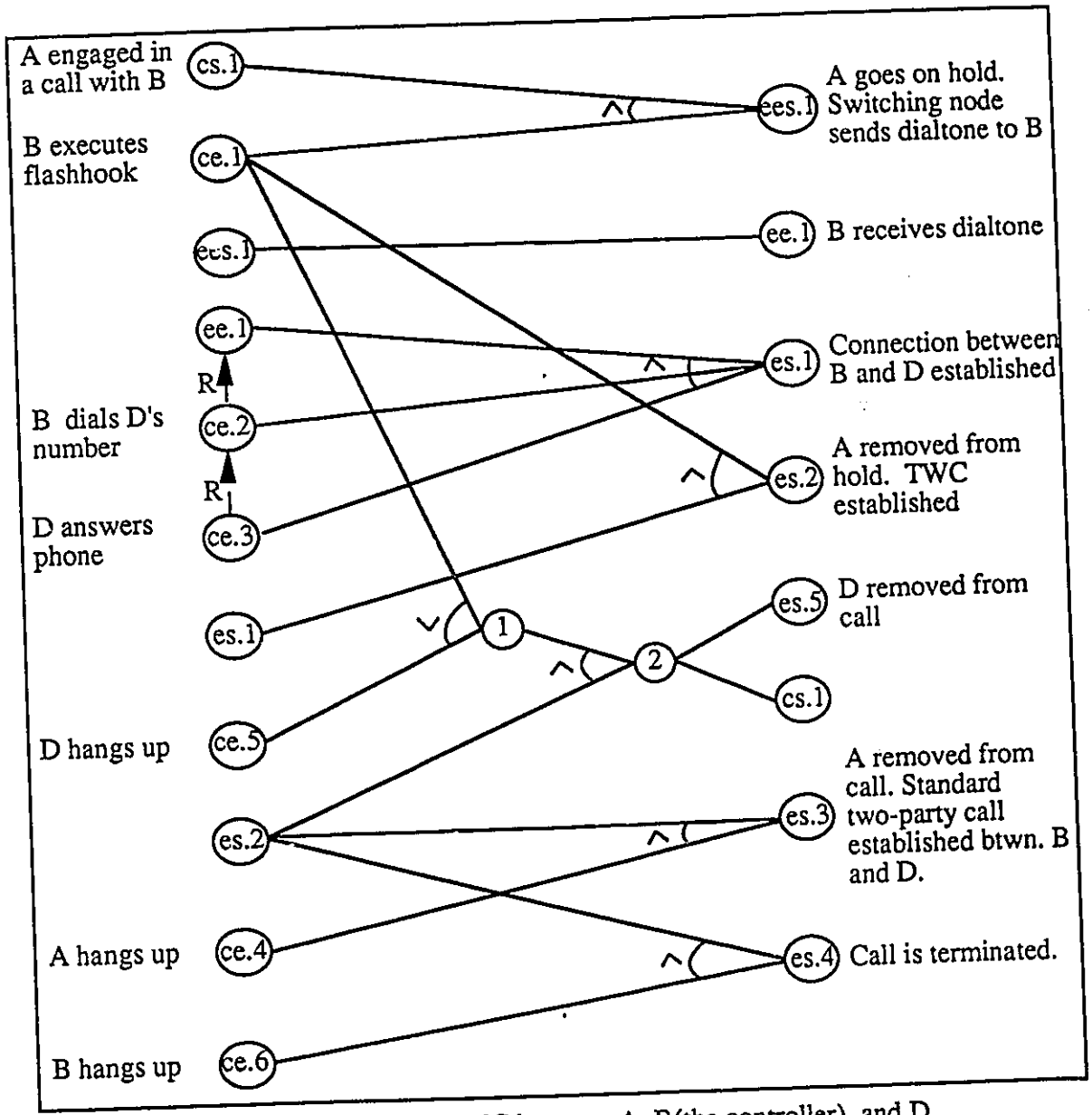


Figure 4.7 ECEG of TWC between A, B(the controller), and D

As it was shown previously, it is important to know how two features will affect the functionality of each other when both are present on the same handset. Moreover, it is

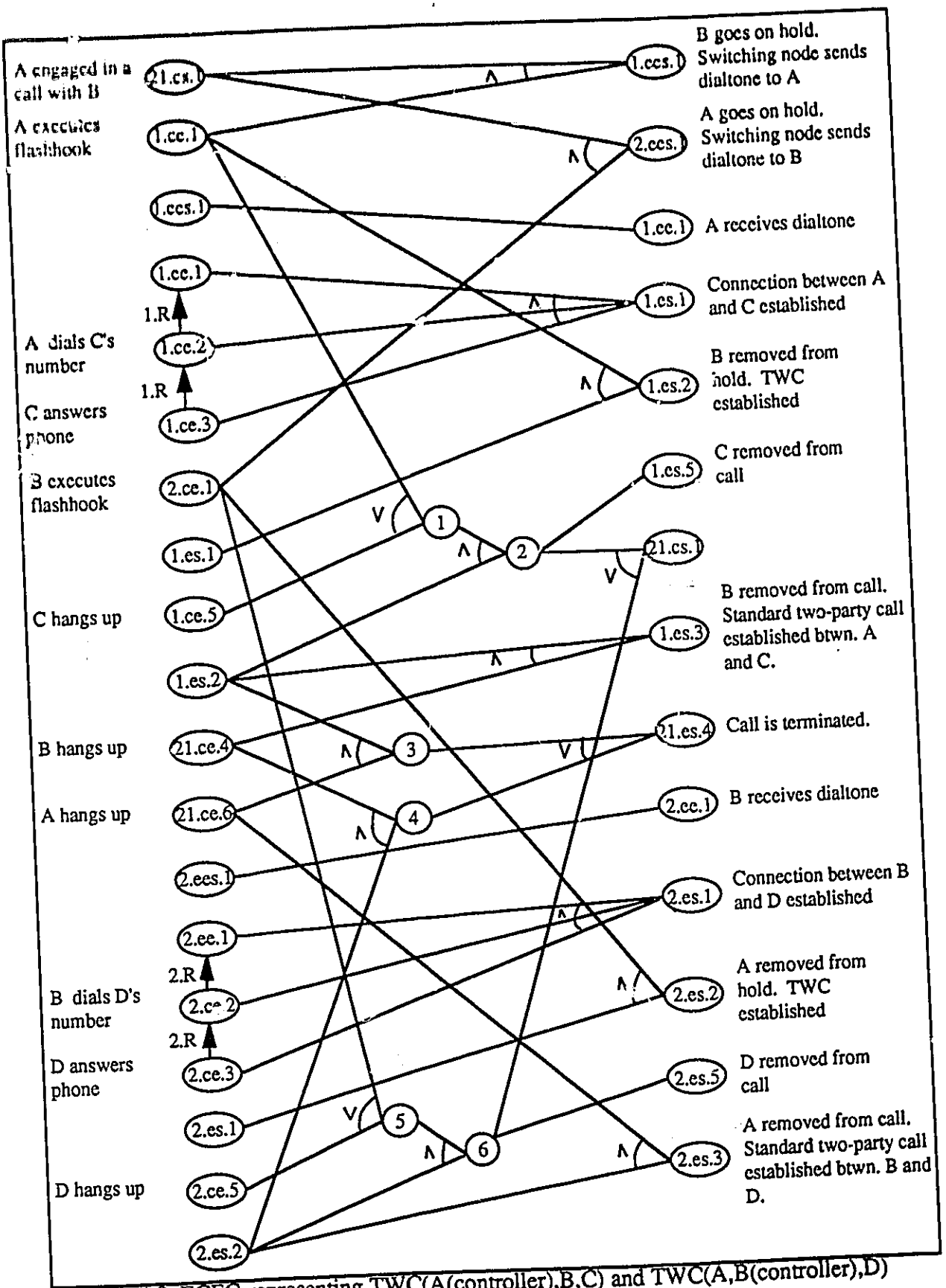


Figure 4.8 ECEG representing TWC(A(controller),B,C) and TWC(A,B(controller),D)

equally important to know how two features will interfere with each other if the features are found on remote handsets. This particular case is of great interest because this is usually where features interfere with each other very often and this case is among the hardest to detect.

The first step in detecting any feature interaction between features on remote handsets is to combine their respective ECEGs together using the same algorithm given in section 4.4. In this section we give an example of combining two features found on remote handsets. Both features are Three-Way Calling. We assume that at first there is a two-party call between A and B where both subscribers have the TWC feature. Our objective is to represent two TWC calls in progress, one between subscribers A(the controller), B, and C, and the other one between A, B(the controller), and D. Figure 4.4 is the ECEG representation of TWC of the first case while Figure 4.7 represents the second case.

Our objective is to combine the ECEG representing TWC between A, B, and C, say $ECEG_1$, and the one representing TWC between A, B, and D, say $ECEG_2$, to obtain $ECEG_0$ which will show how two instances of the same feature can work concurrently.

Since both ECEGs are of the same size and complexity, any one of them can be assigned to $ECEG_0$ at the start of the combination procedure. In this case $ECEG_1$ is the one assigned and the identification number of all the causes, effects, and constraints are preceded with "1". Then all the causes, effects, and constraints from $ECEG_2$ are added to $ECEG_0$ in such a way that $ECEG_0$ will have only one copy of cause, effect, and constraint. Among the causes and effects that have not been added to $ECEG_0$ from $ECEG_2$ are cs.1 (A engaged in a call with B), ce.4 (A hangs up), ce,6 (B hangs up), and es.4 (Call is terminated) as they are already present in $ECEG_0$. All the other causes, effects, and constraints are added to $ECEG_0$ and prefixed with "2". Figure 4.8 shows the resulting ECEG.

As the examples illustrate, the ECEG method is convenient to represent telephony features, and in a larger scale distributed systems. The building of an ECEG can be automated given its text representation . More details of this can be found in Appendix A.

Chapter 5

ECEG-Based Validation of Features and Services

Once a representation in Extended Cause-Effect Graph of each feature is done, we can proceed to the analysis of the graph for validation purposes. The analysis will start by first checking the graph for certain inconsistencies. Then, a decision table is built [Myer79] from the graph and test scenarios derived from the decision table. The process of building a decision table can be fully automated as well as the process of deriving test scenarios. Several methods exist for deriving a decision table from a Cause-Effect Graph, namely using Equivalent Normal Form [Wals83], Test design using Path Sensitization [Arab86], and Myers technique [Myer79]. The same techniques can be used in the case of ECEGs.

When an individual feature has been validated, it is usually integrated with other features to provide a service. During this activity two situations may arise : either all features work just as they do on an individual basis and none of them interfere with the functioning of any of the other features, or the behavior of some features are influenced (either positively or negatively) by other features, i.e feature interactions occur [Dwor89]. In section 5.2 a method is shown where composed features can be validated after the composition algorithm (section 4.4) has been applied to the features.

5.1 ECEG-Based Validation of an Individual Features

When an ECEG is built from an informal specification, some defects could be introduced in the graph due to mistakes made during that process or due to faults present in the specification itself. Some of the symptoms of those defects could be spotted through inconsistencies in the ECEG representation. Figure 5.1 illustrates how addition of constraints to an ECEG can lead to certain undesirable results.

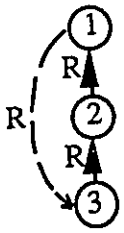
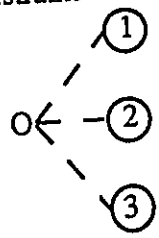
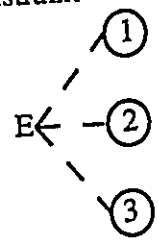
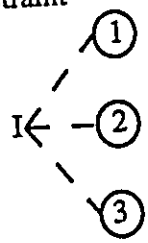
Constraint	Problematic Case	Observation
Requires (R)	Looping Requires Constraint 	Causes 1, 2, and 3 should either all be present or all should be absent. The user could either have added one or more Requires constraint in excess or causes 1, 2, and 3 should actually have been considered as a single cause.
	Multiple state 	The causes "ws.y" and "vs.x" represent cause states. In this case both states should either be present or absent simultaneously. Usually a system can only be in one state at a time. Therefore this case is probably an error. The constraint could be in excess.
One-and-one-only (O)	Wrong Constraint 	This case will be incorrect if an effect requires either the presence of more than one cause among causes 1, 2, and 3, or the absence of all of them. Replacing or removing the constraint could solve the problem. But, the problem can be with the effect too.
Exclusive (E)	Wrong Constraint 	This case will be incorrect if an effect requires the presence of more than one cause among causes 1, 2, and 3. Replacing or removing the constraint could solve the problem. But, the problem can be with the effect too.
Inclusive (I)	Wrong Constraint 	This case will be incorrect if an effect requires the absence of all causes 1, 2, and 3. Replacing or removing the constraint could solve the problem. But, the problem can be with the effect too.

Figure 5.1 (a) Problematic constraints in an ECEG

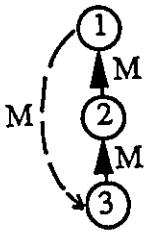
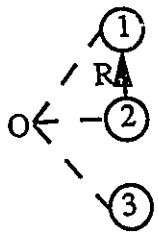
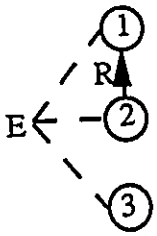
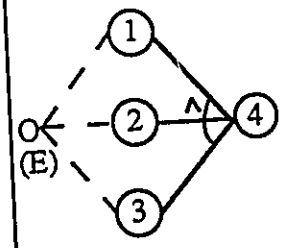
Constraint	Problematic Case	Observation
Mask (M)	Looping Mask Constraint 	Whenever effects 1, 2, and 3 are present simultaneously it cannot be determined which effect will mask the presence of the other one. The user could either have added one or more mask constraint in excess or effects 1, 2, and 3 should actually have been considered as a single effect.
	Impossible case 	This mask constraint would be incorrect if no cause combinations bring about the presence of both effects 1 and 2. In that case, either the mask constraint should be removed or the causes have been wrongly related to the effects.
Requires + One-and-one-only	Impossible case 	In this case it is impossible to abtain a cause combination that will satisfy both constraints. The constraints should either be replaced or removed partially or completely.
Requires + Exclusive	Impossible case 	The only cause combination that can satisfy all the constraints is to set all causes (1, 2, and 3) to 0. The constraints should be revisited.
One-and-one-only (or Exclusive) + "and" relation	Stuck_at_0 	It can easily be observed that node 4 will always be absent whether the Exclusive or One-and-One-Only constraint is annoted to the ECEG. A mistake could either have been caused with the constraint or the relation.

