



THE LIBRARY

THE UNIVERSITY OF TEXAS AT ARLINGTON

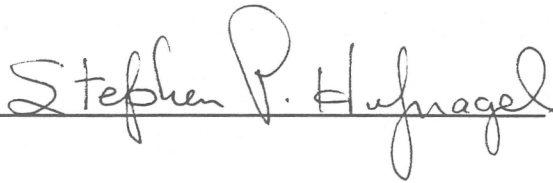
Manuscript Thesis

Any unpublished thesis submitted for the Master's or the Doctor's degree and deposited in the Library of The University of Texas at Arlington is open for inspection, but is to be used only with due regard to the rights of the author. Bibliographical references may be noted, but passages may be copied only in accord with the limitations of the "fair use" provisions of the Copyright Law of the United States, and proper credit must be given in subsequent written or published work. Extensive copying or publication of this thesis in whole or in part requires the permission of the author and the consent of the Dean of the Graduate School of The University of Texas at Arlington.


GRAPHICAL EVENT-DIRECTED SCENARIO BEHAVIORAL SPECIFICATIONS
FOR THE SCENARIO-BASED ENGINEERING PROCESS (SEP)
USING A DOMAIN SPECIFIC SOFTWARE
ARCHITECTURE (ASSA) PHILOSOPHY

The members of Committee approve the masters
thesis of Miao Xia

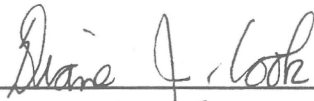
Stephen Hufnagel
Supervising Professor



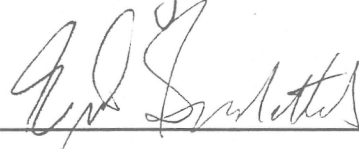
Karan Harbison



Diane Cook



Erik Mettala



Copyright© by Miao Xia 1993

All Rights Reserved

GRAPHICAL EVENT-DIRECTED SCENARIO BEHAVIORAL SPECIFICATIONS
FOR THE SCENARIO-BASED ENGINEERING PROCESS (SEP)
USING A DOMAIN SPECIFIC SOFTWARE
ARCHITECTURE (ASSA) PHILOSOPHY

by

MIAO XIA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 1993

ACKNOWLEDGMENTS

I would like to solemnly express my appreciation to Dr. Hufnagel for the assistance, guidance, and encouragement he has given me during my whole work on this thesis; my efforts were supported by his knowledge and insight tremendously. I also would like to present my thanks to Dr. Harbison, Dr. Cook, and Dr. Mettala for their assistance. Without their assistance, this thesis could not have been completed. Finally, I thank my wife, Qun Zhao, for sacrificing her personal interests for the sake of my graduate work at UTA. This work is dedicated to her.

November 10, 1993

ABSTRACT

GRAPHICAL EVENT-DIRECTED SCENARIO BEHAVIORAL SPECIFICATIONS
FOR THE SCENARIO-BASED ENGINEERING PROCESS (SEP)
USING A DOMAIN SPECIFIC SOFTWARE
ARCHITECTURE (ASSA) PHILOSOPHY

Publication No. _____

Miao Xia, M.S.

The University of Texas at Arlington, 1993

Supervising Professor: Stephen P. Hufnagel

This thesis extends scenarios of a system into real-time event digraphs. A scenario is an event trace resulting from a particular thread of system execution. An event digraph is an event network. Multiple external stimuli result in a wave of execution in distributed and parallel architecture.

* A graph theoretic formal definition for event digraphs is developed with the finite sets of events and the *scenario event order* (SEO). A scenario language is developed from the event digraph. In modeling an event digraph, we introduce the *frontier expansion* mechanism which let the users and the developers go around the question if this is the last occurrence, what will be the next? The behavior of event digraphs is specified with the input and output of nodes and the *event dictionary*. The occurrences of events in event graphs are specified with the token.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	iv
ABSTRACT	v
LIST OF ILLUSTRATIONS	ix
1. INTRODUCTION	1
2. PREPARATION	
2.1 Event.....	3
2.1.1 The Definition of Events.....	4
2.1.2 The Event Abstraction and Properties.....	4
2.1.3 Event Value.....	7
2.2 State.....	7
2.3 Stimuli.....	8
2.3.1 The Definition and the Properties of Stimuli.....	8
2.2.3 Stimuli-system Partition.....	8
2.4 Boundary State.....	9
2.5 Scenarios.....	10
2.5.1 SEO.....	11
2.5.2 More about Event.....	13
2.5.3 The Definition of Scenarios.....	13
3. PERCEPTUAL VIEWING OF EVENT DIGRAPH	
3.1 The Abstraction of Event Digraph.....	14
3.2 Example 1, Vending Machine System.....	15

3.3 Example 2, Cruise Control System.....	23
---	----

4. STRUCTURE AND ANALYSIS OF ED

4.1 The Definition in Formal Form.....	30
4.2 The Behavior of ED.....	32
4.2.1 The Concurrence within the Proceeding Set and the Succeeding Set.....	32
4.2.2 Using Token to Specify the Occurrence Wave.....	34
4.2.3 The Input and Output of Nodes.....	35
4.2.4 The Event Dictionary.....	38
4.3 Modeling.....	38
4.4 The Follow Trees.....	40
4.4.1 The Definition and the Algorithm.....	41
4.4.2 The Pages of the Following Trees for an ED.....	42
4.4.3 The $E\{\wedge\} \leftarrow e$ in Following Trees.....	42
4.5 Scenario Language.....	43
4.5.1 The Scenario Language Definition.....	43
4.5.2 Establish an Gs from an ED.....	43
4.6 Hierarchical Expansion of an ED.....	45
4.6.1 Sub-extension and Subgraph-Contraction.....	45
4.6.2 Establish a Sub-extension Grammar from the Gs.....	46

5. COMPARISON WITH OTHER GRAPHICAL REPRESENTATION

IN REQUIREMENT ANALYSIS PHASE

5.1 Comparison with Petri Nets.....	47
5.2 Comparison with Event Tace Diagram.....	49
5.3 Comparison with DFD.....	49
5.4 Companrison with STD.....	50
6. CONCLUSION AND FUTURE WORK.....	52

ILLUSTRATIONS.....	58
APPENDIX A.....	79
APPENDIX B.....	82
APPENDIX C.....	98
REFERENCES.....	100

LIST OF ILLUSTRATIONS

Figure 1: ED of the Vending Machine System.....	59
Figure 2: Sub-ED of the Vending Machine System.....	60
Figure 3: The Follower Tree (Sub-ED) of the Vending Machine System.....	61
Figure 4: The Followe Tree of the Vending Machine System.....	62
Figure 5: ED of Scenario 1, Cruise Control; Cruise Control System.....	63
Figure 6: ED of Scenario 2, Inc/Dec the Maintained Speed; Cruise Control System...	64
Figure 7: ED of Scenario 3, Resume; Cruise Control System.....	65
Figure 8: ED of Scenario 4, Brake Application; Cruise Control System.....	66
Figure 9: ED of Scenario 5, Terrain Condition; Cruise Control System.....	67
Figure 10: The Follower Tree of Scenario 1; Cruise Control System.....	68
Figure 11: The Follower Tree of Scenario 2, 3, 4, 5; Cruise Control System.....	69
Figure 12: The Paged Follower Tree of the Vending Machine System (page 1).....	70
Figure 13: The Paged Follower Tree of the Vending Machine System (page 2).....	71
Figure 14: The Paged Follower Tree of the Vending Machine System (page 3).....	72
Figure 15: The Paged Follower Tree of the Vending Machine System (page 4).....	73
Figure 16: The Paged Follower Tree of the Vending Machine System (page 5).....	74
Figure 17: The Paged Follower Tree of the Vending Machine System (page 6).....	75
Figure 18: The Transformation from I Arcs to Petri Net Structures.....	76
Figure 19: The Refinement of e9, Hierarchical Extension of an ED.....	77
Figure 20: The Object Model of the Vending Machine System.....	78

CHAPTER 1

INTRODUCTION

Scenarios were incorporated into the life cycle model of Object-Oriented software development in [Wang 91] (SDOOD). In the Scenario-Based Engineering Process (SEP) [Hufnagel & Harbison 93], scenarios are fundamental. Scenarios provide consistent communication mechanism among users, managers, and developers. Scenarios are used to map among a system's requirement, specification, design, implementation, and other documentation. Scenarios fulfill the seamless software development goal [Hufnagel & Liou 90]. However, the method of building scenarios into a requirement document remains ad-hoc. The scenario representation format is very informal. This thesis formalizes that process and makes it deterministic.

This thesis extends scenarios into real-time event digraphs. These digraphs combine system scenarios into graphical structures. A scenario is an event trace resulting from a particular system thread of execution [Rumbaugh 91] (p86), while an event digraph is an event network. Multiple external stimuli result in a wave of execution in distributed and parallel architecture.

This thesis consists of six chapters and three appendices. Chapter 2 describes the fundamental concepts and terminology. It lays the foundation for event digraph construction for reactive systems. A scenario is redefined with a scenario event order (SEO), the system external stimuli, and the scenario boundary states. The boundary states are defined within the system execution. The external stimuli are defined by the property that the stimuli, including the external inputs, can occur spontaneously and randomly with respect to a system. An SEO is described with the assumption that the

followed event is the last occurrence in the current event occurrence wave. Chapter 3 shows the perceptual view of event digraphs and the pragmatic steps to build an event digraph from an actual scenario. The *Vending Machine System* and the *Cruise Control System* are given as two small real-time examples. We give the formal definition and the behavior specifications for event digraphs in Chapter 4. In Chapter 4, a scenario language and the ED hierarchical expansion mechanism are also developed with the event digraph base. Chapter 5 compares event digraphs with other requirement analysis graphical representation. The last Chapter includes the summary, conclusions, and suggested future work.

CHAPTER 2

BACKGROUND

The purpose of preparing developers with scenarios of a system is to get a better understanding of the expected system behavior [Rumbaugh 91] (p170). We will extend scenarios into an event digraph. The purpose of building an event digraph is to formally specify system behavior. The concepts, the terminology, and the abstractions discussed below are used later in the thesis.

2.1 Event

In Coad & Yourdon's kindergarten [Coad & Yourdon 90] (p1), the distinguishing object's feature often is the attribute defining the objects' distribution. We may note that a chair is located beside a table. We also may note that we observed the table first, then the chair, or vice versa. We may say that the table is *previous* to the chair because we saw the table first. The objects' distribution is the foundation of our spatial concept. The occurrence distribution is the foundation of our time concept. In Einstein's relativity, both are integrated together. The order abstraction takes an important part in their integration. The observation order depends on the position relationship among the objects and the observer. So with respect to the observer, the occurrence order is a kind of spatial object order. This consideration is only to illustrate the important relationship between events and observers. In system analysis, the events and the event order abstractions will first be informally described.

2.1.1 The Definition of Events

One of the intuitive human activities is to deal with the real world by partitioning time. Some examples are: *Flight 098 departs from DFW Airport At 2:00 PM, today's seminar will be held after the class CSE 6325, the meeting is scheduled from 1:00 PM to 3:00 PM, etc.*

An event is the boundary of a time partition. In the partition *today's seminar will be held after the class CSE 6325*, the boundary can be represented as *the seminar is finished* or as *the class CSE 6325 begins*. In the partition *the meeting will be held from 1:00 PM to 3:00 PM*, we can set the boundaries as such: *the meeting begins at 1:00 PM* and *the meeting ends at 3:00 PM*. The example *Flight 098 departs from DFW Airport at 2:00 PM* itself is a partition boundary. It divides the time into two parts, *before the departure* and *after the departure*. We use events to mark time. Note that our public standard time is marked with the ticking of the clock. Each tick is a fundamental event. Imagine if there were no event occurrence in the universe, there would have been no time in this universe. "Processes are event-driven rather than clock driven" [Jonathan & W 90] (p133).

2.1.2 The Event Abstraction and Properties

By comparing the adjacent time segments partitioned by the events, we can always define distinguishing attributes. For example: *Flight 098 was on the ground before 2:00 PM; and after 2:00, it was in the air*. During CSE 6325, *the number of student is 30*. When the following class starts, *there are 28 students*. We use the word *parameter* to represent things that change in the different partitions. In the first example, the *parameter* refers to the *position of the plane, Flight 098*; in the second

example, the parameter refers to the *number of student*. The observer determines the significant attributes.

To illustrate other event features, let us ride on a *time train*. This will enable us to see what will appear on the time mark. We observe that the appearance of time marks is instantaneous. The some attributes between adjacent segments have been changed on the time marks. The changing is caused by some mutual effects among objects. We have observed three properties of events: instantaneous attribute, parameter change, and mutual effect. The instantaneous attribute is the essential characteristic; parameter change and mutual effect are the intrinsic parts of the events.

Of course, nothing is really instantaneous. If one of our observations is of a very short time duration relative to another observation, then the former observation can be seen as an event. Anything such as a process, object, concept, etc., can be abstracted as an event, when we study or observe the instantaneous state of its attributes. For an object, its existence can be abstracted as an event for a relatively time scale. The signal, interrupt, occurrence, happening, action, etc., are events that can have the *at a point in time* property.

In computer science, the function, task, operation, process, and state concepts are fundamental. They all have a start time and an end time in their system execution. It is natural to consider the questions:

When is a function called?

When does a process start?

When does a state exist?

When is an object in active?

Etc.

Those *start* and *end* observations are events. They can be modeled with *at a point in time* attribute.

Time is always marked on the point where some parameters change or some mutual effects occur. For example, at 1:00 PM, a car hit a tree (mutual effect), the speed of Flight 098 stops increasing (parameter change), Flight 098 is on land (object state change).

A mutual effect implies some parameter changes. For instance, a car hit a tree implies that the body of the car was damaged, I open the door implies that the position of the door is changed.

A parameter change may not imply a mutual effect. For instance, the stopwatch is set for 2-minutes and 2-minute time passed does not imply any mutual effect. Note that 2-minute time passed itself may be a signal that causes some event occurrences.

In some applications, we only abstract one side of mutual effects. For example a car is hit, the door is opened. A description of events can be done as a parameter change form. They can be either as a one sided effected form, or as a two-sides mutual effect form. In which form an event is abstracted depends on both the observers and the problem domain. With respect to the object-oriented approach, each side in mutual effect may be either an object or an attribute; a parameter change can be either an operation or a state transition; mutual effect can result from message passing.

Applying the group mechanism among events results in the event class concept. Applying the inheritance mechanism among event classes, we get event class hierarchical structures. Examples and discussions about event classes and event inheritance can be find in [Rumbaugh 91] (p85, p98).

2.1.3 Event Value

We define the *event value* as the value the changed parameter conveys. In *Flight 098 departures at 2:00*, the event value is 098. For *the door is opened*, possible values of the door in this abstraction are *open* or *closed*. For the event *check the plan's speed*, the event value may be 200 mph, 600 mph, etc. An event may be described with a group of objects or attributes. The one that is changed conveys the identity of that event.

2.2 State

A state is the event partitioned time interval in which the event effects remain in effect. If we ride on the *time train*, the states appear as the *colored distances* between the events. The *color* here means the superposition of event effects. The current occurred event superposes its effect on the *color* that was made by the previously occurred events. That the next occurred event sweeps out the last event's effect is viewed as the special case of superposition. Each superposition causes a new state. In other words, an event may affect not only its incident state, but also its succeeding states. It follows that a state can be specified by several event values.

In an example of an A/C system, consider the following event sequence

Turn on power → *turn the cool to high* → *turn the thermostat to 11*
 → *turn the cool to low*

Each event causes a change of the A/C. Note that *turns the cool to low* sweeps out the effect of *turn the cool on high*, but not that of *turns the thermostat to 11*. The *turn on power* affects not only its incident state, but also the other states following it. Also note that each state is specified with a value combination of power, fan, and thermostat value.

2.3 Stimuli

2.3.1 The Definition and the Properties of *Stimuli*

We define *Stimuli* as a finite collection of external events of a system; each occurs spontaneously and randomly with respect to the system. For example, consider a TV set with the external events: *turn on power, change to channel 8, adjust the bright button on low*, etc. The TV can not predict which one will occur next.

For an input event, if the input data is queried by the system and the data's coming is inevitable, then this event does not belong to *Stimuli*. A system queries of data from an external storage is not a stimulus, because we assume that the external storage will always supply what the system requires. The occurrence of the queried input is inevitable, and not spontaneous. The occurrence of an input of a system from a user belongs to *Stimuli* because the user may make an error data or the user may input some data even if the system does not require it.

At a point in time, a single stimulus may occur or a group of stimuli may occur together. According to the event exclusive discussion [William 88], we follow the regulation that the two exclusive stimuli cannot stimulate a system at the same time. For instance, we can not turn the TV on and at the same time turn it off; we cannot change the channel to 8 and at the same time to 21.

The occurrence of an event in a *Stimuli* set is an external effect of a system. Some external effect on a system will continue for a duration of time; the stimulus is the start or the end of the external effects.

2.3.2 Stimuli-System Partition

Two well-known software requirement analysis approaches are Structured Analysis

(SA) and Object-Oriented Analysis (OOA). In SA, the data flow diagram (DFD) is a typical graphical representation. To specify a requirement with DFDs, the 0-level DFD is what we give first [Pressman 92] (p209). A 0-level DFD represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. In OOA, we always represent first a top level OO model with an environment-system partition form and a message passing mechanism between the environment and the system, both of which are abstracted as two high level objects.

A common feature of 0-level DFDs and environment-system high level OO model is a 2-partition, which is the first abstraction we developed to specify the relationship between a system and its external existence.

When we specify a requirement with a scenario extended event digraph, we do a similar thing. We take stimuli-system 2-partition as our top level model in this case. The *Stimuli* has been defined in the last section.

We compare stimuli-system abstraction with a 0-level DFD:

1. The external part of DFD includes the system output, while a *Stimuli* doesn't.
2. A *Stimuli* specifies the data input occurrences as well as any system external effective if they are spontaneous and random.

The behavior of the system part in stimuli-system is referred as the function of the stimuli's behavior.

2.4 Boundary State

A boundary state is an externally visible state which users or domain experts or observers abstracted from a system. The identification of a boundary state for a system depends on the abstraction level in the application. Note that the users or

domain experts are permitted to stipulate only one boundary state for a system. The transition back to the boundary state means a scenario is finished or the objects return to their original state. The transition out of the boundary state means the beginning of a complete-scenario. The scenario and the complete-scenario will be introduced later. A boundary state may be the state that no external effective remains in the system or may be the state that no internal events occur in the system. We often view the system's *OFF* mode (When a system is not in execution) as a boundary state. For example: the A/C is in the state *off*, the TV is in the state *off*, etc. We also often refer the mode of system's *waiting clients* (When there is no effect of *Stimuli* remained in a system) as a boundary state. For example: the ATM machine is waiting for customers, the pharmacy is open, etc.

Note that the users or domain experts can only stipulate one above case as the boundary state of a system. If the domain experts say that *OFF* mode is perceived as a boundary state, the *Waiting Customer* is not viewed as a boundary state.

2.5 Scenarios

A scenario is defined in [Rumbaugh 91] (p463) as a sequence of events that occur during one particular system execution. To build an event digraph and to consider multiple execution threads of a system requires a more powerful definition. The definition above deals with a particular execution path, while an event digraph must consider all possible execution paths. A scenario is redefined with the boundary state and the scenario event order (SEO). In this section we discuss the *SEO* first, then define scenarios.

2.5.1 SEO

SEO is a collection of ordered event pairs (tail, head); the tail is assumed to be the *last occurrence* in an occurrence wave with respect to the head and the head is assumed to follow the tail *immediately*. An example could be concurrently enjoying McDonald's Extra Value Meal package that includes Big Mac, fries and a soft drink. A person eats the Big Mac while eating fries and while drinking the soft drink. The person must at least take a bite on every food item. The last swallowed food item cannot be known in advance. If we want to consider the next event for the event *swallowing Big Mac*, we assume that the last swallowed is the Big Mac; if we search for the events that follow the event *swallowing Fries*, the last swallowed is assumed to be fries; and so on. Then we may have the following expression:

$(\text{finish Big Mac, go}), (\text{finish fries, go}), (\text{finish drink, go}) \in \text{SEO}.$

Note that with respect to the event *the person leaves*, the last occurrence may be *finish Big Mac*, or *finish fries*, or *finish drink*.

The same mechanism can be also applied if we want to consider the next occurrences from the occurred events. We suppose that all food and drink are possible from the start eating event. If we want to search for the follower of Big Mac, we say that Big Mac is the last started, and so on. We may have

$(\text{start Big Mac, finish Big Mac}), (\text{start Big Mac, finish fries}),$
 $(\text{start Big Mac, finish drink}),$
 $(\text{start fries, finish Big Mac}), (\text{start fries, finish fries}),$
 $(\text{start fries, finish drink}),$
 $(\text{start drink, finish Big Mac}), (\text{start drink, finish fries}),$
 $(\text{start drink, finish drink}) \in \text{SEO}.$

There are three kinds of *SEO* orders. The tail and the head are causally related in the first kind *SEO*. The head occurs only if or iff the tail occurs. For example: *it rains*

→ *the earth becomes wet (only if); the answer machine signal is flashing ↔ somebody has left a message (iff).*

The tail and head are conditionally causally related in the second kind of *SEO*. The head occurring depends on some particular value of tail or on some special states in which the tail occurs. In other words, the head implies the tail. For instance, if the deposited coin is enough for a candy bar, the machine returns the new inserted coin after the event *insert coin*; if the deposited coin is not enough, the machine sets a new state after the *insert coin*. Therefore, we have the second kind *SEO* as: *insert coin ← sets a new state and insert coin ← returns new inserted coin.*

In the last kind *SEO*, the head, in a particular state, can possibly occur immediately after the occurrence of the tail, and the tail and the head are not causally related. This state may or may not be caused by the tail, and may or may not be result from the head. Note that not every event can occur in a particular state. An example is the following situation: a student is in the engineering building II, and he has the option to leave the building immediately, or he may go into the #2 elevator immediately, but he can not play football immediately.

Again the essence of *SEO* is the relation that under the certain abstraction views the occurrence of an event follows another occurrence immediately either inevitably or possibly. This relation is partitioned into three parts by applying causality mechanism. In system analysis, whether an event order belongs to *SEO* is determined by users, domain experts. Similarly, abstraction viewers can define relations. In other word, whether the causality relation, conditionally causality, or possibly following under certain situation can be assigned between two events is decided by users or domain experts. The *SEO* can also be viewed as the domain experts or users defined logic. Therefore in system modeling both users and analysts can ask the question *If this is the last occurrence, what will be the next?*

2.5.2 More about Events

If two events have the same description (i.e., in the same parameter change form or in the same mutual effect form), but they occur in the different states, we will categorize them as two distinct events. In other words, an event is determined not only by its description but also by its position in time (sequence of events). There is a difference between a student leaving the library before the closing time or after the closing time, even though both *leaving library* events are of the same description.

★ 2.5.3 The Definition of a Scenario

Scenario: A sequence of events where

1. The start event is a stimulus, and after the last event occurs, the system state is the boundary state.

2. Each immediate ordered pair in the sequence belongs to *SEO*.

If the start event of a scenario also occurs in the boundary state, this scenario is called a *complete-scenario*. *SEO* excludes the arbitrary event ordered pairs that do not belong to scenarios. A stimulus is permitted to be appear more than once in the interval of a scenario sequence.

CHAPTER 3

PERCEPTUAL VIEWING OF EVENT DIGRAPH

SEO is actually the mechanism that provides us a heuristic to search for the followers for each event concerned. Since the mechanism for searching the next occurrence is simpler, we will first examine two small examples to show how to use this mechanism to build event digraphs; therefore, the perceptual viewing of event digraphs in this chapter will be illustrated. The formal definition and the analysis of the event digraphs are to be discussed in the next chapter.

3.1 The Abstraction of an Event Digraph

We abstract a system as a finite collection of events E that

$$E \equiv \{e_1, e_2, \dots, e_n\}$$

We refer *SEO* to a relation such that

$$SEO \subseteq E \times E$$

The result of this abstraction (E, SEO) is a directed graph called an event digraph (*ED*).

In building an *ED* according to a requirement document, an *SEO* is applied as the question *if this event was the last occurred, what will be the next?* Developers can ask this question not only for themselves but also for the user's view. This question can be a center of the communication between users and developers if an *ED* is developed. Furthermore, developers and users can work together to build an *ED* since *ED* development only needs the *next occurrence* mechanism, which is intuitive and requires no knowledge of computer technique.

3.2 Example 1, Vending Machine System

This Example comes from the UTA course, *Real-Time Design* (CSE 6325 SPRING 93).

