INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa • University of Ottawa

Abstract

Modern software design tools use finite-state machines (FSMs) arranged in hierarchical fashion. Many techniques have been developed for testing software modelled as an FSM, but none explicitly addressing designs modelled as a hierarchical FSM (HFSM). Additionally, the problem of explosion in the number of test paths precludes the testing of all possible paths through the HFSM [Holzmann 91].

This thesis presents a practical and scaleable method for testing a design modelled as an HFSM. The method is based on graph traversal and uses the hierarchy of the underlying directed graph. A recursive algorithm computes *breadth*, the number of paths needed just to cover all transitions at least once. This idea is extended to cover all states, all inputs, and all outputs. This method is complementary to existing formal methods for conformance testing and protocol testing. Breadth is a lower bound on the number of test paths to cover the HFSM.

Keywords: hierarchical finite-state machine; FSM; software design; statecharts; metrics

A Testing Metric for Designs Modelled as Hierarchical Finite-State Machines

by

Ning Lew

Thesis submitted to
School of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master's of Computer Science

under the auspices of the
Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario

December 2000

Table of Contents

List of Figures and Tables (2001-01-14)

1

Introduction (2001-01-15)

1.1 Context and motivation

In the realm of software quality engineering, software testing remains a challenge with respect to two major considerations: techniques (formal and informal methods) and resources (time and money). Typically, resources are fixed or limited by factors beyond the control of the test engineer. Therefore the key is to employ techniques that make the most efficient use of what is available. Herein lies the essential test management challenge: finding cost-effective strategies for testing designs and code.

This thesis considers systems whose designs are modelled by communicating extended hierarchical finite-state machines (FSMs). These terms will be described further in Chapter 2. For now, suffice it to say that for the general case of (non-hierarchical) communicating extended FSMs (CEFSMs), one of the main problems in testing is the classic "state space explosion problem" [Holzmann 91]. Specifically, the number of reachable states for a reasonably sized system is too large to permit analysis of each state. Since thorough and complete solutions are generally not feasible, we seek ways of obtaining a practical amount of coverage and a useful metric to describe this.

Holzmann's approach is to make the computerised mechanical implementation of state space search very efficient, as demonstrated by his Supertrace algorithm for verifying computer protocols, through efficient coding, compression, and representation [Holzmann 91]. Another general approach is to "prune the testing space," that is, to reduce or relax the requirement for how much testing is to be done and to settle for a more realistic level of coverage by selecting a representative sample.

To reduce the amount of testing, we must choose either to remove areas that we feel are not worth testing or to concentrate on those that demand extra attention. From the point of view of risk management, priority should be given to whatever testing is deemed the most essential, for example, critical scenarios [Karolak 96]. These scenarios need to be identified manually by someone knowledgeable in the customer's business or application. Although a wise tester would never want to bypass such analysis, having some simple automated methods with coverage metrics would make testing more amenable to non-expert testers. The metric and approach proposed in this thesis promote such ease of testing and are advocated for designs modelled as hierarchical finite-state machines (HFSMs), which are summarily explained in sub-section 2.1.4. This overall approach also provides a complementary strategy to the High-Yield testing approach [Myers 79] [Probert 95].

Problem definition

Situation: Modern real-time systems are often designed using hierarchical finite-state machines. General design methods typically consist of multiple parts (often supported by tools):

- One part describes the system behaviour, using FSMs, HFSMs, or some variant thereof.

- Another part describes the system structure, particularly in the case of object-oriented modelling. Descriptions are produced for object classes, relationships, roles, sub-classes, etc.

- A potential third part may include a *reflective* language, a supplementary aspect that captures the rationale behind the development of the model and aids in analysing the model. Examples: message sequence charts (or time sequence diagrams) which provide specific scenarios of the system's behaviour [Harel 97].

The design methods under consideration in this thesis are HFSM-based ones such as Harel's *Statecharts* [Harel 87] and Real-time Object-Oriented Modelling (ROOM) *ROOMcharts* [Selic 92]. A precise definition of the HFSM subset is given in Appendix 8.

By way of explanation, ROOM is an object-oriented software modelling methodology that uses CEFSMs to represent the behaviour of actors. As is the case for statecharts, a key difference between ROOM models and those for other FSM-based software (for example, telecommunications protocol software) is that the FSMs in ROOM are arranged hierarchically. In addition to using HFSMs, ROOM provides for some extensions in aspects such as how input stimuli (triggers) are implemented and where (in the FSM) the code is permitted to be located. The ROOM variant of the HFSM is called a *ROOMchart* [Selic 92].

General problem: During the design phase, we would like to test an HFSM model such as those described above, and we would like some useful information to help direct our testing efforts in a practical manner. Note that it is the design model that is to be tested, and not the code itself. In general, "testing the model" includes whatever is necessary to increase our confidence in the design and can include doing the following: validating the design against the original user requirements; verifying traceability of the requirements to the design and of the design to the code; focussing on the real-time or reactive aspects of the system modelled; examining the object classes and the objects chosen; verifying the data flow; examining the structure, the interfaces, the behavioural aspects, and so forth [Probert 95].

Statecharts and ROOMcharts are both based on HFSMs. Although there are well-established and formal methods for testing software modelled as FSMs, there are no well-established means of testing designs modelled as HFSMs and consequently no commonly

used metrics. Therefore, we conclude that it would be appropriate to try incorporate traditional FSM-based methods but also to specifically address the hierarchical apsect. We wish to examine the underlying HFSM basis of both of these behaviour notations in order to derive a useful metric.

Specific problem: This thesis addresses the specific problem of deriving a metric to give us a lower bound on the number of paths required to cover the HFSM. This metric is denoted the *breadth* of the HFSM.

Additional requirements – scaleability & automatability

We require an automated way of providing an assessment of what can be covered. As for any real-world application, such an approach must satisfy the following two requirements:

Requirement R1. Be able to "scale up";

Requirement R2. Be able to be automated

We will show in Chapter 5 how our approach satisfies these requirements.

Related but different problems

Because of the FSM structure of the HFSM, there is potential for confusing the problem and approach of this thesis with well-known problems in *conformance testing* and *machine identification*. The approaches to these problems are discussed in Section 2.4, "FSM-based test-generation methods." Machine identification proceeds by identifying states and transitions in the code based on observable responses to test sequences. In general, these methods are applied, black-box style, to program code that purports to implement a desired FSM-based specification. Such methods are intended for the latter stages of software development.

In contrast to machine identification methods, identifying the state-machine in this thesis is not an issue because the design machine model itself is completely visible to us. Our method is particularly applicable at the requirements and design stages of system development. The tool can be used to compute *breadth* to give an ongoing indication of the complexity of the design, as the design progresses. The tool can be used to select from amongst alternative design possibilities, since we feel that there is some correlation between *breadth* and testing effort. We see our approach as complementary to traditional FSM-based methods because they apply in the latter part of the development cycle (implementation and conformance testing).

Addressing other attributes of HFSMs such as object orientation, real-time performance, and requirements traceability would also be desirable, however they are considered out of scope for this study. Also, our analysis is concerned with only control flow and not data flow.

1.2 Objective

The objective of this thesis is to present a practical and scaleable approach for producing a testing metric, namely ***breadth***, to bound from below the number of covering paths for designs whose control-flow (behaviour) is modelled as a hierarchical finite-state machine.

Because the number of covering paths is representative of the number of test cases, we use the *breadth* as a measure of the test effort associated with the design of the software. Furthermore, we also take *breadth* as an indicator of the complexity of the design. It is recognised that there is no universally accepted definition of complexity, but most practitioners of software engineering have an intuitive feeling for what it is and what measures could be used as metrics or estimators. For example, the widely accepted McCabe complexity metric is shown in Chapter 5 to be bounded from below by our breadth metric. A discussion of complexity follows in Section 1.3. A thumbnail sketch of the complexity

problem that we are facing is given in Section 3.1.3. A brief survey of metrics appears in Section 2.5.

Graph-theoretical approach

Intuitively, a strategy that takes advantage of the hierarchical structure is more appealing than one that does not. As described in Chapter 2, there are no existing methods that explicitly address testing of HFSMs, however there are many graph algorithms and heuristics that can be used to address or solve a wide variety of related problems.

It is clear that many of the traditional FSM-based testing strategies (Section 2.4) draw heavily from methods in combinatorics, graph theory, and network algorithms. For example, the Transition Tour method and finding Euler circuits both deal with covering all the edges in a graph. Covering all the nodes in a graph is done by finding a Hamiltonian circuit – the traditional "travelling salesman" problem. We take advantage of the relationship between FSM-based testing and graph theory, in particular, network flow methods, to obtain a lower bound on the number of covering test cases.

1.3 Main idea of our approach

The proposed method takes advantage of the hierarchy to reduce the amount of testing required. We start with the basic goal of traversing all transitions, and extend this to account for additional features in statecharts and ROOMcharts (the behavioural portion of ROOM). Our basic recursive algorithm provides a simple method of counting how many paths are required to traverse each transition in an HFSM at least once. With appropriate interpretation of the input format, the algorithm has been extended to account for some of the modelling features in ROOM that are used to supplement the HFSM. The final count provides a measure of the amount of design-level testing that is required. This metric is a lower bound in that

(a) some of the paths inferred may turn out not to be feasible, therefore additional (user-specified) paths would be required to make up for the infeasible combinations, and

(b) using fewer test cases than our metric would leave some transitions uncovered.

In addition to determining the minimum number of paths to cover the graph, our metric also estimates the complexity of the design – the higher the number, the more complex. This interpretation of complexity is consistent with the approach taken by McCabe, who is known for his work in metrics [McCabe 82]. Metrics are discussed in Section 2.5. The relationship of our metric to McCabe complexity is discussed in detail in Section 5.3.2.

It is worth noting the incidental but additional benefit of counting test paths: we also obtain a first approximation to a covering set of test paths (acknowledging that some paths may be infeasible). McCabe also touts this as an offshoot benefit of his metric for cyclomatic complexity and advocates use of manual and automated methods to derive sets of test paths [McCabe+ 94]. By constructing test paths at the design level, we are actually testing the design by exercising design scenarios. This enables us to detect design flaws at design time, thus avoiding later, more expensive rework.

Finally, it should be emphasized that the problem of dealing with infeasible paths also occurs with all FSM-based testing methods because the FSM depicts only the control flow of the system. It is not possible to determine from the FSM alone what data values in what combinations are valid along each control path. Therefore the presence of infeasible paths should not be seen as an obstacle particular to our approach.

1.4 Contributions of this thesis

The main contribution of this thesis is a new and simple graph-traversal approach, based on network flow theory, to bound from below the test coverage requirements of designs modelled as hierarchical finite-state machines. The resulting *breadth* metric gives the

minimum number of test cases required to cover all states, all transitions, all inputs, and all outputs (provided that they are visible in the graph), and is at least as indicative of the complexity of the designs as are widely used metrics. In addition to the breadth metric for coverage and complexity, we also provide some algorithms to derive possible paths for use as test cases. (These paths are subject to the standard concerns regarding infeasible paths in the system.) As we will see, our approach can be scaled up and is amenable to automation. It is also complementary to the more well-known formal methods for FSM-based software testing. Thus, *breadth* is a true lower bound (as compared to McCabe's metric) on any possible covering test path set.

1.5 Organisation of thesis

Chapter 2 provides a review of some of the definitions and terminology used when modelling systems with finite automata. An overview of statecharts is provided, along with a brief description of the variant called ROOMcharts. Software testing methods are reviewed, with particular attention to FSM-based methods. The concept of metrics is introduced, particularly with respect to software complexity.

Chapter 3 presents and discusses our new strategy for estimating the complexity of testing designs modelled as an HFSM. As mentioned, the strategy computes the breadth of the HFSM as a means for estimating the testing effort and the design complexity. A brief review of relevant graph theory terms is also provided in sub-section 3.2.

Chapter 4 accommodates enhancements that are particular to ROOMcharts. These ROOMchart features are extensions to the basic HFSM concepts, and are partly rooted in the origins that ROOMcharts have in Harel statecharts (which are introduced in Section 2.2).

Chapter 5 provides an assessment of the approach used and puts it in the perspective of general software testing approaches.

Chapter 6 sets out a few extensions or directions that can be further pursued.

Chapter 7 summarizes the ideas and findings in this thesis, and closes with some general concluding remarks.

The appendices provide details on the algorithms relevant to this thesis. As well, the main example is contained in Appendix 5. Source code is in Appendix 6.