Figure 5.1 (b) Problematic constraints in an ECEG

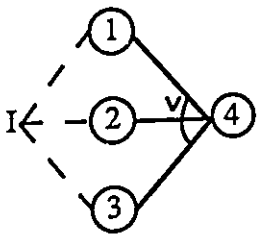
Constraint	Problematic Case	Observation
Inclusive + "or" relation	Stuck_at_1 	It can easily be observed that node 4 will always be present. A mistake could either have been caused with the constraint or the relation.

Figure 5.1 (c) Problematic constraints in an ECEG

Other kind of inconsistencies that can be introduced into an ECEG is the way in which the causes are related to the effects. Figure 2.18 (Chapter 2) shows a case where for an effect to be present it requires that a given cause should be set to 1 and 0 simultaneously, an impossibility. Moreover, if for an effect to be present (or absent) it requires presence of more than one cause state (of the same subscriber) then a mistake is most likely to be present in the ECEG. Similarly, if for an effect to be present (or absent) it requires presence of more than one cause event and no Requires constraint is used to show the ordering of those events then this case should be studied in greater details.

By methodically tracing state conditions in the graph representing a given feature, the graph is converted into a limited-entry decision table using any one of the Path Sensitization technique. Note that both presence and absence of effects are studied. During this exercise it can be determined whether a cause or an effect is stuck at 0 or stuck at 1 (Chapter 2).

All the above analysis of an ECEG validates the syntactic contents of the graph. This means that irrespective of the graph if any of the above symptoms arise, it can automatically be said that the ECEG or the informal specification contains some defects. It should be noted that the ECEG syntax validation can be fully automated. Now, after the syntax of the ECEG has been analyzed, we should look at its semantics, i.e we should determine if the ECEG is a correct translation of the informal specification and at the same time if the specification has been well understood by the user.

One way of validating the specification and the semantics of the ECEG is through ***user-directed*** scenarios. The user-directed scenarios are obtained from the decision table. A user-directed scenario consists of the causes that should be present and those that should be absent as

well as the effect produced. Each column in the decision table will correspond to at least one user-directed scenario (one scenario for each effect observed). The scenarios are given to the client who should approve or disapprove of each one. Consider the case where the user disapprove a given scenario. At that point we can note five potential sources for this problem: the scenario may have been incorrectly derived from the decision table, the decision table may have incorrectly been obtained from the ECEG, the ECEG may be incorrect with respect to the specification, the specification may be incorrect, or a combination of any one of the above. Since deriving a decision table and then the test scenarios are algorithmic steps we will ignore these sources of errors, assuming that the corresponding algorithms have thoroughly been tested and are correct. To know the source of the error, the ECEG should first be checked with respect to the specification to detect any omission or wrong interpretation. If no error is found after doing this, then we can say that the specification is incorrect or it has wrongly been interpreted by the tester. At that point any change in the specification should be made in collaboration with the customer.

As an important side-benefit, test scenarios give a concise, clear description of the functioning of a feature. This way of writing helps the customer and the specifier to more easily detect any possible omissions or mistakes in the specification. During analysis several questions can be asked: What if a cause is not present when it should have been present? What about the delay factor? As in the above specification, we can ask what would happen if cause ce.3 is absent in scenario # 3, i.e what would happen if C does not answer the phone or c's line is busy?. These types of questions are useful for clarifying the given specification, especially with respect to exceptional cases which are frequently overlooked in requirements documentations. These appropriate questions can only be asked by an experienced user in both the telephony systems and ECEG techniques.

5.2 Example : Three-Way Calling

In this section we will show how the ECEG analysis is performed on the Three-Way Calling feature. The syntax of the graph is first validated and no mistake was found. From its ECEG (Figure 4.4) the corresponding decision table is built (given in Table 5.1), based on the Basic Path Sensitization technique given in chapter 2. The derivation has been straight-forward. In each column a bold value of a cause means that this cause is found in the cause set of the effect whose presence or absence is studied for that column, while a bold value of an effect means the effect targeted for that column. Note the difference that we make between events that are both causes and effects: if an event (both cause and effect) is present as a cause this does not mean that this event should be present as an effect too since at any given time this event will either be a cause or an

effect but not both. The same reasoning applies when an event is present as an effect while its corresponding cause could be absent. For instance, for effect ees.2 to be present, both causes ce.4 and es.2 should be present. Even if es.2 acts both as a cause and an effect, only its cause part will be set to 1 while its effect part will be set to 0. This case is represented by column number 7 in the decision table.

From the decision table in Table 5.1 the corresponding test scenarios are derived, at least one test scenario per column. The test scenarios derived are shown in Table 5.2. The derivation has been straight-forward. From the test scenarios the following observations can be made: i) when A executes flashhook and B hangs up at the same time, what could happen ?; ii) what would be the result if C does not answer the phone or his/her line is busy?; iii) what if instead A dials C's number he/she dials B's number?. All these observations shows that the informal specification of TWC gives the minimum information about its functionality. These observations can easily be made from a graphical representation of the whole functionality of the TWC. In that context precisely, the ECEG technique derives its greatest power.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Causes	ce.1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0
	ce.2	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
	ce.3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	ce.4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	ce.5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ce.6	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Causes and effects	cs.1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
	ee.1	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0
	es.1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
	es.2	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1
	ees.1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Effects and Causes	cs.1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	ee.1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	es.1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	es.2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	ees.1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Effects	ees.2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	es.4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	es.5	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.1 Decision Table for TWC

#	Effects	P/A	Test Scenarios
1	A engaged in a call with B (cs.1) — C removed from call (es.5)	P	Starting state (cs.1)
		—	Scenario # 5 **and** C hangs up (ce.5)
		P	Scenario # 5 **and** A executes flashhook (ce.1)
2	B goes on hold, Switching nod sends dialtone tc A(ees.1)	P	Scenario # 1 **and** A executes flashhook (ce.1)
3	A receives dialtone (ee.1)	P	Scenario # 2
4	Connection between A and C established (es.1)	P	Scenario # 3 **and** A dials C' number (ce.2) **and** C answers phone (ce.3)
5	B removed from hold, TWC established (es.2)	P	Scenario # 4 **and** A executes flashhook (ce.1)
6	B removed from call. Std two-party call btw. A and C (ees.2)	P	Scenario # 5 **and** Bhangs up (ce.4)
7	Call is terminated (es.4)	P	Scenario # 5 **and** Ahangs up (ce.6)
8	A engaged in a call with B (cs.1) — C removed from call (es.5)	A	Scenario # 5 **and not** A executes flashhook (ce.1) **and not** C hangs up (ce.5)
		—	
		A	
9	B goes on hold, Switching nod sends dialtone tc A(ees.1)	A	Scenario # 1 **and not** A executes flashhook (ce.1)
10	Connection between A and C established (es.1)	A	Scenario # 3 **and not** A dials C' number (ce.2) **and not** C answers phone (ce.3)
			Scenario # 3 **and** A dials C' number (ce.2) **and not** C answers phone (ce.3)
11	B removed from hold, TWC established (es.2)	A	Scenario # 4 **and not** A executes flashhook (ce.1)
12	B removed from call. Std two-party call btw. A and C (ees.2)	A	Scenario # 5 **and not** Bhangs up (ce.4)
13	Call is terminated (es.4)	A	Scenario # 5 **and not** Ahangs up (ce.6)

Table 5.2 Test scenarios for TWC

5.3 ECEG-Based Validation of Combined Features

Once two ECEGs, ECEG1 and ECEG2, have been composed into ECEG0, we can proceed at the same time as feature interaction analysis (discussed in the previous chapter) to validate the behaviors of the two features, one represented by ECEG1 and the other by ECEG2, when both are present in the same service.

The first validation step consists of checking ECEG0 to see if none of the syntactical errors mentioned in section 5.1 are present in it. In this phase, major constraint errors, impossible cause combinations, nodes stuck at 0, and nodes stuck at 1 can be detected. Moreover, one should see whether ECEG0 = ECEG1 » ECEG2. All and only causes, effects, and constraints present in either ECEG1 or ECEG2 should be present in ECEG0 (except in the case where additional constraints had to be added to ECEG0 after its construction). It should further be checked to see if a constraint between a group of causes, say gc, or a group of effects, say ge, found in one ECEG (ECEG1 or ECEG2) is also present in the other ECEG if gc or ge is present in that graph. Furthermcre, no cause or effect should be present more than once in ECEG0 (except in the case where the graph is drawn separately due to its size. In this case each part of the graph should have unique causes and effects). In addition, it should be checked to see if every cause, effect, and constraint in ECEG0 having their identification number prefixed with "21" are present in both ECEG1 and ECEG2. Similarly, causes, effects, and constraints with their identification number prefixed with "1" ("2") should be present in ECEG1 (ECE2).

Just as before we should now proceed to validate the semantics of ECEG0 to determine if the two features this graph represent can operate successfully when present in the same service, i.e to detect any feature interactions. When two ECEGs are composed together several faults may be detected. Figure 5.2 illustrates some of them.

When the decision table is derived from ECEG0, it should be checked to see if any cause or effect is stuck-at-1 or stuck-at-0. This can easily be spotted in the decision table by looking if all causes and effects take both the values 1 and 0 in different columns.

Just as before user-directed scenarios are obtained from the decision table. The scenarios will help the customer and the specifier to more easily detect any possible omissions or mistakes when the ECEGs of two features are composed. During analysis several questions can be asked: What if a cause or an effect is not present when it should have been present? What about the delay factor? Can two events occur simultaneously? What will happen when two events

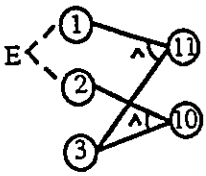
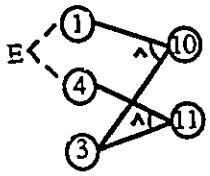
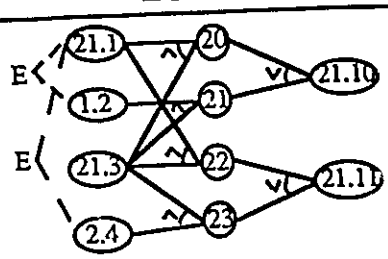
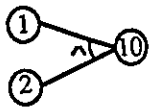
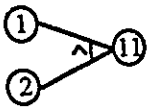
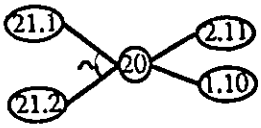
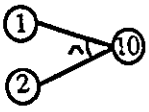
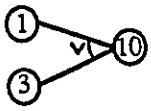
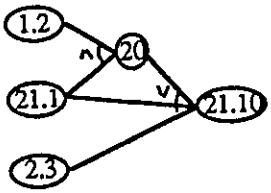
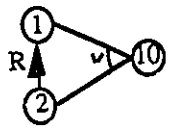
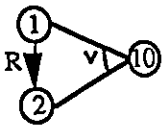
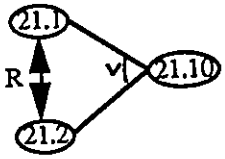
	ECEG1	ECEG2.	ECEG0
1			
	Remarks *In both ECEG1 and ECEG2 effects 11 and 10 could not occur concurrently and this is what the customer required. However, in ECEG0 both effects could occur at the same time (when causes 1.2, 21.3, and 2.4 are set to 1. Thus the resulting graph shows that when two features are present in the same service,they could interfere with the functioning of each other.*		
			
	Remarks *This case is similar to the previous one where effects 11 and 10 could not occur concurrently except that in this case they are found in seperate features. Thus when the features are brought together they interfere with each other.*		
2			
	Remarks *In ECEG1 effect 10 will be present if both causes 1 and 2 are present at the same time while in ECEG2 effect 10 will be present if any one of causes 1 and 3 is present. ECEG0 shows that effect 10 can occur irrespective of cause 2 meaning that this graph does not show this specific behavior of ECEG1.*		
3			
	Remarks *In ECEG1 effect 10 will be present if cause 1 or both causes 1and 2 are present while in ECEG2 effect 10 will be present if cause 2 or both causes 1and 2 are present. In ECEG0 effect 10 will be present only if both causes 1 and 2 are present at the time. This case impose a constraint on the behaviors of both features.*		

Figure 5.2 Examples of defects in composed ECEGs

occurs concurrently? These questions will help the user in detecting an interactions between the two features when present in the same service.

5.4 Examples

In this section we show two examples of validation of composed features. In the first example, the composed behaviors of Three-Way Calling and Call Waiting features are validated. These two features are found on the same handset. The second example consists of validating two instances of Three-Way Calling feature, found on remote handsets.

5.4.1 Example : Three-Way Calling and Call Waiting

The ECEG representing the composed behaviors of TWC and CW features is given in Figure 4.6 of chapter 4. The syntax of the graph is first validated and no mistake was found. From its ECEG the corresponding decision table is built (given in Table 5.3), based on the Basic Path Sensitization technique given in chapter 2. The derivation has been straight-forward. The large number of columns in the decision table is due to the fact that both presence and absence of effects are studied. In each column a bold value of a cause means that this cause is found in the cause set of the effect whose presence or absence is studied for that column, while a bold value of an effect means the effect targeted for that column. The first observation that can be made is that none of the causes or effects are stuck at 0 or 1.