Requirement: The vending machine receives coins, then dispenses candy bars. The machine holds five types of candy bars, 200 of each type. The machine does not give change. The machine only accepts 5, 10, 25 coins. An incomplete transaction will be canceled after 2 minutes, if not successfully completed. Termination of transaction shall result in the return of all deposited money to the customer. Being ultra modern, the machine has an internal phone modem, and can alert the vendor when any of the following conditions occurs:

1. The candy supply of any candy bar is less than 10; a delivery service will then fill the machine to capacity.

2. There are more than 40 units of any coin, since the coin holders can only hold 50 units of any coin. An armored truck will remove all money from the machine. If the armored truck does not arrive within one hour, the machine will shut itself down. It will go into a protect-mode, to prevent robbery.

ED Building: (The result is shown on Figure 1.)

Assumption 1: The services of Armored truck and candy bar delivery are combined into one service, *Truck Service*.

Step 1, Identify stimuli: *Stimuli* is a collection of the external effects in which each element occurs spontaneously and randomly with respect to a system. With respect to the vending machine, we see the following external events:

e₁': *insert a coin*

e₂': *push a candy button*

These events are spontaneous and random. A customer may push the button first, then push the button again, then insert 5-cent coin, then push button, etc. We may

assume that the indicators as *machine asks customer to insert coin* and *machine asks customer to push the candy button* are designed from the machine's view, but it is possible that customers may play with the machine and ignore those operation indications or that the customers may make some wrong operations. So the machine cannot expect what will occur to it, and in what order. The *truck* can be referred to as an external store. The machine puts the coins into and takes the candy bars from the *truck*. The *truck* service occurs only if the machine requests it.

Let $e_1', e_2' \in E$. The primes on e_1', e_2' indicate that those events are stimuli.

Let $SEO = \emptyset$.

Step 2, Determine the Boundary State: We determine that the vending machine enters its boundary state only if there are no coins deposited in the machine.

Step 3, Make the Search Queue, Q: We add every element in E into a queue Q . At first, for the elements in *Stimuli*, the adding order is made by chance. However, the following discussion may offer some clues that will help in deciding what element in *Stimuli* is added into the front of Q . We can add the "normal" inputs, or "normal" stimuli in the front of Q . A "normal" input or a "normal" stimulus is the first event of the "normal" scenarios. The meaning of a "normal" scenario can be found in [Rumbaugh 91] (p170). For a "normal" scenario, we do not consider unusual conditions, the "special" cases such as *omitted input sequences, maximum and minimum values, and repeated values*, the user error cases including *invalid values and failures to respond*. For the example of an ATM machine [Rumbaugh 91] (p151-185), we will look at the input sequence:

user insert a cash card → *user enters his password* →

user selects the kind of transaction → *user enters the \$100*

This is a "normal" input sequence that an ATM machine expects. In our example, the machine normally expects that the a customer first inserts a coin. Therefore, the Q of the vending machine after adding the elements of *Stimuli* looks like

$$Q \equiv (e_1', e_2').$$

Step 4, Search for Followers: We remove Q (Remove- $Q(e_i)$), let the removed element occur, then search the following occurrences in the system in this step. What kind of following occurrences can be searched out depends on our abstractions. Users can help to make this searching more *deterministic*. For each new obtained event if it is not in E , we add it in E and do Add- $Q(e_i)$. Then add the event ordered pair, which is constructed with Remove- $Q(e_i)$ as the tail and the new obtained event as the head to *SEO*. We repeat this step until Q is empty.

In our vending machine example, we do Remove- $Q(e_1')$. What will occur next if Remove- $Q(e_i)$ were the last occurrence on the vending machine system? After the occurrence of e_1 , the *count 2-minute time* should be executed while a new coin value state should be set. However, if the current coin value is already enough for a candy bar, the machine should return the new inserted coin directly instead of setting a new coin value state. That is to say that the machine needs to check the current coin value state before deciding whether to set a new state or to return the new received coin. We design an event *check state 1* to deal with the situation above. The reason that we put a *1* after *check state* will be illustrated later. We have the following abstractions and manipulations:

e_3 : *check state 1*, (a follower of e_1').

e_4 : *count 2-minutes*, (a follower of e_1')

Let $e_3, e_4 \in E$

Let $(e_1', e_3), (e_1', e_4) \in SEO$

e_5 : *return coin 1*, (a follower of e_3).

e_6 : *set state*, (a follower of e_3).

Let $e_5, e_6 \in E$

Let $(e_3, e_5)', (e_3, e_6)' \in E$

Add_Q(e_4), Add_Q(e_5), Add_Q(e_6).

Note that there are primes on $(e_3, e_5)'$ and $(e_3, e_6)'$. We use the prime symbol to make distinguish among the three kinds of *SEO*. A (e_i, e_j) indicates that e_i and e_j are causally related; a $(e_i, e_j)'$ indicates that e_i and e_j are conditionally causally related; a $(e_i, e_j)''$ indicates that e_i and e_j are not causally related and that e_j is "required" or "designed" to follow e_i . In our example, whether e_3 causes e_5 or e_6 depends on the value of the current coin state value (being enough for a candy bar or not), so $(e_3, e_5)'$ and $(e_3, e_6)'$ belong to the second type of *SEO*. The immediate result so far is

$$E \equiv \{e_1', e_2', e_3, e_4, e_5, e_6\}$$

$$SEO \equiv \{(e_1', e_3), (e_1', e_4), (e_3, e_5)', (e_3, e_6)'\}$$

Note that we don't Add_Q(e_3), because the succeeding events of e_3 have already been designed in the same time. The Q right now looks like

$$Q \equiv (e_2', e_4, e_5, e_6).$$

Remove_Q(e_2'), let e_2' occur. What will be the next events if e_2' is the last occurrence on the vending machine? After a candy selecting button has been pushed, the next actions of the machine depend also on the current coin value state. If the current coin value is enough or too much for a candy bar, the machine dispenses the candy bar the customer has chosen. If a customer first pushes a candy button without depositing any coin, the machine will do nothing and continue to keep its boundary state. If a customer pushes a candy button under the state that the coin value is not enough for a candy bar, the machine will keep waiting until the 2-minute time is out or another coin is deposited or candy buttons are pushed again. The same event under different states will cause different state transitions. This is an outstanding characteristic shown on

State Transition Diagrams (*STD*). In *ED* building, this situation requires us to design again a *State Checking* event. For our vending machine *ED* building process, we design the event *check state 2* to occur between *pushes a candy button* and the events that may be caused. We have another group of abstractions and manipulations as follow

e_7 : *check state 2*, (a follower of e_2')

Let $e_7 \in E$

Let $(e_2', e_7) \in SEO$

e_8 : *2-minute time is out*, (a follower of e_7).

e_9 : *decrease the number of a candy bar*, (a follower of e_7).

e_{10}'' : *reset*, (a follower of e_7 , the last event of a scenario).

e_1' : (a follower of e_7 , already in E).

e_2' : (a follower of e_7 , already in E).

Let $e_8, e_9, e_{10}'' \in E$

Let $(e_7, e_8)'', (e_7, e_9)', (e_7, e_{10}''), (e_7, e_1')'', (e_7, e_2')'' \in SEO$

Add_Q(e_8), Add_Q(e_9).

Note that we have used the symbols as e_i , e_i' , and e_i'' to distinguish the three kinds of nodes so far. The e_i' means that the event indicated can be the first event of a scenario, which is also a stimulus. The e_i'' means that the event indicated can be the last event of a scenario. After e_i'' , a system then goes back to its boundary state. The e_i means that the event indicated can only be an interval event in a scenario. In our example, the event *return coin 1* cannot start a scenario. The *return coin 1* is not an external event and can not happen spontaneously and randomly. Furthermore, after the *return coin 1*, there are still some processes going in the system. Something will happen before the boundary state and after the *return coin 1*. According to the definition, only a stimulus can start a scenario. In our example, the *Stimuli* includes e_1'

and e_2' . After the event *reset*, the vending machine system goes back to its boundary state. Actually, if a customer pushes a candy button without deposit, there is no change on machine's state. However, since we view e_7 as an internal event—a part of the system execution, we still let the event *reset* to follow e_7 for our vending machine simulation. That also means that the system gets its boundary state one more time after the stimulus e_2' . The following scenario is an illustration.

Scenario: A Customer's Curiosity

1. A customer pushes a candy button without any deposit (occurs under the boundary state).
2. The machine checks that the current coin value is zero.
3. The machine is resets, and goes back the scenario boundary state.

Actually, no interval event in this scenario is in the active state. This is not a "normal" scenario.

Note that we don't take $\text{Add-Q}(e_7)$ and $\text{Add_Q}(e_{10})$. The situation of e_7 is the same as that of e_3 . We already have the succeeding events of e_7 , and we do not need to consider e_7 's followers any more. We do not add e_{10} into Q because e_{10} is the end of the scenarios; there are supposed to be no followers of the ends of scenarios. The Q looks like $Q \equiv (e_4, e_5, e_6, e_8, e_9)$.

Note that if the current coin value is enough or over for a candy bar, e_7 will cause e_9 . Therefore the arc $(e_7, e_9)'$ has a prime come with as it up script. If the current coin value is not enough, e_1' or e_2' or e_8 may follow e_7 . However e_7 won't cause any one of them; the arcs $(e_7, e_1)''$, $(e_7, e_2)''$, and $(e_7, e_8)''$ come with double primes as their up scripts.

Note that the actions conveyed by e_3 and e_7 are the same — *check state*. However, we design *check state* into two distinct events. With the respect of the action *check state*, the succeeding event set depends on the *history* of the *check state*, i.e., we need

to be concerned with what has been happened before the *check state* occurred. In the graphical representation, we need to examine from where the *check state* comes. For the same value, for example the *current coin value is enough*, of the *check state*, if the *check state* comes from e_1' , the machines return the newly deposited coin; if it comes from e_2' , the machine dispenses a candy bar. An event is identified not only by its instantaneous performance but also by its effect in time. We abstract an event by instantaneous performance of things; we distinguish events by the time effect of an instantaneous performance. Technically, our distinguishing events according to their different effect propagating in time guarantees that the occurrences of the succeeding events are only determined by the performance of the preceding event.

The sub-ED of the vending machine so far is

$$E \equiv \{e_1', e_2', e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}''\}$$

$$\text{SEO} \equiv \{(e_1', e_3), (e_1', e_4), (e_3, e_5)', (e_3, e_6)', (e_2', e_7), (e_7, e_1)'', (e_7, e_2)'', (e_7, e_8)'', (e_7, e_9)', (e_7, e_{10})''\}$$

The graphical representations of the sub-ED are shown on Figure 2. From the Figure 2, we may already have a premonition that an ED's graphical representation for even a medium system may become a chaotic picture. The solution to the chaos is to apply the *follow tree* as one of the ED's graphical representations. Figure 3 is the *follow tree* of the Figure 2. The *follow tree* of an ED is not only a method to help us to organize a good documentation format in ED requirement analysis approach but also a graphical expression emphasizing the question — *what will be the next occurrences?*. More discussions about the *follow tree* is in the next chapter.

We do *Step 3* in iterative way, continuing to removing Q , abstracting and designing the succeeding event set for each removed event by keeping in mind with the question *as if this is the last occurrence, what will be the next?*, then adding the new produced events into Q , until Q is empty. The graphical representations of the vending machine

are shown on Figure 1 and Figure 4. The Figure 4 is the *follow tree* of the final result. Since in our abstraction in the section 3.1, E is finite for a system. We can always get the time that Q becomes empty.

The behavior of a graphical representation is specified by another *ED* component — *Event Dictionary*. An *Event Dictionary* plays a part as that a requirement dictionary [Pressman 90, Thomas 90] plays in DFD. The whole *event dictionary* of the vending machine system is located in Appendix B. We have more discussion about *event dictionaries* in the next chapter.

Note that a scenario in an *ED* appears as a *walk* [Bondy & Murty 77] (p12). The start and the end of a walk are a stimulus and a sink point. The sink point is unique in a scenario; stimuli are not necessary to be distinct in a scenario. Furthermore, it is not required that there must be some interval events happening under the active states.

The process we used to build an *ED* above is similar to the Breadth-First Search (*BFS*) algorithm [Larry & Sanford 90, Thomas 90]. We identify first the stimulus set, then initialize a queue to contain all the events in the stimulus set. While the queue is not empty do the following:

1. Remove an event e_i from the queue.
2. Identify , abstract, and design all events e_j following e_i ;

If e_j is new produced then add e_j to the queue.

More discussion about this *BFS* like process will be given in the next Chapter. Besides the *BFS* like method, there is another way to build an *ED* for a system. The next section shows an alternative *ED* construction method by the example of cruise control system.

3.3 Example 2, Cruise Control System

The *ED* construction method shown in this section is Scenario Architecture, which takes a "normal" scenario as a base, then installs several scenarios on a "normal" scenario to make a net structure. From a requirement document, we make scenarios in textual format first, just as those made in [Wang 91] (p31-34) for PBX on system level or in [Rumbaugh 91] (p171) for ATM machine. Then we pick up a "normal" one from the textual scenarios we just made, map the "normal" scenario to an *ED* form. Next, extend the *ED* by installing the rest of the textual scenarios. Recall that a scenario is a walk in *ED*. In Scenario Architecture, we actually build a single walk for *ED* first.

The stimuli-system partition is still the first step here. This first level 2-partition is the foundation of the event-driven approach.

The example we used in this section comes from the UTA's course CSE 5324 Software Engineering (I) SUMMER 92.

Requirement: A cruise control system must maintain a car's speed within 5 mph of the desired speed, even over varying terrain. The cruise control system's inputs are

1. Cruise control on or off — if on, denotes that the cruise control system should maintain the car's speed.
2. Engine on or off — if on, denotes that the car engine is turned on; the cruise control system is only activated if the engine is on.
3. Pulses from wheel — a pulse is sent for every revolution of the wheel.
4. Accelerator signal — indication of how far the accelerator has been depressed.
5. Brake signal — on when brake is depressed; the cruise control system temporarily reverts to manual control if the brake is depressed.
6. Increase or decrease — increase or decrease the maintained speed; only applicable if the cruise control system is on.

7. Resume speed — resume the last maintained speed; only applicable if the cruise control system is on.

8. Clock pulse — timing pulse every millisecond.

The cruise control system's output is throttle setting — values for the engine throttle setting.

Assumption:

1. We consider that increase or decrease are signals. Each time the driver pushes the increase decrease button, the cruise control system increases or decreases the speed by 5 mph once. The same as that for resume button

2. When brake is released, the cruise control (if it is on previously) goes back to the previous work state.

3. The effect of the accelerator on the cruise control system is the same as that of the brake.

4. The minimum value of the maintained speed is 5 mph. The maximum value of the maintained speed is 110 mph. That is to say that if the driver turns the cruise control on when the car's current speed is only 2 mph, the cruise control will maintain the car's speed at 5 mph. If the car's speed is 140 mph when the cruise control is turned on, the car's speed is maintained at 110 mph.

Step 1, Identify Stimuli: The input *clock pulse* is not a stimulus. With respect to the cruise control system, this *clock pulse* is not spontaneous and random. The *clock pulse* can be viewed as the results of the cruise control system's querying from somewhere.

The stimuli of the cruise control system can be shown as:

Stimuli $\equiv \{e_1', e_2', e_3', e_4', e_5', e_6', e_7', e_8', e_9', e_{10}', e_{11}'\}$ such that

e_1' : engine is on.

e_2' : engine is off.

e_3' : cruise control button is on.

e_4' : *cruise control button is off.*

e_5' : *brake is on*

e_6' : *brake is off*

e_7' : *accelerator is on*

e_8' : *accelerator is off*

e_9' : *increase/decrease signal (Inc/Dec sig).*

e_{10}' : *resume speed signal(RSV sig).*

e_{11}' : *terrain change.*

We can make a further abstraction on *Stimuli* to make things simpler. The cruise control works only if the engine is on and the cruise control button is on. We design the following event to demonstrate this point:

$$CC\ on = e_1' \wedge e_3'$$

By the Demorgan's law, we write the equivalent expression:

$$CC\ off = e_2' \vee e_4'$$

Actually, the *CC on* and *CC off* are the one event with different values. However, for the specific application, we still design two events here. Furthermore, since both the effects of the brake and the accelerator on the cruise control system are related to the manual switch, we have the following design:

$$B/A\ on = e_5' \vee e_7'$$

$$B/A\ off = e_6' \wedge e_8'$$

Other relations in the *Stimuli* are $e_1' \oplus e_2'$, $e_3' \oplus e_4'$, and $e_{11}' \rightarrow e_1'$

We rebuilt *Stimuli* as

$Stimuli \equiv \{e_1', -e_1', e_2', -e_2', e_3', e_4', e_5'\}$ such that

e_1' : *CC on.*

$\neg e_1'$: *CC off.*

e_2' : *B/A on.*

$\neg e_2'$: *BA off.*

e_3' : *increase/decrease signal (Inc/Dec sig).*

e_4' : *resume speed signal (RSV sig).*

e_5' : *terrain change.*

Let $Stimuli \subset E$

Let $SEO = \emptyset$

Step 2, Determine the Boundary State: We make the decision that the cruise control system gets its boundary state only if $\neg e_2'$ is true.

Step 3, Textual Scenarios: (Refer to the Appendix A)

Step 4, The Walk Building for A "Normal" Scenario: Obviously, the first scenario depicted in Appendix A, Cruise Control, seems "longer" than other scenarios. We begin the *ED* building with scenario 1.

(The graphical representation of the scenario 1's *ED* is shown on Figure 5.)

The first event is e_1' , which is already in the *Stimuli*. According the step 3 in the scenario 1, when e_1' happens, the CC system begins sampling the wheel pulses and the clock pulse, then calculates the current speed. We have the following abstractions:

e_6 : *sample whl pulse* (sample wheel pulses, a follower of e_1').

e_7 : *sample clock pulse* (sample clock pulses, a follower of e_1').

e_8 : *calculate C-spd* (calculate current speed, the follower of e_6, e_7).

Let $e_6, e_7, e_8 \in E$

Let $(e_1', e_6), (e_1', e_7), (e_6, e_8), (e_7, e_8) \in SEO$.

Next, according to the step 4 in the scenario 1, the abstractions follow

e_9 : *compare C-spd and M-spd* (compare the current speed and the maintained speed, the follower of e_8).

Let $e_9 \in E$

Let $(e_8, e_9) \in SEO$

However, before the occurrence of e_9 , the CC system must know the maintained speed from somewhere. If we just start the CC system, the first calculated current speed should be recorded somewhere as the maintained speed and should be also stored somewhere as the last maintained speed. The last maintained speed is stored for the resuming.

Therefore the analysis goes as:

If CC system is active, it queries the maintained speed. If the maintained speed is not zero (By the assumption 4, the minimum value of the maintained speed is 5 mph), the next action is to compare the calculated speed with the queried value. If the queried value is zero, we know that the CC system has just started. So the current speed should be recorded as the maintained speed, and at the same time, the current speed should be also stored as the last maintained speed. If a new maintained speed is determined, it should be used in comparison with the current speed. The analysis above is still guided by the question *if this is the last occurrence, what will be the next?*. The events analyzed so far can happen on parallel way with e_6 , e_7 , and e_8 . We have the following abstractions:

e_{10} : *query M-spd (1)* (query the maintained speed, a follower of e_1' so far).

e_{11} : *store L-spd* (store the last maintained speed, a follower of e_{10} and e_8).

e_{12} : *record M-spd* (get the maintained speed ready, a follower of e_{10} and e_8).

Let $e_{10}, e_{11}, e_{12} \in E$

Let $(e_1', e_{10}), (e_{10}, e_9)', (e_{10}, e_{11})', (e_{10}, e_{12})', (e_8, e_{11})'', (e_8, e_{12})'', (e_{12}, e_{10}) \in SEO$.

Note that we do not concern where e_{10} queries M-spd and where e_{11} stores L-spd. Those belong to the software design phase. What we are concerned with is that if some condition is satisfied, e_{11} will follows e_{10} . Similarly, we are not really concerned with the fact that e_{11} may use the data from e_8 . The fact that the e_{11} may use the data from e_8 represents another fact that it is possible for e_{11} to occur after the e_8 .

Still, according to the step 4 in scenario 1, the event *adjusting the car's throttle* follows the e_{11} ; however, it does so conditionally.

e_{13} : *adjust throttle* (adjust the car's throttle, the follower of e_9).

Let $e_{13} \in E$

Let $(e_9, e_{13})' \in SEO$.

According to the step 5 and 6 of the scenario 1, we have

e_{14}'' : *clean L-spdc and M-spdc* (clean the last maintained speed and the maintained speed, a follower of $\neg e_1'$)

Let $e_{14}'' \in E$

Let $(e_{13}, \neg e_1''), (e_{13}, e_{14}'') \in SEO$.

Step 5, The Walk Extension: Now we install scenario 2 on the ED of scenario 1. The first step of scenario 2 says that scenario 2 begins at the occurrence of e_5' . After that, according to step 2, the CC system needs to check if *CC on* is true before querying the maintained speed. If *CC on* is false, the CC system does nothing more. We have the following:

e_{15} : *check CC (1)* (check if CC is on, a follower of e_5').

e_{16} : *query M-spdc (2)* (query the maintained speed by increase or decrease, a follower of e_{15}).

e_{17}'' : *do nothing* (go back to the previous state, a follower of e_{15} so far).

Let $e_{15}, e_{16}, e_{17}'' \in E$

Let $(e_5', e_{15}), (e_{15}, e_{16})', (e_{15}, e_{17}'')' \in SEO$

According to the step 2 and 3 of the scenario 2, we have

e_{18} : *check spdc limit* (check the maintained speed limit, a follower of e_{16}).

e_{19} : *inc/dec M-spdc* (increase/decrease the maintained speed, a follower of e_{18}).

Let $e_{18}, e_{19} \in E$.

Let $(e_{16}, e_{18})', (e_{18}, e_{17}''), (e_{18}, e_{19})', (e_{18}, e_{11})', (e_{19}, e_{12}) \in SEO$.

So far scenario 2 has been connected with scenario 1 through the arcs $(e_{18}, e_{11})'$, (e_{18}, e_{12}) . Next, we will examine both scenarios to see if there are some arcs from the scenario 1 to the scenario 2. We have

$$(e_{12}, e_3)'' , (e_{13}, e_3)'' , (e_{14}, e_3)'' \in \text{SEO}.$$

Now the scenario 2 is completely installed on the scenario 1. The graphical representation of the scenario 2 is shown on Figure 6.

The installation of the scenarios 3, 4, 5 is shown on Figure 7, 8, 9 respectively. Figure 10, 11 are the follow trees. We build these follow trees according to the order of the scenario installation, so we will find that the size of the follower tree of scenario 1 (Figure 10) is larger than the rest. The *event dictionary* of the cruise control system is in Appendix B.

CHAPTER 4

STRUCTURE AND ANALYSIS OF ED

4.1 Definition in Formal Form

An *ED* is a particular kind of directed graph. We use E to indicate the vertex set, SEO to indicate the arc set. E includes two distinct subsets S, V . The vertices in S are the *sources* of *ED* and those in V are the *sinks* of *ED*. S and V are assumed to be disjoint and non empty. SEO is partitioned into three parts, I, C , and T .

Definition:

An *ED* is a 7-tuple, $ED = \{E, SEO, S, V, I, C, T\}$ where

$E = \{e_1, e_2, \dots, e_n\}$ is a finite set of events,

$SEO \subseteq E \times E$ is a finite set of arcs,

$S \subset E, V \subset E$,

$I \subset SEO, C \subset SEO, T \subset SEO$,

$S \neq \emptyset, V \neq \emptyset, S \cap V = \emptyset$,

$I \cap C \cap T = \emptyset, I \cup C \cup T = SEO$.

SEO : Let $e_i, e_j \in E$. If it is possible that the occurrence of e_i immediately follows that of e_j under the state determined by e_j , then $(e_j, e_i) \in SEO$.

S : Let $e_i \in E$. If the occurrence of e_i is spontaneous, then $e_i \in S$. If $e_i \in S$, then $d_{out}(e_i) \geq 1$.

V : Let $e_i \in E$. If the occurrence of e_i has the system state transferred to the boundary state, then $e_i \in V$. If $e_i \in V$, then $d_{out}(e_i) = 0$.

I : Let $(e_i, e_j) \in SEO$. If $e_i \leftrightarrow e_j$ or $e_i \rightarrow e_j$, then $(e_i, e_j) \in I$.

C : Let $(e_i, e_j) \in SEO$. If $e_i \leftarrow e_j$, then $(e_i, e_j) \in C$.

$T: SEO - (I \cup C).$

Theorem 1: $|E| \geq 2.$

Proof: Suppose $E = 1$, then at least one of $S = \emptyset$ and $V = \emptyset$ must be true.