2

Background and Related Work (2001-01-15)

2.1 Introduction to finite-state machines and modelling

Finite-state machines (FSMs) are widely used to model the behaviour of systems, especially of communication protocols. This section reviews the terminology and notation used for FSMs, and shows how they are used to model software.

2.1.1 Basic FSM

The most basic FSM consists of a set of states, S, a set of possible inputs, I, and a transition function, TR, that tells which new state (possibly the same state) to adopt when presented with a combination of state and input; that is, $TR : S \times I \rightarrow S$. It is also normal to designate one state as the initial or starting state. A simple extension to this basic FSM is to add a set of outputs, O, to be used by the transition function; $TR : S \times I \rightarrow S \times O$.

More formally, an FSM, *"M"*, can be defined [Hopcroft+ 79] as a 5-tuple:

$$M = (S, I, O, TR, s0)$$

where

S is a finite set of symbols denoting states,

I is a set of symbols denoting the possible inputs,

O is a set of symbols denoting the possible outputs.

TR is a transition function mapping $S \times I$ to $S \times O$.

$s0 \in S$ is the initial state.

A sample FSM is shown in Fig. 2-1. Here, $S = \{s0, s1, s2\}$, $I = \{x, y\}$, $O = \{0, 1\}$, TR is given by the table or the diagram, and the initial state is $s0$.

	Input	
State	***x***	***y***
s0	0, *s1*	1, *s0*
s1	1, *s1*	1, *s2*
s2	1, *s0*	0, *s1*

Each table entry shows what symbol is output and which new state is to be adopted.

Fig. 2-1a. *A sample finite-state machine. Tabular form.*

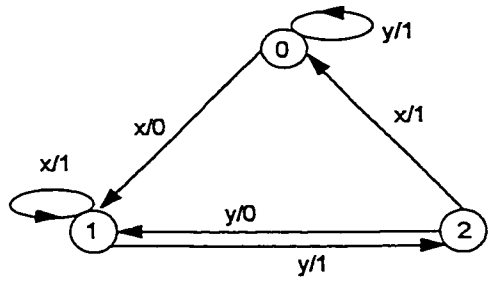


Fig. 2-1b. *Graph form. The label "x/0" means that "when x is input, 0 is output."*

An FSM can be specified in tabular form, as in Fig. 2-1a, or as a directed graph, as in Fig. 2-1b. The directed graph is denoted as $G = (V, E)$ in which the states $\{s1, s2, s3, ..., sn\}$ are represented by nodes or vertices, $V = \{v1, v2, v3, ..., vn\}$, and each transition from state si to state sj, invoked by input ip and generating output oq, is represented by a directed edge (vi, vj) with an input/output label ip/oq. Each edge on the graph will bear a label representing the input and output for that edge.

From a modelling standpoint, the inputs represent the various actions or stimuli to which the system may be subjected. The outputs represent the resultant actions or observable behaviours on the part of the system. An example with no outputs is shown in Fig. 2-2, modelling just the "push" actions in a stack-type data structure.

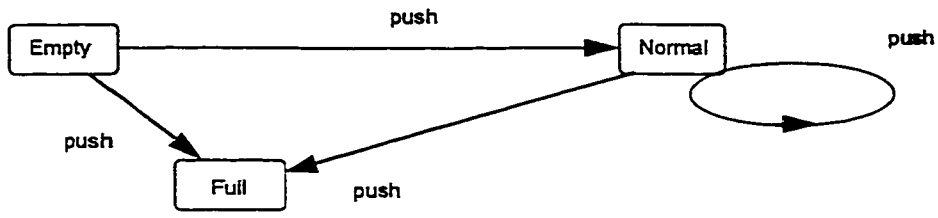


Fig. 2-2. *Partial model of a stack using a basic FSM with no outputs.*

2.1.2 Extended FSM

An extended FSM (EFSM) is an FSM that has been extended to include variables that can be "written" (set), often during an output action, and later "read" (tested), often in conjunction with an input action. We also include two extra types of actions: boolean conditions on variables and assignments of values to variables [Holzmann 91]. The scope of an EFSM variable may be local to the state in which it was defined, globally accessible to all states, or accessible only to the defining state and all its contained substates, depending on the convention being used. Such variables may be used as counters, may represent timers, or may represent external conditions such as time-of-day or real-time sensor information like temperature or voltage. Test strategies for EFSMs require a data-flow coverage component in addition to the normal control-flow portion of a regular FSM (but this is not part of this thesis) [Sidhu 90].

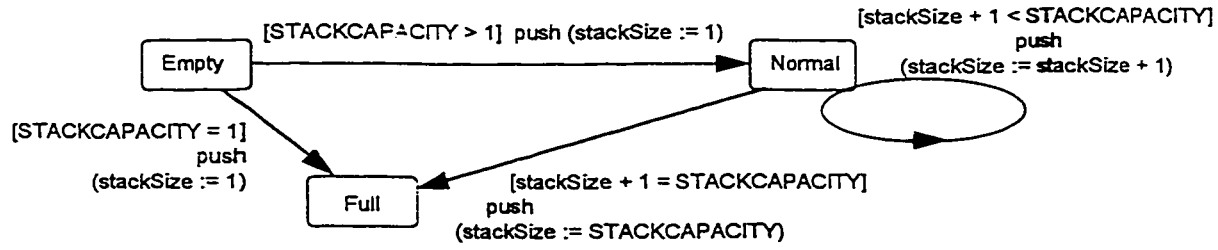


Fig. 2-3. *Partial model of a stack using an EFSM.*

Fig. 2-3 shows an example of a 3-state EFSM with a variable called stackSize and a "constant variable" STACKCAPACITY. The two variables are used in the *guard* conditions, (shown in square brackets) preceding the transition name as well as in the additional *actions* (shown in parentheses) following the transition name.

An extended finite-state machine can be defined as

$$M = (S, I, O, TR, s0, A)$$

where S, I, O, TR, and $s0$ are defined as before, and A is the set of variable names.

2.1.3 Communicating FSM

A communicating FSM (CFSM) is an FSM that can send messages to another by way of a communication channel. Accordingly, the action related to an FSM transition may be a "send" or "receive" action. A channel may be thought of as a first-in first-out (FIFO) queue; all channels are therefore unidirectional (either send or receive). In terms of the basic FSM, it is as though we now permit the simultaneous execution of multiple FSMs, where the inputs and outputs are now funnelled into queues.

Different types of CFSM can be defined. Here are some common points for variation:

(1) whether the FSM has a pair of channels for every other FSM or only a single input queue to be used by all other FSMs;
(2) whether the maximum queue capacity is zero, infinite, or somewhere in-between; and
(3) whether communication channels are assumed to be reliable or should provision be made for messages being lost.

Fig. 2-4 shows an example with two CFSMs communicating by way of a single pair of channels between them [Ural 94].

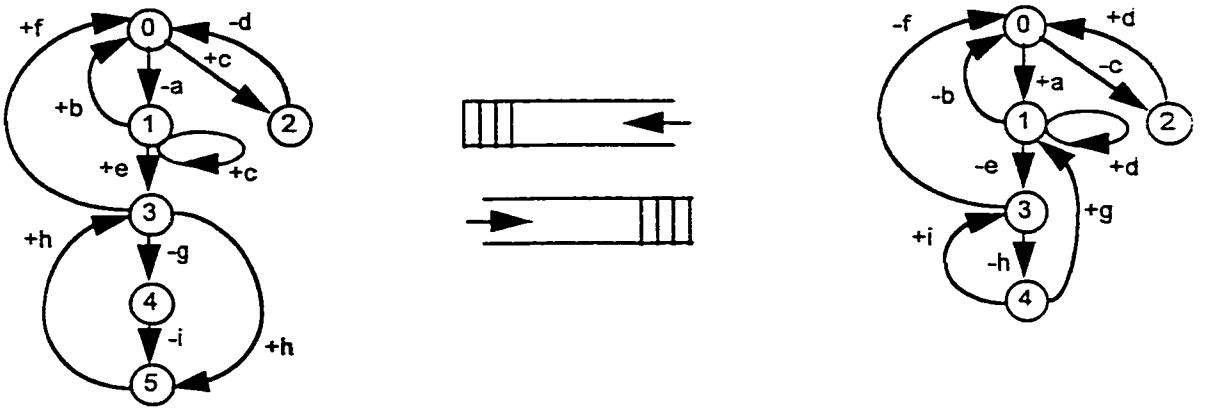


Fig. 2-4. *An example of CFSMs communicating through a single pair of channels.*

The notation "*-a*" on a transition means that message *a* is being transmitted by this CFSM; similarly, "*+a*" means that message *a* is being received.

Note: The communication protocol in this particular example happens to be incompletely specified; for example, although the first state diagram describes what happens in State 0 when it receives message *c* (it goes to State 2), it does not describe what would happen if it were to receive an *a*, *b*, or any other message. Other CFSM problems are discussed in subsection 5.4.4 on Reachability. Fig. 2-4 serves merely as an introductory illustration to the CFSM concept.

Holzmann defines a communicating finite-state machine as a tuple [Holzmann 91]:

$$(S, TR, s0, MQ)$$

where *S*, *TR*, and *s0* are defined as before, and

MQ is a set of *message queues*, each of which is a triple *(QV, NS, QC)*

where

QV is a finite set called the *queue vocabulary*,
NS is an integer that defines the number of slots in that queue,
QC is the queue contents, an ordered set of elements from *QV*.

Notice that what we referred to earlier as the set of inputs, *I*, and the set of outputs, *O*, are now part of the queue vocabulary, *QV*; the elements of *QV* are called *messages*. The action of a basic FSM input occurring is now the *receipt* of an item from a queue; the action of an output occurring is now the *transmission* of an item to a queue. A *system vocabulary*, *SV*, can be defined as the conjunction (the set union) of all queue vocabularies.

2.1.4 Hierarchical FSM

FSMs can be combined in a hierarchical fashion. In this manner, a single state may consist of many sub-states that are related to each other in a (lower-level) FSM. Such a state is said to be *hierarchical state* or a *composite state* – one that contains a lower-level state machine or *sub-machine* [Selic++ 94]. Extending this thinking upwards instead of downwards, an entire FSM may be used to represent the internal behaviour of a larger super-state. In hierarchical models, a single layer of the model represents only a single level of abstraction. When modelling at different levels, it is important to note such things as the scope of any variables being used and which state should be assumed when a previously active FSM is re-entered (that is, whether it takes the last state upon the previous exit or always takes the initial state) and what values the variables will have. Fig. 2-5 shows an HFSM structure.

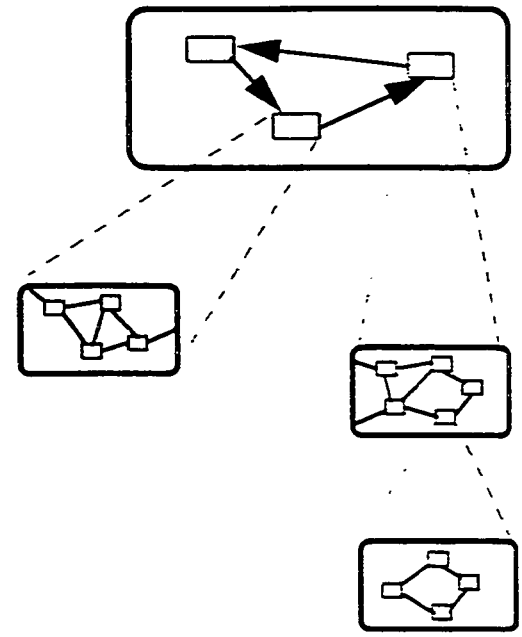


Fig. 2-5. *Structure of HFSM.*

2.1.5 Sample FSM model

The modelling of systems using FSMs can best be seen with a few examples. Recall that it is the externally visible behaviour that will be modelled. If it is necessary to model internal behaviour, then another "layer" of FSM can be used within a given state. It should be noted that there are other structured modelling techniques, many of which tend to be based on data flow and data-flow diagrams. Fig. 2-6 shows an FSM model representing the behaviour of just the transmission portion of a simple fax machine (the "FPS 2000" from [Probert 94]).