From the decision table in Table 5.3 the corresponding test scenarios are derived, at least one test scenario per column. The test scenarios derived are shown in Table 5.4. The derivation has been straight-forward. From the test scenarios the following observations can be made: i) when B is placed on hold, and A is connected to C, how will a flashhook made by subscriber A be interpreted by the telecommunications systems? In that particular case, under the TWC feature B will be removed from hold and a Three-Way call will be established between A, B, and C, while under the CW feature C will go on hold and A connected to B.; ii) what would be the result when A hangs up? Under the TWC policy the call will terminate while under CW A will be rung back.; iii) it can be noted that the general idea behind these features is that a TWC will be established between subscribers A, B, and C if A calls C while a CW feature will take over if C calls A. Does this mean that a TWC call cannot be established if C calls A and a CW cannot be established if A calls C? Is the caller important to established who is going to control the call? ; (iv) A can disconnect C from a TWC call by simple doing a flashhook, while under CW this action will put either B or C on hold depending which subscriber was on hold before ; (v) how will the system

reacts if when C calls A, the latter do not flashhook?. All these observations clarify the behaviors of these two features on the same handsets and at the same time introduce further questions about the interactions between Three-Way Calling and Call Waiting features.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Causes	1.ce.1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ce.6	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
	2.ce.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	2.ce.3	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
	21.ce.2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.ce.3	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1
	21.ce.4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1
	21.ce.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Causes and effects	1.es.1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0
	1.es.2	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0
	1.ee.1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	1.ee.2	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	1.ee.3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	2.ee.1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	2.es.2	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
	21.cs.1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
	2.ees.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects and Causes	1.es.1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
	1.es.2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	1.ee.1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ee.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	1.ee.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
	2.ee.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
	2.es.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.cs.1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
	2.ees.1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects	1.es.3	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0
	2.es.4	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1
	21.es.4	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.es.5	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0

Table 5.3(a) Decision Table of TWC and CW

		19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Causes	1 ce 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 ce 6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	2 ce 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 ce 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 ce 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 ce 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 ce 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 ce 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Causes and effects	1 cs.1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0
	1 cs.2	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
	1 cc.1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	1 cc.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 cc.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 cc.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 cs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 cs.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 cs.1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 ccs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects and Causes	1 cs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 cs.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 ce.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 ce.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1 ce.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 ce.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 cs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 cs.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 cs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 ccs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects	1 cs.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2 cs.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 cs.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21 cs.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.3(b) Decision Table of TWC and CW

#	Effects	P/A	Test Scenarios
1	B goes on hold. Switching nod sends dialtone tc A(2.ees.1)	P	Scenario # 4 **and** C performs flashhook up (21.ce.2)
2	Signal tone heard on A's line, and audible ringing on C's line (1.ee.1)	P	Scenario # 4 **and** C calls A (1.ce.1)
3	A is connected to C (21.es.4)	P	Scenario # 11 **and** A answers (1.ce.6) **and not** B hangs up (21.ce.3) **and not** Scenario # 5 **and not** Scenario # 7 (This applies only for effect 21.es.4)
	— — — — — —	—	Scenario # 5 **and** B hangs up (21.ce.3) **and not** A answers (1.ce.6) **and not** scenario # 11 **and not** Scenario # 7
	B is disconnected (1.es.3)	P	Scenario # 7 **and** B hangs up (21.ce.3) **and not** A answers (1.ce.6) **and not** scenario # 11 **and not** Scenario # 5
4	Two-party call between A and B (21.cs.1)	P	Starting state (21.cs.1)
			Scenario # 10 **and** A answers (1.ce.6) **and not** A performs flashhook (21.ce.2) **and not** C hangs up (21.ce.4) **and not** Scenario # 6 **and not** Scenario # 7 (This applies only for effect 21.es.5)
	— — — — — —	—	Scenario # 6 **and** C hangs up (21.ce.4) **and not** A answers (1.ce.6) **and not** A performs flashhook (21.ce.2) **and not** Scenario # 10 **and not** Scenario # 7
	C is disconnected (21.es.5)	P	Scenario # 7 **and** C hangs up (21.ce.4) **and not** A answers (1.ce.6) **and not** A performs flashhook (21.ce.2) **and not** Scenario # 10 **and not** Scenario # 6
			Scenario # 7 **and** A performs flashhook (21.ce.2) **and not** A answers (1.ce.6) **and not** C hangs up (21.ce.4) **and not** Scenario # 10 **and not** Scenario # 6
5	B is placed on hold, A is connected to C (1.es.1)	P	Scenario # 6 **and** A performs flashhook (21.ce.2) **and not** Scenario # 2
			Scenario # 2 **and** A performs flashhook (21.ce.2) **and not** Scenario # 6
6	C is placed on hold, A is connected to B (1.es.2)	P	Scenario # 5 **and** A performs flashhook (21.ce.2)
7	B removed from hold, TWC established (2.es.2)	P	Scenario # 9 **and** A performs flashhook (21.ce.2)
8	A receives dialtone (2.ee.1)	P	Scenario # 1
9	Connection between A and C established (2.es.1)	P	Scenario # 8 **and** A dials C's number (2.ce.2) **and** C answers phone (2.ce.3)

Table 5.4(a) Test scenarios of TWC and CW

#	Effects	P/A	Test Scenarios
10	A is rung back due to B (1.ee.2) C is disconnected (21.ce.5)	P P	Scenario # 5 **and** A hangs up (21.ce.5)
11	A is rung back due to C (1.ee.3) B is disconnected (1.es.3)	P P	Scenario # 6 **and** A hangs up (21.ce.5)
12	Call is terminated (2.es.4)	P	Scenario # 7 **and** A hangs up (21.ce.5)
13	B goes on hold. Switching nod sends dialtone tc A(2.ees.1)	A	Scenario # 4 **and not** C performs flashhook up (21.ce.2)
14	Signal tone heard on A's line, and audible ringing on C's line (1.ee.1)	A	Scenario # 4 **and not** C calls A (1.ce.1)
15	A is connected to C (21.es.4)	A	Scenario # 11 **and not** A answers (1.ce.6) **and not** B hangs up (21.ce.3) **and not** Scenario # 5 **and no** Scenario # 7 (This applies only for effect 21.es.4)
			Scenario # 5 **and not** B hangs up (21.ce.3) **and not** A answers (1.ce.6) **and not** scenario # 11 **and no** Scenario # 7
	B is disconnected (1.es.3)	A	Scenario # 7 **and not** B hangs up (21.ce.3) **and not** A answers (1.ce.6) **and not** scenario # 11 **and no** Scenario # 5
16	Two-party call between A and B (21.cs.1)	A	Scenario # 10 **and not** A answers (1.ce.6) **and not** A performs flashhook (21.ce.2) **and not** C hangs up (21.ce.4) **and not** Scenario # 6 **and not** Scenario # 7 (This applies only for effect 21.es.5)
			Scenario # 6 **and not** C hangs up (21.ce.4) **and not** A answers (1.ce.6) **and not** A performs flashhook (21.ce.2) **and not** Scenario # 10 **and not** Scenario # 7
	C is disconnected (21.es.5)	A	Scenario # 7 **and not** C hangs up (21.ce.4) **and not** A answers (1.ce.6) **and not** A performs flashhook (21.ce.2) **and not** Scenario # 10 **and not** Scenario # 6
			Scenario # 7 **and not** A performs flashhook (21.ce.2) **and not** A answers (1.ce.6) **and not** C hangs up (21.ce.4) **and not** Scenario # 10 **and not** Scenario # 6
17	B is placed on hold, A is connected to C (1.es.1)	A	Scenario # 6 **and not** A performs flashhook (21.ce.2) **and not** Scenario # 2
			Scenario # 2 **and not** A performs flashhook (21.ce.2) **and not** Scenario # 6

Table 5.4(b) Test scenarios of TWC and CW

#	Effects	P/A	Test Scenarios
18	C is placed on hold, A is connected to B (1.es.2)	A	Scenario # 5 **and not** A performs flashhook (21.ce.2)
19	B removed from hold, TWC established (2.es.2)	A	Scenario # 9 **and not** A performs flashhook (21.ce.2)
20	Connection between A and C established (2.es.1)	A	Scenario # 8 **and not** A dials C's number (2.ce.2) **and not** C answers phone (2.ce.3)
			Scenario # 8 **and** A dials C's number (2.ce.2) **and not** C answers phone (2.ce.3)
21	A is rung back due to B (1.ee.2) _ _ C is disconnected (21.ce.5)	A _ A	Scenario # 5 **and not** A hangs up (21.ce.5)
22	A is rung back due to C (1.ee.3) _ _ B is disconnected (1.es.3)	A _ A	Scenario # 6 **and not** A hangs up (21.ce.5)
23	Call is terminated (2.es.4)	A	Scenario # 7 **and not** A hangs up (21.ce.5)

Table 5.4(c) Test scenarios of TWC and CW

5.4.2 Example : Three-Way Calling and Three-Way Calling (Remote Handset)

The ECEG representing the composed behaviors of two instances of TWC feature is given in Figure 4.8 of chapter 4. The syntax of the graph is first validated and no mistake was found. From its ECEG the corresponding decision table is built (given in Table 5.5), based on the Basic Path Sensitization technique given in chapter 2. The derivation has been straight-forward. The large number of columns in the decision table is due to the fact that both presence and absence of effects are studied. In each column a bold value of a cause means that this cause is found in the cause set of the effect whose presence or absence is studied for that column, while a bold value of an effect means the effect targeted for that column. The first observation that can be made is that none of the causes or effects are stuck at 0 or 1.

From the decision table in Table 5.5 the corresponding test scenarios are derived, at least one test scenario per column. The test scenarios derived are shown in Table 5.6. The derivation has been straight-forward. From the test scenarios the following observations can be made: i) suppose that A and B flashhook at approximately the same instance. How will the telecommunications

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Causes	1.ce.1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	1.ce.2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	1.ce.3	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
	1.ce.5	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
	2.ce.1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	2.ce.2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	2.ce.3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	2.ce.5	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
	21.ce.4	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
	21.ce.6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Causes and effects	1.ee.1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	1.es.1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	1.es.2	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0
	2.ee.1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
	2.es.1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
	2.es.2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1
	2.es.1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	21.cs.1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ees.1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ees.1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Effects and Causes	1.ee.1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.es.1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	1.es.2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	2.ee.1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	2.es.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.cs.1	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0
	1.ees.1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ees.1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects	1.es.5	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0
	2.es.5	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	1.ees.2	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
	2.ees.2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
	21.es.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.5(a) Decision Table representing two instances of TWC

systems react? Is there a priority set for each subscriber? ; ii) suppose both subscribers engage in a TWC : A has placed B on hold and B has placed A on hold. When either A or B flashhooks what would happen? ; iii) suppose that a TWC call is established between A, B, and C with A the controller. Suppose now that B wants to bring subscriber D into the call. When B flashhooks will A go on hold? ; (iv) following the same principal as in (iii) what would happen if subscriber D is

busy and then B performs another flashhook? Will subscribers A, B, and C hear the busy tone? ; (v) how will the system reacts if both subscribers A and B try to establish a TWC call with subscriber C at the same time?. Note that this last case could happen very often, it takes only a misunderstanding between subscribers A and B. All these observations clarify the behaviors of the TWC feature on remote handsets and at the same time introduce further questions about the interactions between two instances of Three-Way Calling feature.

		17	18	19	20	21	22	23	24	25	26	27	28	29	30
Causes	1.ce.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ce.2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	1.ce.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ce.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ce.1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	2.ce.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ce.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ce.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.ce.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.ce.6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Causes and effects	1.ee.1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
	1.es.1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	1.es.2	0	0	0	0	0	1	0	1	1	0	0	0	0	0
	2.ee.1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	2.es.2	0	0	0	0	0	0	0	1	0	0	1	0	0	1
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.cs.1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	1.ees.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ees.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects and Causes	1.ee.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.es.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ee.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.es.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.es.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.cs.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ees.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ees.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Effects	1.es.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.es.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.ees.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2.ees.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	21.es.4	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.5(b) Decision Table representing two instances of TWC

#	Effects	P/A	Test Scenarios
1	B goes on hold, Switching node sends dialtone to A (1.ees.1)	P	Scenario # 7 **and** A executes flashhook (1.ce.1)
2	A goes on hold, Switching node sends dialtone to B (2.ees.1)	P	Scenario # 7 **and** B executes flashhook (2.ce.1)
3	A receives dialtone (1.ee.1)	P	Scenario # 1
4	B receives dialtone (2.ee.1)	P	Scenario # 2
5	Connection between A and C established (1.es.1)	P	Scenario # 3 **and** A dials C's number (1.ce.2) **and** C answers phone (1.ce.3)
6	B removed from hold. TWC established btw A, B, C (1.es.2)	P	Scenario # 5 **and** A executes flashhook (1.ce.1)
7	A engaged in a call with B (21.cs.1)	P	Starting state (21.cs.1)
			Scenario # 6 **and** A executes flashhook (1.ce.1) **and not** C hangs up (1.ce.5) **and not** B executes flashhook (2.ce.1) **and not** D hangs up (2.ce.5) **and not** Scenario # 11 (Does not apply to 2.es.5)
	C removed from call (1.es.5)	P	Scenario # 6 **and** C hangs up (1.ce.5) **and not** B executes flashhook (2.ce.1) **and not** A executes flashhook (1.ce.1) **and not** D hangs up (2.ce.5) **and not** Scenario # 11 (Does not apply to 2.es.5)
			Scenario # 11 **and** B executes flashhook (2.ce.1) **and not** C hangs up (1.ce.5) **and not** A executes flashhook (1.ce.1) **and not** D hangs up (2.ce.5) **and not** Scenario # 6 (Does not apply to 1.es.5)
	D removed from call (2.es.5)	P	Scenario # 11 **and** D hangs up (2.ce.5) **and not** C hangs up (1.ce.5) **and not** A executes flashhook (1.ce.1) **and not** B executes flashhook (2.ce.1) **and not** Scenario # 6 (Does not apply to 1.es.5)
8	B removed from call. Std two-party call established btw A and C (1.ees.2)	P	Scenario # 6 **and** B hangs up (21.ce.4)