We call a vertex in S an S vertex. Similarly, we have V vertex, I arc, C arc, and T arc.

We take an ED to model the event SEO order in the system executions. The events we abstracted in the system requirement domain are represented with the vertices. The SEO orders defined in Chapter 2 are represented with the arcs. The first SEO is represented with the arcs in I ; the second SEO , in C ; the third, in T . The *Stimuli* described in Chapter 2 is represented with S . The *Sink* described in Chapter 3 is represented with V .

Definition: An S -walk is a *walk* of an ED with the *origin* in S and the *terminus* in V .

Theorem 2: An S -walk is equivalent to a scenario.

Proof: \Rightarrow : Consider an S -walk, say w , in ED . By the definition of a scenario in Chapter 2, w is a scenario.

\Leftarrow : Let s a scenario. By the definition of ED , there is a correspondent sequence, say w , of s in ED ; the *origin* and the *terminus* of this sequence are in S and V respectively. The E does not include repeated nodes; the w is unique.

Corollary: The collection of all S -walks in the ED of a system is the scenarios of the system.

Since we take S to model a *Stimuli*, we can think that S includes a group of subsets, say $S1$, $S2$, etc. If the stimuli corresponding the elements in S within the same subset, those stimuli can not occur at the same time.

The examples of modeling a system (Vending Machine System and Cruise Control System) with ED are illustrated in Chapter 3. The graphical representations of the examples are on Figure 1—10. We represent an S vertex with a circle with small dots

within it, in V with a circle with slice sloped lines within it. The arcs in I are represented with the strong arrowhead lines; the arcs in C , with thin arrowhead lines; the arcs in T , with dashed arrowhead lines. The E , S , V , I , C , and T of the ED for examples are shown on Appendix C.

4.2 The Behavior of ED

An ED is an event network. The occurrence of an event, say e_i , in ED may be caused by different event occurrence. The set of the events that may cause the occurrence of e_i is call the *preceding set* of e_i . e_i may cause the occurrence of several other events. The set of events that may be caused by e_i is call *succeeding set* of e_i . For an event in an ED , some of its preceding events or succeeding events may or may not be required to occur at the same time. We use a group of symbols and symbol expressions to specify the concurrence or non-concurrence within a succeeding set or a preceding set.

An ED can also be viewed as an *event machine*. We use tokens to specify the occurrences of events in an ED . The token behaviors as well as the input and output of arcs assigned for each vertex are specified in the *Event Dictionary*, which is viewed as a component of an ED .

4.2.1 The Concurrence within Preceding Set and Succeeding Set

With respect to an event, some of its preceding events are concurrent, and some are not. For example on Figure 3, e_5 and e_6 are the succeeding set of e_3 and within the preceding set of e_1' ; e_5 and e_6 are not concurrent. On Figure 5, e_6 and e_7 are the preceding set of e_8 and within the succeeding set of e_1' ; e_6 and e_7 are required to be concurrent.

Within the preceding set or the succeeding set of an event, we specify the concurrence or non-occurrence among the elements with a group of symbols. The *connector* $\{\wedge, \vee, \oplus\}$ specifies the concurrence or non-concurrence of events. The *arc indicator* $\{\leftrightarrow, \rightarrow, \leftarrow, \dashv\}$ specifies the arc types. An expression is consists of several events connected with several connectors and an arc indicator. Either left hand side or right hand side of the arc indicator is a single event. The examples are:

$$(e_i \wedge e_j) \vee e_k \dashv e_l$$

$$e_i \leftarrow e_j \oplus e_k \oplus e_l$$

If the single event is on the right hand side, the left hand side is the preceding set. If the single event is on the left hand side, the right hand side is succeeding. We use the symbol $E\{\wedge, \vee, \oplus\}$ to represent an expression with events as the literal and $\{\wedge, \vee, \oplus\}$ as the connectors. We assign the meaning to the symbols as:

$E\{\wedge\}$ means that the events in the expression are required to occur at the same time.

$E\{\oplus\}$ means that the events in the expression can not occur at the same time.

$E\{\vee\}$ means that the events in the expression may or may not occur at the same time.

Let $e_i, e_j \in E$.

$e_i \leftrightarrow e_j$ means that $(e_i, e_j) \in I$ and e_j occurs iff e_i occurs.

$e_i \rightarrow e_j$ means that $(e_i, e_j) \in I$ and e_j occurs only if e_i occurs.

$e_i \leftarrow e_j$ means that $(e_i, e_j) \in C$.

$e_i \dashv e_j$ means that $(e_i, e_j) \in T$.

The formal expression of symbols is on Appendix B.1.1

4.2.2 Using Tokens to Specify the Occurrence Wave

At a point of time, we simulate a *Stimuli* by assigning a group of *token creations* on some elements in S . We call that *one assignment*. For each *assignment*, we assume the two regulations:

1. For each exclusive subset, only one element is allowed to create a token.
2. Exactly one element in a conflict pair should be assigned a token.

For the example of the cruise control system, in *one assignment*, only one of e_1' and $\neg e_1'$ is allowed to create a token and at least one of them has to create a token.

We assign each vertex in E the ability to create a token at one time. However only the vertices in S can create a token by an assignment. The token creations in S stir other vertices to create tokens. Therefore we say that an *assignment* can cause a *token creation wave* in an ED network.

A token creation wave is propagated along path's arcs. This wave can only be absorbed in V . For example in Figure 3, we let e_1' create a token, then e_3 and e_6 create tokens, then perhaps the e_6 creates a token, then maybe e_8 , then e_{11} , and then e_{10}'' . The event e_{10}'' is a sink that expires the creation wave.

Some S vertices stop creating tokens when their exclusive partners begin creating tokens. For the example in Figure 3, if e_3 and e_4 create tokens, the token in e_1' is removed. In this situation, it seems that the tokens created in S travel in the ED network. Some other S vertices stop creating tokens when their exclusive partners begin creating tokens. For example on Figure 5, if we let e_1' create a token, e_1' will continue its creation until we let $\neg e_1'$ create a token. In this situation, the stimulus seems to *hold* its token once it has created one, but we prefer that the vertex continues to create tokens. What S vertices should be designed with the *holding* ability depends on the applications. For the cruise control example in the last Chapter, we assign e_1' , e_2' , and $\neg e_2'$ with the *holding* ability.

4.2.3 The Input and Output of the Nodes

A vertex, for example e_i in E , may be stirred up by its preceding vertices to create a token. The creation of this token will stir other token creations in succeeding vertices. The I input and output can be determined by an ED structure. The C and T inputs and outputs are determined by the semantic of applications. All inputs and outputs are specified in the *Event Dictionary* associated with an ED .

Suppose that the input and output arcs of e_i include all three kinds of SEO arcs, I , C , and T . If any one of e_i 's I tails have created a token, e_i creates a token; if e_i has created a token, all of e_i 's I heads create tokens. In other words, between the I input arcs, the logic relationship is OR; between the I output arcs the logic relation is AND. Therefore, the I input/output is a logic. For the example of the vending machine system in Figure 3, the I input logic can be found on e_{15} that creates a token only if either e_{14} or e_{13} creates a token; the I output logic can be found on e_{16} . The token creation on e_{16} will cause both e_{16} and e_{19} to create tokens, respectively.

Therefore by the I input/output logic, if an S vertex is designed to be of the *holding* ability, the token creation of this vertex will cause a sub- ED connected with I arcs to allow all of its vertices to hold their tokens; in this case the sub- ED is said to be *saturated*. For the example of the cruise control system of Figure 5, if $e_{1'}$ continues its token creation, the sub- $ED = \{e_{1'}, e_6, e_7, e_8, e_9, e_{10}\}$ is in saturation. A saturated sub- ED corresponds to a state of the ED and the events saturated in the sub- ED become some processes or activities of the system the ED modeled.

The I input and output logic of an ED comes directly from the logic defined in the first SEO in which the head occurs only if or iff the tail occurs. The I input and output logic can be determined completely by token creations.

The specification of I input of e_i can be $E\{\wedge, \vee, \oplus\} \rightarrow e_i$ or $E\{\wedge, \vee, \oplus\} \leftrightarrow e_i$. Not that the expression of $e_i \oplus e_j \rightarrow e_k$ does not mean that if e_i and e_j occur at the same time

the e_k will not occur; this merely shows that it is not possible for e_i and e_j to occur at the same time. So $e_i \oplus e_j \rightarrow e_k$, $e_i \wedge e_j \rightarrow e_k$, and $e_i \vee e_j \rightarrow e_k$ mean the same / input logic. The specification of / output can be $e_i \rightarrow E\{\wedge\}$ or $e_i \leftrightarrow E\{\wedge\}$. There is no application of the expressions like $e_i \rightarrow E\{\oplus, \vee\}$, etc.

The input and output for C arcs cannot be completely determined by token creation because the logic between the tail and the head, as defined in the second SEO , is such that the occurrence of the head implies the occurrence of the tail. For a C arc, the token creation on a tail only means a probability that the C arc is *active* even if this C arc is the only output of the vertex. For the example in Figure 5, the arc (e_9, e_{13}) is the only output of e_9 . However, the fact that the cruise control system is comparing the current speed and the maintained speed (e_9) does not mean that the throttle is being controlled by the cruise control system (e_{13}); the throttle may be in manual control even if the cruise control system is still running.

A C arc may remain active if this C arc comes from a saturated sub- ED . Suppose that A_1 and A_2 are two C arcs coming out from e_i . There are three cases of activity logic of C arcs. One is that A_1 and A_2 are active together. An example is Figure 5 in which (e_{10}, e_{11}) and (e_{10}, e_{12}) are active simultaneously. We use the symbol $e_{10} \leftarrow e_{11} \wedge e_{12}$ to specify this case. The next case is that A_1 and A_2 cannot be active together; only one of them is active at one time. An example is Figure 6 in which either (e_{15}, e_{16}) or (e_{15}, e_{17}) is active. We use the symbol $e_{15} \leftarrow e_{16} \oplus e_{17}$ to specify the second case. The last case is that A_1 and A_2 may be active together or may not; whichever the case is does not matter. An example is Figure 3, where it is possible that e_{16} and e_{17} occur together; (e_{15}, e_{16}) and (e_{15}, e_{17}) may or may not be active together. We use the symbol $e_{15} \leftarrow e_{16} \vee e_{17}$ to specify the last case. Generally, we use $e_i \leftarrow E\{\wedge, \vee, \oplus\}$ to represent the C output assigned for e_i . Note that a C arc in active does not mean the head will create a token unless that the C arc is the only input C arc of the head. On Figure 10, (e_9, e_{13})

in active does not mean that e_{13} will create a token. However, in Figure 3, (e_9, e_{13}) is the only C input of e_{13} , so (e_9, e_{13}) in active means that e_{13} will create a token.

The C input of a vertex is similar to the C output. The C input of e_i is specified with $E\{\wedge, \vee, \oplus\} \leftarrow e_i$. An expression as $e_j \wedge e_k \leftarrow e_i$ means that e_i may create a token if both (e_j, e_i) and (e_k, e_i) are active. For the example in Figure 10, e_{13} keeps creating tokens if both (e_9, e_{13}) and (e_2', e_{13}) stay active. An expression as $e_j \vee e_k \leftarrow e_i$ means that either (e_j, e_i) in active or (e_k, e_i) in active may stir e_i to create a token, and that the occurrences of e_j and e_k may or may not be simultaneous. For example, a wet backyard means that it is raining, that the sprinkler is on, or that it is both raining and sprinkling at the same time. An expression as $e_j \oplus e_k \leftarrow e_i$ means that either (e_j, e_i) in active or (e_k, e_i) in active may stir e_i to create a token and that e_j and e_k cannot occur at the same time. An example is the game that A, B, and C throw two coins. The regulation is such that if A gets exactly one head, D wins; if B gets two heads, D wins. So if D wins, this means that either A was throwing or B was throwing, but A and B cannot throw coins at the same time.

The T arcs don't stir the token creation between the tails or the heads. The token creation on a T head removes the token from the T tails if some T tails have tokens. The expression $E\{\wedge, \vee, \oplus\} \dashv e_i$ means that e_i may remove the tokens on the events in $E\{\wedge, \vee, \oplus\}$ and that there are some $\{\wedge, \vee, \oplus\}$ relationship among those events. The expression $e_i \dashv E\{\wedge, \vee, \oplus\}$ means that the token on e_i may be removed by the events in $E\{\wedge, \vee, \oplus\}$ and that there are some $\{\wedge, \vee, \oplus\}$ relationship among those events.

The concurrent relation among I, C, T arcs with respect to a vertex is \vee . For the example in Figure 3, e_{12} and e_{13} may or may not occur together; $\{e_9 \wedge e_{10}'\}$ and e_2' may or may not occur together.

4.2.4 The *Event Dictionary*

If we apply the *SEO* abstraction and develop an *ED* from a system in the real world, an associated *event dictionary* is necessary. An event dictionary answers the following questions:

What are the contents of the vertices represented in an ED with respect to the system modeled?

What is the value of the events?

How does the value of an event, if any, determine the C input and output?

In Figure 3, for example, if e_7 gets the value *coin is enough or no deposit*, the *C I/O* is applied; if the e_7 's value is *coin is not enough*, then the *T I/O* is applied

An event dictionary is an organized listing of all the events in *E* that have been abstracted from a system; the event contexts are precisely described. Both the users and the system analysts will have a common understanding about what an *ED* means with respect to a system, and both can execute the system on an *ED* in advance.

The examples of the event dictionaries associated with the vending machine system *ED* and the cruise control system *ED* are in Appendix B. In Appendix B, the format of an event dictionary is also suggested.

4.3 Modeling

We abstract the events from a system into the vertices of an *ED*. By each occurrence, we may abstract all possible immediate followers. The relationship of an event with its immediate followers is modeled as the arcs in an *ED*. This abstraction depends highly on the individual analysts; therefore, the result is non deterministic. However, if there are some users taking part in this development process, the outcome will be less non deterministic.

In the last chapter, using the vending machine and the cruise control system examples, we introduced two methods with which to model a system into an *ED*. The first method mimics the Breadth First Search Algorithm (*BFS*). "Breadth First Search is so named because it expands the frontier between the discovered and the undiscovered nodes uniformly across the breadth frontier." [Thomas 90] (p469). For an *ED*, we expand the frontier between abstracted events and the events that will be abstracted across the breadth frontier. The question *if this event is the last occurrence in the system, what will be the next?* is the key guideline in our expansiveness. The second method installs the scenarios together to form an *ED*.

Sometimes, some immediate followers of an event depend greatly on the states. In other words, an event may possibly occur under several states. These states may cause different succeeding sets of the event. In this case, the analyst design the event *check state* in *ED*; the different states become the values of the event *check state*. The e_3 and e_7 in Figure 3, e_{15} and e_{18} in Figure 6, e_{20} in Figure 7, and e_{24} in Figure 9 are the examples.

The following are the general steps applied to the mechanism of the frontier breadth expansiveness:

1. Establish a *Stimuli*
2. Let $E = \text{Stimuli}$
3. Initialize a queue, say Q , with a "normal" sequence of the vertices in E .
4. While Q is not empty, do
 - a. Remove a vertex v from Q .
 - b. For each abstracted follower w of v do the following:
 - i. If w has no succeeding set, do

$$w \in E.$$

Add w into Q .

ii. let $(v, w) \in SEO$.

The vending machine system in the last Chapter is an example of the application of the *BFS*-like approach.

The cruise control system in the last chapter is an example of an alternative way to model a system with an *ED*. This method actually transfers the scenarios of a system from textual form to graphical form. We have the following steps as the guide:

1. Identify the *Stimuli*
2. Define a boundary state.
3. For each stimulus, write a textual scenario.
4. Select a "normal" scenario, do the following
 - a. For each event v in the scenario event sequence
 - If $v \notin E$, let $v \in E$.
 - b. For each ordered pair (v, w) in the scenario event sequence
 - If $(v, w) \notin SEO$, let $(v, w) \in SEO$.
5. For the remaining textual scenarios, do 4.

4.4 The *Follow Trees*

We introduce follower trees of an *ED* to emphasize the development *follow* feature and to have a chaotic *ED* organized using the hierarchical representation form. *Follow tree* that is a mechanism for braking a large *ED* down into a series of *pages* make the software analysis document more easily understood.

4.4.1 The Definition and the Algorithm

A *follow tree* is a copied hierarchical structure produced in an iteration of the following algorithm:

1. For each node, say r , in *Stimuli* do the following
 2. Copy r ;
 3. Initialize the queue, say Q , with r ;
 4. While Q is not empty do
 5. Remove Q , say v ;
 6. For each adjacent head, say h , of v , do the following
 7. Copy h as a child of v ;
 8. If $h \notin S$ and $h \notin V$ and h is not colored, do the following
 9. Color h in ED ;
 10. Add h into Q ;

Note that the *follow trees* are not mathematical trees instead of a documentation form with the tree structures.

A *follow tree* is of a tree structure.

Proof: No operation in the algorithm produces an arc connecting two copied structures. Consider the i -th iteration of the Algorithm. The Copy operation of step 2 creates a node, and each Copy operation of the step 7 creates a node, as well as an edge connected with a parent node. Therefore, we have a connected graph, say (FE, FA) , that $|FA| = |FE| - 1$.

A group of *follow trees* produced from the same ED has the following features.

1. A node in E , say e_i , corresponds $d_{\text{Bin}}(e_i)$ number nodes in the *follow tree* group. Exactly one of those nodes corresponded by e_i has children; the rest are leaves.

2. A root belongs to S .

Proof 1: Step 6 and 7 in the Algorithm say that each arc in ED corresponds to a head copy. The e_i will be copied $d_{\text{in}}(e_i)$ times.

The first step of the Algorithm adds each S vertex into Q exactly one time. The color mechanism of the Algorithm adds each immediate vertex in ED exactly one time in Q . Only one vertex is removed from Q each time, then each vertex has only one chance to have children.

Proof 2: It is obvious from the first step in the Algorithm.

Figure 4 shows an example of a group of *follow trees* produced from the ED in Figure 1. In this example, e_2' is the first taken as a root.

4.4.2 The *Pages of the Follower Trees for an ED*

If we come across an ED with several hundreds of events, it is sometimes necessary to group the vertices into *page* size for the sack of documentation. A page is just a notational convenience, not a logical construction.

The size of a page is the number of vertices indicated in a sub *follow tree*. We take George Miller's magical number "the magical number seven, plus or minus two" [Miller 56] to be the reference number of the vertices in a page. An example of paging a group of *follow trees* is shown by Figures 12, 13, 14, 15, 16, and 17.

4.4.3 The $E\{\wedge\} \leftarrow e_i$ in *Follow Trees*

For our convenience, we make the following rule: for each node in the follower tree, say e_i , if e_i has more than one C inputs with the " \wedge " relation each other, we list all those nodes as e_i 's parents. For example, on figure 10, e_{13} has two parents, e_9 and e_2' . There is no harm done in maintaining the tree's property by this rule.

4.5 Scenario Language

4.5.1 Scenario Language Definition

The scenario grammar is a regular grammar [Sudkamp 88] (p57), say

$$G_s = (V, \Sigma, P, \text{Scnr})$$

V : Finite set of variables;

Σ : Finite set of terminal symbols which maps E of ED .

Scnr : Distinguished element of V of G_s which implies the name *Scenario* and starts derivations;

P : Finite set of rules which is the form

$$A \rightarrow aB$$

$$A \rightarrow \lambda.$$

The scenario language is defined as:

$$L_s = \{w \mid \text{Scnr} \bullet \Rightarrow w\},$$

(w : Sentence;

$\bullet \Rightarrow$: The derivation utilizes the rules of G_s).

4.5.2 Establishing an G_s from an ED

Consider an ED . We execute the following algorithm:

Let $\Sigma = E$

Use Scnr to name a S -Walk in the ED .

Create V as:

1. $V = \emptyset$;
2. For each $e_i \in \Sigma$, create a variable A_i , and let $A_i \in V$

Create P as:

1. For each $e_i \in \Sigma$
2. If e_i corresponds an element in *Stimuli*,
3. let $Scnr \rightarrow e_i A_i \in P$;
4. Else if e_i corresponds an element in *Sink*,
5. let $A_i \rightarrow \lambda \in P$;
6. Else for each $(e_i, e_j) \in SEO$
7. let $A_i \rightarrow e_j A_j \in P$.

All *S-Walks* in an *ED* can be derived by *Gs*. In other words, all scenarios of the system can be derived by *Gs*.

Proof: Consider a *S-Walk* = $e_1 e_2 \dots e_n$. $e_1 \in \text{Stimuli}$, so we apply $Scnr \rightarrow e_1 A_1$. $(e_1, e_2) \in SEO$, so we apply $A_1 \rightarrow e_2 A_2$. Suppose we have already applied the rules in *P* i times and have derived the string $e_1 e_2 \dots e_i$, we apply $A_i \rightarrow e_{i+1} A_{i+1}$ because $(e_i, e_{i+1}) \in SEO$. At the end of the string, we apply $A_{n-1} \rightarrow e_n A_n$ and $A_n \rightarrow \lambda$. Note that $e_n \in \text{Sink}$.

A *Gs* only derives the *S-Walks* of the *ED*.

Proof: Consider a string $Str = e_1 e_2 \dots e_n$ which is not a *S-Walk* of an *ED*.

Case 1: Some terminal symbols in *Str* do not belong *E*. Let $e_i \notin E$, then $e_i \notin \Sigma$. *Gs* can not derives a string with the terminal symbol e_i included.

Case 2: Some concatenations in *Str* have no correspondence to the elements in *SEO*. Let $(e_i, e_j) \notin SEO$. Let $e_i e_j$ is included in *Str*, then $A_i \rightarrow e_j A_j$ must be applied. However, $A_i \rightarrow e_j A_j \notin P$.

Case 3: $e_1 \notin \text{Stimuli}$, then we can not start the derivation for *Str* with *Gs*.

Case 4: $e_n \notin \text{Sink}$, then we can not eliminate the variable in the sentential form $e_1 e_2 \dots e_n A_n$ with *Gs*.

4.6 Hierarchical Expansion of an *ED*

In a certain specific application, we may refine some events into the atomicity processes. For example in Figure 1, we may refine e_9 as

e_{9-1} : *motor connects the screw for the i -th bar type;*

e_{9-2} : *motor starts running;*

e_{9-3} : *open the i -th type door;*

e_{9-4} : *i -th type bar comes out;*

e_{9-5} : *signal e_{12} ;*

e_{9-6} : *decrease # i type by 1*

as well as *SEO* abstraction as

$(e_7, e_{9-1})'$, (e_{9-1}, e_{9-2}) , (e_{9-1}, e_{9-3}) , $(e_{9-2}, e_{9-4})'$, $(e_{9-3}, e_{9-4})'$, (e_{9-4}, e_{9-5}) , (e_{9-4}, e_{9-6}) , (e_{9-5}, e_{12}) , (e_{9-6}, e_{12}) , $(e_{9-6}, e_{13})'$. The illustration is on Figure 20.

4.6.1 Sub-Extension and Subgraph-Contraction

Definition: Let $G(E, SEO)$ be a graph. A *sub-extension* of G is a graph that can be obtained from G by the following:

- 1). Let $t = 1$, $n \leq |E|$
- 2). While $t \leq n$, do
 - a). Let $G_t'(E_t', SEO_t')$ be a graph. Let $e_t \in E$;
 - b). $E = E \cup E_t' - \{e_t\}$ that $e_t \notin E_t'$;
 - c). Replace each input arc of e_t , say (e_i, e_t) , with one or more arcs, say $(e_i, e_{G_k}), (e_i, e_{G_{k+1}}), \dots$ that $e_{G_k}, e_{G_{k+1}} \dots \in E_t'$;
 - d). Replace each output arc of e_t , say (e_t, e_j) , with one or more arcs, say $(e_{G_l}, e_t), (e_{G_{l+1}}, e_t), \dots$ that $e_{G_l}, e_{G_{l+1}} \dots \in E_t'$;
 - e). $t = t + 1$.

With respect of a *sub-extension*, G is called *subgraph-contraction*.

Note that a *subgraph-contraction* must correspond a *sub-extension*. Only if a *sub-extension* operation be applied, can a *subgraph-contraction* operation be applied.

Definition: The lower-level expansion of an ED is the *sub-extension* of the ED that I , C , and T arcs are replaced with I , C , and T arcs respectively.