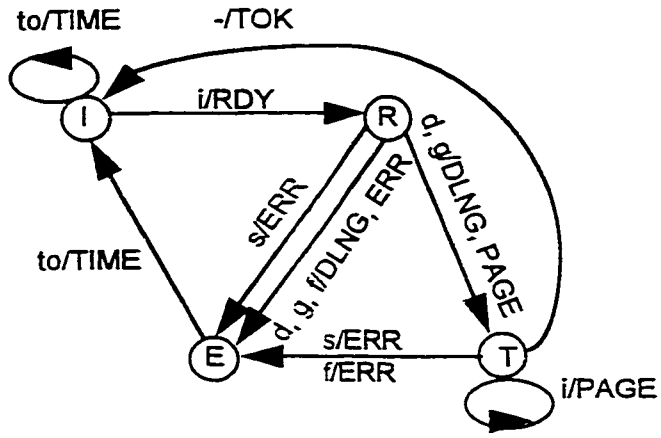


Fig. 2-6. *FSM for a simple fax machine (transmission portion only).*

Notation used in Fig. 2-6

States = {I, R, T, E}	**Input events = {i, d, g, s, to, f}**	**Output responses = {TOK, TIME, PAGE, DLNG, ERR, RDY}**
I = Idle	i = insert page	TOK = "Transmission okay"
R = Ready to transmit	d = dial destination number	TIME = (Time of day)
T = Transmitting	g = press "Go" button	PAGE = "Transmitting page"
E = Error	s = press "Stop" button	DLNG = "Dialling"
	to = timeout (5 seconds elapsed)	ERR = "Error"
	f = failure of the attempted or pending action	RDY = "Document ready"

2.1.6 FSM assumptions

Regardless of which of the above FSMs is used, certain properties generally hold when applying formal methods. First of all, it is assumed that the FSM is *deterministic* in that, at a given state, there cannot be more than one possible transition to take for any given input; thus, no two outgoing edges can have the same input label at any vertex. For modelling purposes, this implies that the results are predictable and repeatable.

Another common assumption is that the transition function must be *fully specified*; that is, there is a known result for every combination of state and input. If not fully specified, it can be made so (implicitly or explicitly) by declaring that any transitions not specified will be treated as though it were a transition back to the same state and with a "null" action.

An FSM shall also be *minimal* meaning that there exists no other FSM that is *equivalent* yet has fewer states; two FSMs are said to be equivalent if both FSMs produce identical output sequences for every input sequence.

Finally, it is assumed that the FSM graph is *strongly connected*, meaning that it should be possible to (eventually) reach every state from any given state; that is, all states are *reachable*. In practice, this also includes the additional restriction that there are no "trap" states, dead-end states from which the system cannot recover, thereby causing all other states to be non-reachable.

Commercial modelling and testing tools may or may not incorporate all of these assumptions.

2.1.7 What constitutes a state

The most basic issue in using state-machine modelling does not have an obvious answer, specifically, "What is a *state*?" If we were to model the use of a telephone, we might consider the set of states to be OnHook, DialTone, BusySignal, RingingSignal, and Talking. If actions are allowed to occur only during transitions, then the granularity of states may be quite fine. If a state may represent an entire phase of activities, then the granularity would be rather coarse. Clearly, the level of detail in the model will influence what is declared a state. A discussion in [Beizer 90] covers ideas on what kinds of states a "good" state graph should have.

The variables used in EFSM models may or may not affect which transition is taken and consequently which state is next entered. Variables that simply record something about the history of a process (for example, the number of coins dropped into a vending machine) can be eliminated by creating separate states to represent the intermediate stages of process progress. Elimination of all variables simplifies testing by permitting the use of standard (non-extended) FSM-based techniques, however this is not always practical.

In addition to deciding how communication is to be handled, CFSMs raise the issue of how states should be interpreted. From a local viewpoint, each CFSM makes transitions through its own set of states. The question in the mind of a tester is whether it is possible to determine the outcome of a test of a distributed system by examining the behaviour at each CFSM separately. Or, is there a need to verify that certain sequences of interleaving events occurred in a specific relative order? The concept of a *global state* is used to record and represent something regarding the history of the distributed system as a whole. The most common definition for a global state, applicable to the CFSM model in Fig. 2-4, is the combination of each local state along with the sequences of messages that are pending in all the queues.

Many formal languages have been developed for modelling distributed systems. International efforts include SDL (Specification and Description Language), Estelle [ISO 89, Diaz++ 89], and LOTOS (Language of Temporal Ordering and Specification) [ISO 8072]. The ISO and ITU have collaborated in producing sets of standards. Two varieties of SDL are currently in service: SDL-88 [CCITT 89] and SDL-92 [ITU-T 94]. In this thesis, the modelling languages are Statecharts and ROOM, both discussed in the following section.

2.2 An overview of Harel statecharts and ROOM

This section provides an overview to the visual formalism called Statecharts and draws significantly from Harel's main 1987 paper [Harel 87]. Statecharts are an extension of conventional state machines and state diagrams. They are intended to be a visual but formal way of representing behaviours in reactive (or "event-driven") systems. The main extensions involve the notions of *depth* (hierarchy) and *orthogonality* (concurrency and having composite states). We will briefly discuss the motivation for statecharts, explain the basic notation and summarise the main advantages of statecharts over conventional state transition diagrams.

2.2.1 Background and motivation for statecharts

It has already been shown that finite-state machines are used for modelling systems. Many systems are "data driven" in that the main activity of the system is to wait for input data, process (or *transform*) the data, send results somewhere, and then revert to an idle or ready state until further inputs arrive. Such *transformational* systems are neatly represented by data-flow diagrams. More difficult to depict are the many systems that are "event driven," also called *reactive* systems or "control driven" systems. Examples include telecommunication systems, industrial plants, control systems, and even desktop software, where the system is reacting to stimuli that are provided by way of button presses, switches, sensors or other monitoring devices, keypad or keyboard.

One basic and obvious difficulty with conventional FSMs is that they are simply flat. When attempting to depict complex behaviour, the number of states increases quickly with even a modest increase in complexity. With the numerous transitions drawn in, flat FSM diagrams become large and cluttered very quickly, making it difficult for designers or modellers to use and understand.

Consider the simple example of modelling part of the user interface in a standard piece of word processing software [Horrocks 99]. Most word processors have three separate buttons for marking text as bold, italic, or underlined. Independently, each button can be either On or Off. It is clear that these three features are independent of each other and may be combined; as examples, text may be formatted with bold On, italics Off, underline On; or, all three Off. Conceptually, a designer and user will think of three small state machines, each with two states: On and Off (and transitions to go between the two). But with conventional FSMs, this requires eight separate states to model each of the eight combinations for the three switches, with many transitions going between the states. Clearly, the size of an FSM has the potential to grow exponentially with the complexity.

The other main difficulty with a conventional FSM lies in its sequential nature. Reactive systems often tend to be large and complex, and exhibit internal concurrent behaviour. They are typically designed in modular fashion, often with a "top-down" approach, for example, using subroutines in computer programming, or using known components in electronics design. This is a very natural and intuitive way for humans to think about things. Different levels of detail are reflected at different levels of description, documentation or modelling.

Orthogonality can be seen when considering the different combinations of switch positions on the dashboard of a car, the different statuses of systems on board a ship, or the different settings for hardware and software in a microcomputer. The concept of hierarchy can also be seen in the latter case. Modern operating systems and software applications use an arrangement of hierarchical menus for setting or adjusting configurations. As the user clicks on a pull-down menu or clicks on a tab, the user is presented with different choices or even additional menus. The point is that the settings are not only independent of one another, but that they are arranged hierarchically instead of as one large flat structure. For example, in the Microsoft Windows 95 "Control Panel," after selecting "System," we will still be offered a choice of the tabs "General", "Device Manager", "Hardware Profiles", and "Performance" (See Fig. 2-7.) And then under the tab "Device Manager", there will be categories of devices, underneath which there will be individual devices.

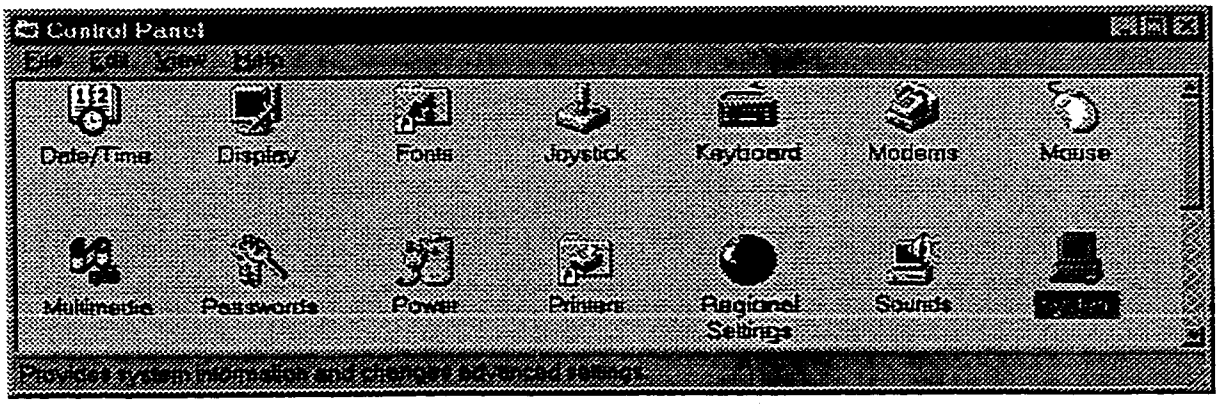


Fig. 2-7. *The Control Panel. Hierarchical composite states (from Microsoft Windows 95).*

In summary, conventional FSMs suffer from an exponential increase in the number of states, cannot be abstracted at different levels for designers and users, and are sequential and do not cater to concurrency. By comparison, statecharts are intended to be a visual, expressive, compact, and structured formalism for describing the control portion of a reactive system, including:

- the ability to cluster states into a superstate ("zooming-out")
- orthogonality (simultaneous state-components)
- ability to refine a state with lower-level states ("zooming-in")

Note that for modelling purposes, statecharts would still need to be combined with a physical (structural) and functional (scenario-based) portion. In addition to adding the notions of *depth* (hierarchy) and *orthogonality* (concurrency), statecharts also add *broadcast communications* as a mechanism for communicating between concurrent components. Harel summarizes this as:

statecharts = state-diagrams + depth + orthogonality + broadcast communications

In the next sub-section, we look at the basics of statechart notation to see how Harel extends the FSM notation.

2.2.2 Statechart notation

In his keystone paper [Harel 87], Harel's main instructional example is that of a multi-function, digital wristwatch. Fig. 2-8 shows the top-level representation of the watch. States are drawn as rectangles with rounded corners ("roundtangles"). The transitions are shown as labelled arrows, with a pre-condition specified in parentheses if applicable. Composite states are shown with dotted lines separating the concurrent components. As an example, Fig. 2-8 shows a composite state (the "Operational" state), formed by the main display AND alarm 1 status AND alarm 2 status AND the status of the light.

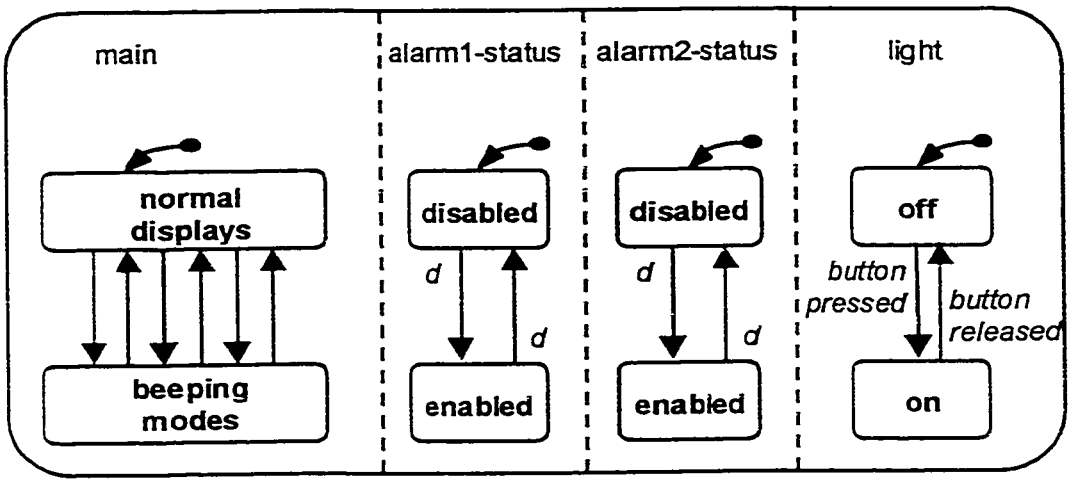


Fig. 2-8. *A composite state (in a wristwatch) comprising four orthogonal states. [after Harel 87]*

A small arrow from a round circle shows the *default* state, which is the state entered unless one is otherwise specified. The default state is analogous to the initial state, *s0,* in finite automata (as first described in sub-section 2.1.1).