Table 5.6(a) Test Scenarios representing two instances of TWC

#	Effects	P/A	Test Scenarios
9	Call is terminated (21.es.4)	P	Scenario # 6 **and** A hangs up (21.ce.6) **and not** B hangs up (21.ce.4) **and not** Scenario # 11
			Scenario # 11 **and** B hangs up (21.ce.4) **and not** A hangs up (21.ce.6) **and not** Scenario # 6
10	Connection between B and D established (2.es.1)	P	Scenario # 4 **and** B dials D's number (2.ce.2) **and** D answers phone (2.ce.3)
11	A removed from hold. TWC established btw A, B, D (2.es.2)	P	Scenario # 10 **and** B executes flashhook (2.ce.1)
12	A removed from call Std two-party call btw B and D (2.ees.2)	P	Scenario # 11 **and** A hangs up (21.ce.6)
13	B goes on hold, Switching node sends dialtone to A (1.ees.1)	A	Scenario # 7 **and not** A executes flashhook (1.ce.1)
14	A goes on hold, Switching node sends dialtone to B (2.ees.1)	A	Scenario # 7 **and not** B executes flashhook (2.ce.1)
15	Connection between A and C established (1.es.1)	A	Scenario # 3 **and not** A dials C's number (1.ce.2) **and not** C answers phone (1.ce.3)
			Scenario # 3 **and** A dials C's number (1.ce.2) **and not** C answers phone (1.ce.3)
16	B removed from hold. TWC established btw A, B, C (1.es.2)	A	Scenario # 5 **and not** A executes flashhook (1.ce.1)
17	A engaged in a call with B (21.cs.1)	A	Scenario # 6 **and not** A executes flashhook (1.ce.1) **and not** C hangs up (1.ce.5) **and not** B executes flashhook (2.ce.1) **and not** D hangs up (2.ce.5) **and not** Scenario # 11 (Does not apply to 2.es.5)
	C removed from call (1.es.5)	A	Scenario # 11 **and not** B executes flashhook (2.ce.1) **and not** C hangs up (1.ce.5) **and not** A executes flashhook (1.ce.1) **and not** D hangs up (2.ce.5) **and not** Scenario # 6 (Does not apply to 1.es.5)
	D removed from call (2.es.5)	A	
18	B removed from call. Std two-party call established btw A and C (1.ees.2)	A	Scenario # 6 **and not** B hangs up (21.ce.4)

Table 5.6(b) Test Scenarios representing two instances of TWC

#	Effects	P/A	Test Scenarios
19	Call is terminated (21.es.4)	A	Scenario # 6 **and not** A hangs up (21.ce.6) **and not** B hangs up (21.ce.4) **and not** Scenario # 11
			Scenario # 11 **and not** B hangs up (21.ce.4) **and not** A hangs up (21.ce.6) **and not** Scenario # 6
20	Connection between B and D established (2.es.1)	A	Scenario # 4 **and not** B dials D's number (2.ce.2) **and not** D answers phone (2.ce.3)
			Scenario # 4 **and** B dials D's number (2.ce.2) **and not** D answers phone (2.ce.3)
21	A removed from hold. TWC established btw A, B, D (2.es.2)	A	Scenario # 10 **and not** B executes flashhook (2.ce.1)
22	A removed from call Std two-party call btw B and D (2.ees.2)	A	Scenario # 11 **and not** A hangs up (21.ce.6)

Table 5.6(c) Test Scenarios representing two instances of TWC

In this chapter a catalogue of specification errors or logical inconsistencies found in an ECEG representation is given. It is also shown how logical errors can arise when composing together ECEGs. The telephony specifications and ECEGs given in Chapter 4 are also validated through decision tables and test scenarios.

Chapter 6

Conclusion and Areas for Further Study

Cause-Effect graphing is basically a hardware testing technique adopted for software testing by Elmendorf [Elme73]. Since then, several researchers have further developed this strategy [Elme74, 75, Myer76, 79, Howd80, Adgr83, Wals83, Prob85, Soft91]. Cause-Effect graphing is a technique that selects systematic way, a high-yield set of test cases. It has a beneficial side effect in pointing out points of incompleteness and ambiguity in the specification [Myer79]. Finally, this method is the only black-box testing technique that explores the effects of combinations of causes

In this thesis, Myers Cause-Effect Graphing technique is first explained. Some logical inconsistencies in his technique have been spotted and possible solutions proposed. Then, it is shown how Myers technique is extended to best represent telecommunications features and services. This extension is called the extended cause-effect graphing technique. In addition we have shown how this technique can represent basic SDL constructs. During that same phase, an analysis of natural language specifications is made.

Furthermore, we have shown how the extended cause-effect graphing technique can be used to validate telecommunications specifications. We have experimented with this technique on several natural language specifications, including telecommunications specifications. It has also been used to validate combination of features in order to see if any two features can proceed successfully when present in the same service. We have found that this is a very reliable technique since it detects omissions, ambiguities, feature interactions, and several other common mistakes in informal specifications. The most significant cost-benefit accrues when this technique finds major errors even before the design phase of the software development cycle has been undertaken. Besides, it is the only black-box testing technique that focuses on combination of causes and at the same time it looks for both presence and absence of effects.

The process of validating a telecommunication feature from its informal requirements is shown and it is illustrated by Figure 6.1, while the process of validating combinations of features is shown in Figure 6.2.

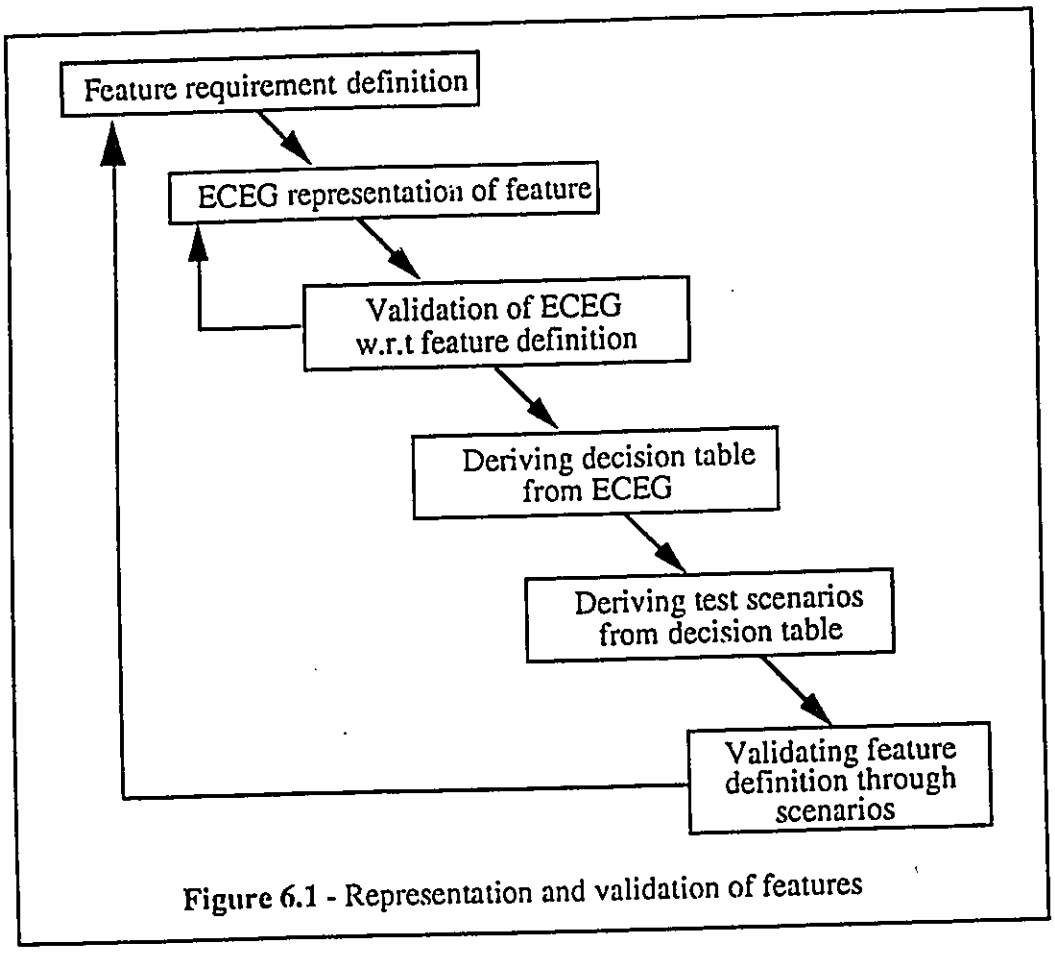


Figure 6.1 - Representation and validation of features

Only one commercial CEG tool exists presently on the market, namely the SoftTest tool [Soft91]. However, the objective of most of them is to derive test cases from the cause-effect graph in order to validate an implementation. Our goal is to validate the informal specification itself. Presently, we are developing a tool where we will derive the decision table and the test scenarios automatically from a given Extended Cause-Effect Graph. The ECEG is input in a text format; then our tool automatically draws the graph and derive the decision table and the test scenarios.

This requirement validation approach is based on heuristics which in turn are based on a limited survey of the practices in the application domain. This technique possesses many benefits. However, it does possess some drawbacks too. The main drawback is probably the up-front cost of deriving causes, effects, and constraints from a given informal specification. This process will

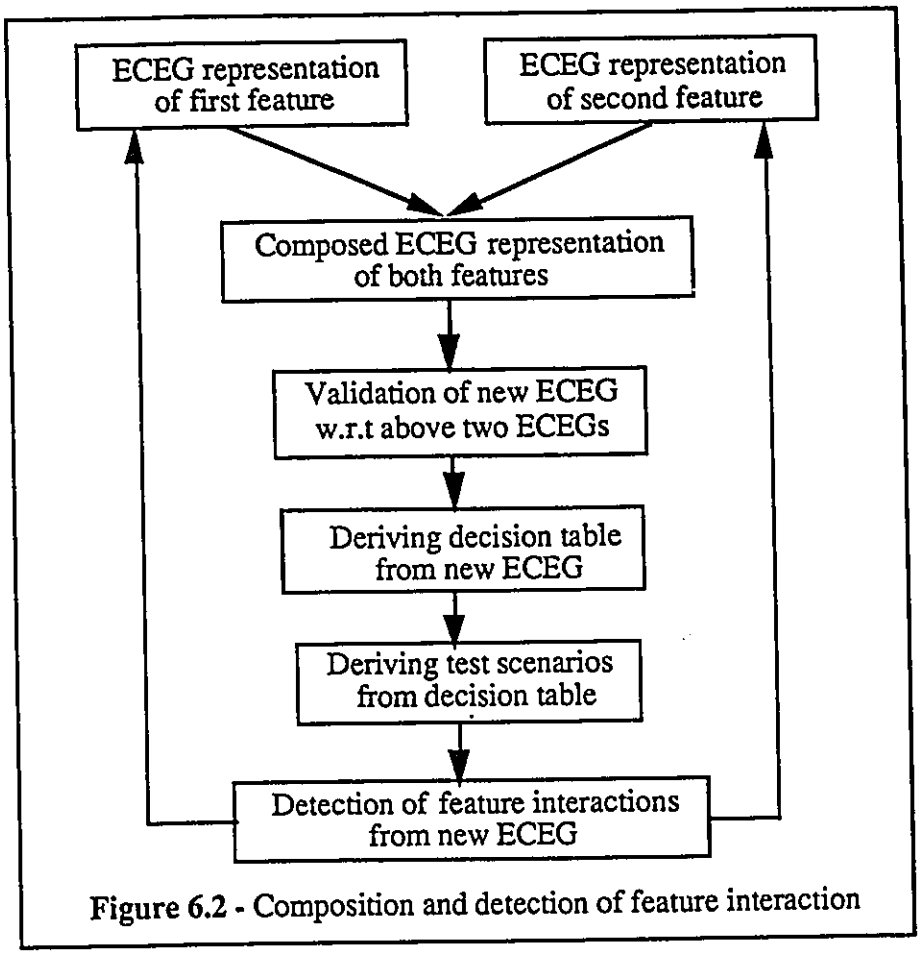


Figure 6.2 - Composition and detection of feature interaction

require domain knowledge and training in the ECEG method. However, up-front costs are small compared to the high potential for major downstream savings in rework and operational problems.

References

[Adgr83] Adler, M and Gray, M. A., "A Formalization of Myers Cause-Effect Graphs for Unit Testing", ACM SIGSOFT, Vol. 8, No.5, Oct. 1983, pp. 24-32.

[Arab86] S. Arabatzis, "Path Sensitization Technique for Test Case Design", Master's Thesis, University of Ottawa, 1986.

[Arms66] D. B. Armstrong, "On finding a nearly minimal set of fault detection tests for combinational logic nets", IEEE Transaction on Electronic Computers, Vol. EC-15, no. 1, pp. 63-70, Feb 1966.

[BaGo79] Balzer R., and N. Goodman, "Criteria for Good Specifications and their Impact on Specification Languages", Proceedings, IEEE Conference on Specifications, 1979.

[BeHo89] Ferenc Belina, Dieter Hogrefe, " Introduction to SDL", North Holland, Computer Networks and ISDN Systems 16, 1989, pp. 311-341.

[BoBr87] Bolognesi B., and Brinskma E. , "Introduction to the ISO specification Language LOTOS", Computer Networks and ISDN Systems 14 (1987), pp. 25-59.

[BoMi90] Boudriga N., A. Mili, F. Mili and R. Zalila, "Specification and verification of data types", Thirteenth Australian Computer Science Conference, Melbourne, February 7-9, 1990.

[Boum91] R. Boumezbeur, "Design, Specification, and Validation of Telephony Systems in LOTOS", Masters thesis, TR-91-30, University of Ottawa, July 1991.

[Boum93] R. Boumezbeur, L. Logrippo, "Specifying Telephone Systems in LOTOS", IEEE Communications, Vol. 31, No. 8, August 1993, pp 38-45

[Broo87] F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, vol. 20, no. 4, April 1987.