4.6.2 Establish a Sub-Extension Grammar from the G_s

We can establish a sub-extension grammar $G_{ss} = \{V_s, \Sigma, P_s, Scnr\}$ from G_s by

1). Extend an $ED = (E, SEO)$ with $(Et_1, SEOt_1), (Et_2, SEOt_2), \dots, (Et_n, SEOt_n)$;

2). Let $E' = Et_1' \cup Et_2' \cup \dots \cup Et_n'$;

3). $\Sigma = \Sigma \cup E' - \{e_t \mid e_t \in E\}$;

4). $V_s = V \cup \{A_i \mid A_i \text{ has one-to-one correspondence with an elements in } E'\} - \{A_j \mid A_j \text{ has one-to-one correspondence with an elements in } \{e_t \mid e_t \in E\}\}$;

5). Let $P_s = P$;

6). For each rule, say r , in P_s , if there is no r 's correspondence arc in SEO ,

then

$$P_s = P_s - \{r\};$$

7). For each arc, say seo , in SEO , if there is no seo 's correspondenc in P_s ,

then

a). create a rule, say r , according to the algorithm in 4.5.2;

b). $P_s = P_s + \{r\}$.

We can use the same proof methods in 4.5.2 to prove that all scenarios in a sub-extension can be derived by G_{ss} , while a G_{ss} can only derive the terminal strings which correspond the elements in the S-Walks of a sub-extension. Obviously, a G_{ss} is a extended scenario grammar.

CHAPTER 5
COMPARISON WITH OTHER GRAPHICAL REPRESENTATION
OF REQUIREMENT ANALYSIS

5.1 Comparison with Petri Nets

A Petri Net is also a particular kind of directed graph [Tadao 89], [Peterson 81], [Wolfgang 92], and [Joanthan 92] with bipartite structure of *places* and *transitions*. An *ED* is not necessary to be a bipartite directed graph.

In modeling, Petri Net uses bipartite abstraction on the real world. The structure of *Input Place* → *Transition* → *Output Place* is often interpreted as ([Tadao 89])

Precondition → *Event* → *Postcondition*

Input Data → *Computation* → *Output Data*

Input Signals → *Signal Processor* → *Output Signals*

Resources needed → *Task or Job* → *Resources Released*

Conditions → *Clause in Logic* → *Conclusion(s)*

Buffers → *Processor* → *Buffers*

Etc.

A place in a Petri Net cannot be *Input data*, as well as a *Condition*, as well as a *Signal*, etc., at the same time. If a place fires several tokens simultaneously, it is not possible that some tokens go to *Output Data* by *Computing*, some go to *Conclusion(s)* by *Clause in Logic*, and some go to *State by Occurrences*. If it is necessary to model all those aspects for a system, several Petri Nets on different abstraction levels are necessary. However, in the Object-Oriented world, an event may change a state for the first object, send a message to stir an operation in the second object, and also be

an input data for a process of the third object, all at the same time. A Petri Net has difficulty describing such diverse situation at the same time because the bipartite abstraction implies a kind of mutual constraint, or mutual dependence between the two partitioned parts. That is to say that transitions could not exist if there were not places; without places, the transitions lose their significance. If we abstract *Buffer* from the real world, we have to take *Processor* or some other related abstraction as the bipartite partner for *Processor*, we can not take *Clause in Logic* as the bipartite partner of *Buffer*. Furthermore, the semantic relationship between two bipartite parts has to be kept consistent throughout a whole Petri Net.

The foundation of an *ED* modeling is time (a scenario, a sequence of events with respect to an observer). An *ED* only considers the *immediate following* relation. Regardless of whatever the input data, the buffer, the computation, the procedure, the operation, the job, the condition, the clause in logic, the message passing, etc., we can model them as the vertices in an *ED* only if we perceive them on a point in time or only if they are of time point attributes.

It is a straightforward approach to make the transformation from the *I* input/output logic to Petri Net structures. This is illustrated by Figure 19. However, the transformation from the *C* input/output or the *T* arc behavior to some Petri Net structure is complex. This complexity is due to the " \vee " relation in preceding or succeeding sets of a node in an *ED*. In Figure 1, for example, we have $e_{15} \dashv e_1' \vee e_2'$. This means that sometimes e_1' and e_2' occur simultaneously after e_{15} , and sometimes they do not. The corresponding Petri Net structure is then required to have its tokens going two places simultaneously as well as alternatively. Whether it is possible to make the transformation from an *ED* to a Petri Net is reserved for future work.

5.2 Comparison with Event Trace Diagram

There are discussions about the Timing Diagram in [Booch 91] (p173-174) and the Event Trace in [Rumbaugh 91] (p86-87). Both are event graphical representations of objects developed in Static Object Models.

An event trace diagram corresponds with a *walk* of an *ED*, because for each event in an Event Trace or in a Timing Diagram, only one possible follower is specified. An Event Trace or a Time Diagram is a 1-D time line; an *ED* represents all possible event sequences for a system.

An event trace diagram is developed on the basis of a static object model; an *ED* is developed on the basis of scenario using the methods of frontier expansion and scenario installation.

5.3 Comparison with DFD

Both *ED* and DFD make 2-partition abstractions on their top level. We have the following structure: Inputs → System → Output for a 0-level DFD, *Stimuli* → System for the first step of an *ED* modeling. A *Stimuli* does not consider the output data of a system. In an *ED* model, the response of a system is viewed as a part of the system's behavior. Not all input data belongs to *Stimuli*; input data that is not spontaneous or random to the system does not belong to *Stimuli*.

In a DFD, there are not only the note representative *processes* but also other notations to represent such things as *actor* and *data store* that are different from *processes* [Rumbaugh 91] p(124). Processes and data stores are the computer oriented terms, which prevent users from taking an active part in a system specification phase. An *ED* has only one kind vertex of *event* represented. An event can be

abstracted not only as a starting (or an ending) process but also as a getting into a state, a message passing, updating a data storage, etc.

An arc in a DFD conveys data. In an *ED*, an arc represents the time relationship between two occurrences. An *ED* does not consider to where the data goes and from where the data comes. With respect to a user, the problem of to where the data goes and from where the data comes belongs to the *how* scope instead of the *what* scope.

5.4 Comparison with STD

A STD consists of a finite collection of states and the finite collection of connections among the states. Each arc is associated with a label to describe the event that causes the state transition. This model can also be specified with the formal graphical specification. However, in modeling there are following disadvantages compared with *ED*.

1. Not every user is of finite state machine concept, which is not good for the communication between the users and the analysts.

2. The abstraction of states is complex to that of events. If we went to identify a state, we need to consider the features for every element in a system or changes for every attribute of an object. However, if we went to identify an event, it is sufficient for us to take care the instantaneous behavior of only one element in a system or only one attribute of an object.

3. Suppose we apply the *frontier expansion mechanism* to model a STD. The question around which both the user and the analyst go should be: "If the system under this state, what events are possible or inevitable?". At the beginning step, corresponding to a node in STD, the state in analyst's brain is hard to keep consistence with that in the user's brain because only if the analyst completely understands the whole system's behavior, can he figure out each state.

Unlike STD that takes the finite state machine base, *ED* takes the scenario event base. If different states cause different followers for an event, analysts design an event *check state*. The analysts only abstract these states as the values of the event *check state* without taking care about what users mean by those states first.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The goal of this thesis was to solve the ad-hoc situation of scenario building, which inevitably requires the formal representation and the rigorous definition of scenarios.

In [Rumbaugh 91] (p462), we have the definition of scenarios for Rumbaugh's dynamic modeling as *a sequence of events that occur during one particular execution of a system*. We extended this definition to general cases with the *SEO*, the *Stimuli*, and the boundary state. *SEO* assigns the essence to scenarios, in that each event pair in the event sequence must either be of some causality (1st *SEO*, 2nd *SEO*) or be determined (*rehearsed*) under certain situation (3rd *SEO*) by the users/domain experts. *SEO* states that not every ordered event pair can be the source material of a story, and different situations provide the different possibilities. The boundary state and the *Stimuli* provide the limit on a scenario; as a result, scenarios must carry something through to the end; just like a drama needs the rise of the curtain at beginning and the curtain call at the end. The *SEO*, the *Stimuli* and the boundary state have scenarios well defined. A scenario is defined as an event sequence that each event order in the sequence belongs to *SEO*; the start event must be in *Stimuli* and the end event has the system state transferred to the boundary state.

From the observations of the scenario model building for the Vending Machine System and the Cruise Control System, it is concluded that *SEO* provided the foundation with the question of *if this is the last occurrence, what will be the next?* to carry the frontier expansion mechanism. The advantage of this is that users and

developers can work together to specify the behavior of a system; each frontier expansion of an *ED* can be "determined" by users.

All events and *SEO* in a system construct a directed graph (*ED*); a graph is a formal structure. We have proved that a scenario corresponds to an *S-walk* in an *ED* and established the correspondence between scenarios and the collection of all *S-walks* of an *ED*. Therefore, the scenarios of a system are formally represented within one structure. Therefore, the forthcoming conclusion is that an *ED* model, in the entire life cycle of a software system, inherits all the advantages of scenarios. From the observation of the *ED* of the Vending Machine System and the Cruise Control System, it is also concluded that an *ED* specifies the behavior of scenarios concurrently.

ED provides that the occurrences of the events in the scenarios of a system are specified by the token propagation wave and that the behaviors of the tokens are described with input and output. Therefore an *ED* can be a machine or a prototype. The advantage is that developers or even users can execute the specified software in advance. Upon the observation of the *ED* of the Vending Machine System and the *ED* of the Cruise Control System, we know that we can "insert" coins into the Vending Machine *ED* and then "push" the select button to see if the "dispensing bar" happens, or we know that we may "push" the resume button, "push" the inc/dec button, "depress" the break, etc., on the Cruise Control *ED* to see if the expected happens.

Another formal method development for scenarios in this thesis is the scenario language. A scenario language is a collection of terminal symbol strings that are derived from the scenario grammar. The scenario grammar is a regular grammar that is inferred directly from an *ED*. We also have proved that the all scenarios for a system can be derived by the scenario language and the scenario language can only derive the scenario of the system.

The future work concerning *ED* scenario modeling is to provide a full development environment with automatic tools to support *ED* scenario modeling. These tools are needed to maintain consistency of design documents such as *ED* matrix, *ED* graphical representation, Paged *Follow Trees*, *Event Dictionary* and to retrieve information quickly from the design documents.

One more future work is to develop a Static Object Model from an *ED*. The following is an attempt.

Fact 1: So far no proof says that only from a textual requirement document can an Object Model be established.

Fact 2: So far no proof says that the best way to develop an Object Model is to started with the textual requirement document.

Observation: The object identification is an art. "The identification of classes and object is the hardest part of object-oriented design. Our experience shows that identification involves both discovery and invention." [Booch 91] (p133). There are two reasons. The first is that the two essential themes, the *encapsulation* and the *inheritance*, don't support the object identification directly in object-oriented design. The *encapsulation*, separating the external aspects from the internal implementation in detail, is the mechanism that is better to be thought as designed, invented, and constructed for an object than to be thought as identified, or discovered. The *inheritance* deals with a kind of relationship among objects. It implies that if we have already gotten a group of objects, we can consider their commonality. The second reason is that we define objects in isolated way ("We define an object as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand." [Rumbaugh 91] (p21)); however when identifying them, we have to let them interact each other ("An object has state, exhibits some well-defined behavior, and has a unique identity" [Booch 91] p(77). The same philosophy can be summed up in the

phase, *only if he is dancing, can we say he is a dancer*). To execute something in order to identify something is a dilemma. We solve this dilemma with scenario models (*ED*). The goal of developing an Object Model from an *ED* is to lessen the difficulty of object identification.

Assumption: An event in *ED* occurs either on an object or among objects. "Each event transmits information from one object to another" [Rumbaugh 91] (p86). Actually, when we make the Stimuli — System partition, we have already implied that there are two objects in the top level abstraction in an *ED* approach. Both Stimuli and System are viewed as collections of events; the objects that will be developed from an *ED* are also first viewed as collections of events. The encapsulation property and the inheritance relationship can be invented or built afterward.

Method: (The example is on Figure 19 of Vending Machine System. The notation comes from [Coad & Yourdon 92])

1. Construct objects by grouping the events in *ED* according to the principles such as coupling, cohesion [Booch 91] p(124), and inheritance. The inheritance can be applied not only among objects but also among events, behavior, etc. "When we classify, we seek to group things that have a common structure or exhibit a common behavior" [Booch 91] (p133). Here Booch's *things* are events.

(After the step 1, we get a new directed graph: $OD = (O, M)$ (see Figure 19). O is the collection of vertices. Each vertex in O corresponds to an event group that has just been approached. If an arc in *SEO* is totally in a group, contract the arc. For example: Figure 19, (e_1', e_3) has been contracted in *coin mechanism*; M is the collection of arcs which includes the all rest first *SEO* and second *SEO*. If an event is grouped into two groups, we add the arc (actor, passive) between the two groups. For example in Figure 19, both *customer* and *coin mechanism* have e_1' , so we have the message passing from *customer* to *coin mechanism* to convey the arc (actor, passive) caused by e_1' .

This situation corresponds with the *mutual effect* case discussed on Chapter 2. Note that if an event causes an (actor, passive) arc, the inputs of the event in *ED* are considered on the actor side, while the outputs are considered on the passive side. For example, in Figure 19, consider e_{18} in *truck* and *candy mechanism*. The input of e_{18} , (e_{16} , e_{18}), is conveyed in the message passing from *Modem* to *truck*; the output of e_{18} , (e_{18} , e_{10}), is conveyed in the message passing from *candy mechanism* to *coin mechanism*.)

2. Within each event group, map the events into operations or services. "The operations in the object model correspond to events in the dynamic model and functions in the functional model" [Rumbaugh 91] (p18).

3. For each group, build data structures with the reference on the behavior of the services. Name each group with an object name. Behaviors have to be determined by requirement documents, while structures can be somehow constructed or invented by the developers themselves.

(From now on, we already have an object model with each event group as an object and arcs in *M* as message passing representation.)

4. For each node in *O*, develop the aggregation and generalization relationships if it is necessary. (For example, in Figure 19, we may further develop objects as *coin state*, *shut down* as the parts of *coin mechanism*).

5. Cancel the redundant multiple arcs between any two objects. This means that we use a message passing notation to represent more than one of the arcs.

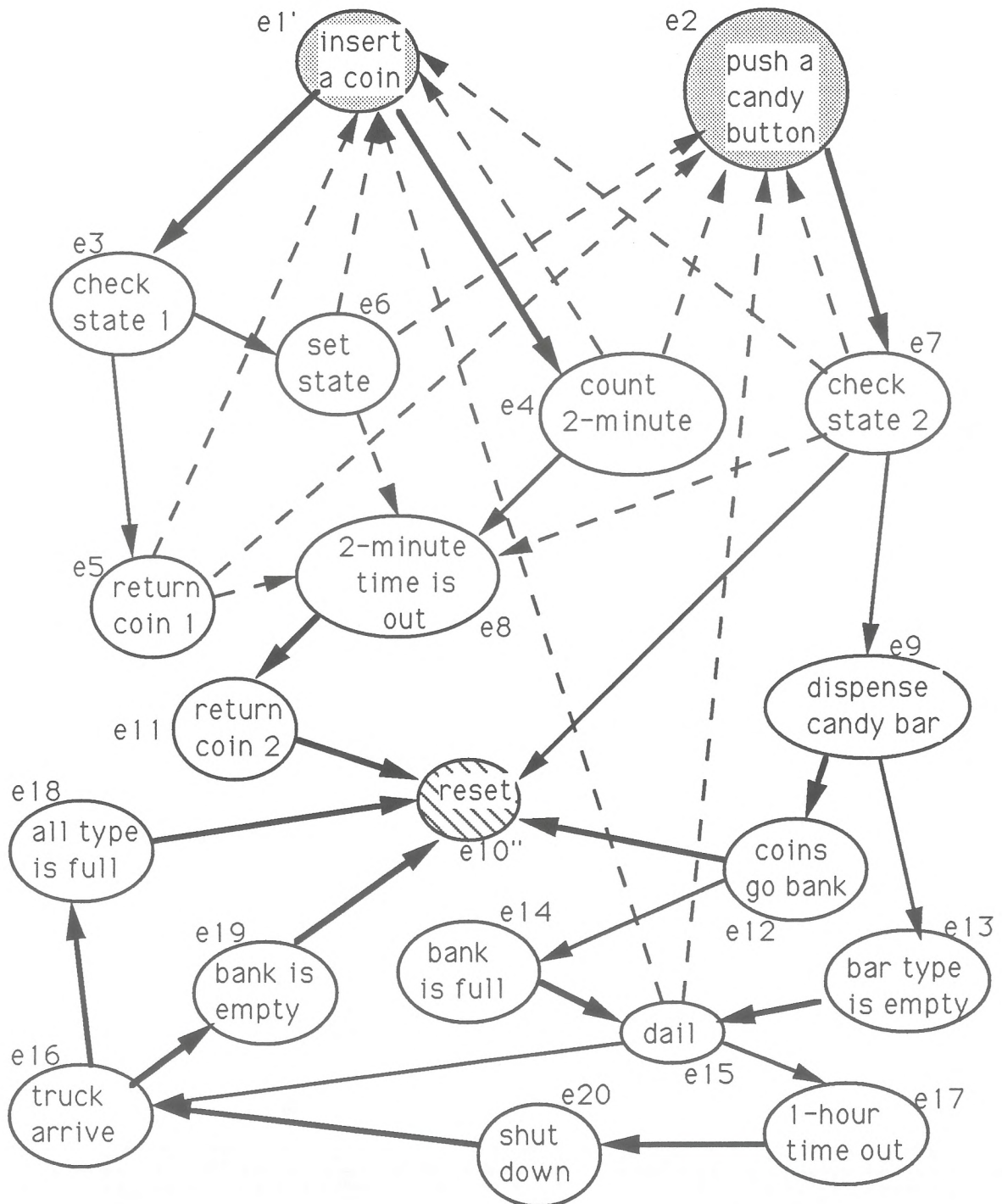
6. Check if all the 1st and 2nd *SEO* are "resolved". That means that for each 1st and 2nd *SEO*, either it is contracted or it is conveyed by some message passing notation in (*O*, *M*).

7. Make refinement.

Note that on Figure 19, the event e_{15} is abstracted as an object, which is permitted in [Shlaer & Mellor 88] (p15), [Ross 87] (p9), and [Coad & Yourdon 90] (p62) that events can be candidate classes and objects.

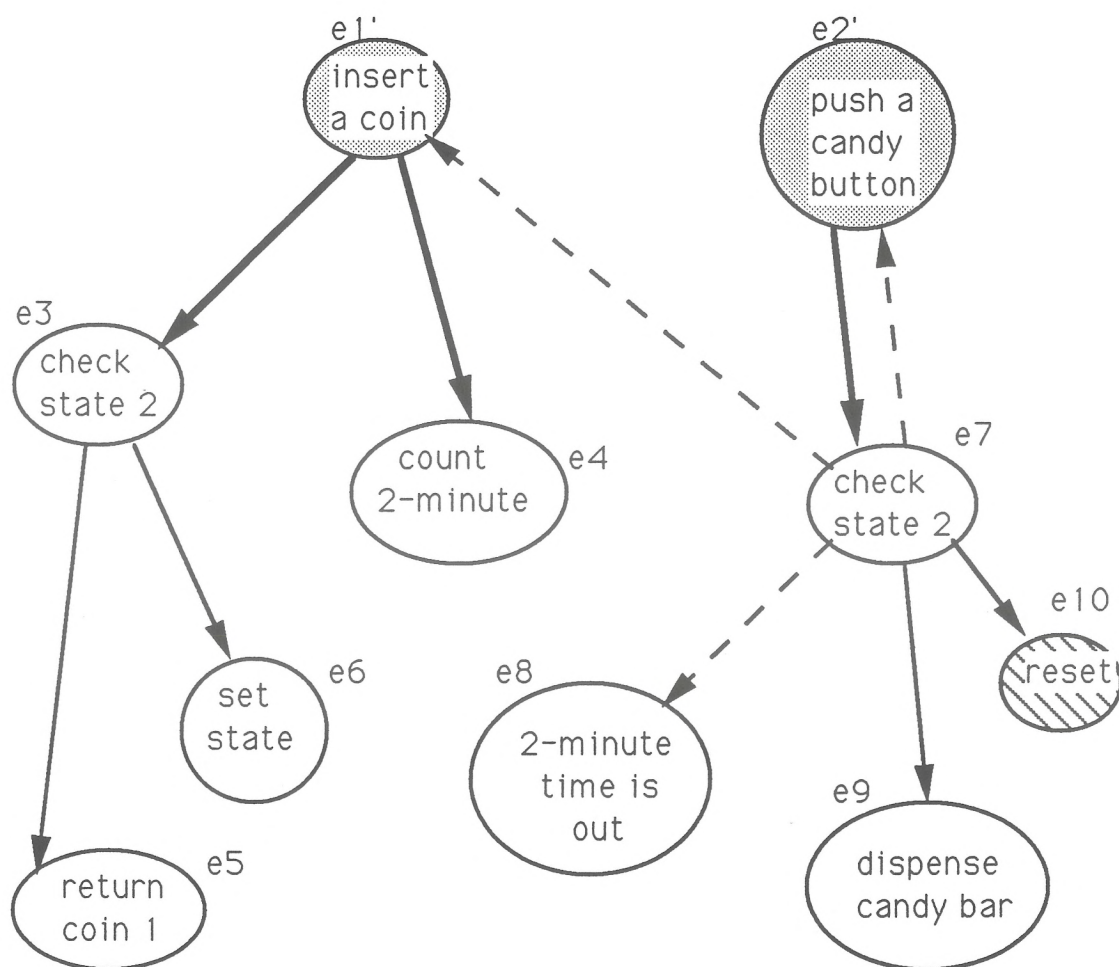
Note that the grouping some events together into an object is different from that for a regular module; this grouping is easier to manage [Pressmen 92] (p324). If we want to divide an *ED* simply into modules, our principle is to partition the *ED* so that each part has a small number of cut edges and a large number of cordiality with a minimum number of edges contracted within the partitioned part. Each module performs a single task. If we group events into an object, the main principle is the application of semantic abstraction, inheritance and encapsulation.

ILLUSTRATIONS



Symbol Dictionary: (Refer Figur 2.)