Certain features of the statechart formalism are described in a later chapter:

- a "clustered state" with a *group transition* emanating from it, regardless of substate; (described in sub-section 4.1.1)
- a *top-level group transition*; for example, "when selection button is pressed, enter selected mode"; (described in sub-section 4.1.2)

- the *history* of a state, a virtual substate representing the actual substate that the state was last in (upon previous departure) so that incoming transitions may return to whatever the previously held substate was; (shown in sub-section 4.1.3)

- conditional entrances to a state, also called *choicepoints* (described in sub-section 4.1.4)

Other features of the statechart formalism include reliable (error-free) instantaneous broadcast communications; delays and timeouts; lower and upper bounds on timers; "parameterized states" (multiple instances of identical states; examples *alarm1* and *alarm2*, or "all the telephones in one exchange" 555-0001, 555-0002, ..., 555-000*i*, ...); and "overlapping states" (states that share some of their respective substates). The reader is referred to [Harel 87] for a description of these features and many others. Their mention here is simply to show that statecharts are a highly structured and expressive notation, and are particularly well-suited to the specification and design of complex discrete-event systems.

2.2.3 Main advantages of statecharts

The statechart notation has many desirable characteristics. The notation is very visually oriented and makes "specification by diagram" very plausible. It is amenable to being implemented in automated fashion (see next sub-section). And, it is more precise than a natural-language specification.

Compared with conventional FSMs, statecharts are easier to create and easier to read. They add the notion of hierarchy, which permits abstraction of the design at different levels, and orthogonality, which allows for concurrent components in the design. They also add a broadcast mechanism for communication amongst concurrent components. These extensions make it practical to model large and complex reactive systems. The resultant diagrams are expressive, but more compact than the conventional FSM diagrams, making state-based notations much more amenable to industrial use.

2.2.4 Existing work with statecharts

The work of Harel has inspired others to try their hand. However, not many have said much about testing. Some have restricted themselves to a subset of Harel's vision; others have made specific extensions to address issues in their particular area of interest.

Research

As seen from the current literature, many scholars have concentrated on trying to make the original semantics more rigorous, including by way of creating variants. The following is a short list of some of the areas currently under research.

- A "model checker" for statecharts has been developed to test whether an operational specification (given by a statechart) satisfies the descriptive specification of the system requirements [Day 93]

- *Event priorities* were not handled by Harel's original statecharts; in states having multiple conditions associated with transitions, it is possible that more than one condition could hold true. A simple extension to the notation has been proposed to avoid introducing non-deterministic behaviour [Horrocks 99]. A more detailed investigation has also been done to determine what syntactic and semantic extensions would be needed to model transition-based priorities [Maggiolo-Schettini+ 97]. The two options considered were "explicit priority relations between transitions" and priorities modelled by "negated events in triggers of transitions," and it was shown that they could be translated into one another. It was also shown how a pre-emptive interrupt could be modelled by a suitable interpretation of the hierarchy of states.

- The *equivalence* of statecharts has been studied in an attempt to substitute specifications with equivalent more concise ones [Maggiolo-Schettini+ 96]. An attempt is made to translate statechart terms into labelled transition systems (LTS) and then to look for equivalence of the LTSs.

- A *transition structure*, which is a basic representation of a statechart in terms of its transitions, is shown to be sufficient to describe a statechart, and that the states, the hierarchy on states, and the representation on parallelism can be derived from the transition structure [Peron 95].

- TSP is a discrete timed process language that can be used for modelling a discrete timed version of Harel statecharts. A compositional verification method is proposed for TSP [Levi 97]. Future research may extend this approach to a "dense time" environment, in which signals may appear at arbitrary close instants of time; it is not clear if the hypothesis that "reaction of the system to inputs is instantaneous" will still be reasonable.

- A variant of statecharts called Real-time Object-Oriented Modeling (ROOM) was developed to address real-time requirements and to take advantage of the object paradigm [Selic 92]. ROOM is described further in the rest of this chapter and in Chapter 4.

Application of statecharts

Now, we will discuss some of the actual applications or implementations of statecharts. For Harel, statecharts were more than an academic exercise. This notation was developed while Harel was consulting for an aircraft corporation in Israel; statecharts were used as the main method for specifying the behaviour of an avionics system. In 1987, statecharts were implemented in Harel's STATEMATE system, a commercial tool for the specification and design of real-world complex systems [Harel+ 96]. Statecharts, which describe system behaviour, were combined with Object-model diagrams, which specify the system structure, to form a language set that now constitutes the core part of the Unified Modeling Language (UML), a commercial modelling methodology [Harel+ 97]. A fully executable model is created; the current implementation produces code in C++.

Another real-world application of statecharts has been in the construction of user interfaces by Horrocks, a user interface designer at a British telecommunications firm [Horrocks 99]. Although different techniques are listed for specifying the dynamic behaviour of systems (namely: natural language, finite-state machines, decision tables, program design language, statecharts, requirements engineering validation system, requirements language processor, specification and description language, PAISLey [Zave 91], and Petri nets), most of the author's comparisons of statecharts are with natural language and conventional FSMs. Case studies and pseudo-code are given

The ROOM methodology mentioned earlier in this sub-section has been implemented in the ObjecTime toolset, a commercial CASE tool [Selic++ 94], now available as Rose RealTime. As the name suggests, the focus of ROOM is on the development of concurrent, event-driven, real-time distributed systems. The developers of ROOM have their roots in a large Canadian telecommunications research company. The toolset includes a graphical modelling environment, produces executable models, and can automatically generate complete C and C++ code from the design model.

The Requirements State Machine Language (RSML) is not actually based on statecharts but includes many of its characteristics and is worthy of mention [Heimdahl+ 96]. RSML is a state-based requirements specification language developed by an American safety research group using the Traffic alert and Collision Avoidance System II (TCAS II) as a test bed. RSML was designed specifically for black-box requirements specifications, for behaviours expressed only in terms of what is externally visible. Analysis does not require generating a global reachability graph, and tool support is available.

2.2.5 ROOM Actors

We now turn our attention to ROOM, which is based on the notion of *statecharts* which are themselves an extension of conventional state machines and state diagrams [Harel 87]. The following sub-sections form a basic introduction to some of the main concepts in ROOM (Real-time Object-Oriented Modeling), a language and methodology that has been implemented by a toolset called ObjecTime. In these sub-sections, discussion will be centred on aspects of behaviour modelling using hierarchical finite-state machines. Additional details on extensions to this basic model will be provided, later in Chapter 4 and as specific enhancements to the main idea of this thesis. Readers interested in a thorough treatment of ROOM should consult the primary reference [Selic++ 94].

In keeping with ROOM's object-oriented approach, models are centred on actor classes created by the designer. An actor is said to exhibit *structure* and *behaviour*. The

structure is defined using a *structure chart*, which shows all of this actor's ports, and any other actors that may be contained within it. An actor's behaviour is given by its *ROOMchart*, which is a form of hierarchical finite-state machine that includes a set of conventions for special activities such as exiting states, entering states, and re-entering states. (See Fig. 2-9.)

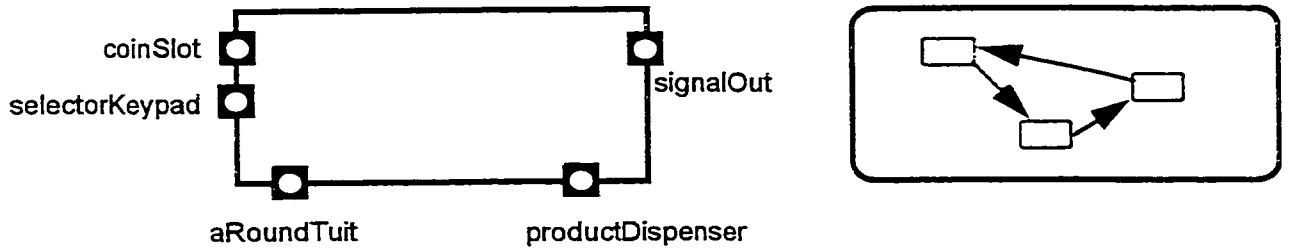


Fig. 2-9. *Example of a structure chart and a ROOMchart depicting respectively* actor structure *(left) and* actor behaviour *(right).*

This thesis addresses the behaviour portion of an actor.

2.2.6 Ports, signals, and protocol classes

Actors interact or communicate with other actors by sending messages through their interfaces called *ports*. Since a port is defined to be of a certain type (a *protocol class*), the message types (called *signals*) and their direction of travel (in or out) are automatically determined. Unlike in some communications models, ports do not transmit in "broadcast" mode; rather, a port is *bound* directly to another port (of another actor). A receiving port that accepts the same message types, but with in/out directions reversed, is called a *conjugate* port. In structure charts, port bindings are shown explicitly with dark lines from one port to the other, within a container (higher-level) actor.

The following is an example of a protocol class.

```
protocol class DeviceControl:
        in: {deviceCommand}
        out: {deviceStatus}
```

Thus, every port automatically has a fixed set of signals associated with it. The class shown here could be used for ports on devices that can accept commands and report their own status. Likely, such a device would be communicating with a controller device, acting on behalf of a user or an operator.

A transition is taken if its trigger condition is satisfied. A *trigger* is the receipt of a message at a port, optionally combined with a *guard* condition, a Boolean expression involving data values or simply their absence or presence. A transition may have multiple triggers, any of which will cause that transition to be taken. To see the triggers in ObjecTime, the user can double-click on a transition.

2.2.7 Actions and states

In contrast to the FSMs described in the previous section where actions are associated only with transitions, actions in ROOM can also occur upon entry to a state and upon exit from a state. It is generally up to the modeller or designer to determine how to model the action. For example, a *continual* action with a state may be modelled as either a start-action upon entry to the state or a stop-action upon exiting.

A ROOM model is a collection of actors represented by communicating extended finite-state machines, and usually set in the environment of one big actor for the entire system. ObjecTime does not attempt to capture any notion of global state. Each of the actors functions independently and potentially concurrently; messages are sent and received without any guarantee on their delivery time or their sequencing. In this sense, the overall system exhibits non-determinism because it has no control over the relative order of input events.

2.2.8 Run-Time Systems

The development environment of the ObjecTime toolset includes the ability to execute a design model. A model compiler is used to translate the ROOM notation to a high-

level language such as C++ which is then further compiled to run on a "ROOM virtual machine." Some degree of animation is also provided to assist the modeller in doing the design. If the window for an actor's ROOMchart is open during execution of the run-time system, then the transitions and states will be temporarily highlighted (with bold lines) as control passes through. At the same time, if the actor's structure chart window is open, the port bindings will be highlighted in an animated fashion so as to show a message moving from one port to another.

Additional windows may also be opened to provide the modeller with some observation and control. For example, one may wish to inject inputs into certain ports, monitor messages being received at ports, or monitor the activity on a given transition.

2.2.9 Differences between ROOM and statecharts

It is worth summarizing here the main differences between ROOM and statecharts.

a. In ROOM, communications are not broadcast; each source has to be connected to each destination desired. Even the use of any timing services to note timeouts or the passage of time must be explicitly modelled by the designer.

b. ROOM does not assume reliable communications, since a real-world distributed system tends to run over unreliable communication links.

c. ROOM does not incorporate concurrent states, as such, but this can be emulated to some degree. Since it is possible to have different actors in different states, ROOM achieves orthogonality by having each concurrent state represented by a separate ROOM actor.

d. ROOM follows object-oriented concepts such as encapsulation and inheritance.

2.2.10 Elaboration on scope of thesis

To situate the problem properly, we have briefly described modelling (and we will describe testing). Real-world modelling requires both a structural description and a behavioural description. This thesis focusses on the behavioural portion only. In the case of ROOM, this implies only the ROOMcharts. In the case of statecharts, all of it is behavioural; however, the richness of the language precludes our addressing everything. We will limit ourselves to "the basics": the hierarchy, clustering, and refinement of states; we also address (in ROOM terminology) group transitions, top-level group transitions, the history mechanism, and choicepoints. We will not address advanced features such as delays, timeouts, parameterized states, overlapping states, or temporal logic. Keeping the strategy basic also increases its applicability to the other variants of statecharts.

2.3 Principles of testing

This section provides a brief review of basic testing terminology and test design principles. The primary references are [Probert 94], [Beizer 90], and [Myers 79]. The final section is devoted entirely to FSM-based testing methods.