[BuDe87] S. Budkowski, P. Dembinski, "An introduction to ESTELLE: A specification language for distributed systems", Comput. Networks and ISDN Syst., Vol 14, no. 1, pp. 3-23, 1987.

[Came91] E. Jane Cameron, Yow-Jian Lin, "A Real-Time Transition Model for Analyzing Behavioral Compatibility of Telecommunications Services", ACM 1991.

[Came93] E. J. Cameron, H. Velthuijsen, "Feature Interactions in Telecommunications Systems", IEEE Communications, Vol. 31, No. 8, August 1993, pp 18-23

[Dwor89] F. S. Dworak, T.F. Bowen, C.H. Chow, G.E. Herman, N. Griffeth, and Y.-J. Lin, "Feature Interaction Problem in Telecommunication Systems". In Proceedings of the seventh International Conference on Software Engineering for Telecommunication Switching Systems, pages 59-62, July 1989. Bournemouth, United Kingdom.

[Elme73] Elmendorf, W. R., "Cause-Effect Graphs in Functional Testing", TR_00.2487, IBM Systems Development Division, Poughkeepsie, N.Y., Nov. 1973.

[Elme74] Elmendorf, W. R., "Functional Analysis using Cause-Effect Graphs", IBM Systems Development Division, Poughkeepsie, N. Y. 12602, presented at session L201 of SHARE XLIII, Chicago, Illinois, Aug. 26th-30th, 1974, pp. 566-577.

[Elme75] Elmendorf, W. R., "Functional Testing of Software Using Cause-Effect Graphs", IBM Systems Product Division, Poughkeepsie, N. Y., 12602, Automatic Support Systems Conference (ASSC) for Advanced Maintainability, Island Inn, Westbury L.I. New York, Oct. 28th-30th, 1975.

[Erra92] Mohammed Erradi, Ferhat Khendek, Rachida Dssouli, Gregor v. Bochmann, "Dynamic Extension of Object-Oriented Distributed System Specifications", Proceedings of International Workshop on Feature Interactions in the Telecommunications Software Systems, St. Petersburg, December 3-4, 1992.

[Fits92] Proceedings of International Workshop on Feature Interactions in Telecommunications Software Systems, IEEE Communication Society. St. Petersburg, December 3-4, 1992.

[Geha82] Narain Gehani, "Specifications: Formal and Informal - A case study", Software-Practice and Experience, vol. 12, pg 433-444, 1982.

[Glas81] Robert L. Glass, "Persistent Software Errors", IEEE Transactions on Software Engineering, vol. SE-7, n0. 2, March 1981.

[GoGe75] Goodenough J. B. and Gerhart S. L., "Towards A Theory Of Test Data Selection", IEEE Transaction in Software Engineering, Vol. SE-1, pp. 156-173, June 1975.

[Grif93] Griffeth, Nancy, "Defining features, services, and feature interactions", Note to feature Interaction Interest Group, January, 1993.

[Hall91] P.A.V. Hall, "Relationship between specifications and testing", Information and Software Technology, vol. 33, no. 1, January February 1991.

[Haml88] Richard Hamlet, "Special section on software testing", Communications of the ACM, vol. 31, no. 6, June 1988.

[Howd80] Howden, W. E., "Functional Testing and Design Abstraction", The Journal of System and Software 1, 1980, pp. 307-313.

[Inou92] Yasuaki Inoue, Kazumasa Takami, Tadashi Ohta, "Method for Supporting Detection and Elimination of Feature Interaction in a Telecommunication System", Proceedings of International Workshop on Feature Interactions in the Telecommunications Software Systems, St. Petersburg, December 3-4, 1992.

[Kuis93] E. Kuish, R. Janmaat, H. Mulder, I. Keesmaat, "A Practical Approach to Service Interactions", IEEE Communications, Vol. 31, No. 8, August 1993, pp 24-31

[LiBe79] Liskov Barbara, and V. Berzins, "An Appraisal of Program Specifications", Research Directions in Software Technology, P. Wegner, Editor. MIT Press, 1977.

[LiZi77] Liskov Barbara, and Stephen Zilles, "An introduction to formal specifications of data abstractions", Current Trends in Programming Methodology : Software

Specification and Design, R.T. Yeh, editor. Englewood Cliffs, NJ : Prentice hall, 1977.

[Meye85] Meyer B., "On Formalism in Specifications", IEEE Software, pp 6-26, January 1985.

[Myer76] Myers G. L., "Software Reliability : Principles and Practices", Wiley-Interscience, New York, 1976.

[Myer79] Myers G. L., "The Art of Software Testing", Wiley-Interscience, New-York, 1979.

[Naur69] Naur P., "Programming by Action Clusters", BIT, Vol. 9, No. 3, 1969, pp. 250-258.

[NuPr93] Nursimulu K., and R. Probert, " Cause-Effect Validation of Telecommunications Service Requirements", Proceedings of the Sixth International Conference in Software Engineering and its Applications, CNIT Paris-La Défense, November 15-19, 1993. To appear.

[Parn72] Parnas D L., "A Technique for Software Module Specification with Examples", Communications of the ACM, Vol 12, No 10, pp 330-336, May 1972.

[Prob85] Probert, R. L., Ural, H. and Smith, E., "CEG Advisor User's Guide", Department of Computer Science, University of Ottawa, July 1985.

[Rana92] A. Ranade, J. C. Petrides, "A Pragmatic View of Feature Interaction Management", Proceedings of International Workshop on Feature Interactions in the Telecommunications Software Systems, St. Petersburg, December 3-4, 1992.

[Soft91] Softest, "Requirements Based Testing CASE Tool", Release 3.2, Bender and Associates, inc., 1991.

[TuMa87] Turski W. M., and T.S.E. Maibaum, "The Specification of Computer Programs", Reading, MA : Addison Wesley, 1987.

[Wals83] Walsh, P. J., "An Analysis of Test case Selection", IBM Corporation, General Technology Division, Endicott, New York 13760, Phoenix Conference on Computers and Communications 2nd, 1983.

[[Woit92] Denise M. Woit, "An Analysis of Black-Box Testing Technique", Technical Report, Department of Computing and Information Science, Queen's University, February 1992.

[Zave93] Pamela Zave, "Feature Interactions and Formal Specifications in Telecommunications", IEEE Computer, August 1993.

Appendix A

Automatic Derivation of a Decision Table from an Extended Cause-Effect Graph

As Myers stated, the derivation of a decision table from its corresponding Cause-Effect graph is algorithmic [Myer79]. This means that an algorithm can be implemented on a computer to undertake this action. Actually, such an algorithm has already been implemented by different organizations, namely, in the ***SoftTest tool*** [Soft91] by Bender and Associates and at IBM research centers [Myer79]. These implementations follow Myers test derivation procedure. In [Arab86] a similar algorithm is implemented and it is based on Arabatzis Path Sensitization Technique.

All of the above implementations consist of deriving a decision table from a cause-effect graph. In this section we present an implementation to derive a decision from an extended cause-effect graph. The algorithm is based on our Basic Path Sensitization Technique which is given in Chapter 2, section 2.3.4.

A.1 The Derivation Procedure

The algorithm was implemented on a Macintosh computer using the Think Pascal programming language. As this algorithm cannot stand by itself, a module that will read in an Extended Cause-Effect graph (given in a text format) and build its internal structure is added to the implementation. We have also added a module that writes a decision table on the computer screen. These added modules allow to exercise and indirectly to test the algorithm.

The following sequence of operations is performed during execution of the algorithm:

(i) Read the Extended Cause-Effect graph text file and build its internal structure.
(ii) Read the constraints of the Extended Cause-Effect graph and save them internally.

(iii) Derive the Decision Table from the Extended Cause-Effect graph.

(iv) Apply the constraints to the Decision Table.

(v) Output the resulting Decision Table.

Note that the Extended Cause-Effect graph along with its constraints are given in the same input text file.

A.1.1 The Extended Cause-Effect graph.input file

This input file is a text file and it can be created by any text editor that use only ascii characters. The input file is a written representation of the Extended Cause-Effect graph. This input file should follow a well-defined syntax given in Figure A.1.

In the ECEG input text file, the reserved word "NODES" should first be written followed by the description of each node. Each such description should be written on one and only one line. If a node is reused then the character "*" should be written before giving defining the node. If some constraints are present in the graph, the reserved word "CONSTRAINTS" should follow the description of the nodes. Then each constraint should be described on one and only one line. Afterwards, the reserved word "RELATIONS" must be written before stating the relations in the graph.

Note that in the identifiers used to named the nodes should be alphanumeric. The identifier of a cause event node should start with a "ce.", a cause state node with "cs.", an intermediate node with "in.", an effect event node with "ee.", an effect state node with "es.", and an effect event and state with "ees.".

A.1.2 The Output Produced

When the ECEG input text file is fed to the implementation, the internal structure of the ECEG is built, a corresponding decision table is built, and then the latter is written on the computer screen. Figure A.2 shows the output format.

Note that the output is slightly different from the usual format of a decision table in the sense that in each test case is represented by a row instead of a column as it used to be. This new format can more easily be implemented than the usual one.