Figure 1: ED of the Vending Machine System



Symbol Dictionary:






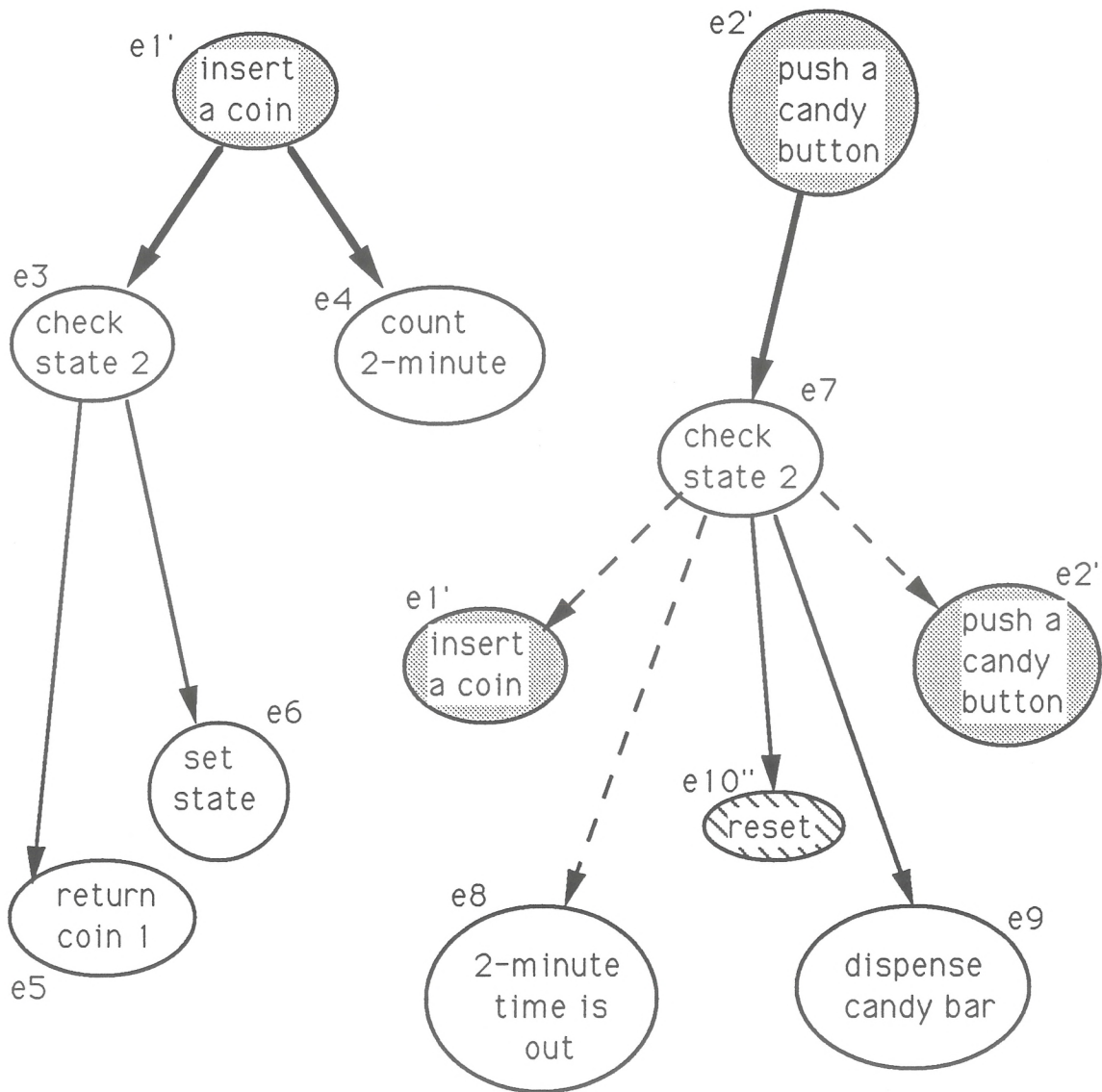
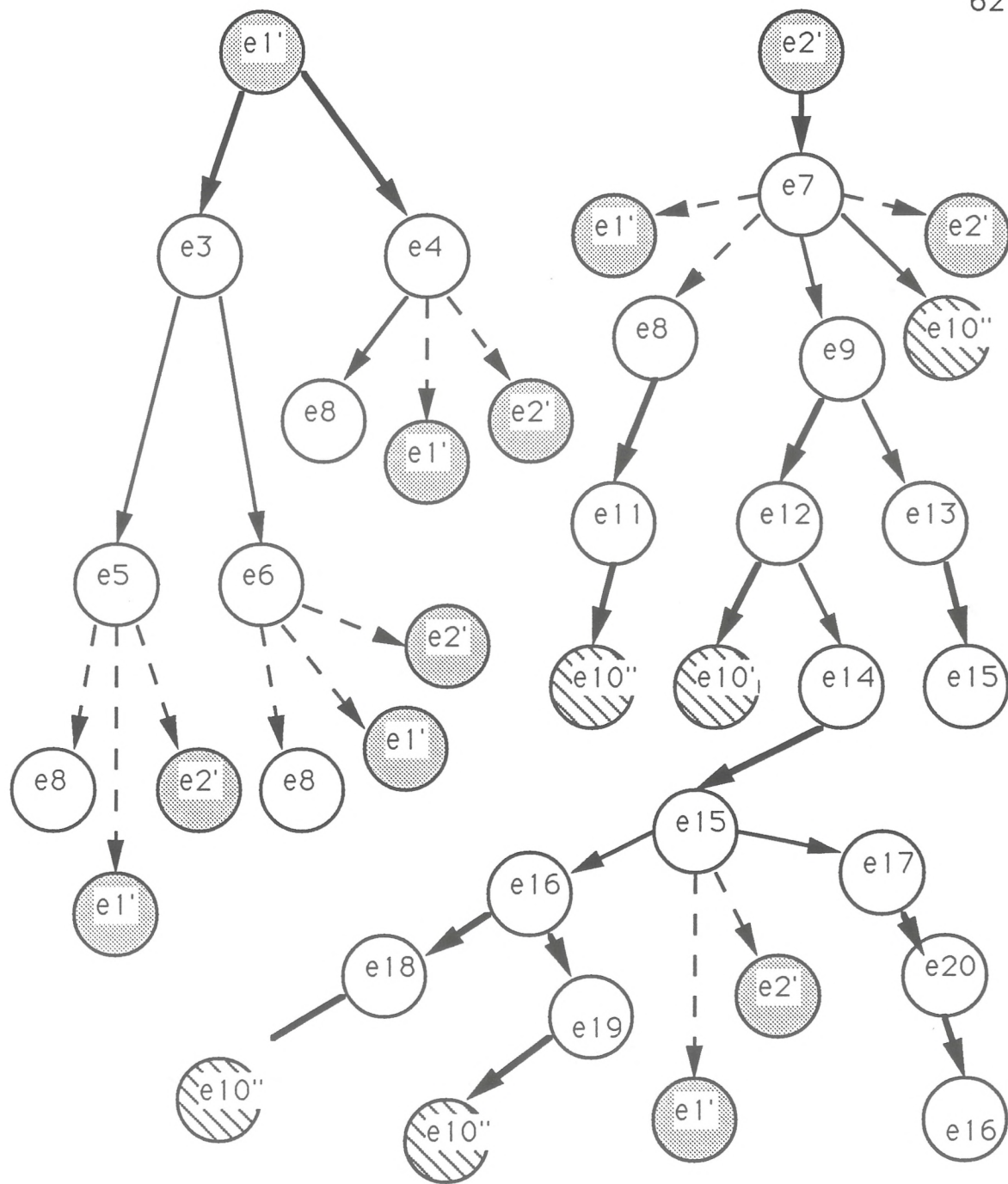
-  A resource (stimulus), an origine of scenarios.
-  A sink (sink point), a termination of scenarios
-  The third SEO, the tail and the head are not causally related.
-  The second SEO, the tail and the head are conditionally causally related.
-  The first SEO, the tail and the head are causally related.

Figure 2: Sub-ED of the Vending Machine System



Symbol Dictionary: (Refer Figure 2.)

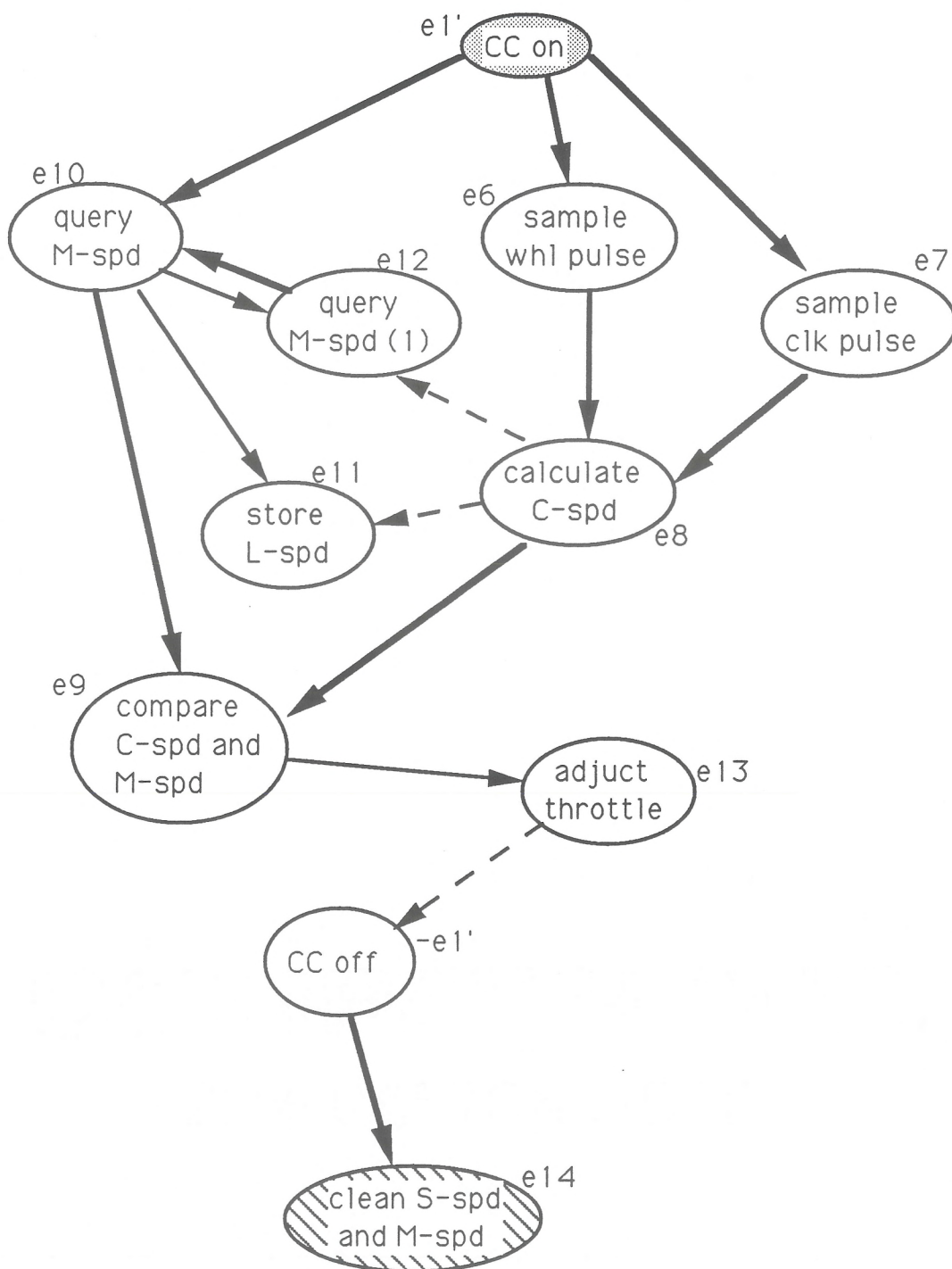
Figure 3. The Follower Tree (Sub-ED) of the Vending Machine System



Symbol Dictionary: (Refer Figure 2.)

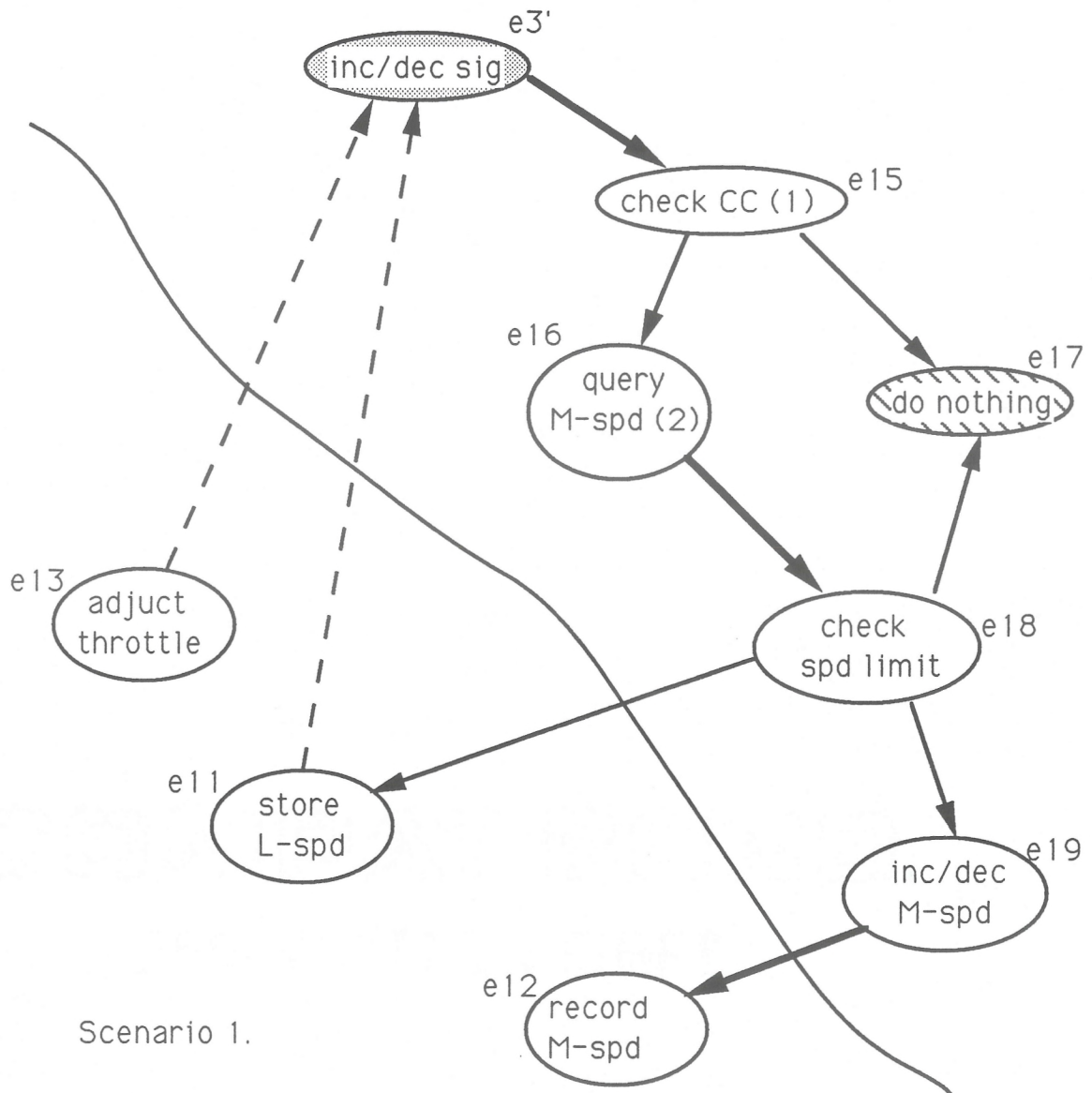
The behavior is referred on the Event Dictionary or on Figure 3.

Figure 4. The Follower Tree of the Vending Machine System



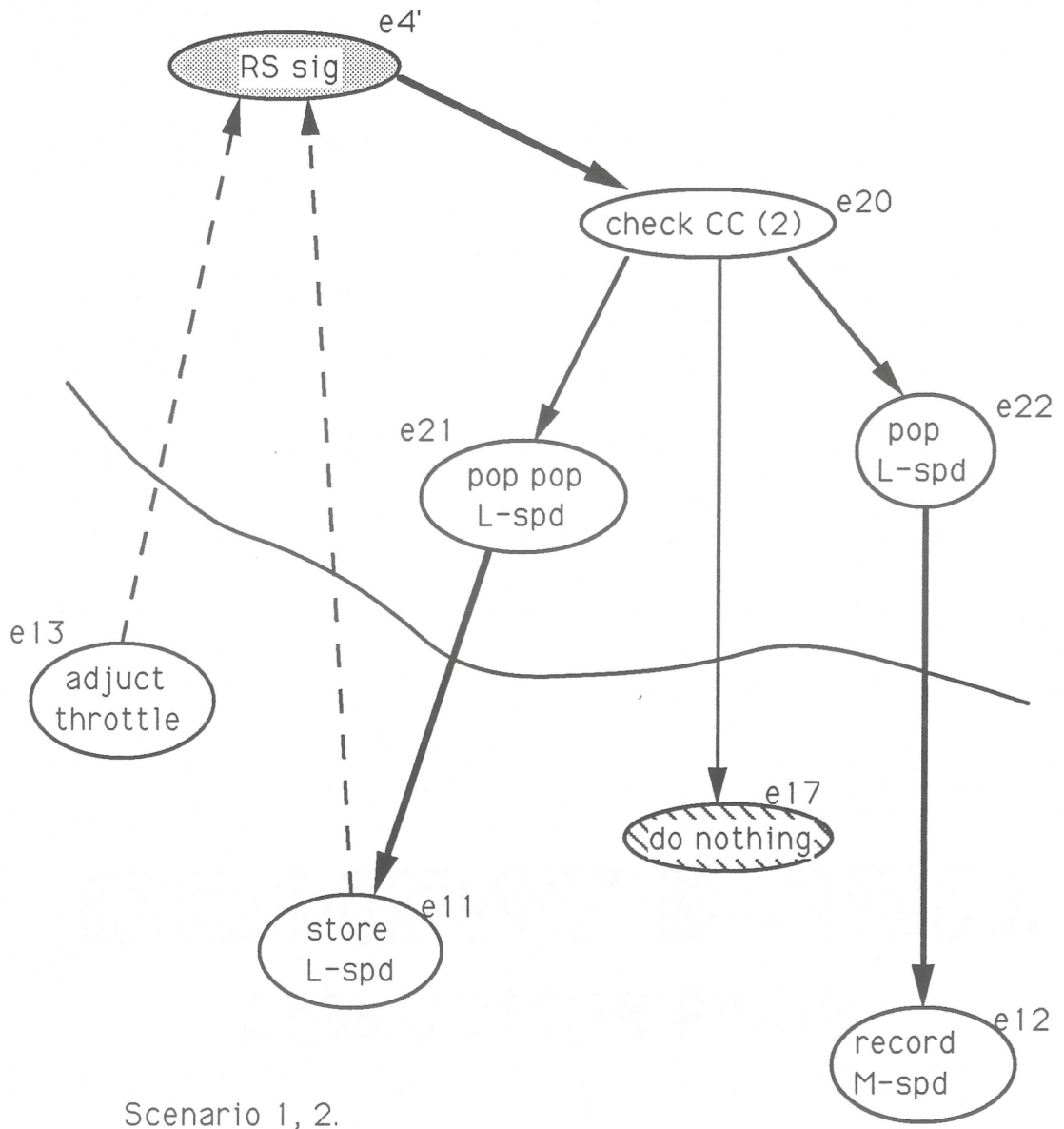
Symbol Dictionary: (Refer Figure 2).

Figure 5: ED for the scenario 1, Cruise Control;
Cruise Control System



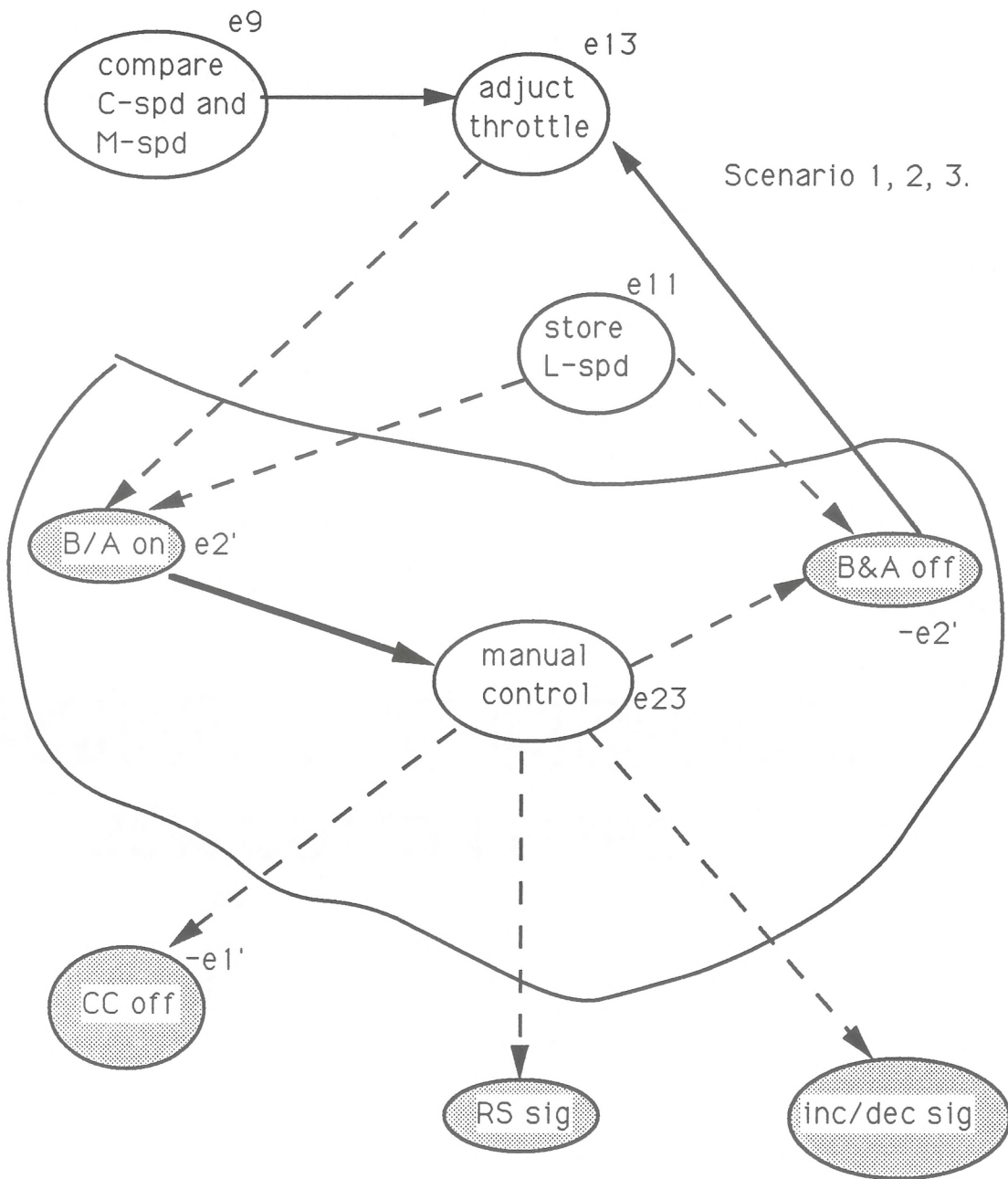
Symbol Dictionary: (Refer Figure 2).

Figure 6: ED of the Scenario 2, Inc/Dec the Maintained Speed. Cruise Control System



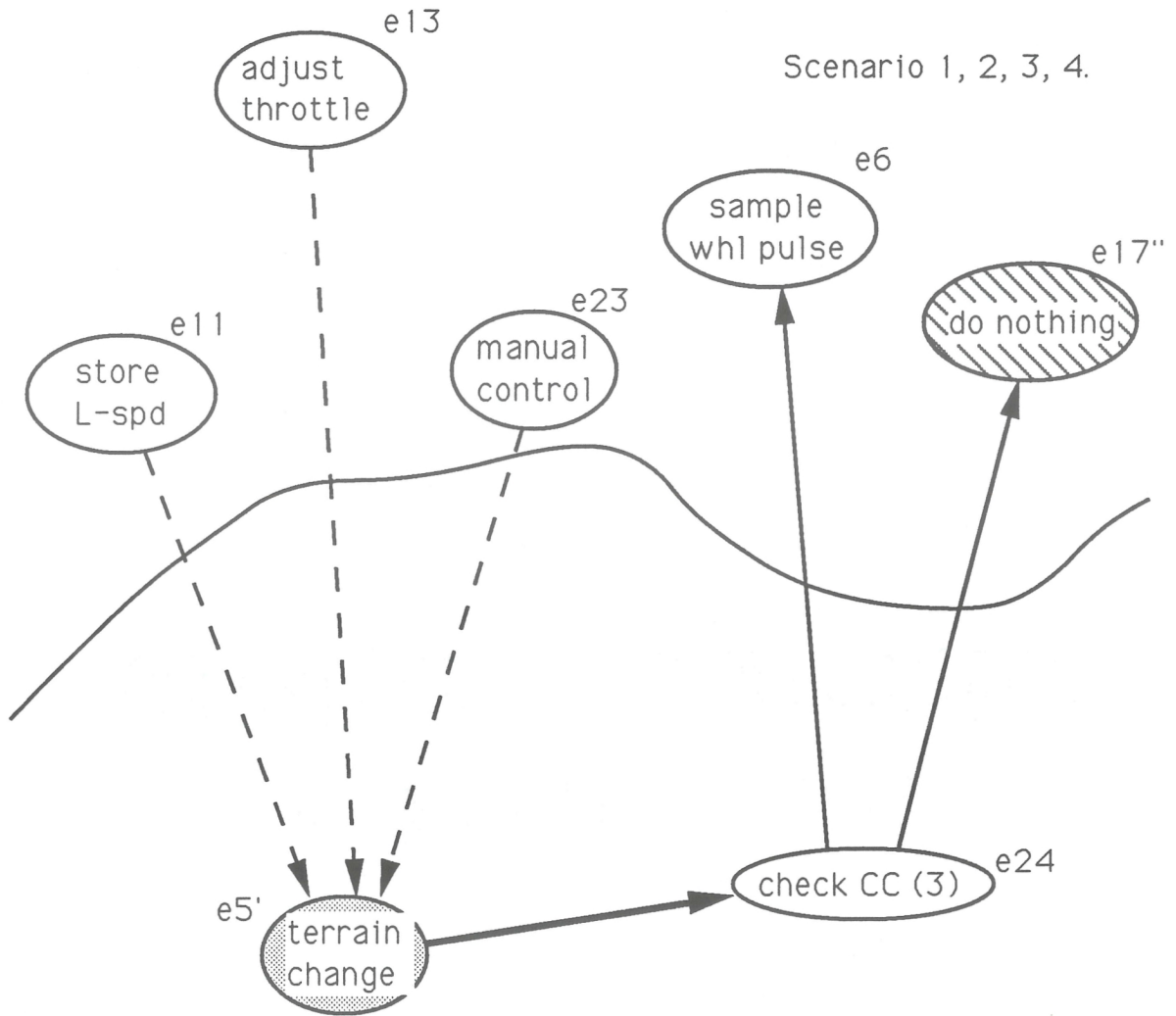
Symbol Dictionary: (Refer Figure 2).