2.3.1 What is meant by "testing"

The term "testing" refers to module testing (or "unit testing"), interoperability testing, performance testing, conformance testing, and acceptance testing, to name a few. For each kind of testing, the associated activities generally include test planning, test design, test-case generation, test execution, results reporting, management of test sessions, test results analysis, and test records management. In this section, the emphasis will be put on the design of test cases that can be applied to software modules, but with the understanding that the methods should be scaleable to larger units.

2.3.2 Test case design

Test design is the stage at which we hope to achieve great efficiencies in revealing faults. A designer tries to write test cases that will reveal defects in the program and that do so in an efficient manner. Simply trying every possible input (exhaustive testing) is usually not feasible. For example, testing a compiler program would require inputting every possible program that could be written. Since this is not possible, randomly selected inputs could be attempted, but this is not likely to reveal as many faults as when a systematic method is applied to select the inputs [Myers 79].

There are many different methods for designing test cases. Some of the better-known methods are discussed here. Since the different methods have different capabilities in revealing defects, a good tester will use some combination of them to make testing as effective as possible.

2.3.3 Scenario-based methods

A common and intuitive way of generating test cases is to consider various scenarios of execution. The scenarios can be drawn from the *customer operational profile*, if this is known. Such a profile can also be used as a guide to determine where greater testing effort should be applied. Assessing *risk* is another way of choosing scenarios. Risk can be roughly "calculated" as the probability of a particular defect occurring multiplied by the cost, consequence, or severity of this error. From this, high-risk scenarios can be identified and tested.

For the purposes of optimizing testing effort, it may be worthwhile to think of behaviours in terms of "normal" and "abnormal" (or "exceptional"). A good scenario to start with is the longest normal path in an FSM. Long paths from start to finish (or back to the start) will typically reflect the behaviour desired by a customer. Unfortunately, test cases based on these normal customer-directed scenarios tend to be "low-yield" in terms of revealing faults because the programmer will almost certainly have tested them. Neverthe-

less, it is prudent to take advantage of the operational profile of the customer in including such tests, because the consequence of not discovering an error here will be immediate and great (as well as embarrassing). It is typical (and efficient) to try to cover as many normal behaviours as possible in one test case.

Test cases derived from exception-handling behaviours (hopefully with recovery behaviours as well) tend to be "high-yield." The intent is to check for both abnormal system behaviour (by injecting potential system faults) and abnormal user behaviour. The exception recovery sequences will usually be short but numerous. A separate test case should be used for each, so that results will be easy to interpret. The use of high-yield cases can lead to improved overall robustness of the system.

2.3.4 Black-box methods [Beizer 95, Myers 79, Probert 95]

Black-box methods assume no knowledge of the program code. They are based on the specifications of the program or its advertised features. To write black-box test cases, it is necessary to examine the observable behaviour or functionality of the system or program. Thus, black-box testing is also called *functional testing* or behaviour-based testing. To execute such test cases, inputs or stimuli are applied to the system, and the corresponding outputs or responses are then observed to see if they are compliant with the expected responses. Because it may not be possible to try all input values, a common technique is to try values at the boundaries of allowable ranges and to try values just outside or just inside the boundaries. Such values can be taken as representative; if the values are discrete, an arbitrary selection may be made.

Black-box methods are useful in the early stages of program or system development for ensuring basic functionality. Such early testing can also provide worthwhile feedback with regard to its design. Black-box methods are also used in the later stages, when the system is considerably larger and the amount of code becomes too great for efficient white-box analysis. Yet, the desired behaviour of the overall system remains clear, making

functional testing very practical. Clearly, any modelling methodology that permits a user to perform behaviour testing would be highly desirable.

2.3.5 White-box methods

White-box methods rely on the test designer being able to see the structure of the code being tested. Tests may be written to verify that the if-then-else branches are working properly or that looping constructs terminate properly. The simplest kind of white-box testing is ensuring that each statement in the program is exercised at least once ("statement coverage"). More effective is ensuring that each branch is covered ("decision coverage") or, better, that each branching condition is exercised ("condition coverage"). *Path coverage* implies that all combinations of branches have been exercised; often, this is not achievable because of the large number of possible combinations of branches and because certain combinations of branches may not be permissible. White-box testing is also called *structural testing*, clear-box testing, glass-box testing, or code-based testing.

Also categorized as white-box are static (non-executing) methods such as code inspection and data-flow analysis. *Data-flow* testing techniques for general software involve first locating where a variable is *defined* (created, initialized, or modified), *used* in a calculation, used in a predicate, or *released*. Different execution paths (and sub-paths) are then examined to see if there are any invalid or anomalous behaviours. Examples of anomalous behaviour: a variable is defined but never used; or, it is defined and re-defined again before any use; or, it seems to be used without first being defined. Selecting enough paths to cover all definition/use combinations for all variables would be a strong strategy [Weyuker 88].

In the case of EFSMs, it may be sufficient to restrict attention to only those variables that could affect the flow of control. Fortunately, such variables are often also the ones visible by a customer, like parameter settings, timer or temperature inputs, capacities or

limits. This visibility restriction restores a black-box flavour to this kind of testing and avoids the cost of testing a greater number of paths.

White-box methods that depend directly on analysing code are best applied after the code has achieved some degree of stability.

There also exist formal methods for mathematically proving the correctness of code, but these methods are tedious and are usually reserved only for very critical code, for example, the microprocessor microcode in safety-critical applications [MillerS+ 95].

2.3.6 Grey-box methods

Additionally, there are methods that draw partly upon the behaviour and partly upon the structure of the program or system. These are sometimes referred to as *grey-box* methods [Probert 94]. The structure examined is the design representation, rather than the actual code. For example, to verify an indication such as a customer-visible alarm (a behavioural feature), it may be necessary to examine various locations in the program (code-based analysis) to see what combinations of trigger parameters will result in its activation. (Hopefully, these correspond to the customer's specified requirements.) Another example may be a case in which the code is not seen, but where structural details of execution behaviour are known and used to formulate test cases.

2.4 FSM test-generation methods

In the current literature, FSM-based test generation is based on one general problem: "Given a behaviour *specification* (in the format of an FSM) and an actual software *implementation* (whose code is not visible), can it be determined whether or not this implementation conforms to the specification?" The general approach is to derive a sequence of inputs (and expected outputs) which when applied successfully would give a tester confidence that the software correctly implements the FSM specification as claimed.

Because the code and internal machinations of the software implementation are hidden from the tester, some of the changes in state may remain unseen. The tester depends on externally observable behaviour to identify the current state. Consequently, these methods are considered black-box methods. If a given action or output is not unique to a particular state or unique to a specific input, it becomes necessary to apply a succession of inputs (and to observe the corresponding outputs) in order to draw a conclusion as to what the original state must have been. A set of assumptions is used often to make the problem manageable; amongst the most common: that the FSM is strongly connected; that the FSM is *minimal* (in the sense that there is no other FSM with fewer states that accomplishes the same thing); that the FSM is fully specified; and that there exists a ***reliable reset*** feature (a trusted ability to return the FSM to a known initial state). [Aho++ 91], [Dahbura++ 90], [Gonenc 70],

Amongst the possible errors that an implementor might make: the provision of a wrong output for a given input (*output error*); transition to an incorrect state (*next state error*); inclusion of a transition that was not in the specification; or inclusion of an extra state. The test sequence generation methods discussed here address only the first two kinds of error and assume that there is at most one error.

2.4.1 Verifying states and transitions

Identification or verification of states is achieved by examining the input/output combinations (that is, the transition function) in the FSM specification. As an example, consider the FSM specified in the table below.

	Input	
State	***x***	***y***
s0	*0, s1*	*1, s0*
s1	*1, s1*	*1, s2*
s2	*1, s0*	*0, s1*

It can be seen that if we apply an *x* to the machine and we receive (observe) a *0*, then we can assert that the machine must have been in state *s0*. If, however, we receive a *1*, then there remains a question as to whether the machine was in state *s1* or *s2*. Similarly, whenever we apply the input *y* and receive a *0*, we can conclude that we must have been in state *s2*. To identify state *s1*, we can look for a *sequence* of inputs for which the corresponding output sequence could not have been produced starting from any other state. For example, applying the input sequence *x.x* at state *s1* produces the output sequence *1.1* (where a dot indicates concatenation), whereas applying this same sequence at *s0* would have resulted in *0.1*, or in *1.0* if from *s2*. Thus, *x/1.x/1* is a *unique input/output (UIO) sequence* for state *s1*.

Aside from using UIO sequences [Sabnani+ 88], there are other state identification methods such as the Distinguishing Sequence (DS) [Kohavi 78] for the D-method [Gonenc 70], the Characteristic Set (W-method) [Chow 78], the UIOv-method [Chan++ 89], and the Wp-method [Fujiwara++ 91], and all of these methods are also based on finding a distinct sequence of applied inputs and expected outputs.

The verification of a transition has two parts: verifying that the expected output is received, and verifying that the new state attained is truly the one that it is claimed to be. The *transition tour* (or T-method) is a simple method that consists of verifying only the output, and not the new state [Naito+ 81]. Though not very thorough, it is comparatively inexpensive and easy to design. The aim of a transition tour is to exercise each transition (at least) once. An example of a transition tour for the FSM of Fig. 2-1, starting from the state *s0*, would be *x/0.x/1.y/1.y/0.y/1.x/1.y/1*. Methods more sophisticated than the T-method attempt to verify the identity of the state that is reached after each transition is tested. The vast majority of the methods currently being researched involve some combination of transition selection strategy and state identification method, with slightly varying techniques for selection, identification, or combination. [Aho++ 88], [Chen+ 95], [Dahbura++ 90]

2.4.2 Test sequences

The 5-step structure commonly used in combining state and transition verifications is as follows:

Step 1. Reset the system to a known (usually initial) state;

Step 2. Use a *preamble* (or *transfer sequence*) to get to the starting state of the transition to be tested;

Step 3. Exercise that transition, observing the correct output;

Step 4. Apply a state identification sequence to verify that the transition reached the correct state; and

Step 5. Use a *postamble* to prepare the system for the next test input by noting the new current state and using a transfer sequence to get to the next starting state (as in Step 2), or by simply resetting the system to the initial state (as in Step 1).

The input/output sequence obtained from Steps 2, 3, and 4 together form a *test segment* for the transition being tested. A collection of test segments that verifies all transitions is a *test sequence*. Anything in between, for example, testing just one path in an FSM, is sometimes referred to as a *test subsequence*. [Sidhu 90] provides a survey of the formal methods used in test generation and outlines some of the issues in protocol conformance testing.

Some degree of optimization can be obtained (especially in Step 5) by finding ways in which to combine the test segments so that the total length of the test sequence is the smallest. Combinatorial heuristics and exhaustive comparisons are used when there is no precise algorithm. Shorter sequences can be had by not resetting the system after each test segment; there is a theoretical risk that an error will go undetected, but there may be significant cost savings realised in practice. Another method proposed is to combine the test segments in an overlapping fashion, resulting in a test sequence whose total length is considerably shorter than the sum of the individual segments [Chen++ 90], [MillerR+ 93]. Again, the trade-off is between thoroughness of test and economy of resources. [Ural++ 97] shows

how to find minimal-length *checking sequences*, which are sequences, starting at a specific state, that will distinguish the given FSM from any other FSM that is not isomorphic to the given one.

Some of these FSM-based test generation methods assume that a "reset" feature (to bring the FSM to the initial state from any state) is correctly implemented in the implementation under test. Also, any error must not increase the number of states in the implementation. On the basis of these assumptions, claims can be made as to the effectiveness of error detection. However, there still remain the practical problems of locating, diagnosing, and reparing the error.

2.4.3 Pragmatic FSM-based methods

In comparison to other methods for software testing, the classical, formal, FSM-based techniques are more difficult to implement and are less popular in the general software industry. However, certain types of software, for example, communication protocol software, are amenable to FSM-based methods. It is possible to develop pragmatic FSM-based approaches to satisfy our coverage requirements.

There is a natural tendency, when analysing only the control flow of a program, to focus on the structure of the flow, rather than on the meaning or significance of the actual function. Probert covers both the structural aspects and the functional aspects in classifying a tester's desired coverage using the following three categories [Probert 95]:

- 'untargeted' coverage, based on structure
- 'targeted' coverage, based on structure
- risk-directed coverage, based on function

Untargeted coverage is achieved by exercising all transitions in an FSM transition tour. The objective is to derive as few scenarios as possible but which, in total, will exercise

all transitions. This level of coverage is suitable for testing normal behaviours and can be used by designers for regression testing.