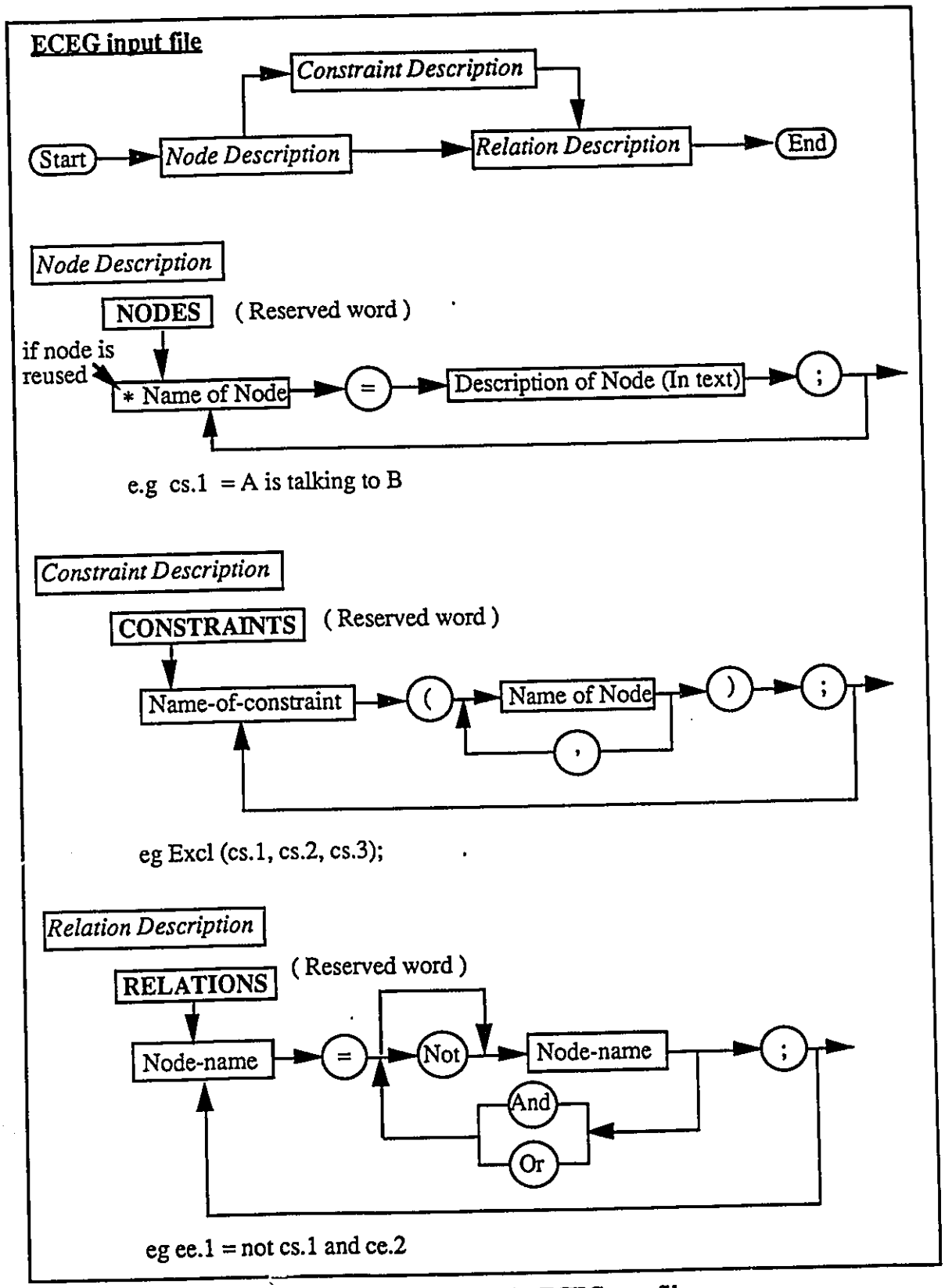


Figure A.1 Syntax of the ECEG text file

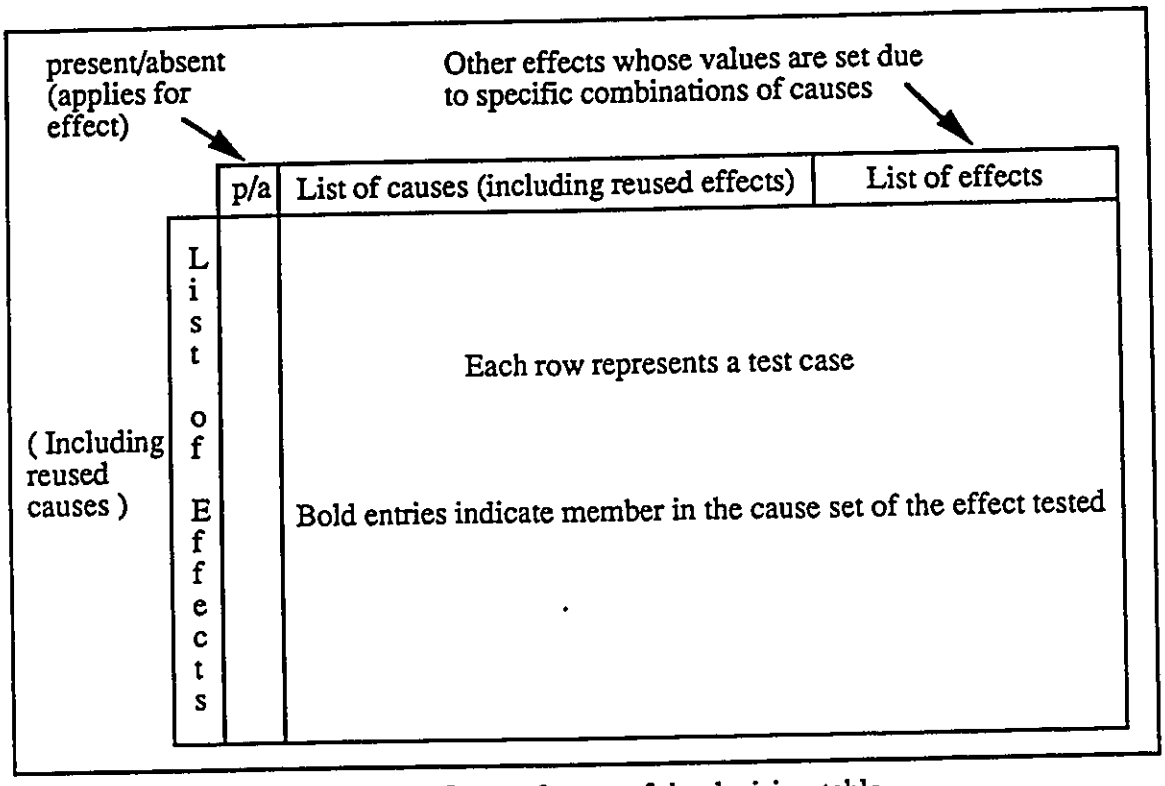


Figure A.2 Output format of the decision table

A.2 The Source Code

The implementation of the system is made in the ***Think Pascal language*** in the Macintosh environment. The system has been decomposed into four separate modules, namely the "MyStringDef" unit which contains the necessary tools to read and process an ECEG text file, the "ECEG" unit which contains tools to construct an internal representation of an ECEG given its text representation, the "DecisionTable" unit which deals with deriving a decision table from an internal ECEG representation, and finally the main program which controls the functioning of the whole process.

In the sections to come, we will give the source code for each of the above-named units.

A.2.1 The MyStringDef Unit

```
unit MyStringDef;
{
This unit provides the necessary data structures and string functions to read an ECEG text file before
creating its corresponding internal representation
}
```

```
Interface

  type
    ConsTable = array[1..10] of string;    { Table used to store temporily the constraints read }
                                           { from the text file.                                 }

    RelRecord = record                     { Record typeto store items in relation equations.    }
                  id: string;              { Node name.                                          }
                  Negation: Boolean;       { Set to true if a "not" is set on the edge relating  }
                end;                       { node "id" and its corresponding forward node.       }

    RelTable = array[1..10] of RelRecord;  { Table used to store temporily the relations read   }
                                           { from the text file.                                 }


  function EmptyStr (Str: string): boolean;
    { Returns true when str is empty }

  function EqualStr (Str1, Str2: string): Boolean;
    { Returns true when str1 = str2 }

  procedure GetToken (var Ligne, Token: string; var i: integer);
    { Returns in "Token" the word read from position i in string "Ligne" }

  procedure CopyStr (var str1: string; str2: string);
    { Copy string str2 onto string str1 }

  procedure GetNodeInfo (var Ligne, id, Desc: string; var Reuse: boolean);
    {
    Extract the node identifier as well as its description from string "Ligne" into variables "id" and
    "Desc" respectively. If the node is reused, then "Reuse" will be set to true.
    }

  procedure GetConsInfo (var Ligne, Cons: string; var Elements: ConsTable; var NumElem:
  integer);
    {
    Extract the constraint name as well as the items in the constraints into variables "Cons" and
    "Elements" respectively. "NumElem" will contain the number of items in the constraint.
    }

  procedure GetRelInfo (var Ligne, RelNode: string; var Rel: Integer; var Elements: RelTable; var
  NumElem: integer);
    {
    Extracts the node id on the LHS of the relation equation into "RelNode", then all the nodes found
    in the relation into "Elements" with NumElem giving the number of items found in that relation,
    and finally place inot "Rel" the nature of the relation : 0 - direct, 1- or, 2 - and.
    }

Implementation

  function EmptyStr (Str: string): boolean;
    var
      i: integer;
  begin
```

```
  i := 1;
  while (Str[i] = ' ') and (i <= length(Str)) do
    i := i + 1;
  EmptyStr := i > Length(Str);
end;


function EqualStr (Str1, Str2: string): Boolean;
  var
    k: integer;
    final: boolean;
begin
  k := 1;
  final := (length(str1) = length(str2));
  while (final) and (k <= length(str1)) do
    begin
      final := (str1[k] = str2[k]);
      k := k + 1;
    end;
  EqualStr := Final;
end;


procedure Uppercase (var Str: string);
    { Set to sppercase string "str" }
  var
    i: integer;
begin
  for i := 1 to length(Str) do
    begin
      if (Str[i] >= 'a') and (Str[i] <= 'z') then
        str[i] := chr(ord(str[i]) - 32);
    end;
end;


procedure GetToken (var Ligne, Token: string; var i: integer);
begin
  while (Ligne[i] = ' ') do      { discard blanks }
    i := i + 1;
  Token := '';
  while (Ligne[i] <> ' ') and (i <= Length(Ligne)) do
    begin
      Token := Concat(Token, Ligne[i]);
      i := i + 1;
    end;
  Uppercase(Token);              { Set token to uppercase }
end;


procedure CopyStr (var str1: string; str2: string);
begin
  Str1 := copy(Str2, 1, Length(Str2));
end;
```

```
procedure GetNodeInfo (var Ligne, id, Desc: string; var Reuse: boolean);
   {
   Extract the node identifier as well as its description from string "Ligne" into variables "id" and
   "Desc" respectively.  If the node is reused, then "Reuse" will be set to true.
   }
  var
    Token: string;
    i: integer;
begin
  i := 1;
  getToken(Ligne, Token, i);
  Reuse := EqualStr(Token, '*');      { if node is reused, a star will predece its idenfier in the file }
  if Reuse then
    getToken(Ligne, Token, i);
  CopyStr(id, Token);
  getToken(Ligne, Token, i);          { Ignore '=' }
  while (Ligne[i] = ' ') do           { Discard blanks }
    i := i + 1;
  Desc := '';
  while not (Ligne[i] in [';']) do    { Get the description of the node }
    begin
      Desc := Concat(Desc, Ligne[i]);
      i := i + 1;
    end;
end;


procedure GetConsInfo (var Ligne, Cons: string; var Elements: ConsTable; var NumElem: integer);
  {
    Extract the constraint name as well as the items in the constraints into variables "Cons" and
    "Elements" respectively.  "NumElem" will contain the number of items in the constraint.
  }
  var
    Token: string;
    i, k: integer;
begin
  i := 1;
  getToken(Ligne, Token, i);
  CopyStr(Cons, Token);
  getToken(Ligne, Token, i);          { ignores '(' }
  k := 1;
  repeat
    getToken(Ligne, Token, i);        { Fills in the Elements table }
    CopyStr(Elements[k], Token);
    k := k + 1;
    getToken(Ligne, Token, i);        { ignores ',' }
  until EqualStr(')', Token);
  Numelem := k - 1;
end;
```

```
procedure GetRelInfo (var Ligne, RelNode: string; var Rel: Integer; var Elements: RelTable; var
NumElem: integer);
  {
    Extracts the node id on the LHS of the relation equation into "RelNode", then all the nodes found in
    the relation into "Elements" with NumElem giving the number of items found in that relation, and
    finally place inot "Rel" the nature of the relation : 0 - direct, 1- or, 2 - and.
  }
  var
    Token: string;
    i, k: integer;
begin
  i := 1;
  getToken(Ligne, Token, i);
  CopyStr(RelNode, Token);          { Reads the LHS into RelNode }
  getToken(Ligne, Token, i);        { ignores '=' }
  k := 1;
  repeat
    getToken(Ligne, Token, i);      { Reads every item in the relation }
    Elements[k].Negation := (EqualStr('NOT', Token));       { Test for NOT }
    if Elements[k].Negation then
      getToken(Ligne, Token, i);
    CopyStr(Elements[k].id, Token);
    getToken(Ligne, Token, i);
    if (EqualStr('AND', Token)) then  { Test for the nature of the relation }
      Rel := 2
    else if (EqualStr('OR', Token)) then
      Rel := 1;
    k := k + 1;
  until (EqualStr(';', Token));     { End of string is marked by ";" }
  if k = 1 then
    Rel := 0;
  NumElem := k - 1;                 { Set the number of items in the relation }
end;

end. { Unit MyStringDef }
```

A.2.2 The ECEG Unit

```
unit Eceg;

Interface
  uses
    MyStringDef;

  type
    MyFiletype = text;              { The ECEG text file type }
    NodeIdType = string[6];         { Identier type of each node }
    NodeDescType = string[100];     { Node description }

    NextPrevPtr = ^NextPrevNode;    { Pointer to either forward or backward nodes of a given node }
    NextPrevNode = record
        id: NodeIdType;             { Node id }
```

```
    ArcRel: Boolean;              { Set to false if a "not" is on the edge linking this node }
    Next:  NextPrevPtr;
  end;

NodePtr = ^NodeType;              { Pointer to a node }
NodeType = record
    id: NodeIdType;               { Node id }
    Desc: NodeDescType;           { Node description }
    Class: Integer;               { 1- Cause, 2- Inter, 3- Effect, 4 - Cau+Eff, 5 - Eff+Cau }
    Reuse: Boolean;               { Set to true if node is reused }
    Relation: Integer;            { 0 -> direct, 1-> or, 2 -> and }
    ListForwNode:  NextPrevPtr;   { List of forward nodes of this node }
    ListBackNode:  NextPrevPtr;   { List of backward nodes of this node }
    Next: NodePtr;
  end;

ConsElemPtr = ^ConsElem;          { Pointer to a constraint element }
ConsElem = record
    id: NodeIdType;               { Node id found in the constraint }
    Next: ConsElemPtr;
  end;

ConsPtr = ^Cons;                  { Pointer to a constraint }
Cons = record
    Class: string;                { Constraint type: EXCL, INCL, ONE, REQ, MASK }
    Elem: ConsElemPtr;            { List of nodes affected by the constraint }
    Next: ConsPtr;
  end;

procedure BuildECEG (var f: MyFileType; var PtrNode: NodePtr; var PtrCons: ConsPtr);
  {
    Builds an internal ECEG representation given the ECEG text representation file "f". PtrNode will
    contain the pointer to the ECEG, while PtrCons will contain the pointer to the list of constraints
    found in the ECEG.
  }

function FindPtrNode (Ptr: NodePtr; str: string): NodePtr;
  {
    Returns the pointer to a node whose identifier is given by "str" in the internal ECEG
    representation.
  }
```