Figure 7: ED of the Scenario 3, Resume Cruise Control System



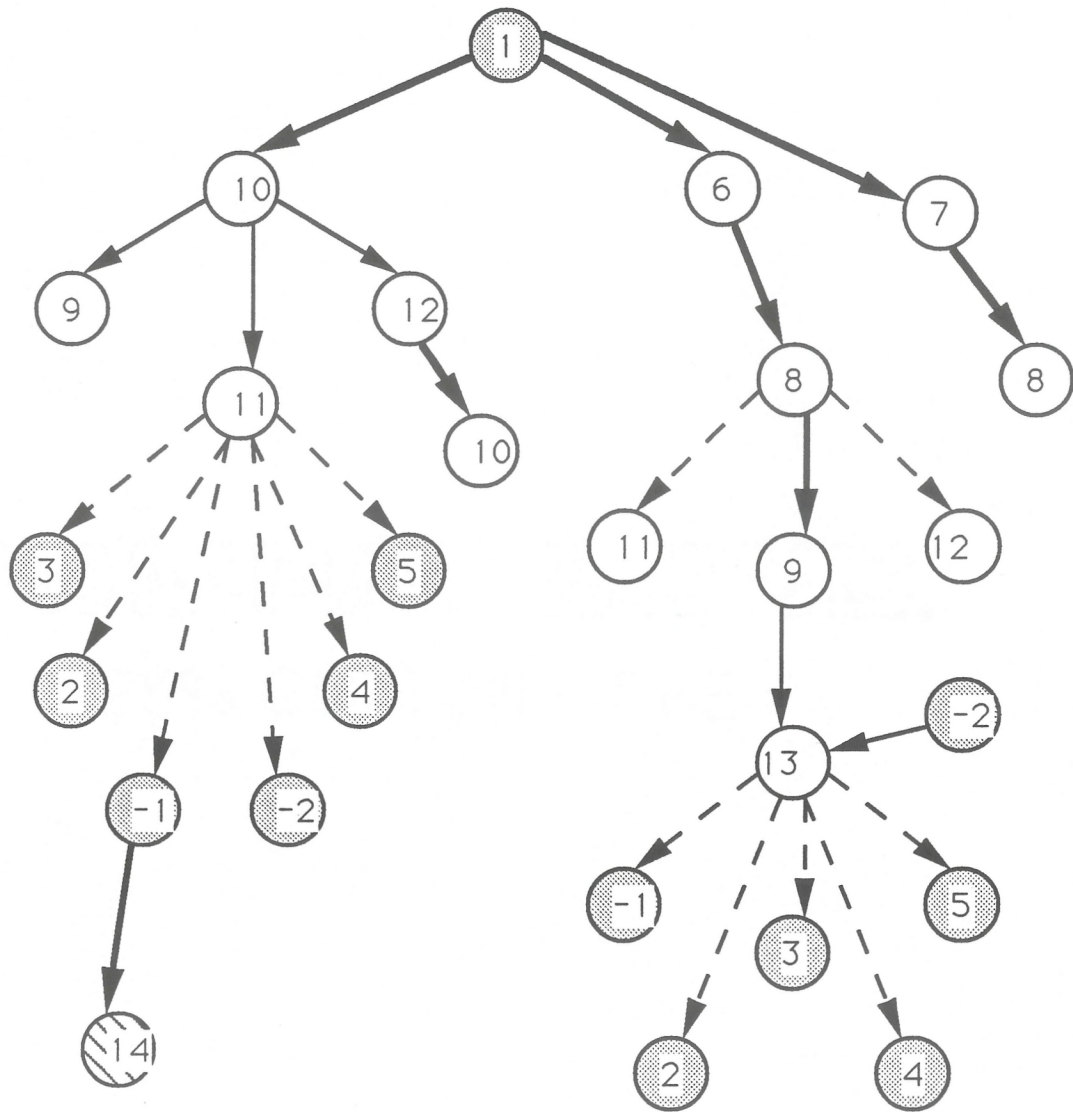
Symbol Dictionary: (Refer Figure 2).

Figure 8: ED for the scenario 4, Brake Application.
Cruise Control System



Symbol Dictionary: (Refer Figure 2.)

Figure 9: ED for scenario 5, Terrain Condition Changing Cruise Control System

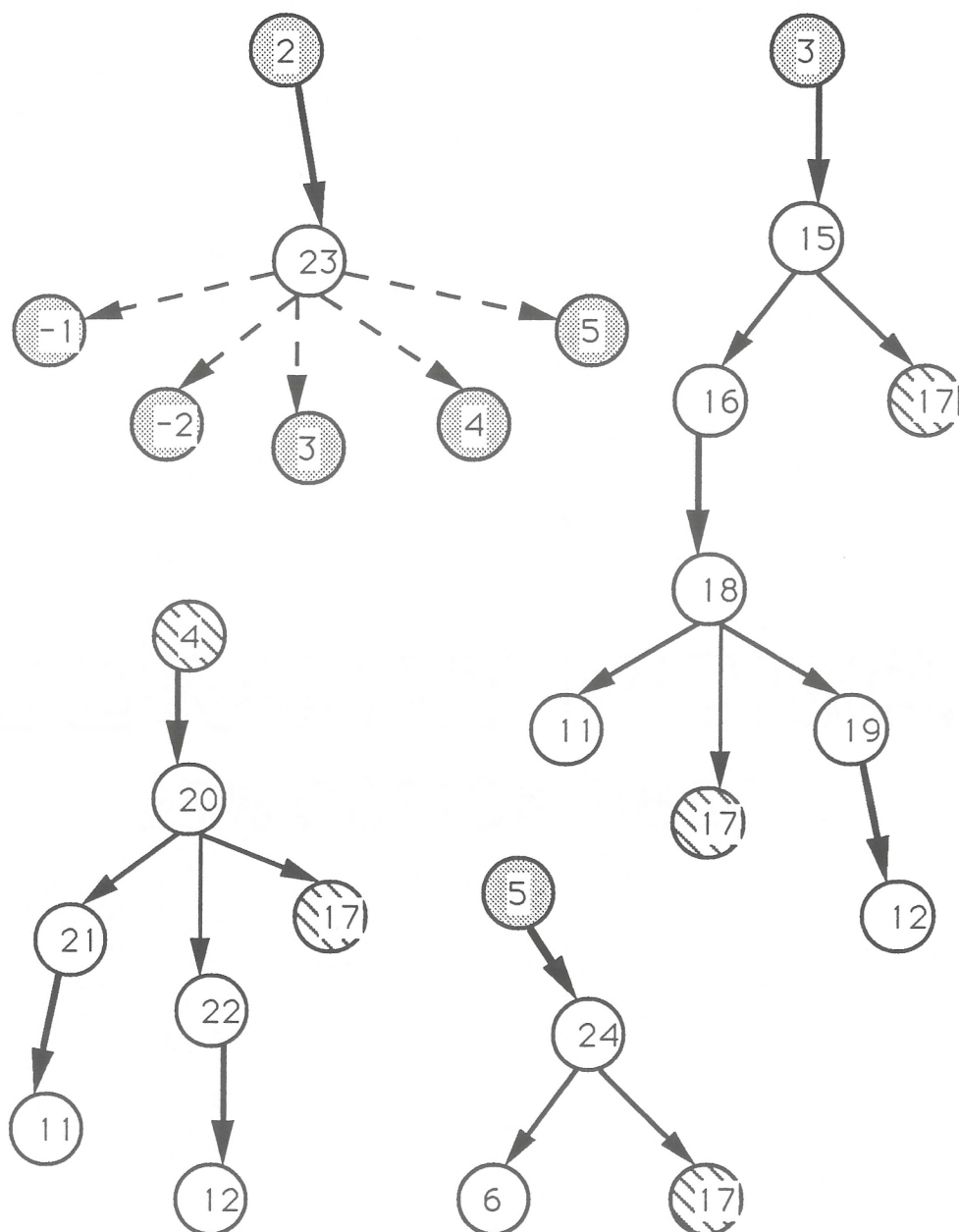


The behavior is referred on the Event Dictionary.

The Construction Order: Scenario(1, 2, 3, 4, 5).

Symbol Dictionary: (Refer Figure 2.)

Figure 10: The Follower Tree of The Scenario 1 for Cruise Control System.

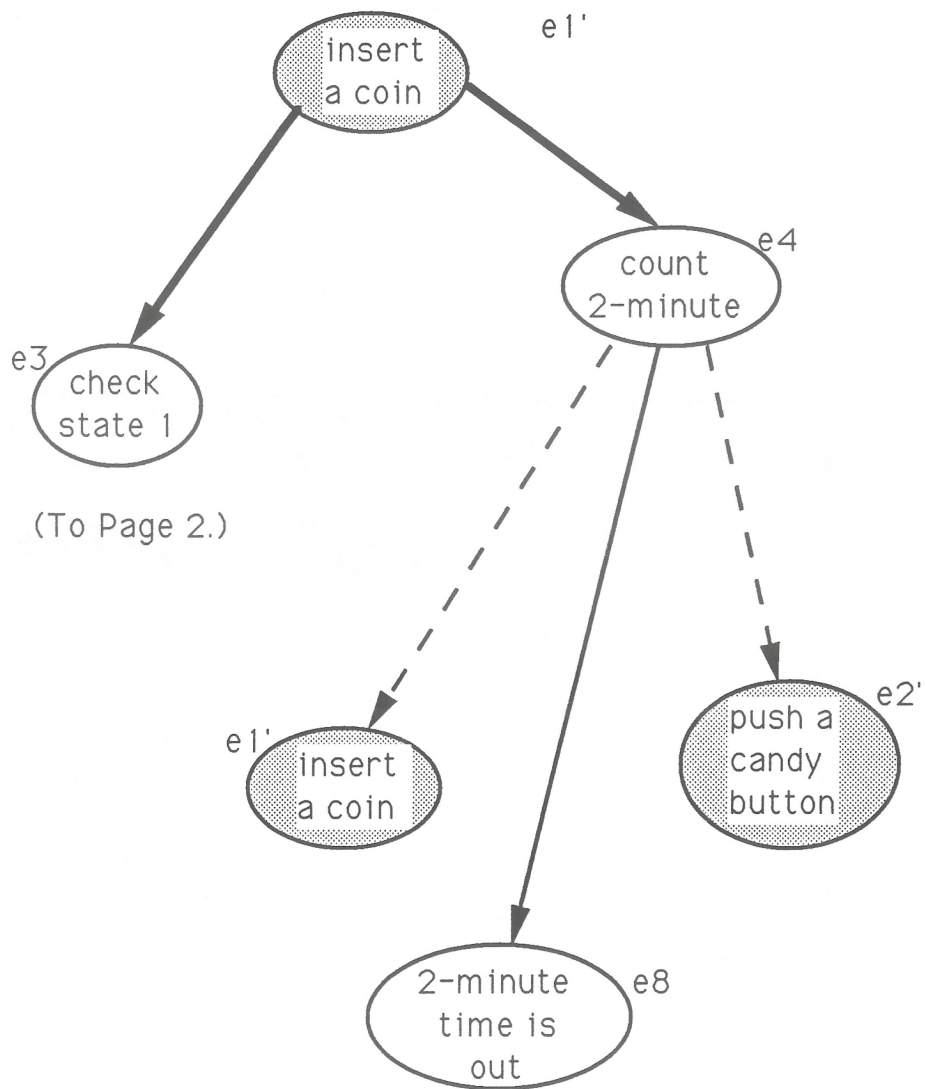


The behavior is referred on the Event Dictionary.

The Construction Order: Scenario(1, 2, 3, 4, 5)

Symbol Dictionary: (Refer Figure 2.)

Figure 11: The Follower Trees of The Scenario 2, 3, 4, 5 for Cruise Control System.

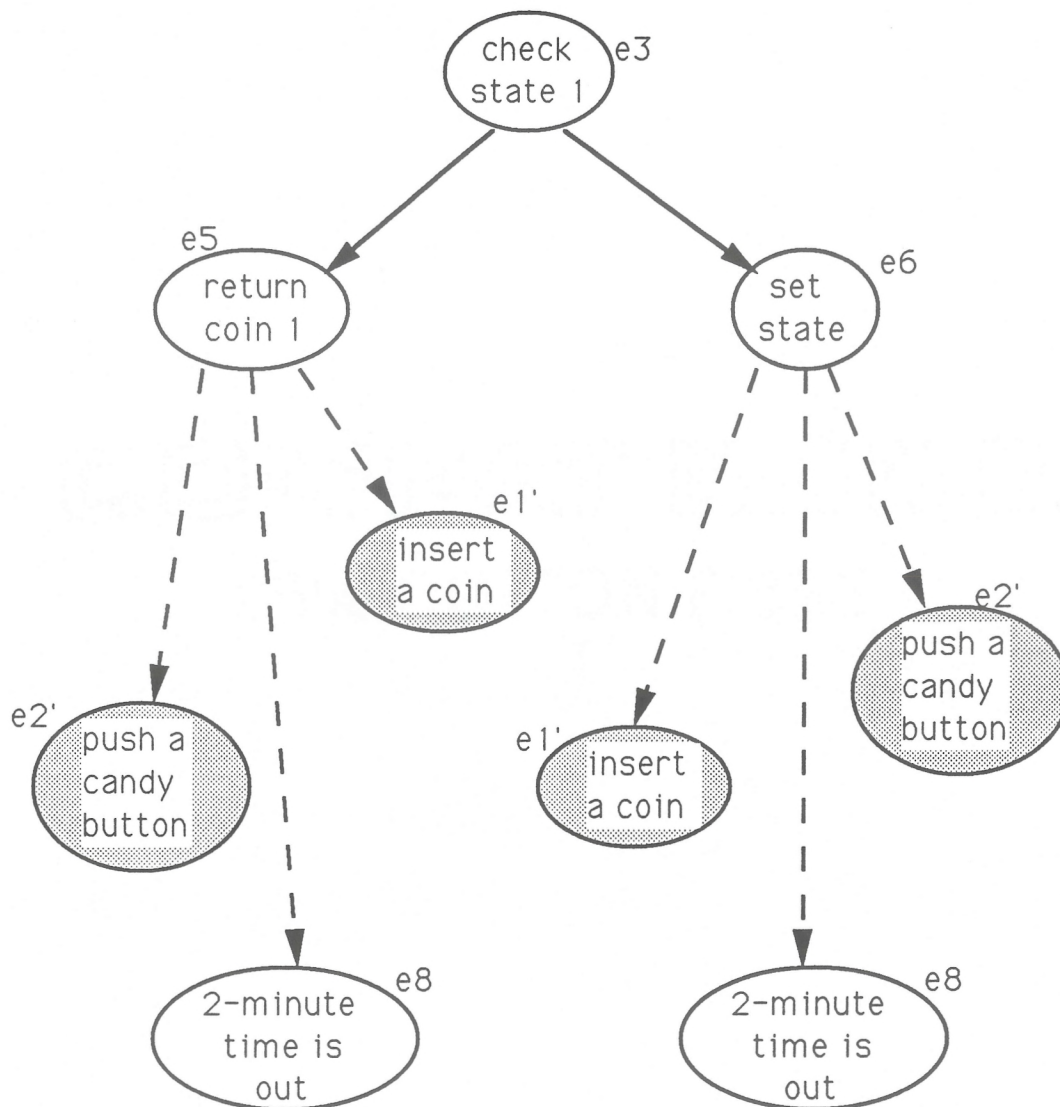


Page 1.

Symbol Dictionary: (Refer Figure 2.)

Figure 12: The Paged Follower Tree
of the Vending Machine System
(Page 1)

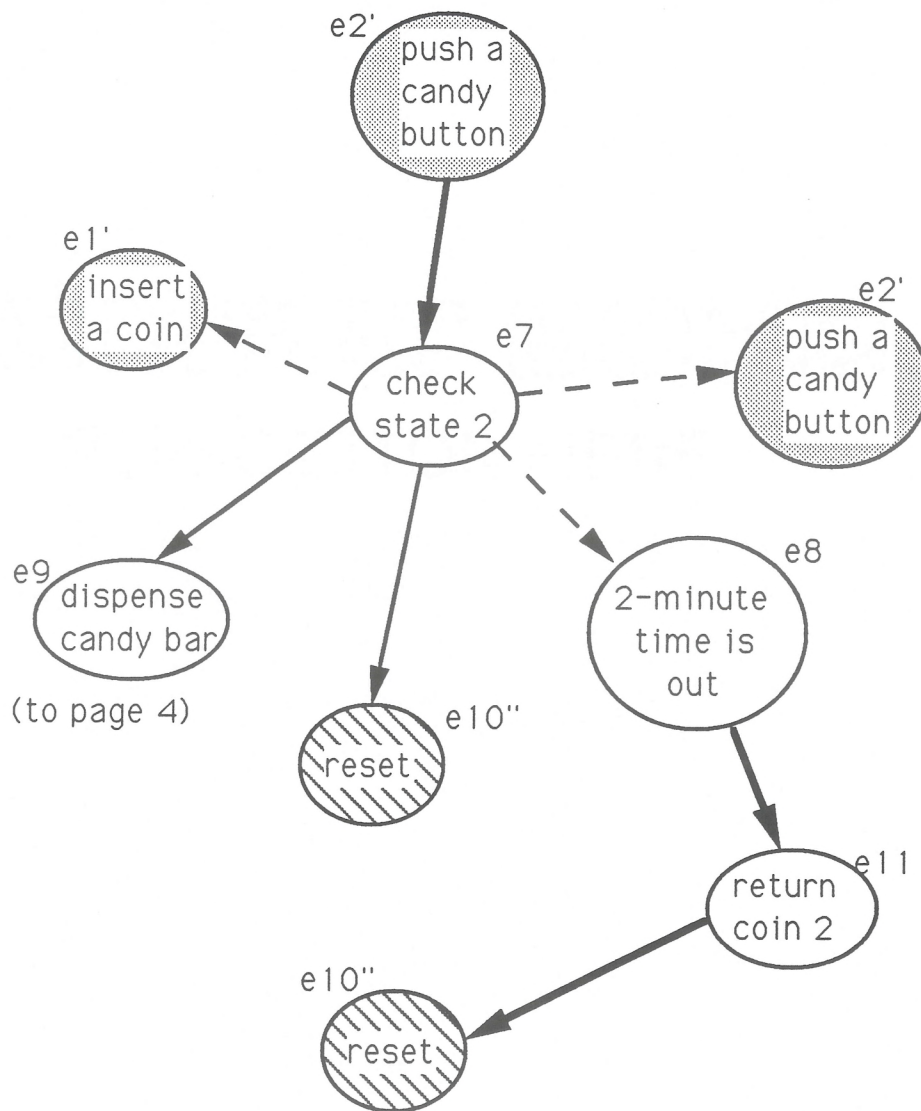
(From Page 1.)



Page 2.

Symbol Dictionary: (Refer Figure 2.)

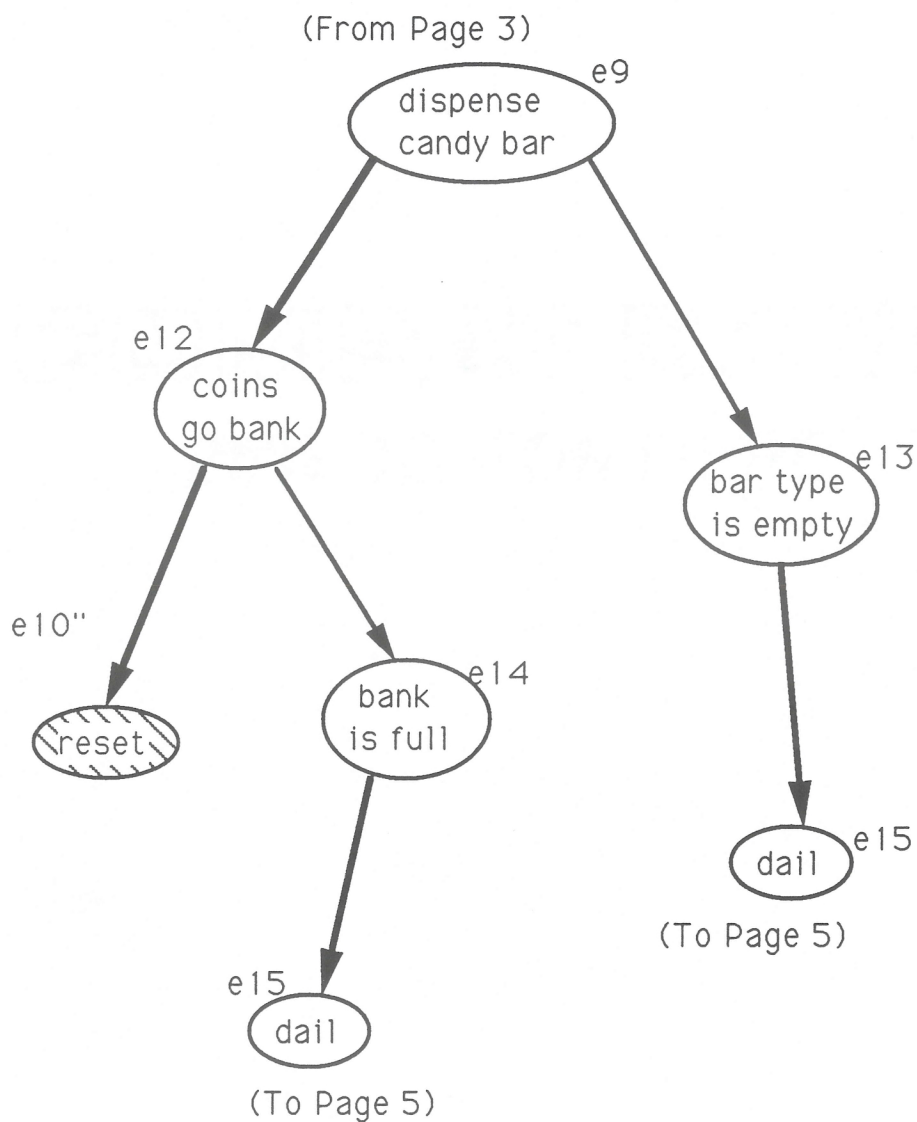
Figure 13: The Paged Follower Tree
of the Vending Machine System
(Page 2)



Page 3

Symbol Dictionary: (Refer Figure 2.)