Targeted coverage means that each transition is targeted one at a time, and that a separate scenario will be drawn up to exercise each transition. This approach is very common for generating tests to conduct conformance testing of telecommunication protocols.

A *functional, risk-directed* coverage strategy implies that an attempt has been made to understand which scenarios represent normal behaviour and which represent abnormal or risky behaviour. A practical way to analyse an FSM is to first identify natural phases in the execution of the system. Then, for each phase, the normal behaviour path(s) can be charted, and for each of these 'normal' paths, the abnormal transitions can be noted – normally, these deviations from the norm are where the exceptions are handled. The overall strategy is to cover the normal paths with as few scenarios as possible, but to cover each selectively, in accordance with its assessment of risk (if applicable).

2.4.4 Other methods

The methods described thus far in this section involve *control-flow* testing because no regard is given to the data values, only the sequence of states and transitions. *Data-flow* methods were discussed in the section on white-box testing (Section 2.3.5).

Although the testing of EFSMs will not be discussed at length in this thesis, a few words are worth mentioning. Special attention needs to be paid to the variables that are used by the system. Because the actual values of the variables might be used to determine which transitions are executed, the test designer must take this into account when writing test cases. Paths that seem feasible when looking at just the FSM may turn out to be infeasible when the variables are also considered. For example, in Fig. 2-3, the path Empty-to-Normal followed

by Normal-to-Full (two successive 'Push'es) is infeasible if STACKCAPACITY is less than two.

Where applicable, timeouts and other time-related actions will have to be considered. For example, there may be some exception-handling scenarios that are valid only when a timer has expired, therefore any attempt to combine these with normal scenarios (where the timer is still alive and running) would not be meaningful. Because EFSM testing involves knowing the values of system variables, EFSM test methods tend to resemble white-box methods (where you can see the code, know the stack capacity, etc.).

In ROOM, a limited degree of data-flow testing can be conducted simply by examining and varying the values of the externally visible parameters, as identified in the design representation (which in this case would be the ROOMchart). By knowing the functional relationships between input and output parameters (causes and effects), test cases can be written to verify the different possibilities. Standard black-box techniques such as trying values on or near the boundaries can be applied as well.

Clearly, all of these methods are of only limited value if there are very few places in the system to make inputs or to see results. One important aspect of "design for testability" is the inclusion of a sufficient number of *points of control or observation* in the system. The incorporation of such points should be done early in the development cycle if possible. Vuong and others provide a discussion of design for testability specifically for communication protocol software [Vuong++ 94].

Summary

It can be seen that there are many ways of extending the basic concept of a state machine (not all of which were described were in this chapter). Statecharts added the notions of hierarchy and orthogonality. ROOM retained hierarchy, but not true orthogonality, and then added object-oriented extensions. Faced with this interesting combination, we considered very fundamental software testing techniques, and then focussed on those specific to FSMs. Acknowledging the limitations of formal FSM-based methods, we presented more

practical approaches, noting that the combination of structural analysis with scenario-based analysis appears to be cost-effective.

2.5 Software complexity metrics

A *metric* is a simply a number that shows the value or "score" of a particular item based on the measurement of pre-defined attributes or characteristics. It can then be compared to some norm or standard, or recorded to create a history (which may eventually lead to the declaration of a norm or standard) [Perry 95]. Metrics are often used in an attempt to quantify something that is usually more of a qualitative nature.

Software metrics can be broadly grouped into three categories: process, product and resource (or "project"). *Process metrics* concern themselves with any software related activity that takes place over time, in particular those measurable entities that can contribute to the improvement of a process. Examples include the rate at which defects arrive and the effectiveness of defect removal during development. *Product metrics* describe characteristics of the actual deliverables, the documents or other items that arise from the process. These metrics often deal with size, complexity, and performance. Finally, *resource metrics* are those that describe items that are inputs into the processes, such as the number of programmers on staff or the costs of testing [Grady 92].

Metrics themselves have a set of intuitive attributes against which they may be judged; examples: ease of understanding, maintainability, usefulness, and ease of implementation. Sometimes, these are taken as an indication of the "quality" of the metric. It is also possible to come up with more formalised properties that metrics should satisfy [Weyuker 88]. Two examples are paraphrased here:

- Adding code to a program cannot decrease its complexity

- It is possible to find two program bodies, *A* and *B*, having equal complexity, that could be separately concatenated to a third program, *C*, but the resultant larger programs, *C+A* and *C+B*, would have two different complexities.

Many metrics are also concerned with the ability to predict something, such as cost or reliability. This implies that a causal relationship exists somewhere. An interesting example is Halstead's formula that claims that the final length of a program can be estimated by

$$n1 \times \log(n1) + n2 \times \log(n2)$$

where

n1 = the number of unique operators (including keywords)
n2 = the number of unique operands (data objects)
log = the Base 2 logarithm. [Halstead 77]

Observations have shown that the correlation is often quite high. The problem with this metric is that the final numbers to be input might not be available until the program is nearly complete, therefore the ability to predict might be rather limited [Kan 95].

For this thesis, we will concern ourselves with primarily with *complexity metrics* which, although they are measured directly from the product itself, may also be indicative of the process that was used to create the product. It is tempting to look for a single number that accurately reflects all aspects of complexity of software; however, even for something as specific as control-flow complexity (only), it is not possible derive a general measure because "there is no general notion of control-flow complexity of programs which can be measured on an ordinal scale" using the set of Real numbers and a relationship operator "<". [Fenton 94]

2.5.1 Number of lines of code

Probably the simplest metric for complexity is the number of lines of code in the program. Even this simple metric has some variations. One could count

- only lines with the executable code
- executable code + data definitions
- all lines that are not comments
- all lines
- physical lines on an input screen

The general line of reasoning for this metric is that the more code there is, the more likely there is a defect. A *defect* can be taken to mean any anomaly in the product, but like "number of lines of code," may also have other definitions.

It has also been found that the *defect density*, the number of defects per amount of software, is related to the number of lines of code. One study looking at programs written in Ada included software modules from as small as 63 lines to more than 1000 lines and found that the minimum defect density occurred for modules around 250 lines [Withrow 90]. Obviously, this number may be different for different programming languages, types of project, environments, stages of testing, manner in which "lines" were counted, and so forth.

2.5.2 Halstead's *Elements of Software Science*

Halstead's metrics can be described as *linguistic* metrics, meaning that they apply to the text of the code without taking into account the meaning of the text. One property of such a metric is that it does not change if the code is re-arranged. Halstead treats a program as a collection of "tokens" that will later be compiled. Using the following notation, Halstead expresses a number of different measures [Halstead 77].

Notation.

$n1$ = the number of distinct operators in the program (including keywords)

$n2$ = the number of distinct operands in the program (data objects)

$N1$ = the total number of operator occurrences

$N2$ = the total number of operand occurrences

log = the Base 2 logarithm

Some of his **key equations** are summarised here:

Vocabulary (n)	$n := n1 + n2$	(by definition)
Length (N)	$N := N1 + N2$	(by definition)
	$N = n1 \times \log(n1) + n2 \times \log(n2)$	(predicted)
Volume (V)	$V := N \times \log(n)$	(by definition)
Number of bugs (B)	$B = V / S^*$	(predicted)

where S^* is a constant equal to 3000.

Halstead took 3000 to be "the mean number of mental discriminations (decisions) between errors." ■

(Note: See also the earlier discussion in the introduction to Section 2.5)

As controversial as they may have been, through time and much experimentation, Halstead's metrics have established themselves as the primary linguistic metric.

2.5.3 McCabe's cyclomatic complexity

In comparison to Halstead's metrics, which were linguistic metrics, McCabe's cyclomatic complexity is a *structural* metric, meaning that it is based on control-flow complexity or data-flow complexity. Control-flow is depicted by a *program graph* of nodes and edges, where each node represents a task (of one or more statements) and the edges represent transfer of program control (branching) amongst tasks. This typical graph structure permits the application of well established results from graph theory.

By definition, the McCabe *cyclomatic complexity* is [McCabe 76]

$$MCC = \#EDGES - \#NODES + (2 \times \#PARTS)$$

where

#EDGES = the number of edges (or "links") in the graph

#NODES = the number of nodes (or "vertices") in the graph

#PARTS = the number of disconnected parts of the graph
(for example: main program and subroutines)

In a program containing only binary decisions (true / false), this metric is also equal to the number of decisions in the program plus 1. In a general program with decisions other than just binary ones, a *k*-way decision can be counted as *k*–1 binary decisions. The name *cyclomatic complexity* is taken from graph theory, where this computed value is simply the traditional *cyclomatic number* giving the number of regions in a planar graph. When applied to software, this number is the "number of paths" in a module with a single point of entry and a single point of exit. Note that this counts looping constructs as separate paths.

Here are some simple examples:

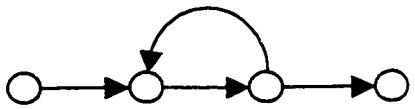


Fig. 2-10. *A control graph with cyclomatic complexity of 2 [After Beizer 90].*

Example. In Fig. 2-10 (above), we see that *#EDGES* = 4, *#NODES* = 4, *#PARTS* = 1.

$MCC = 4 - 4 + 2\,(1) = 2.$ ■

Example: In Fig. 2-11 (below), we see that *#EDGES* = 9, *#NODES* = 6, *#PARTS* = 1.

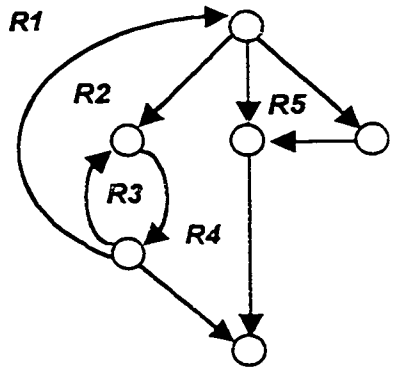


Fig. 2-11. *A control graph showing the number of regions (5). [After Pressman 92]*

$$MCC = 9 - 6 + 2(1) = 5.$$

Note that there are indeed five regions.

Also, we see that there are two 3-way decisions, each of which is counted as 2 binary decisions, and that this graph is planar. Therefore:

MCC = number of binary decisions + 1

= 2 + 2 + 1 = 5 (which is consistent with the previous result). ■

As with Halstead, the complexity metric of McCabe has also become very well accepted.

2.5.4 Other complexity metrics

Other structure-related metrics deal with the interactions between modules (as opposed to within modules); for example, the fan-in and fan-out measures from [Yourdon+ 79] and [Myers 78]. *Fan-in* is the number of modules that call the module concerned; *fan-out* is the number of modules called by the module concerned.

Another metric is the number of INCLUDEs in a module (as run on an IBM Application System AS/400 computer) [Kan 95]. INCLUDEs are used for subroutines and declarations

There are also many variants on the Halstead and McCabe metrics, in an attempt to address weaknesses in the original metrics or to make distinctions where the originals did not. For example, McCabe does not differentiate between the complexities of loops or CASE statements versus and IF-THEN-ELSE structure. As in the case of Halstead, all linguistic metrics ignore the intricacies of structure; therefore, for example, programs that contain heavily nested structures may receive the same score as programs that do not. A number of so-called hybrid metrics can be found in the literature. None seem to have gained recognition on the same scale as Halstead and McCabe yet.

2.5.5 Application of metrics to testing

The application of metrics to testing is commonplace in industry. In general, testing metrics are simply considered as one more tool to help improve the test process, the effectiveness of testing, or the use of test resources. Metrics are still less common at the design phase, but are making inroads here as well. Here are some sample general test metrics and what general purpose they are used for: [Perry 95]

	Metric	Use of metric
1.	instructions exercised (can be as a percent of total instructions)	extent of testing
2.	test cost	resources consumed in testing
3.	defects uncovered in testing	effectiveness of testing
4.	asset value of test (what is spent for testing as a percent of the assets controlled by the system)	effectiveness of testing
5.	paths tested	extent of testing
6.	start-up failure (number of program changes versus the number of failures the first time the changed program is run in production)	effectiveness of testing
7.	cost to locate defect (cost of testing versus the number of defects located in testing)	resources consumed in testing

■

It is generally recognised that complexity increases costs and reduces testability. For example, McCabe sees it as given that greater testing effort be directed to components and subsystems of greater complexity, therefore one should be able to determine in advance (from the metric) how many tests will be needed [McCabe 82]. Because modules having a McCabe complexity of greater than 10 are deemed to be at risk of having a high probability of defects, some organisations flag such modules for special inspections or tests.