Implementation

```
procedure CopyNodePtr (var Ptr1, Ptr2: NodePtr);
  { Copies the node pointed by Ptr2 onto the node pointed by Ptr1 }
begin
  CopyStr(Ptr1^.id,  Ptr2^.id);
  Ptr1^.Class := 5;
  CopyStr(Ptr1^.Desc,  Ptr2^.Desc);
  Ptr1^.Reuse := True;
  Ptr1^.ListForwNode := nil;
```

```
  Ptr1^.ListBackNode := nil;
  Ptr1^.Next := nil;
end;


procedure CreateNode (var Ptr: NodePtr; id, Desc: string; Reuse: boolean);
  { Creates a node. If a node is reused, then two nodes will be created, one designating the cause
    while the other one designating the effect }
  var
    p, q: NodePtr;
begin
  New(p);
  CopyStr(p^.Id, id);
  CopyStr(p^.Desc, Desc);
  case id[1] of
    'C':
      P^.class := 1;
    'I':
      P^.Class := 2;
    'E':
      P^.Class := 3;
    otherwise
      p^.Class := 0;
  end;
  p^.Reuse := Reuse;
  p^.ListForwNode := nil;
  p^.ListbackNode := nil;
  p^.Next := nil;
  if reuse then
    begin
      New(q);
      CopyNodePtr(q, p);
      P^.Next := q;
      P^.Class := 4;
    end;
  Ptr := p;
end;


procedure CreateCons (var ptr: ConsPtr; var Cons: string; var Elements: ConsTable; NumElem:
integer);
  { Create the internal representation of a constraint }
  var
    p: ConsPtr;
    q, w: ConsElemPtr;
    i: integer;
begin
  New(P);
  CopyStr(P^.Class, Cons);
  P^.Next := nil;
  New(q);
  CopyStr(q^.Id, Elements[1]);
  q^.Next := nil;
  P^.Elem := q;
```

```
 w := q;
 i := 2;
 while (i <= NumElem) do
   begin
     New(q);
     CopyStr(q^.Id, Elements[i]);
     q^.Next := nil;
     w^.Next := q;
     w := q;
     i := i + 1;
   end;
  Ptr := p;
end;


procedure AppendNode (var Ptr1, Ptr2: NodePtr);
  { Appends node pointed by Ptr2 to list of nodes pointed by Ptr1 }
  var
    P: NodePtr;
begin
  if Ptr1 = nil then
    Ptr1 := Ptr2
  else
    begin
      P := Ptr1;
      while P^.Next <> nil do
        P := P^.Next;
      P^.Next := Ptr2;
    end;
end;


 procedure AppendCons (var Ptr1, Ptr2: ConsPtr);
   { Appends a constraint  pointed by Ptr2 to list of constraints  pointed by Ptr1 }
   var
     P: ConsPtr;
 begin
   if Ptr1 = nil then
     Ptr1 := Ptr2
   else
     begin
       P := Ptr1;
       while P^.Next <> nil do
         P := P^.Next;
       P^.Next := Ptr2;
     end;
  end;


  function FindPtrNode (Ptr: NodePtr; str: string): NodePtr;
    { Returns the pointer to a node whose identifier is given by "str" in the internal ECEG
      representation.}
    var
      p: NodePtr;
```

```
begin
  p := Ptr;
  while (p <> nil) and (not (EqualStr(P^.id, str))) do
    p := p^.next;
  FindPtrNode := p;
end;


procedure AppendForwNode (var PtrNode: NodePtr; RelNode: string; Eleme: RelTable; NumElem:
Integer);
  { Appends the list of the corresponding forward nodes to a given node }
  var
    p: NodePtr;
    q:  NextPrevPtr;
    i: integer;
begin
  for i := 1 to NumElem do
    begin
      p := FindPtrNode(PtrNode, Eleme[i].id);
      New(q);
      CopyStr(q^.id, RelNode);
      q^.ArcRel := not (Eleme[i].Negation);
      q^.Next := p^.ListForwNode;
      p^.ListForwNode := q;
    end;
end;


procedure AppendBackNode (var PtrNode: NodePtr; RelNode: string; Rel: integer; Eleme: RelTable;
NumElem: Integer);
  { Appends the list of the corresponding backward nodes to a given node }
  var
    p: NodePtr;
    q, w: NextPrevPtr;
    i: integer;
begin
  p := FindPtrNode(PtrNode, RelNode);
  if p^.Reuse then
    p := p^.Next;
  p^.Relation := Rel;
  New(q);
  CopyStr(q^.id, Eleme[1].id);
  q^.ArcRel := not (Eleme[1].Negation);
  q^.Next := nil;
  p^.ListBackNode := q;
  w := q;
  i := 2;
  while (i <= NumElem) do     { Adds each node to the list }
    begin
      New(q);
      CopyStr(q^.id, Eleme[i].id);
      q^.ArcRel := not (Eleme[i].Negation);
      q^.Next := nil;
      w^.Next := q;
```

```
      w := q;
      i := i + 1;
    end;
end;


procedure GetLineToken (var f: myFileType; var Ligne: string; var Finish: Boolean);
  { Reads one line from file "f" into string "Ligne". When end of file is reached, "Finish" is set to
  true }
  var
    ch: char;
begin
  repeat
    Ligne := '';
    while not (EOF(f) or EOLN(f)) do   { Reads one character at a time }
      begin
        Read(f, ch);
        Ligne := concat(Ligne, ch);
      end;
    if not EOF(f) then
      Readln(f);
  until not (EmptyStr(Ligne)) or Eof(f);
  Finish := Eof(f);
end;


procedure BuildECEG (var f: MyFileType; var PtrNode: NodePtr; var PtrCons: ConsPtr);
  {
    Builds an internal ECEG representation given the ECEG text representation file "f". PtrNode will
    contain the pointer to the ECEG, while PtrCons will contain the pointer to the list of constraints
    found in the ECEG.
  }
  var
    Ligne, Token, Cons, RelNode: string;
    id: NodeIdType;
    Desc: NodeDescType;
    Reuse, Finish: Boolean;
    i, NumElem, Rel: integer;
    Elements: ConsTable;
    Eleme: RelTable;
    Ptr1: NodePtr;
    Ptr2: ConsPtr;
begin
  i := 1;
  GetLineToken(f, Ligne, Finish);              { Reads a line }
  GetToken(Ligne, Token, i);                   { Get the first token from the line read . First the }
  GetLineToken(f, Ligne, Finish);              { reserved word NODES is expected. }
  repeat
    i := 1;
    GetNodeInfo(Ligne, id, Desc, Reuse);       { Read each node description until the word    }
    CreateNode(Ptr1, id, Desc, Reuse);         { "Constraints" or "Relations" is encountered }
    AppendNode(PtrNode, Ptr1);
    GetLineToken(f, Ligne, Finish);
    GetToken(Ligne, Token, i);
```

```
  until (EqualStr(Token, 'CONSTRAINTS') or EqualStr(Token, 'RELATIONS'));
  GetLineToken(f, Ligne, Finish);              { Reads a line }
  if EqualStr(Token, 'CONSTRAINTS') then       { If the word "constraints" was met before then }
    begin                                      { the constraints are processed                 }
      repeat
        i := 1;
        GetConsInfo(Ligne, Cons, Elements, NumElem);
        CreateCons(Ptr2, Cons, Elements, NumElem);
        AppendCons(PtrCons, Ptr2);
        GetLineToken(f, Ligne, Finish);
        GetToken(Ligne, Token, i);
      until (EqualStr(Token, 'RELATIONS'));  { Stop constraint processing when the word }
    GetLineToken(f, Ligne, Finish);          { "relations" is met.                       }
    end;
   repeat
    GetRelInfo(Ligne, RelNode, Rel, Eleme, NumElem);      { Processes the relations }
    AppendForwNode(PtrNode, RelNode, Eleme, NumElem);
    AppendBackNode(PtrNode, RelNode, Rel, Eleme, NumElem);
    GetLineToken(f, Ligne, finish);
    GetToken(Ligne, Token, i);
   until (Finish);
 end;

end. { Unit EGEG }
```

A.2.3 The DecisionTable Unit

```
unit DecisionTable;    { Used to derive a decision table from an internal ECEG representation }

interface
  uses
    MyStringDef, ECEG;

  type

    FixCauseListPtr = ^FixCauseList;  { Pointer to a list of causes whose values have been set }
    FixCauseList = record             { in order to sensitize an effect to a particular cause    }
        FixCause: string;
        Value: Integer;
        Next: FixCauseListPtr;
      end;

    CauseListPtr = ^CauseList;        { Pointer to a list of sensitized causes }
    CauseList = record
        SensitiCause: string;         { The sensitized cause }
        RelWithEff: Boolean;          { The relation between this cause and a specific effect }
        NumberToKeep: Integer;        { 3 - presence+absence , 2- presence, 1- absence , 0 - none }
        ListFixCause: FixCauseListPtr; { List of causes whose values are set }
        Next: CauseListPtr;           { Next sensitized cause }
      end;

    DetTabPtr = ^DetTab;              { Pointer to the decision table }
```

```
DetTab = record
    Effect: string;                { Effect Name }
    ListCause: CauseListPtr;       { Cause set of effect }
    Next: DetTabPtr;               { Next effect }
  end;

procedure CreateDT (PtrNode: NodePtr; PtrCons: ConsPtr; var PtrDT: DetTabPtr);
  {
  This procedure creates a decision table from an ECEG (pointed by PtrNode) having the constraints
  given by PtrCons.  PtrDt will contain a pointer to the decision table built.
  }


implementation

 type

   CauseInfo = record          { structure used to store the cause set of an effect }
      id: NodeIdType;
      Class: Integer;
    end;

   CauseSet = array[1..20] of CauseInfo;      { structure of a cause set of an effect }


 procedure AppendSensiCause (var p: DetTabPtr; q: CauseListPtr);
   { Appends a list of sensitized cause "q" to a decision table pointed by "p" }
   var
     t: CauseListPtr;
 begin
   t := p^.Listcause;
   if t = nil then
     p^.Listcause := q
    else
     begin
       while t^.Next <> nil do
         t := t^.Next;
       t^.next := q;
     end;
 end;


   procedure AppendFixCause (var p: CauseListPtr; q: FixCauseListptr);
     { Appends a list of cause "q" to a sensitized cause whose structure is pointed by "p" }
     var
       t: FixCauseListPtr;
   begin
     t := p^.ListFixcause;
     if t = nil then
       p^.ListFixcause := q
     else
       begin
         while t^.Next <> nil do
           t := t^.Next;
```

```
      t^.next := q;
    end;
end;


procedure AppendEffect (var Ptr: DetTabPtr; p: DetTabPtr);
  { Appends an effect with structure pointed by "p" to a decision table pointed by "Ptr" }
  var
    t: DetTabPtr;
begin
  if ptr = nil then
    ptr := p
  else
    begin
      t := ptr;
      while t^.Next <> nil do
        t := t^.Next;
      t^.next := p;
    end;
end;


procedure AppendEffectDT (var Ptr: DetTabPtr; id: string; var Tab: CauseSet; NumElem: Integer);
  { Appends an effect "id" with cause set "tab" (with NumElem elements) to a decision table pointed
    by "Ptr"}
  var
    p: DetTabPtr;
    q: CauseListPtr;
    r: FixCauseListPtr;
    i, j: integer;
begin
  New(p);
  CopyStr(p^.Effect, id);
  p^.ListCause := nil;
  p^.Next := nil;
  for i := 1 to NumElem do
    if (Tab[i].Class = 1) or (Tab[i].Class = 4) then    { cause or reused effect }
      begin
        New(q);
        CopyStr(q^.Sensiticause, Tab[i].Id);
        q^.NumberToKeep := 3;
        q^.ListFixCause := nil;
        q^.Next := nil;
        for j := 1 to NumElem do
          begin
            if (i <> j) and ((Tab[j].class = 1) or (Tab[j].Class = 4)) then  { cause or reused effect }
              begin
                New(r);
                CopyStr(r^.FixCause, Tab[j].Id);
                r^.Value := 1;
                r^.Next := nil;
                AppendFixcause(q, r);
              end;
          end;
```

```
        AppendSensiCause(p, q);
      end;
  AppendEffect(Ptr, p);
end;


function NotPresent (var Tab: CauseSet; str: string; NumElem: Integer): boolean;
  { Returns true if "str" is not present in "Tab" }
  var
    i: integer;
begin
  i := 1;
  while (i <= NumElem) and not EqualStr(tab[i].id, str) do
    i := i + 1;
  NotPresent := i > NumElem;
end;


procedure CreateCauseset (Ptr, temp: NodePtr; var Tab: CauseSet; var NumElem: integer);
  { Build a cause set for an effect pointed by "temp". "Tab" will contain the cause set while
  "NumElem" will give the number of elements in the cause set. }
  var
    i: integer;
    p: NextPrevPtr;
begin
  p := temp^.ListBackNode;
  while p <> nil do
    begin
      temp := FindPtrNode(Ptr, P^.id);
      if (NotPresent(Tab, temp^.id, NumElem)) then
        begin
          NumElem := NumElem + 1;
          copyStr(Tab[NumElem].id, temp^.id);
          Tab[NumElem].Class := temp^.Class;
          if temp^.Class = 2 then
            CreateCauseSet(Ptr, temp, tab, NumElem);
        end;
      p := p^.Next;
    end;
end;


function FindPtrDet (Ptr: DetTabPtr; str: string): DetTabPtr;
  { Returns the pointer to the effect "Str" in the decision table pointed by "Ptr" }
  var
    p: DetTabPtr;
begin
  p := Ptr;
  while (p <> nil) and (not (EqualStr(P^.Effect, str))) do
    p := p^.next;
  FindPtrDet := p;
end;
```

```
procedure WriteOutput (ptrDt: DetTabPtr);
  { Writes the decision table on the output screen }
  var
    p: DetTabPtr;
    q: CauseListPtr;
    r: FixCauseListPtr;
    i: integer;
begin
  writeln('DECISION TABLE');
  p := PtrDt;
  while p <> nil do
    begin
      q := p^.ListCause;
      while q <> nil do
        begin
          i := 1;
          if q^.NumberToKeep <> 0 then
            begin
              repeat
                write(q^.NumberToKeep : 4 - i, q^.SensitiCause, ' :');
                r := q^.ListFixcause;
                while r <> nil do
                  begin
                    write(r^.fixcause, '=', r^.Value : 3);
                    r := r^.next;
                  end;
                write('  =>', p^.Effect, '= ');
                if (q^.NumberToKeep >= 2) and (i = 1) then
                  begin
                    writeln(ord(q^.RelWithEff));
                    i := i + 1;
                  end
                else
                  begin
                    writeln(ord(not  q^.RelWithEff));
                    i := 3;
                  end;
              until i > 2;
            end;
          q := q^.next;
          writeln;
        end;
      p := p^.next;
      writeln;
    end;
  end;


  function FindPtrCauseList (q: CauseListPtr; str: string): FixCauseListPtr;
    { Returns a pointer to a cause "str" in the list "q" }
    var
      r: FixCauseListPtr;
  begin
    r := q^.ListFixCause;
```

```
  while (r <> nil) and not (EqualStr(r^.Fixcause, Str)) do
    r := r^.Next;
  FindPtrCauseList := r;
end;


procedure BackTrack (PtrNode: NodePtr; var q: causeListPtr; Nodeid, FromNode: string;
ValueToSet: boolean);
  { Backtracks a node "nodeid" to set the corresponding causes in the list "q" to specific values.
  "FromNode" is the node from which backtracking started in the first place }
  var
    p: NodePtr;
    r:  NextPrevPtr;
    s: FixCauseListPtr;
begin
  p := FindPtrNode(PtrNode, Nodeid);
  r := p^.ListBackNode;
  while r <> nil do
    begin
      if not (EqualStr(r^.id, FromNode)) then
        begin
          if not (r^.ArcRel) then
            ValueToSet := not (ValueToSet);
          BackTrack(PtrNode, q, r^.id, FromNode, ValueToSet);
        end;
      r := r^.Next;
    end;
  s := FindPtrCauseList(q, p^.id);
  s^.value := ord(ValueToSet);
end;


  procedure SensitizeCauseEffect (PtrNode: NodePtr; var q: CauseListPtr; Cause, Effect: string; var
  Relation: Boolean; Tab: CauseSet; NumElem: integer);
    { This procedure sensitize an effect "Effect" to a cause "Cause".  It will do so by setting the values
      of each cause (except "cause") in the cause set "Tab" to a specific value such that presence of
      effect "effect" will depend entirely on presence or absence of cause "cause" }
    var
      p: NodePtr;
      r:  NextPrevPtr;
  begin
    p := FindPtrNode(PtrNode, Cause);
    r := p^.ListForwNode;
    while r <> nil do
      begin
        if not (NotPresent(Tab, r^.Id, NumElem)) or EqualStr(r^.id, Effect) then
          begin
            p := FindPtrNode(PtrNode, r^.id);
            if not (r^.ArcRel) then
              Relation := not Relation;
            BackTrack(PtrNode, q, p^.id, Cause, not (p^.Relation = 1));
            SensitizeCauseEffect(PtrNode, q, p^.id, Effect, Relation, Tab, NumElem);
          end;
        r := r^.next;
```

```
    end;
end;


procedure DeriveTestCases (PtrNode: NodePtr; var PtrDT: DetTabPtr; Effect: string; var Tab:
CauseSet; NumElem: integer);
  { This procedure derives the test cases for each effect.  At this step it does not consider the impact
    of the constraints, if any, on the final test set. }
  var
    p: DetTabPtr;
    q: CauseListPtr;
    s: NodePtr;
    Relation: boolean;
begin
  p := FindPtrDet(PtrDT, Effect);
  q := p^.ListCause;
  while q <> nil do
    begin
      Relation := true;
      s := FindPtrNode(PtrNode, q^.SensitiCause);
      SensitizeCauseEffect(PtrNode, q, q^.Sensiticause, Effect, Relation, Tab, NumElem);
    q^.RelWithEff := relation;
      q := q^.Next;
    end;
end;


function ElementOf (str: string; q: ConsElemPtr): boolean;
  { Returns true if the element "str" is found in the list "q" }
  var
    p: ConsElemPtr;
begin
  p := q;
  while (p <> nil) and not EqualStr(str, p^.id) do
    p := p^.Next;
  ElementOf := p <> nil;
end;


function SetNumberToKeep (ConsType, Number: integer; SenCauseInCons: boolean): integer;
  { This function considers the constraint given by "Constype" and determines the number of test
    cases to be built for a given sensitized cause }
begin
  case ConsType of
    1:
      if Number = 1 then
        SetNumberToKeep := ord(not SenCauseIncons) * 2 + 1
      else
        SetNumberToKeep := ord(Number = 0) * 3;
    2:
      if Number = 0 then
        SetNumberToKeep := ord(SenCauseIncons) * 2
      else
        SetNumberToKeep := 3;
```

```
    3:
      if Number = 1 then
        SetNumberToKeep := ord(not SenCauseInCons) * 2 + 1
      else if Number = 0 then
        SetNumberToKeep := ord(SencauseIncons) * 2
      else
        SetNumbertoKeep := 0;
  end;
end;


procedure ProcessConsEIO (q: ConsElemPtr; ConsType: Integer; var PtrDt: DetTabPtr);
  { Processes the Excl, Incl, and One-and-only-one constraints }
  var
    s: DetTabPtr;
    t: CauseListPtr;
    v: FixCauseListptr;
    Number: integer;
    SenCauseInCons: Boolean;
begin
  s := PtrDt;
  while s <> nil do
    begin
      t := s^.ListCause;
      while (t <> nil) and (t^.NumberToKeep <> 0) do
        begin
          SenCauseIncons := ElementOf(t^.SensitiCause, q);
          Number := 0;
          v := t^.Listfixcause;
          while (v <> nil) do
            begin
              if ElementOf(v^.fixCause, q) then
                Number := Number + v^.value;
              v := v^.Next;
            end;
          Number := SetNumberToKeep(ConsType, Number, SenCauseInCons);
          if t^.NumberToKeep = 3 then
            t^.numberToKeep := Number
          else if (t^.NumberToKeep <> Number) and (Number <> 3) then
            t^.NumberToKeep := 0;
          t := t^.next;
        end;
      s := s^.Next;
    end;
  end;


  procedure ProcessConsReq (q: ConsElemPtr; var PtrDt: DetTabPtr);
    { Processes the Requires constraint }
    var
      s: DetTabPtr;
      t: CauseListPtr;
      v, w: FixCauseListptr;
      Number: integer;
```

```
    SenCauseInCons: Boolean;
begin
  s := PtrDt;
  while s <> nil do
    begin
      t := s^.ListCause;
      while t <> nil do
        begin
          v := FindPtrCauseList(t, q^.id);
          if EqualStr(q^.id, t^.SensitiCause) or (v <> nil) then
            begin
              if ((v <> nil) and (v^.value = 1)) or ((v = nil) and (t^.NumberToKeep >= 2)) then
                begin
                  w := t^.Listfixcause;
                  while (w <> nil) do
                    begin
                      if ElementOf(w^.fixCause, q) and (w^.Value = 0) then
                        t^.NumberToKeep := ord((v = nil) and (t^.NumberToKeep = 3));
                      w := w^.Next;
                    end;
                end
            end;
          t := t^.next;
        end;
      s := s^.Next;
    end;
end;


procedure ApplyConstraint (var PtrDt: DetTabPtr; PtrCons: ConsPtr);
  { Controls and processes all the cause constraints. The mask constraint will be taken into
    consideration when the output will be written }
  var
    p: ConsPtr;
begin
  p := PtrCons;
  while p <> nil do
    begin
      if Equalstr(p^.class, 'EXCL') then
        ProcessConsEIO(p^.Elem, 1, PtrDt)
      else if EqualStr(p^.Class, 'INCL') then
        ProcessConsEIO(p^.Elem, 2, PtrDt)
      else if EqualStr(p^.Class, 'ONE') then
        ProcessConsEIO(p^.Elem, 3, PtrDt)
      else if EqualStr(p^.Class, 'REQ') then
        ProcessConsReq(p^.Elem, PtrDt);
      p := p^.Next;
    end;
end;


procedure CreateDT (PtrNode: NodePtr; PtrCons: ConsPtr; var PtrDT: DetTabPtr);
  { This procedure creates a decision table from an ECEG (pointed by PtrNode) having the constraints
    given by PtrCons.  PtrDt will contain a pointer to the decision table built. }
```

```
var
    p: NodePtr;
    Tab: causeSet;
    NumElem, i: integer;
begin
  p := PtrNode;
  while p <> nil do
    begin
      if (p^.class = 3) or (p^.class = 5) then
        begin
          NumElem := 0;
          CreateCauseSet(PtrNode, p, Tab, NumElem);
          AppendEffectDT(PtrDT, p^.id, Tab, NumElem);
          DeriveTestCases(PtrNode, PtrDT, p^.id, Tab, NumElem);
        end;
      p := p^.Next;
    end;
  ApplyConstraint(PtrDt,  PtrCons);
  WriteOutput(ptrDt);
end;

end. { Unit Decision Table }
```