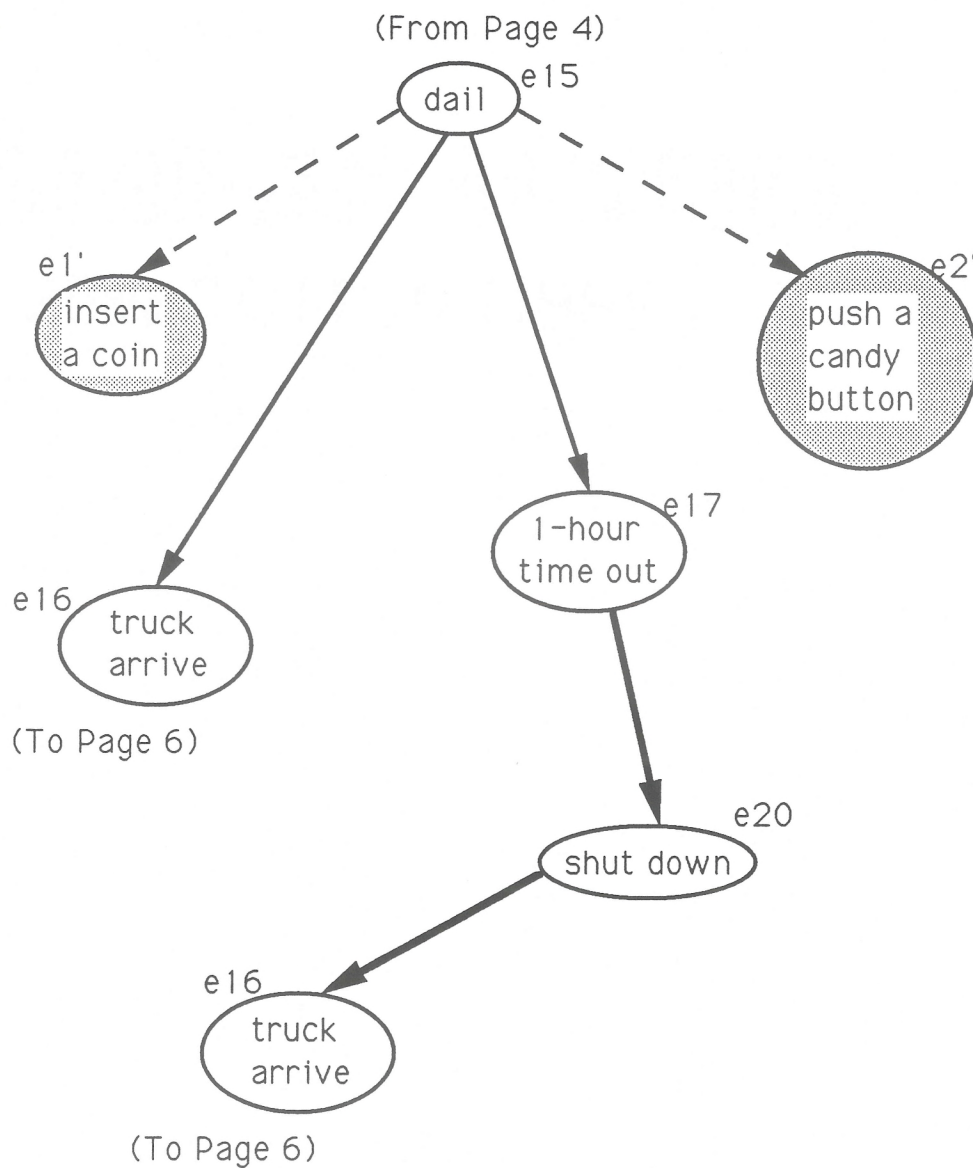
Figure 14. The Paged Follower Tree
of the Vending Machine System (Page 3)



Page 4

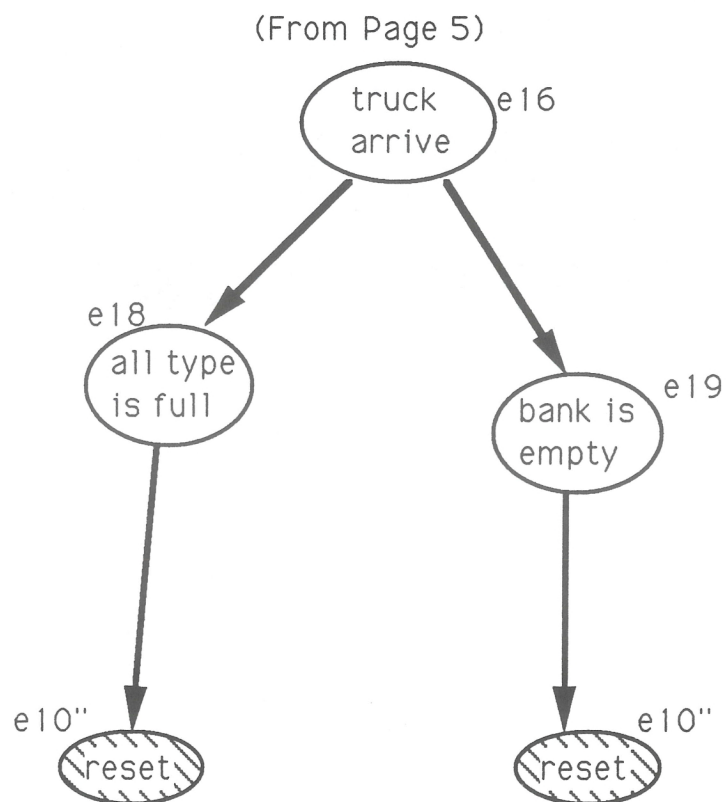
Symbol Dictionary: (Refer Figure 2.)

Figure 15: The Paged Follower Tree
of the Vending Machine System (Page 4)



Symbol Dictionary: (Refer Figure 2.)

Figure 16: The Paged Follower Tree
of the Vending Machine System (Page 5)



Page 6

Symbol Dictionary: (Refer Figure 2.)

Figure 17: The Paged Follower Tree
of the Vending Machine System (Page 6)

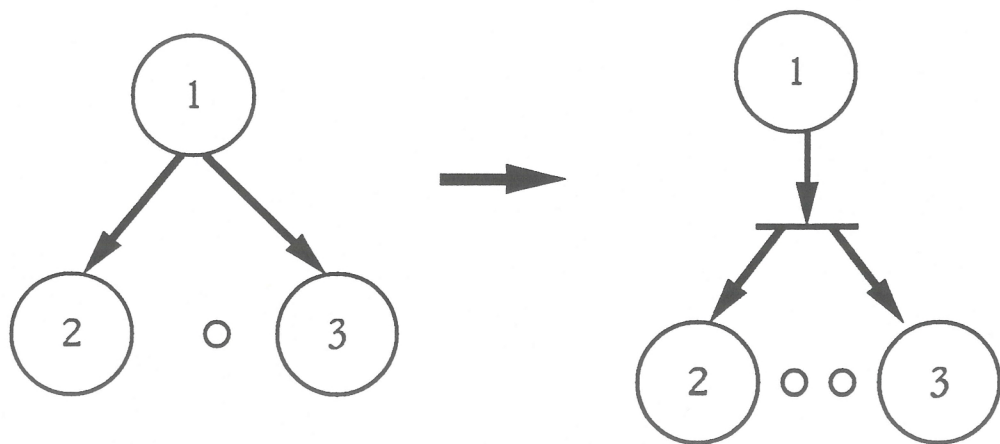
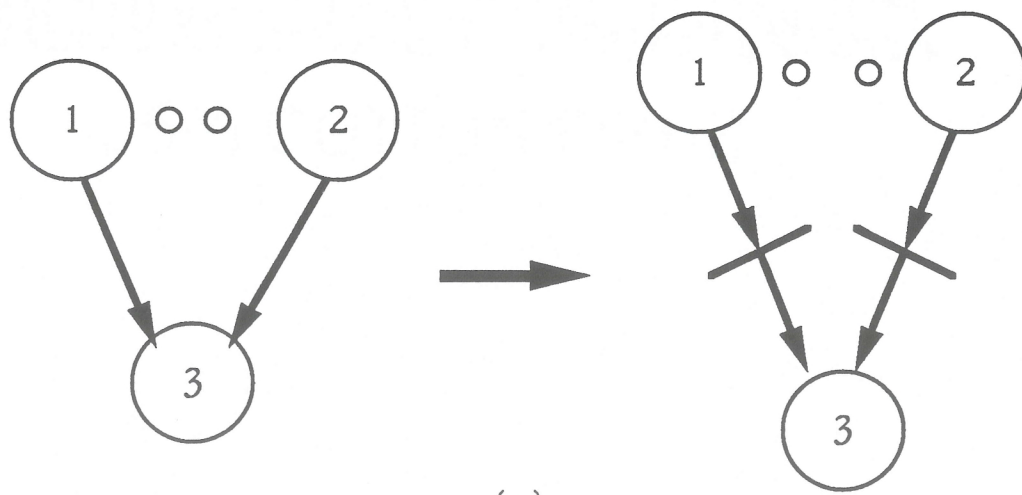
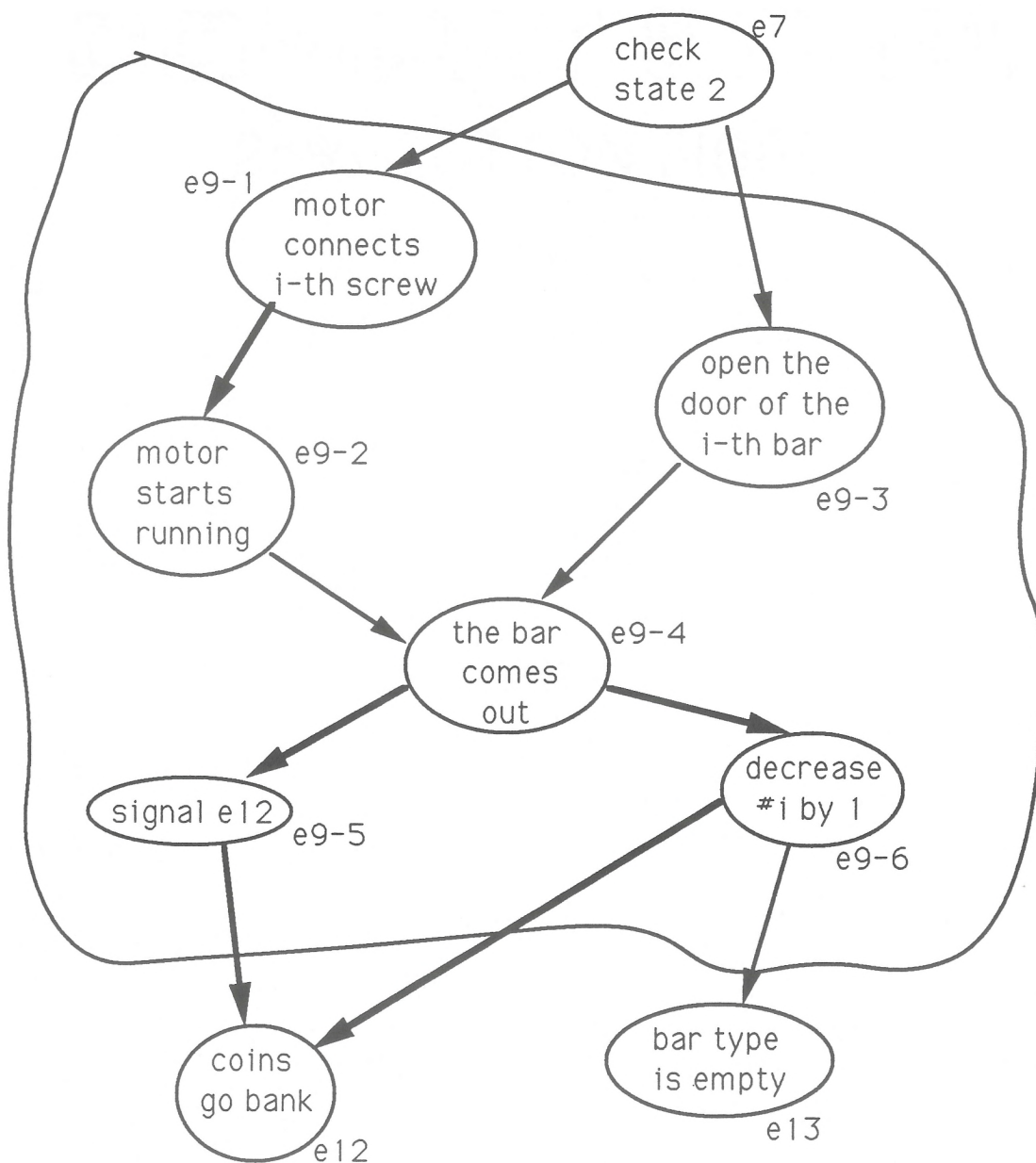
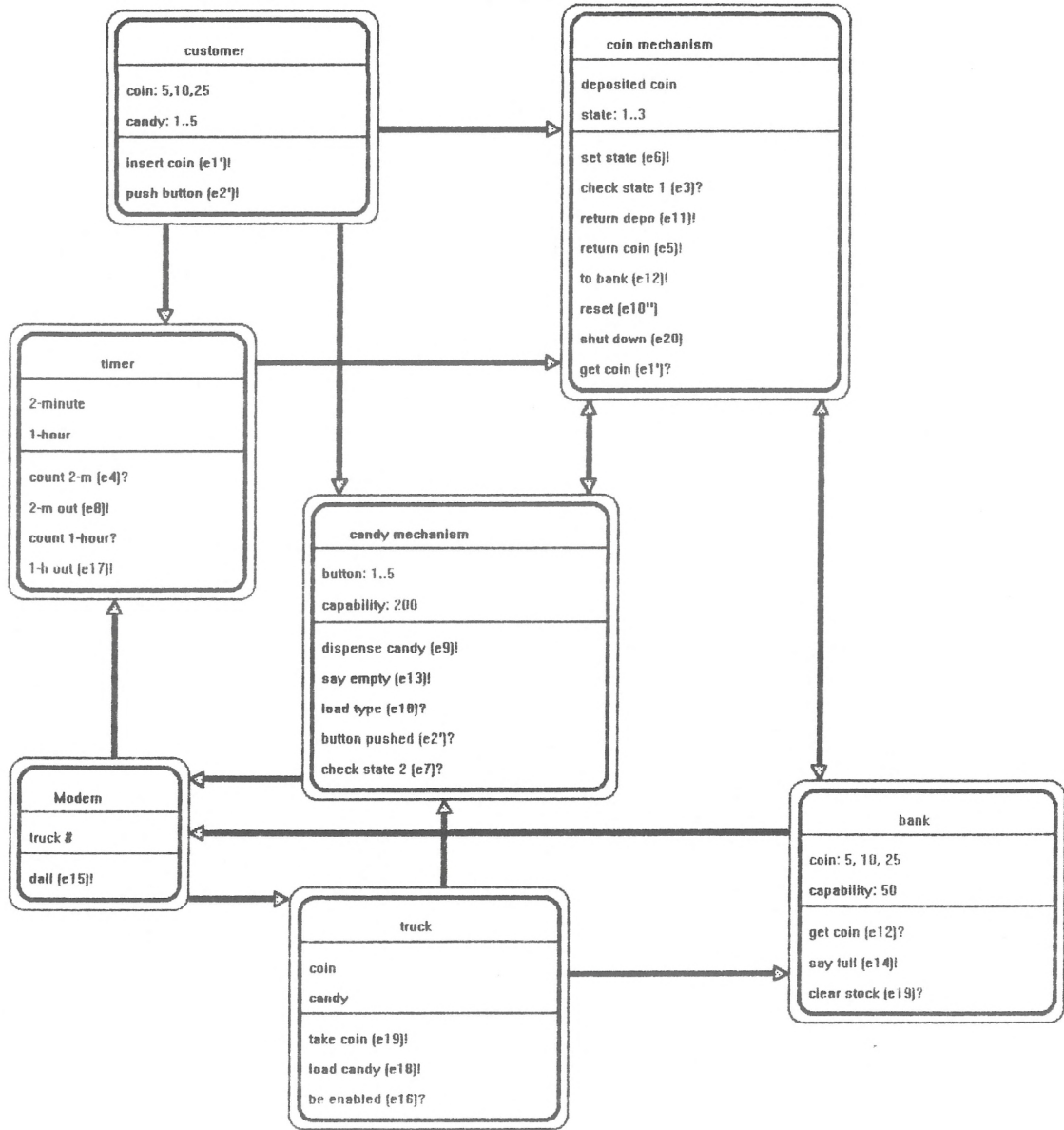


Figure 18: The Transformation from I Logic to Petri Net Structure



The symbol dictionary: (refer Figure 2.)

Figure 19: The Refinement of e9, Hierarchical Extension of an ED.



The Scenarios: (Refer Figure 1)

Figure 20: The Object Model of the Vending Machine System

APPENDIX A
THE SCENARIOS FOR CRUISE CONTROL SYSTEM

Scenario 1: Cruise Control (CC)
Date Created: June 12, 1992
Date last Revised: July 30, 1993
Level: System Level

Situation: The car is located in a parking lot.

1. The driver gets into the car and starts the engine.
2. When the speed of the car gets to 60 mph, the driver puts the CC button on.
3. The CC system picks up the wheel pulse and the clock pulse, then calculates the current speed.
4. The CC system takes the speed that has just been calculated as the maintained speed and compares it with the next speed. The compared result is used to adjust the throttle.
5. The driver turns the CC button off.
6. The CC system clears any stored information, and the work stops.

Scenario 2: Increase/Decrease the Maintained Speed
Date Created: June 12, 1992
Date last Revised: July 30, 1993
Level: System Level

Situation: The CC system is working. The maintained speed is 60 mph

1. The driver pushes the increase button.
2. The CC system checks if the change makes the maintained speed out of range.
3. The change will be all right. CC system increases the maintained speed by 5 mph. The current maintained speed becomes 65 mph.
4. The driver pushes the increase button again.
5. CC system checks if the change makes the maintained speed out of range.
6. The change will be okay if CC system increases the maintained speed by another 5 mph. The current maintained speed becomes 70 mph.
7. The driver pushes the decrease button.
8. The maintained speed goes back to 65 mph.

Scenario 3: Resume the Last Maintained Speed
Date Created: June 12, 1992
Date last Revised: July 30, 1993
Level: System Level

Situation: The CC system is working. The maintained speed is 60 mph; the last maintained speed is 65 mph. The speed before the last maintained is 50 mph.

1. The driver pushes the resume button.
2. The CC system queries the last maintained speed, say 65 mph.
3. The current maintained speed is replaced by 65 mph.
4. The driver pushes the resume button one more time.
5. The CC system queries the speed 50 mph.
6. The current maintained speed is replaced by 50 mph.

Scenario 4: Brake Application

Date Created: June 12, 1992

Date last Revised: July 30, 1993

Level: System Level

Situation: The CC system is working. The maintained speed is 60 mph.

1. The driver sees a dog on the street. He depresses the brake.
2. The throttle control is switched from the CC system to manual operation.
3. The driver releases the brake.
4. The throttle control is switched back to the CC system
5. The CC system works with 60 mph as the maintained speed.

Scenario 5: Terrain Condition Changing

Date Created: June 12, 1992

Date last Revised: July 30, 1993

Level: System Level

Situation: The CC system is working. The maintained speed is 60 mph.

1. The car is climbing a slope. The wheel pulse becomes slow.
2. The calculated speed becomes low, say 55 mph.
3. The CC system compares the calculated speed with the maintained speed of 60 mph and finds that the current speed is 5 mph low.
4. The CC system commands that the throttle opens wider until the current speed is equal to the maintained speed.

APPENDIX B
THE EVENT DICTIONARY — FORMAT AND EXAMPLES

B.1 The Event Dictionary Format and Symbol

B.1.1 The Symbols

Symbol = Arc \cup Operator \cup { \neg }

Arc \equiv { \leftrightarrow , \leftarrow , \rightarrow , \downarrow }

Operator \equiv { \wedge , \vee , \oplus }

Event \equiv { e_i , $\neg e_i$ }, $i = 1, 2, \dots, n$

Events: Event | Event Operator Event | (Events) Operator (Events)

Events Arc Events : The left is tails (or a tail); the right is heads (or a head).

Events \oplus Events : Exclusive Or of the occurrences of the left and the right.

Events \vee Events : The left hand side and the right hand side occur parallel to each other.

Fan_In : Events Arc Event | Fan_In; Fan_In

Fan_Out: Event Arc Events | Fan_Out; Fan_Out

\wedge : In Fan_In, the states specified by the left and the right are both true.

In Fan_Out, the occurrences of the left and the right are both true.

\leftrightarrow : The first SEO, the tail and the head are identity.

\rightarrow : The first SEO, the tail implies the head.

\leftarrow : The second SEO, the head implies the tail.

\downarrow : The third SEO, the tail and the head have no logical relation, but the head may be an immediate follower in the same state in which both the tail and the head occur. The tail may causes a transition into the state; the head may causes a transition out of the state.

B.1.2 The Format

Event Dictionary: The Name of a System.
 Date Created:
 Date Last Revised:
 Group:

Event (The same as that in ED): The Name of the Event
Definition: The Description of the Event
Value: The Value of the Event
Behavior With
Preceding Set: The List of the Event;
 Fan_In
 The Explanation of Fan_In
Succeeding Set: The List of the Event;
 Fan_Out
 The Explanation of Fan_Out

B.2 The Event Dictionary of the Vending Machine System

Event Dictionary: The Vending Machine System.
 Date Created: June 1, 1993
 Date Last Revised: August 5, 1993
 Group:

e1': INSERT COIN.

Definition: A customer inserts a coin into the machine. A stimulus.

Value: 5, 10, 25 (cent).

Behavior With

Preceding Set: e4, e5, e6, e7, e15;

$e4 \vee (e5 \oplus e6) \vee e7 \vee e15 \lrcorner e1'$

It may occur in the scenario boundary state;

Or it may occur after the occurrences of e4, e5, e6, e7 and e15.

Succeeding Set: e3, e4;

$e1' \leftrightarrow e3 \wedge e4$

The system checks the current coin value (e3) and the 2-minute time limit counting begins (e4) iff it occurs.

e2': PUSH A CANDY BUTTON.

Definition: A customer pushes a candy selection button. A stimulus.

Value: Candy bar type; 1, 2, ..., 50.

Behavior With

Preceding Set: (the same as those of e_1').

Succeeding Set: e_7 ;

$e_2' \leftrightarrow e_7$

The system checks the current coin value for e_2' sake (e_7) iff it occurs.

e_3 : CHECK STATE 1

Definition: Machine checks the current coin value after a coin is inserted.

Value: Boolean; current coin value is enough (Y) for a candy bar or not (N).

Behavior With

Preceding Set: e_1' ;

$e_1' \leftrightarrow e_3$

It occurs iff a coin is inserted into the machine (e_1').

Succeeding Set: e_5, e_6 ;

$e_3 \leftarrow e_5 \oplus e_6$

If its value is Y, then the new deposited coin is returned (e_5), else a new coin value state is calculated and set (e_6).

e_4 : COUNT 2-MINUTE

Definition: Counting 2-minute time begins.

Value: [0, 2 min]

Behavior With

Preceding Set: e_1' ;

$e_1' \leftrightarrow e_4$

It occurs iff a coin is inserted into the machine (e_1').

Succeeding Set: e_1', e_2', e_8 ;

$e_4 \downarrow e_1' \vee e_2'; e_4 \leftarrow e_8$

e_8 implies that e_4 happened.

The next occurrence to e_4 may be e_1', e_2', e_1' and e_2' , or e_8 .

e_5 : RETURN COIN 1

Definition: The machine returns the new deposited coin.

Value: 5, 10, 25 cent

Behavior With

Preceding Set: e_3 ;

$e_3 \leftarrow e_5$

It implies that the machine has checked the state for e_1' sake.

Succeeding Set: e_1', e_2', e_8 ;

$e_4 \downarrow (e_1' \oplus e_8) \vee (e_2' \oplus e_8)$

The next occurrence to e_4 may be e_1', e_2', e_1' and e_2' , or e_8 .

e_6 : SET STATE

Definition: A new coin state is set.

Value: (The same as those of e_3)

Behavior With

Preceding Set: (The same as those of e_5);

Succeeding Set: (The same as those of e_5).

e_7 : CHECK STATE 2

Definition: Machine checks the current coin value after a candy button is pushed.

Value: 1. no deposit;

2. coin value is not enough for a candy bar.

3. coin value is enough for a candy bar.

Behavior With

Preceding Set: e_1' ;

$e_1' \leftrightarrow e_7$

It occurs iff a candy select button is pushed (e_2').

Succeeding Set: $e_1', e_2', e_8, e_9, e_{10}''$;

$e_7 \leftarrow e_9 \oplus e_{10}''$; $e_7 \downarrow (e_1' \oplus e_8) \vee (e_2' \oplus e_8)$

If its value is 1, then the machine reset (e_{10}'');

else if its value is 2, then the next occurrence may be $e_1', e_2', e_1' \wedge e_2'$, or e_8 ;

else the machine dispense a candy bar (e_9).

e_8 : 2-MINUTE TIME IS OUT

Definition: 2-minute time is out.

Value:

Behavior With

Preceding Set: e_4, e_5, e_6, e_7 ;

$e_4 \leftarrow e_8$; $(e_5 \oplus e_6) \vee e_7 \downarrow e_8$

e_8 implies that e_4 has happened but not necessary to be the last happened.

e_8 implies that the last occurrence may be e_4, e_5, e_6 , and e_7 .

Succeeding Set: e_{11} ;

$e_8 \leftrightarrow e_{11}$

the machine returns all deposits iff it occurs.

e_9 : DISPENSE CANDY BAR

Definition: A candy bar is dispensed.

Value: The amount for each bar type (0, 1, ...,40).

Behavior With

Preceding Set: e_7 ;

$e_7 \leftarrow e_9$

It implies that e_7 gets the value 3.

Succeeding Set: e_{12}, e_{13} ;

$e_9 \leftrightarrow e_{12}$; $e_9 \leftarrow e_{13}$

The machine put the deposit to the bank iff e_9 has occurred.

That the amount of a bar type is zero implies that e_9 has occurred.

e₁₀'': RESET

Definition: The machine is reset. A sink.

Value: The deposit is zero.

Behavior With

Preceding Set: e₇, e₁₁, e₁₂, e₁₈, e₁₉;

$e_{11} \oplus e_{12} \oplus (e_{18} \wedge e_{19}) \leftrightarrow e_{10}''$; $e_7 \leftarrow e_{10}''$

e₁₈ and e₁₉ happen together. After that e₁₀'' is inevitable.

e₁₀'' occurs only if e₁₁ or e₁₂ occur.

e₁₀'' implies that e₇ gets the value 1.

Succeeding Set: (N/A).

e₁₁: RETURN COIN 2

Definition: The machine returns all the deposited coins.

Value: 5x cent

Behavior With

Preceding Set: e₈;

$e_8 \leftrightarrow e_{11}$

It occurs iff 2-minute time is out.