It is also commonly recommended that testers use the McCabe complexity metric as a guide in ensuring test coverage, but as Beizer puts it, "the relation between cyclomatic complexity and the number of tests needed to achieve branch coverage is circumstantial." Nevertheless, "the reported results confirm the utility of McCabe's metric as a convenient rule of thumb that is significantly superior to statement count [Beizer 90].

Much of the current research in industry attempts to relate McCabe's complexity measure to the defect rate. Studies have found "moderate to strong" correlation, but there is some question as to the true correlation, because it is recognised that both of these are strongly influenced just by program size [Kan 95].

Complexity metrics can be used to help identify overly complex areas that warrant special testing attention, identify simple areas that can be skimmed to save time, estimate long-term maintenance effort, and direct testing efforts. Commercial software tools are available for all of the common complexity metrics discussed here.

In the next chapter, we define "breadth", a design complexity metric equal to the maximum cut of the HFSM.

3

The Approach: Counting Paths – The Breadth Metric

In the previous chapter, the FSMs we looked at were primarily non-hierarchical and generally within the context of the conformance testing problem or machine identification problem. In this chapter, we study a different problem requiring a different approach. This chapter discusses our method, which is based on path-counting to create a "breadth" metric that can be used to indicate the testing complexity of designs modelled as hierarchical finite-state machines (HFSMs). The number of covering paths is indicative of the amount of testing required. Of course, the number of possible paths and the length of possible paths is unbounded for general HFSMs. Although the process of counting paths implicitly generates sample test cases, it should be noted that we are not performing formal test-case generation or conformance testing. Before we discuss ***breadth***, we must first look at how to deal with hierarchy.

Motivation for the counting method is first described in Section 3.1 which provides more detail on the nature of the problem, particularly the combinatorial aspects and the development of our graph-theoretic approach. The subsequent sections put forth rules for handling different cases of graphs.

3.1 Breadth: Estimating the complexity of testing HFSMs

As seen in the preceding chapter, test methods for finite-state machines involve traversal of paths (nodes and edges). In considering test generation strategies for HFSMs (and ROOM in particular), two points are striking. First of all, the academic test generation methods are intended for a flat FSM, not a hierarchical one. This will influence any traversal algorithm. Secondly, because the ObjecTime toolset generates its own code, the ROOM-charts are not only used in design, but in implementation as well. Therefore testing can be done at the ROOMchart level (the design level) instead of on the target code. This is very appealing from a tester's standpoint, since errors can be detected early in the development process.

3.1.1 Flattening a hierarchy

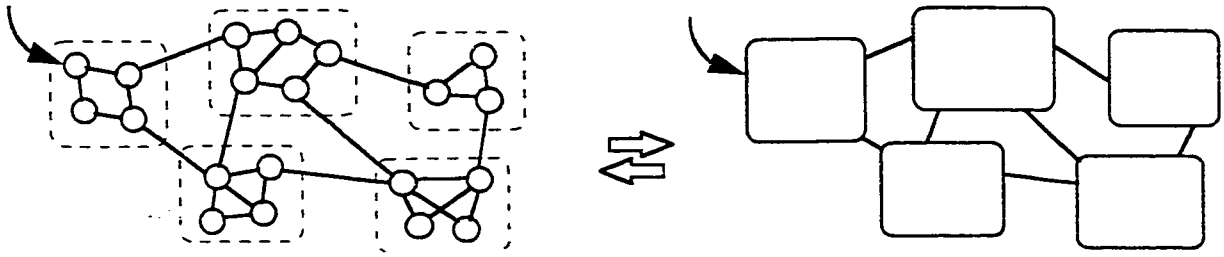


Fig. 3-1. *Smaller states can be clustered into larger ones.*

A hierarchical FSM can be thought of as successive groupings of states. The lower-level states in a very large FSM might first be clustered into many small groupings. In turn, these may be further clustered into larger groupings. In this manner, a hierarchical structure is developed. This kind of HFSM is referred to here as a *simple HFSM.*

There is another type of HFSM, more complex, and which does not start with a flat machine. In this type (see Fig. 3-2), the inner substates do not necessarily have a direct link outside to the other states; rather, they depend on their parent state to make transitions to other "parent states." The parent state "remembers" which substate it was in when it exited and has the ability to return to that substate when being re-entered.

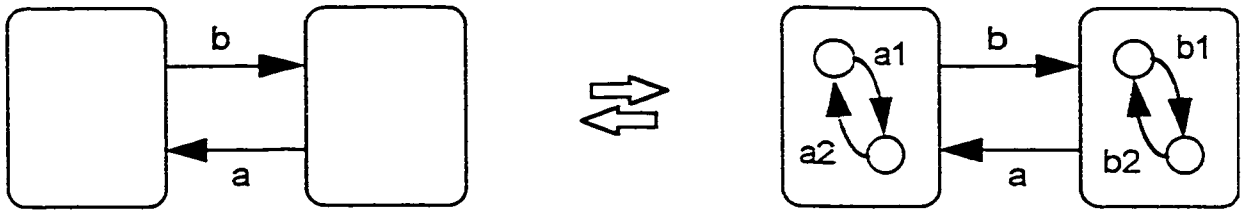
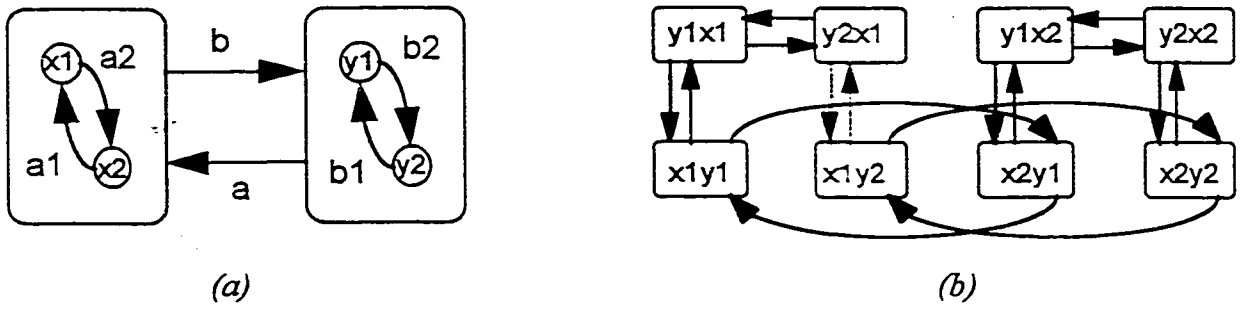


Fig. 3-2. *States can be refined into substates, but with no direct external links. (From [Selic++ 94])*

The opposite activity to building a hierarchy in an FSM is "flattening" it. Completely flattening a ROOMchart would permit the use of standard academic test generation methods, but this would not take any advantage of the inherent organization and structure of the hierarchical ordering. Flattening the HFSM in Fig. 3-1 is easy to do: simply remove all groupings, thereby returning to a single, large FSM. However, flattening the HFSM in Fig. 3-2 will produce a combinatorial increase in the number of states (see Fig. 3-3). The additional states are required to "remember" which substate the other parent state was in before exiting.



(a) *(b)*

Fig. 3-3. *Flattening the HFSM in (a) produces the FSM in (b). (After [Selic++ 94])*

If the amount of testing effort is to be limited, then it would be desirable to avoid the explosion in the number of states that occurs when flattening HFSMs of the type in Fig. 3-2. Therefore, using the hierarchy presents itself as a prime candidate for consideration. In fact, this is the method that we will try first.

3.1.2 Testing at the design level

With most formal test methods, a great deal of effort is devoted to the verification (identification) of every state that is reached, since one cannot be certain that the programmer has correctly implemented the desired FSM. But in ObjecTime, one works at the design level, and the design is made visible using window-based graphics that include animation during execution of the Run-Time System. The designer can always see which transitions are executing and which state an actor is in. Consequently, when any testing is conducted at the ROOMchart level, there is no possibility of ending up in the wrong state (without knowing it). Thus, it is not necessary to apply a distinguishing sequence as is usually employed with conventional FSM-based techniques. Nevertheless, errors may occur in the actions along a transition. Therefore, it is still desirable that all states and transitions be covered.

3.1.3 Magnitude of the problem

The real problem is that there are an extremely large number of different paths to be tested in an HFSM. Before an approach is developed, let us examine the magnitude (complexity) of the problem. It is reasonable to start with the goal of trying to cover all states and transitions. Let us consider first what would be required to achieve coverage of all transitions. We need to branch along various paths to go through the hierarchical structure. Suppose that we start at the top-level chart ("level 0") for the system as a whole. A tour can be written to cover all transitions at this high level. If a state contains substates, then passing through that state implies that some path was taken through some of the substates as well. Applying this logic at each successive layer downward, we may infer that exactly one path is traversed at each level. To simplify the mathematics, let us assume a *fixed* hierarchical structure for the ROOMchart to be analysed.

3.1.3.1 Example. *Analysis of the magnitude of the problem.*

Suppose that

a. there are four levels of fan-out or "break-out" (see Fig. 3-4). (Notice that the top-level node is just a single state and the lowest-level states are "atomic", that is, they do not contain substates.) Thus, the number of levels of break-out is 4 (levels); $NLEV = 4$;

b. there are *exactly* six different ways ("path segments") of passing through any node (six ways at that level only, not counting lower levels); thus, the break-out at *each* level is 6 (ways); $BREAKOUT = 6$;

c. each path segment through any node goes through *exactly* four smaller nodes (four of its substates); in other words, the length of the path segment is four (nodes), $P = 4$.

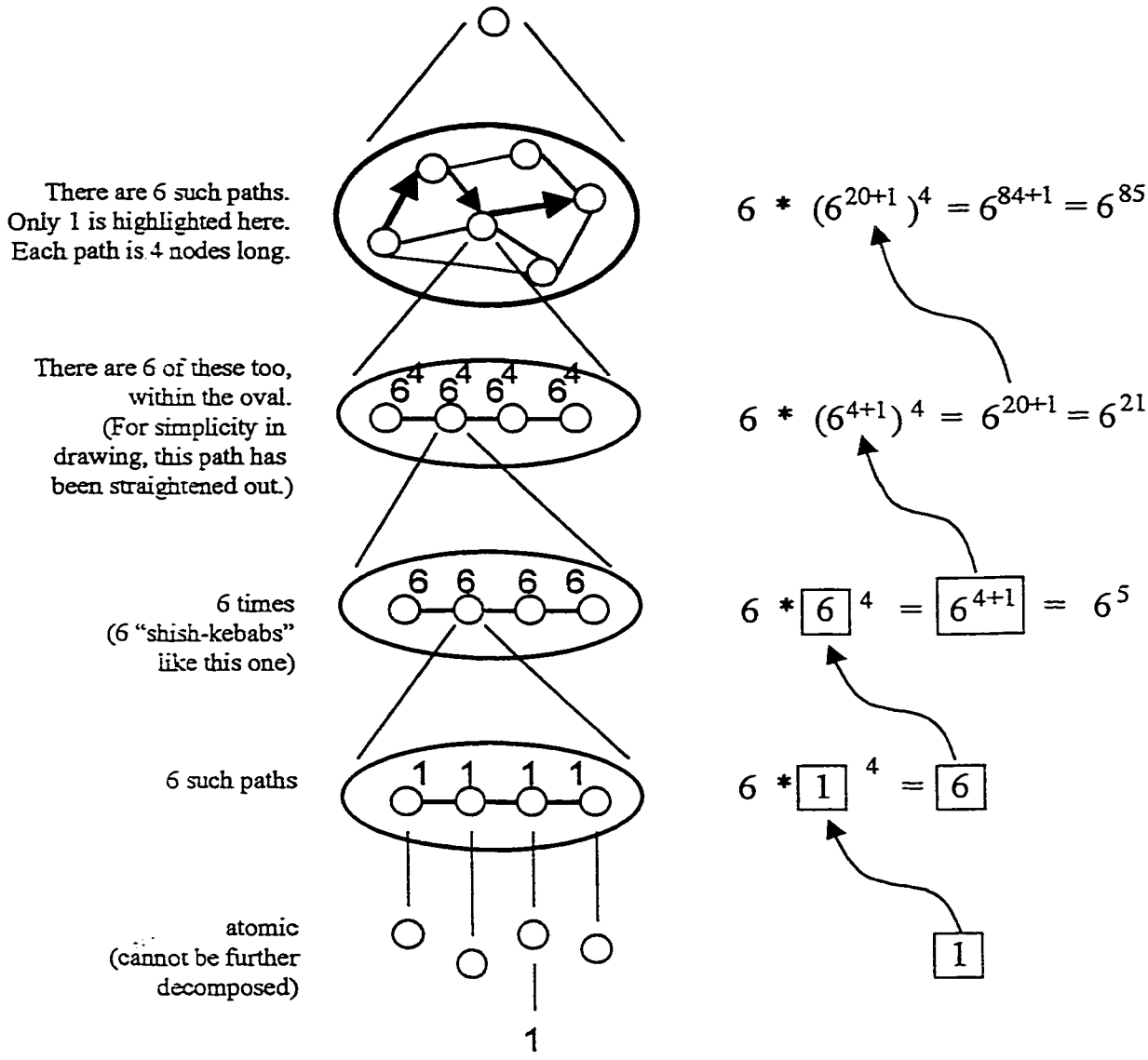


Fig. 3-4. *Showing how atomic states (at bottom) are combined upwards into larger states.*

At each level, there are actually six paths, but only one is shown (to keep the graphic simple). The right-hand side shows the cumulative number of ways through. Subnodes become increasingly more complex, the higher the level. The single node at the top would represent an entire software program. The number on the subnode is NWT.