A.2.4 The Main unit

```
program Main;

uses
ECEG, DecisionTable;

var
  F: MyFileType;
  PtrNode: NodePtr;          { Pointer to an ECEG }
  PtrCons: ConsPtr;          { Pointer to the constraint list }
  PtrDet: DetTabPtr;         { Pointer to a decision table }

begin
  reset(f,  'fic');          { "fic" contains the text eceg representation }
  PtrNode := nil;
  PtrCons := nil;
  PtrDet := nil;
  BuildECEG(f, PtrNode, PtrCons);          { Building of an ECEG }
  close(f);
  CreateDT(PtrNode, PtrCons, PtrDet);      { Building of the decision table }
end.
```

A.3 Examples of Application of the Above Implementation

In this section we will give two examples of application of the above implementation. The first one is about Naur Text Reformatter Problem [Naur69] while the other one is derived from a the Three-Way Calling specification given in Chapter 4, section 4.4.1.

A.3.1 Naur Text Reformatter Problem

The CEG of Figure 2.12 given in Chapter 2 is translated into its equivalent ECEG input text file. This file is shown in Figure A.3 below. The identifiers given to the nodes have been

```
NODES
    ce.1 = Character is not BL or NL or ET;
    ce.2 = Word is too long;
    ce.3 = Character is NL;
    ce.4 = Character is BL;
    ce.5 = Word found will fit on current line;
    ce.6 = At least one character found and not printed;
    ce.7 = A word was already printed on this line;
    ce.8 = Character is ET;
    in.20 = Character is a break;
    in.21 = A word is found;
    in.22 = A word found will fit on current line which already contains a word;
    ee.90 = No output (character put in buffer);
    ee.91 = Alarm is raised;
    ee.92 = BL and buffer printed;
    ee.93 = NL and buffer printed;
    ee.94 = No output (multiple or preceding break characters;

CONSTRAINTS
    One( ce.1,  ce.3, ce.4, ce.8 );
    Req(ce.2, ce.1 );

RELATIONS
    in.20 =  ce.3 or  ce.4  or ce.8 ;
    in.21 = ce.6 and in.20 ;
    in.22 =  ce.5 and ce.7 ;
    ee.90 =  ce.1 and not ce.2 ;
    ee.91 =  ce.2 and ce.1  ;
    ee.92  = in.21 and  in.22  ;
    ee.93  =  in.21  and not in.22 ;
    ee.94  =  in.20 and  not ce.6 ;
```

Figure A.3 ECEG text representation of the Naur Text Reformatter Problem

prefixed with "ce", "in", or "ee" to indicate a cause, intermediate, and effect node. This has been done in order to conform to the required input text format. As it can be noted, in order to produce such a file the user should have made a graphical representation of the graph on paper first. The text input produced is then fed into the implementation which in turn produces the output given in Figure A.4. It can be observed that the decision table produced in this way is the same as the one given in Figure 2.22 of Chapter 2.

		Causes								Effects				
	p/a	ce.1	ce.2	ce.3	ce.4	ce.5	ce.6	ce.7	ce.8	ee.90	ee.91	ee.92	ee.93	ee.94
ee.90	1	1	0	0	0	-	-	-	0	-	0	0	0	0
ee.91	1	1	1	0	0	-	-	-	0	0	-	0	0	0
ee.92	1	0	0	1	0	1	1	1	0	0	0	-	0	0
ee.92	1	0	0	0	1	1	1	1	0	0	0	-	0	0
ee.92	1	0	0	0	0	1	1	1	1	0	0	-	0	0
ee.93	1	0	0	1	0	1	1	0	0	0	0	0	-	0
ee.93	1	0	0	0	1	1	1	0	0	0	0	0	-	0
ee.93	1	0	0	0	0	1	1	0	1	0	0	0	-	0
ee.93	1	0	0	1	0	0	1	1	0	0	0	0	-	0
ee.93	1	0	0	1	0	1	1	0	0	0	0	0	-	0
ee.94	1	0	0	1	0	-	0	-	0	0	0	0	0	-
ee.94	1	0	0	0	1	-	0	-	0	0	0	0	0	-
ee.94	1	0	0	0	0	-	0	-	1	0	0	0	0	-
ee.94	1	0	0	1	0	1	0	1	0	0	0	0	0	-
ee.94	1	0	0	1	0	1	0	0	0	0	0	0	0	-
ee.90	0	0	0	0	0	-	-	-	0	0	0	0	0	0
ee.92	0	0	0	0	0	1	1	1	0	0	0	0	0	0
ee.93	0	0	0	0	0	1	1	0	0	0	0	0	-	0
ee.94	0	0	0	0	0	-	0	-	0	0	0	0	0	-
ee.94	0	0	0	1	0	-	1	-	0	0	0	0	0	-

Effects

Figure A.4 Decision produced for the Naur Text Reformatter Problem

A.3.2 Three-Way Calling

The Three-Way Calling specification along with a corresponding Extended Cause-Effect Graph are given in chapter 2. From the ECEG representation, a text representation is made and it is given in Figure A.5. Note how some nodes description are preceded by "*" in order to indicate reused nodes. Figure A.6 gives the decision table produced when the ECEG text file is fed into the computer.

NODES

* cs.1 = A engaged in a call with B;
  cc.1 = A executes flashhook;
  cc.2 = A dials C's number;
  cc.3 = C answers phone;
  cc.4 = B hangs up;
  cc.5 = C hangs up;
  cc.6 = A hangs up;
  in.1 = Either C hangs up or A executes flashhook;
  in.2 = Either C hangs up or A executes flashhook during TWC btw. A,B,and C;
* ccs.1 = B goes on hold. Switching node sends dialtone to A;
* cc.1 = A receives dialtone;
* cs.1 = Connection between A and C established;
* cs.2 B = removed from hold. TWC established;
  ccs.2 = B removed from call. Standard two-party call established btwn. A and C;
  cs.4 = Call is terminated;
  cc.2 = C removed from call;

CONSTRAINTS

Req(cc.2, cc.1);
Req(cc.3, cc.2);

RELATIONS

cc.1 = ccs.1;
cs.1 = cc.1 and cc.2 and cc.3;
cs.2 = cc.1 and cs.1 ;
cc.5 = in.2;
cs.4 = in.2;
cc.4 = cs.2 and cc.6;
ccs.2 = cs.2 and cc.4;
ccs.1 = cs.1 and cc.1;

Figure 4.5 ECEG text representation of the Three-Way Calling specification

			Causes											Effects							
		p/a	ce.1	ce.2	ce.3	ce.4	ce.5	ce.6	cs.1	ee.1	es.1	es.2	ees.1	cs.1	ee.1	es.1	es.2	ees.1	ees.2	es.4	es.5
Effects	cs.1	1	0	0	0	0	1	0	0	0	0	1	0	-	0	0	0	0	0	0	1
	cs.1	1	1	0	0	0	0	0	0	0	0	1	0	-	0	0	0	0	0	0	1
	ee.1	1	0	0	0	0	0	0	0	0	0	0	1	0	-	0	0	0	0	0	0
	es.1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	-	0	0	0	0	0
	es.2	1	1	0	0	0	1	0	0	0	1	0	0	0	0	0	-	0	0	0	0
	ees.1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	-	0	0	0
	ees.2	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	-	0	0
	es.4	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	-	0
	cs.1	0	0	0	0	0	0	0	0	0	0	1	0	-	0	0	0	0	0	0	0
	ees.1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	-	0	0	0
	es.1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	-	0	0	0	0	0
	es.1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	-	0	0	0	0	0
	es.2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	-	0	0	0	0
	ees.2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	-	0	0
	es.4	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	-	0

Figure A.6 Decision Table produced for Three-way Calling