Succeeding Set: e₁₀'';

$e_{11} \rightarrow e_{10}''$

If it occurs, the machine is reset.

e₁₂: COINS GO BANK

Definition: The machine puts the all deposited coins into the bank.

Value: The amount for each coin type, 1,...,40.

Behavior With

Preceding Set: e₉;

$e_9 \leftrightarrow e_{12}$

It occurs iff a candy bar has been dispensed.

Succeeding Set: e₁₀'', e₁₄;

$e_{12} \rightarrow e_{10}''$; $e_{12} \leftarrow e_{14}$

If it occurs, the machine is reset;

That the amount of a bar is 5 (e₁₃) implies that it has occurred.

e₁₃: BAR TYPE IS EMPTY

Definition: The amount of a bar is 5.

Value:

Behavior With

Preceding Set: e₉;

$e_9 \leftarrow e_{13}$

It implies that one value of e₉ has got 5.

Succeeding Set: e₁₅

$e_{13} \rightarrow e_{15}$

If it occurs, the machine dials the truck service.

e_{14} : BANK IS FULL

Definition: The amount of a coin is less than 35.

Value:

Behavior With

Preceding Set: e_{12} ;

$e_{12} \leftarrow e_{14}$

It implies that one value of e_{12} has got 35.

Succeeding Set: (The same as those of e_{13}).

e_{15} : DIAL

Definition: The machine dials the truck service.

Value:

Behavior With

Preceding Set: $e_1', e_2', e_{13}, e_{14}$;

$(e_{13} \vee e_{14}) \rightarrow e_{15}$

It happens if e_{13}, e_{14} happen or if e_{13} and e_{14} happen together.

Succeeding Set: $e_1', e_2', e_{16}, e_{17}$;

$e_{15} \leftarrow e_{16} \oplus e_{17}; e_{15} \downarrow (e_1' \vee e_2')$

Truck service implies that e_{15} occurred and e_{17} won't occur.

1-hour time out implies that e_{15} happened and e_{16} doesn't happen.

Before e_{16} or e_{17} happen, the e_1' or e_2' may also happen. But only e_{16} or e_{17} can remove the token from e_{15} .

e_{16} : TRUCK ARRIVE

Definition: The service truck accesses the machine.

Value:

Behavior With

Preceding Set: e_{20}, e_{15} ;

$e_{15} \leftarrow e_{16}; e_{20} \rightarrow e_{16}$

e_{16} is the only resolution for the shut down state (got into with the e_{20}).

It may be the next occurrence after the machine has dialed for the truck.

Succeeding Set: e_{18}, e_{19} ;

$e_{16} \leftrightarrow e_{18} \wedge e_{19}$

e_{18} and e_{19} happen together iff it occurs.

e_{17} : 1-HOUR TIME OUT

Definition: The 1-hour time is out.

Value:

Behavior With

Preceding Set: e₁₅;

e₁₅ ← e₁₇

It implies that the machine has dialed for the truck.

Succeeding Set: e₂₀;

e₁₇ ↔ e₂₀

The machine is shut down iff it occurs.

e₁₈: ALL TYPE IS FULL

Definition: The service truck fills the amount for each bar type.

Value: (The same as those of e₉)

Behavior With

Preceding Set: e₁₆;

e₁₆ ↔ e₁₈

It occurs iff the service truck arrives (e₁₆).

Succeeding Set: e_{10''};

e₁₈ → e_{10''}

If it occurs the machine is reset (e_{10''}).

e₁₉: BANK IS EMPTY

Definition: The service truck takes all the coins from the bank.

Value: (The same as those of e₁₂)

Behavior With

Preceding Set: (The same as those of e₁₈).

Succeeding Set: (The same as those of e₁₈).

e₂₀: SHUT DOWN

Definition: The machine is shut down.

Value:

Behavior With

Preceding Set: e₁₇;

e₁₇ ↔ e₂₀

It occurs iff the 1-hour time is out (e₁₇).

Succeeding Set: e₁₆;

e₂₀ → e₁₆

If it occurs only e₁₆ can changes state caused by it.

B.3 The Event Dictionary of the Cruise Control System

Event Dictionary: The Cruise Control System.

Date Created: July 1, 1993

Date Last Revised: August 10, 1993

Group:

e_1' : CC ON

Definition: The CC system starts working if the engine is on and the CC is turned on. Its effect can only be canceled by its complement event $\neg e_1'$. A Stimulus.

Value: True (false for $\neg e_1'$), False (true for $\neg e_1'$).

Behavior With

Preceding Set: (None)

(It may happen after any stimulus occurrence. The possible occurrence order in a stimuli don't need to specify in the ED.

e_{11} and e_{13} are of the assumption that the CC system is working (starting with e_1').

If we didn't do the abstraction that

"engine is on" and "CC button is on" = CC on

"engine is off" or "CC button is off" = CC off,

we would have the arc ("engine is on", "manual control"). But that abstraction will cause the event "manual control" to be out of the CC system domain.)

Succeeding Set: e_{10} , e_6 , e_7 ;

$e_1' \leftrightarrow e_6 \wedge e_7 \wedge e_{10}$

If it occurs, then

The maintained speed is queried (e_{10})

The wheel pulse is queried (e_6)

The clock pulse is queried (e_7).

$\neg e_1'$: CC OFF

Definition: The CC system stops working if the engine is off and the CC is turned off. Its effect can only be canceled by its complement event e_1' . A stimulus.

Value: True (false for e_1'), False (true for e_1').

Behavior With

Preceding Set: e_{13} , e_{23} ;

$e_{13} \oplus e_{23} \downarrow \neg e_1'$

It may happen in the manual control state (starting with e_{23}) or in the state that the throttle is being controlled by the CC system (starting with e_{13}).

Succeeding Set: e_{14}

$\neg e_1' \leftrightarrow e_{14}$

If it occurs, then the CC system is reset (e_{14}).

e_2' : B/A ON

Definition: The driver depresses the brake or accelerator. It has the complement event $\neg e_2'$. A stimulus.

Value: True (false for $\neg e_2'$), False (true for $\neg e_2'$).

Behavior With

Preceding Set: e_{11}, e_{13} ;

$(e_{11} \vee e_{13}) \downarrow e_2'$

It may happen in the state that the throttle is being controlled by CC system (starting with e_{13}). Or if e_{11} (a speed is stored for the further resuming) were the last occurrence, it may be the next.

Succeeding Set: e_{23} ;

$e_2' \rightarrow e_{23}$

If it occur then the car gets into manual control state (e_{23}).

$\neg e_2'$: B&A OFF

Definition: Both the brake and the accelerator are released. It has the complement event e_2' . A stimulus.

Value: True (false for e_2'), False (true for e_2').

Behavior With

Preceding Set: e_{11}, e_{23} .

$e_{11} \vee e_{23} \downarrow \neg e_2'$

This stimulus may happen in the manual control state (starting with e_{23}).

If e_{11} (a speed is stored for the further resuming) were the last occurrence, it may be the next.

Succeeding Set: e_{13} .

$\neg e_2' \leftarrow e_{13}$

It is necessary for the CC system to control the throttle. ($\neg e_2' \wedge e_9 \leftrightarrow e_{13}$).

e_3' : INC/DEC SIG

Definition: Driver pushes the increase/decrease button once. A stimulus.

Value:

Behavior With

Preceding Set: e_{11}, e_{13}, e_{23} ;

$e_{11} \vee (e_{13} \oplus e_{23}) \downarrow e_3'$

This stimulus may occur in the state of manual control (starting with e_{23}) or of the cruise control (starting with e_{13}). Or if e_{11} (a speed is stored for the further resuming) were the last occurrence, it may be the next.

Succeeding Set: e_{15}

$e_3' \leftrightarrow e_{15}$

If it occurs, then the CC system needs to see if the CC button is on.

e_4' : RS SIG

Definition: Driver pushes the resume button once. A stimulus.

Value:

Behavior With

Preceding Set: (The same as those of e_3').

Succeeding Set: e_{20} ;

$e_4' \leftrightarrow e_{20}$

If it occurs, then the CC system needs to see if the CC button is on.

e5': TERRAIN CHANGE

Definition: The change of the terrain condition occurs. A stimulus.

Value: [0, f(140 mph)]

Behavior With

Preceding Set: (The same s those of e3').

Succeeding Set: e24.

$e5' \leftrightarrow e24$

If it occurs, then CC system need to see if CC button is on.

e6: SAMPLE WHL PULSE

Definition: The CC system samples the wheel pulses.

Value: [0, f(140 mph)]

Behavior With

Preceding Set: e1', e24.

$e1' \leftrightarrow e6; e24 \leftarrow e6$

The CC system samples the wheel pulse iff the CC button is on.

If e24 gets it value "CC on", then e6's value will be changed.

Succeeding Set: e8.

$e6 \leftrightarrow e8$

If it occurs, then the CC system calculates the current speed.

e7: SAMPLE CLK PULSE

Definition: CC system samples the clock pulses.

Value: Constant

Behavior With

Preceding Set: e8;

$e7 \leftrightarrow e8$

If it occurs, then the CC system calculates the current speed.

Succeeding Set: (The same as those of e6).

e8: CALCULATE C-SPD

Definition: The CC system calculates the current speed.

Value: [0, 140 mph]

Behavior With

Preceding Set: e6, e7;

$e6 \wedge e7 \leftrightarrow e8$.

e8 uses the data from e6 and e7 iff e6 and e7 are being processed.

Succeeding Set: e9, e11, e12;

$e8 \leftrightarrow e9; e8 \downarrow e11 \wedge e12$

If it occurs, then the current speed and the maintained speed are compared.

e11 and e12 will use its value if the CC system starts working.

e₉: COMPARE C-SPD AND M-SPD

Definition: The CC system processes the current speed and the maintained speed in order to adjust the throttle.

Value:

Behavior With

Preceding Set: e₈, e₁₀;

$$e_8 \wedge e_{10} \leftrightarrow e_9$$

e₉ uses the data from e₈ and e₁₀ iff e₈ and e₁₀ are being processed.

Succeeding Set: e₁₃.

$$e_9 \leftarrow e_{13}$$

It is necessary for the CC system to control the throttle. ($\neg e_2' \wedge e_9 \leftrightarrow e_{13}$).

e₁₀: QUERY M-SPD (1)

Definition: The CC system queries the maintained speed in order to adjust the throttle.

Value: {Null, [5 mph, 110 mph]}

Behavior With

Preceding Set: e_{1'}, e₁₂;

$$e_1' \leftrightarrow e_{10}; e_{12} \rightarrow e_{10}$$

If e₁₂ occurs then e₁₀ will query the new value.

It keeps querying iff the CC button is in the "on" state got into with e_{1'}.

Succeeding Set: e₁₁, e₁₂, e₉;

$$e_{10} \leftarrow e_{11} \wedge e_{12}; e_{10} \leftrightarrow e_9$$

If the state which was caused by e₁₀ keeps going, then the comparison of the current speed and the maintained speed will keep going.

If its value is Null, then e₁₁ and e₁₂ occur.

e₁₁: STORE L-SPD

Definition: The CC system pushes the maintained speed into a stack for the future resuming.

Value:

Behavior With

Preceding Set: e₈, e₁₀, e₁₈, e₂₁;

$$e_{10} \oplus e_{18} \oplus e_{21} \leftarrow e_{11}; e_8 \downarrow e_{11}$$

If e₁₀ is Null or e₁₈ is within the speed limit or e₂₁ occurs, then it happens. But the last stored speed has to be only one at a time.

If e₁₀ is Null, the last maintained speed is e₈'s value.

Succeeding Set: $\neg e_1'$, e_{2'}, $\neg e_2'$, e_{3'}, e_{4'}, e_{5'};

$$e_{11} \downarrow (\neg e_1' \oplus e_5') \vee (e_2' \oplus \neg e_2') \vee (e_3' \oplus e_4')$$

All elements in *stimuli* except e_{1'} may follow e₁₁. The e₁₁ is a termination of some causality chain. After the occurrence of e₁₁, the system is in some states except the scenario boundary state.

e12: RECORD M-SPD

Definition: CC system determines the maintained speed for the querying.

Value: [5 mph, 110 mph]

Behavior With

Preceding Set: $e_8, e_{10}, e_{18}, e_{22}$;

$e_{10} \oplus e_{18} \oplus e_{22} \leftarrow e_{12}; e_8 \downarrow e_{12}$

If e_{10} is Null or e_{18} is within the speed limit or e_{22} occurs, then it happens. But the current maintained speed has to be only one at a time.

If e_{10} is Null, the current maintained speed is e_8 's value.

Succeeding Set: e_{10} ;

$e_{12} \rightarrow e_{10}$

If a maintained speed is newly determined, that speed should be immediately used by CC system.

e13: ADJUST THROTTLE

Definition: The CC system adjusts the throttle.

Value:

Behavior With

Preceding Set: $\neg e_2', e_9$;

$\neg e_2' \wedge e_9 \rightarrow e_{13}$

If the brake and the accelerator are off, the state got into with e_{13} keeps going.

Succeeding Set: $\neg e_1', e_2', e_3', e_4', e_5', e_{17}''$;

$e_{13} \downarrow (\neg e_1' \oplus e_5') \vee e_2' \vee (e_3' \oplus e_4') \vee e_{17}''$

If a maintained speed is newly determined, that speed should be immediately used by the CC system.

e14: CLEAN S-SPD AND M-SPD

Definition: The CC system clears the record maintained speed and stored maintained speed. A sink point (system goes back scenario boundary state).

Value:

Behavior With

Preceding Set: $\neg e_1'$;

$\neg e_1' \leftrightarrow e_{14}$

Succeeding Set: (N/A).

e15: CHECK CC (1)

Definition: CC system's state is checked for e_3' sake.

Value: CC on, CC off.

Behavior With

Preceding Set: e_3' ;

$e_3' \leftrightarrow e_{15}$

If driver pushes inc/dec button, it occur.

Succeeding Set: e_{16}, e_{17}'' ;

$e_{15} \leftarrow e_{16} \oplus e_{17}''$

If its value is CC on, then e_{16}

else go scenario boundary state (e_{17}'').

e_{16} : QUERY M-SPD (2)

Definition: The maintained speed is queried for inc/dec sake.

Value: [5 mph, 110 mph]

Behavior With

Preceding Set: e_{15} ;

$e_{15} \leftarrow e_{16}$

If e_{15} gets value CC on, it occurs

Succeeding Set: e_{18} ;

$e_{16} \leftrightarrow e_{18}$

The maintained speed limit is checked iff it occurs.

e_{17} : DO NOTHING

Definition: The system goes back scenario boundary state.

Value:

Behavior With

Preceding Set: e_{15}, e_{18}, e_{20} ;

$e_{15} \oplus e_{18} \oplus e_{20} \leftarrow e_{17}$

Succeeding Set: (N/A).

e_{18} : CHECK SPD LIMIT

Definition: To check if the maintained speed ± 5 mph is out of the limit.

Value: True, False

Behavior With

Preceding Set: e_{16} ;

$e_{16} \leftrightarrow e_{18}$.

e_{18} occurs iff the maintained speed is queried for inc/dec sake.

Succeeding Set: e_{11}, e_{17}'' , e_{19} ;

$e_{18} \leftarrow (e_{19} \wedge e_{11}) \oplus e_{17}$

If its value is False, then do nothing

else store the maintained speed and inc/dec the maintained speed.

e_{19} : INC/DEC M-SPD

Definition: Let the maintained speed ± 5 mph.

Value: [5 mph, 110 mph]

Behavior With

Preceding Set: e_{18} .

$e_{18} \leftarrow e_{19}$.

It implies that the speed limit is OK.

Succeeding Set: e_{12} .

$e_{19} \rightarrow e_{12}$.

Determine a new maintained speed with the e_{19} 's value.

e_{20} : CHECK CC (2)

Definition: The CC system's state is checked for e_4 ' sake.

Value: CC on, CC off.

Behavior With

Preceding Set: e_4' ;

$e_4' \leftrightarrow e_{20}$.

If driver pushes resume button, it occur.

Succeeding Set: e_{17}'' , e_{21} , e_{22} ;

$e_{20} \leftarrow (e_{21} \wedge e_{22}) \oplus e_{17}''$

If its value is CC off, then e_{17}''

else pope the maintained speed stack twice.

e_{21} : POP POP L-SPD

Definition: The CC system queries the element next to the top of the maintained speed stack.

Value: [5 mph, 110 mph]

Behavior With

Preceding Set: e_{20} ;

$e_{20} \leftarrow e_{21}$

It implies that e_{20} 's value is CC on.

Succeeding Set: e_{11} ;

$e_{21} \rightarrow e_{11}$

The top point of the maintained speed goes down one position.

e_{22} : POP L-SPD

Definition: The CC system queries the top of the maintained speed stack.

Value: [5 mph, 110 mph]

Behavior With

Preceding Set: e_{20} ;

$e_{20} \leftarrow e_{21}$

It implies that e_{20} 's value is CC on.

Succeeding Set: e_{12} ;

$e_{21} \rightarrow e_{12}$

Determine a new maintained speed with the e_{21} 's value.

e_{23} : MANUAL CONTROL

Definition: The CC system is in the manual control state (get into with e_{23}). The assumption of the e_{23} is that the CC system is still running.

Value:

*Behavior With**Preceding Set:* e_2' ;

$$e_2' \leftrightarrow e_{23}$$

It occurs iff the brake or the accelerator are depressed.

Succeeding Set: $\neg e_1', e_3', e_4', e_5'$.

$$e_{23} \downarrow (\neg e_1' \oplus e_5') \vee (e_3' \oplus e_4')$$

In the state started with e_{23} , the elements in the set $\{\neg e_1', e_3', e_4', e_5'\}$ are spontaneous and random. e_{24} : CHECK CC (3)*Definition:* The CC system's state is checked for e_5' sake.*Value:* CC on, CC off.*Behavior With**Preceding Set:* e_5' ;

$$e_5' \leftrightarrow e_{24}$$

It occurs iff the terrain condition changes.

Succeeding Set: e_{17}'' , e_6 ;

$$e_{24} \leftarrow e_6 \oplus e_{17}''$$

If its value is CC off, then e_{17}''
else the value of e_5' is sampled (e_6).

APPENDIX C
THE ED'S FORMAL REPRESENTATION AND
ADJACENCY MATRIX FOR
VENDING MACHINE SYSTEM (EXAMPLE 1)
AND
CRUISE CONTROL SYSTEM (EXAMPLE 2)

C.1 The Formal Representation of the Vending Machine's ED

The graphical representation is on Figure 3.

$$E = \{e_1', e_2', e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}'', e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}\},$$

$$S = \{e_1', e_2'\},$$

$$V = \{e_{10}''\},$$

$$SEO = I \cup C \cup T,$$

$$I = \{(e_1', e_3), (e_1', e_4), (e_2', e_7), (e_8, e_{11}), (e_{11}, e_{10}''), (e_9, e_{12}), (e_{12}, e_{10}''), (e_{13}, e_{15}), (e_{14}, e_{15}), (e_{16}, e_{18}), (e_{16}, e_{19}), (e_{17}, e_{20}), (e_{18}, e_{10}''), (e_{19}, e_{10}''), (e_{20}, e_{16})\},$$

$$C = \{(e_3, e_6), (e_3, e_5), e_4, e_8), (e_7, e_9), (e_7, e_{10}''), (e_9, e_{13}), (e_{12}, e_{14}), (e_9, e_{13}), (e_{15}, e_{16}), (e_{15}, e_{17})\},$$

$$T = \{(e_4, e_1'), (e_4, e_2'), (e_5, e_1'), (e_5, e_2'), (e_5, e_8), (e_6, e_1'), (e_6, e_2'), (e_6, e_8), (e_7, e_1'), (e_7, e_2'), (e_7, e_8), (e_{15}, e_1'), (e_{15}, e_2')\}.$$

C.2 The Formal Representation of the Cruise Control System's ED

The graphical representation is on Figure 5, 6, 7, 8, 9.

$$E = \{e_1', \neg e_1', e_2', e_3', e_4', e_5', e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}'', e_{15}, e_{16}, e_{17}'', e_{18}, e_{19}, e_{20}, e_{21}, e_{22}, e_{23}, e_{24}\},$$

$$S = \{e_1', \neg e_1', e_2', e_3', e_4', e_5'\},$$

$$V = \{e_{14}'', e_{17}''\},$$

$$SEO = I \cup C \cup T,$$

$$I = \{(e_1', e_6), (e_1', e_7), (e_1', e_{10}), (e_6, e_8), (e_7, e_8), (e_8, e_9), (\neg e_1', e_{14}'), (e_2', e_{23}), (e_3', e_{15}), (e_{16}, e_{18}), (e_{19}, e_{12}), (e_4', e_{20}), (e_{21}, e_{11}), (e_{22}, e_{12}), (e_5', e_{24})\},$$

$$C = \{(e_{10}, e_9), (e_{10}, e_{12}), (e_{10}, e_{11}), (e_9, e_{13}), (\neg e_2', e_{13}), (e_{15}, e_{16}), (e_{15}, e_{17}''), (e_{18}, e_{11}), (e_{18}, e_{17}''), (e_{18}, e_{19}), (e_{20}, e_{21}), (e_{20}, e_{22}), (e_{20}, e_{17}''), (e_{24}, e_6), (e_{24}, e_{17}'')\},$$

$$T = \{(e_8, e_{11}), (e_8, e_{12}), (e_{11}, e_3'), (e_{11}, e_2'), (e_{11}, \neg e_1'), (e_{11}, \neg e_2'), (e_{11}, e_4'), (e_{11}, e_5'), (e_{13}, \neg e_1'), (e_{13}, e_2'), (e_{13}, e_3'), (e_{13}, e_4'), (e_{13}, e_5'), (e_{23}, \neg e_1'), (e_{23}, \neg e_2'), (e_{23}, e_3'), (e_{23}, e_4'), (e_{23}, e_5')\}.$$

REFERENCES

- [Wang 91] Wei Wang, "An Application-Specific, Scenario-Driven, Object-Oriented Software Development Technique", MS-Thesis, UTA, 1991.
- [Hufnagel & Harbison 93] Stephen P. Hufnagel & Karan Harbison, "Health Care Mega-System Analysis & Design: Using DSSA Scenario-Based Engineering Process (SEP)", Health Care Information System Proposal, UTA, 1993.
- [Hufnagel & Liou 90] Hufnagel S. and T. Liou, "Seamless Objected-Oriented Specifications, Design, Design and Implementation Method (SOSDIM)", Midcon 1990, Dallas, Texas, September 1990.
- [Pressman 92] Roger S. Pressman, "Software Engineering: A Practitioner's Approach", McGraw-Hill Inc., NY 1992.
- [Rumbaugh 91] James Rumbaugh, et al., "Object-Oriented Analysis and Design" Prentice Hall, Englewood Cliffs, NJ 07632, 1992.
- [Larry & Sanford 90] Larry Nyhoff & Sanford Leestma, "Data Structures and Program Design in Modula-2", Macmillan Publishing Company, New York, 1990.
- [Thomas 90] Thomas H. Cormen & et al., "Introduction to Algorithms", McGraw-Hill Book Company, 1990.
- [Bondy & Murty 77] J. A. Bondy & U. S. R. Murty, "Graph Theory with Applications", American Elsevier Publishing Co. INC, New York, 1977
- [Coad & Yourdon 91] Peter Coad & Edward Yourdon, "Object-Oriented Analysis", Prentice Hall Building, Englewood Cliffs, NJ 07632, 1991.

- [Miller 56] Miller G. A., "The magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *Studies in Long Term Memory*, edited by A. Kennedy. Wiley, 1975.
- [Tadao 89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", *Proceeding of the IEEE*, VOL. 77, NO. 4, April 1989.
- [James 81] James L. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall, INC., Englewood Cliffs, NJ 0762, 1981.
- [Wolfgang 92] Wolfgang Reising, "A Primer in Petri Net Design", Springer-Verlag Berlin Heidelberg New York, 1992.
- [Shlaer & Mellor 88] Shlaer, S. and Mellor, S. "Object-Oriented System Analysis: Modeling the World in Data", Englewood Cliffs, NY, Yourdon Press, 1988.
- [Ross 87] Ross, R., "Entity Modeling: Techniques and Application", Boston, MA, Database Research Group, 1987.
- [Jonathan & W 90] Jonathan S. Ostroff & W. Murray Wonham, "A Framework for Real-Time Discrete Event Control", *Transactions on Automatic Control*, IEEE Volume 35, Number 4, April 1990.
- [Jonathan 92] Jonathan S. Ostroff, "Formal Methods for the Specification and Design of Real-Time Safety Critical Systems", *The journal of Systems and Software*, April 1992.
- [Sudkamp 88] Thomas A. Sudkamp, "Languages and Machines: An Introduction to the Theory of Computer Science", Addison-Wesley Publishing Company, Inc., 1988.