From the figure, we see that there are 6^{85} paths through this ROOMchart. ■

3.1.3.2 Pattern and formula.

We see from Fig. 3-4, starting at the bottom, and working our way upward, that there is a pattern:

$$NEW = 6 * OLD^4$$

We can express this pattern using a recursive definition:

$$NWT_{i+1} = BREAKOUT \cdot NWT_i^{\,P}$$

and $$NWT_0 = 1$$

from which we conclude

$$NWT_{NLEV} = BREAKOUT ** (P^{NLEV-1} + P^{NLEV-2} + \ldots + P^2 - P - 1)$$

where

NWT_{NLEV} = the total number of ways (*including* all sub-nodes) through a graph containing $NLEV$ levels of breakout

$NLEV$ = the number of levels of "break-out" in the graph

$BREAKOUT$ = the number of ways (path segments) through one node at one level

P = the length of each path segment.

In our example above, $NLEV = 4$, $BREAKOUT = 6$, and $P = 4$.

The total number of paths $NWT = 6 ** (4^3 + 4^2 + 4 + 1) = 6 ** (64 + 16 + 4 - 1)$

which confirms that there would be 6^{85} paths through this ROOMchart.

Recall the assumption here was that the path lengths and breakouts are fixed. In a "real" ROOMchart, the structure will not be so uniform, but if averages are known for the path lengths and breakouts, the preceding formula may be used to derive a quick estimate of the total number of paths through the ROOMchart.

3.1.4 Graph-theoretic approach: transition coverage

With the structure of the ROOMchart clearly visible, white-box methods would seem appropriate. As shown in the previous subsections, it is reasonable to start with the goal of trying to cover all states and transitions. We need to branch along various paths to go through the hierarchical structure. Suppose that we start at the top-level chart ("level 0") for the system as a whole. A tour can be easily written to cover all transitions at this high level. If a state contains substates, then passing through that state implies that a path was taken through some of the substates as well. Applying this logic at each successive layer downward, we may infer that exactly one path is traversed at each level.

Intuitively, a path is a combination of transitions taken in a particular sequence. In the previous sub-section, it was demonstrated that even a fairly modest-sized ROOMchart would demand a very large number of tests in order to cover each path (6^{85}).

Clearly, achieving complete *path coverage* of a ROOMchart would be difficult, in general. As a compromise, we now consider methods for covering only all the transitions in the hierarchically structured ROOMchart, without attempting to do all paths. Suppose that we can cover all transitions in the top-level state by using five paths, call them *a, b, c, d, e.* Suppose additionally that there is one substate, common to all five paths, that in turn requires five of its own paths (subpaths), call them *1, 2, 3, 4, 5,* to cover all of its contained transitions. Complete path testing as above would require 5 x 5 = 25 paths, *a1, a2, ..., a5, b1, b2, ... , e4, e5.* By comparison, simple *transition coverage* would require only five paths, for example, *a1, b2, c3, d4, e5.*

Start with transition coverage

The difference between path coverage and all-transitions coverage becomes more significant if there are more levels. Fig. 3-5 shows an example with three levels, each with about five paths. If we were to select a path at each level, there would be 4 x 6 x 5 = 120

paths for all combinations. However, only max{4, 6, 5} = 6 paths are required for transition coverage (in an ideal situation having complete independence of sub-paths).

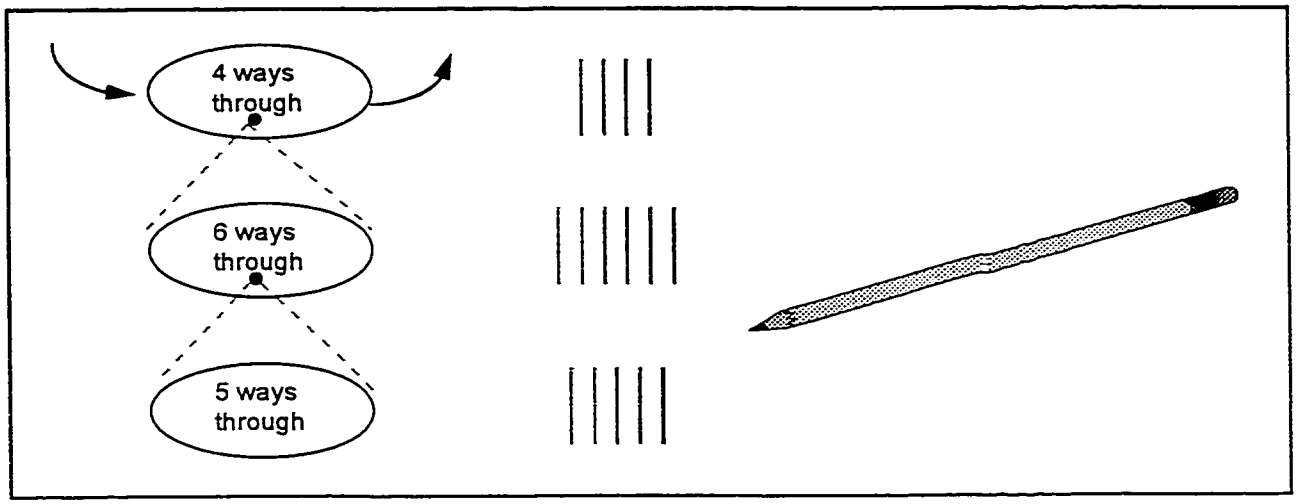


Fig. 3-5(a). *Finding combinations of paths across 3 levels.*

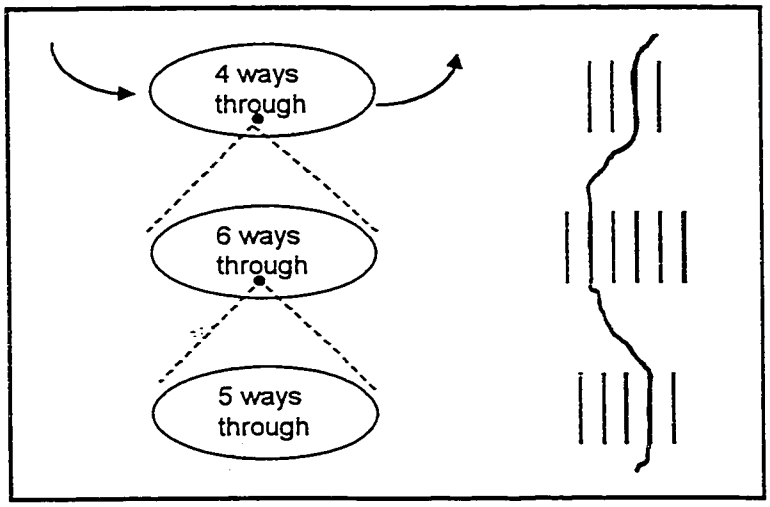


Fig. 3-5(b). *A combination is selected.*

It is interesting to note that as one descends in the hierarchy, an additional level does not necessarily increase the total number of paths. Even when it does, the total grows only linearly, not exponentially. For example, suppose that there was an additional (fourth) layer below. If there were six (or fewer) ways through it, then the total number of paths would not increase at all. If there were seven ways through it, we would require only one more path. For any ROOMchart exhibiting this particular structure (of having only a single sub-state that

expands at each level), the total number of paths required is determined simply by the number required at the layer with the most paths, that is, the "breadth" of the "fattest" node.
Result: *We see that we may use the hierarchy without flattening it.* The rest of this chapter discusses strategies that may be used to accommodate different shapes of HFSM.

3.1.5 Derived graph-theoretic requirement: maximum cut & Definition of breadth

Since most ROOMcharts will not necessarily exhibit the repetitive structure that was assumed for the calculations in the preceding sub-section, it would not be possible to apply a simple formula to obtain the number of paths required to cover all transitions in an arbitrary ROOMchart. In general, for a single node at any level, the number of paths can be counted by inspection, but the user must apply a heuristic to derive the actual routes of each path.

Summary of requirement

> *We require a way of **counting** paths and, optionally, suggesting possible paths in a general ROOMchart. We want a **lower bound** on the total number of paths needed to cover all transitions, based on the required number of paths at each level and on the topology.*

How will we satisfy this requirement? From our quick investigation of transition coverage in preceding sub-section, the concept of "fattest" node gives us the idea of looking at the ***maximum cut*** in a program graph. In graph theory, the graph optimization problem of MAXCUT can be expressed in narrative form as follows:

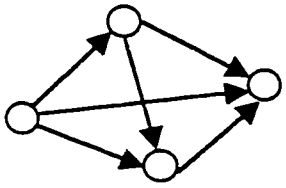
> "MAXCUT- Given a graph with weights on edges, we would like to partition its nodes into two sets such that the sum of the weights of the crossing edges is maximized." [Lu 96]

This is an NP-complete problem [Garey+ 79]. For practical purposes, there are heuristic algorithms for finding a max cut; for small graphs, this can be found by inspection. For

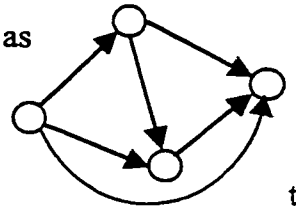
special cases of graphs, such as *planar* graphs (graphs that can be depicted in the plane, possibly re-drawn, without any edges crossing other edges), MAXCUT is polynomial-time solvable [Hadlock 75].

The "planar" graphs described above should not be confused with the "flat" graphs that we referred to in this paper; in our instance, "flat" simply means "non-hierarchical."

Example: The graph

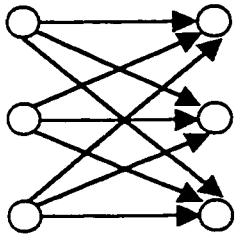


can be re-drawn as



therefore it is planar.

But, this graph



cannot be re-drawn in any manner so as to have no crossing edges. Therefore, it is not planar. Yet, it is still "flat." ■

A more formal definition of the MAXCUT problem is given by the following integer quadratic formulation [Burer+ 98]:

> "Let G be an undirected, simple graph (i.e., a graph with no loops or parallel edges) with vertex set $V = \{1, ..., n\}$ and edge set E whose elements are unordered pairs of distinct vertices denoted by $\{i, j\}$. Let $W = (w(i, j)) \in S^n$ be a matrix of non-negative weights such that $w(i,j) = w(j,i) = 0$ whenever $\{i, j\} \notin E$. For $S \subseteq V$, the set $\delta(S) = \{\{i, j\} \in E : i \in S, j \notin S\}$ is called the cut determined by S. (When $S = \{i\}$, we denote $\delta(S)$ simply by $\delta(i)$.) The maximum cut (MAXCUT) problem on G is to find $S \subseteq V$ such that

$$w(\delta(S)) \equiv \sum_{\{i,j\} \in \delta(S)} w(i,j)$$

is maximized. We refer to $w(\delta(S))$ as the weight of the cut $\delta(S)$. The MAXCUT problem can be formulated as the integer quadratic program:

$$\text{maximize} \quad 0.5 \sum_{i<j} w(i,j)\,(1 - y(i)\,y(j))$$

$$\text{subject to} \quad y(i) \in \{-1, 1\}, \qquad i = 1, \ldots, n" \quad \blacksquare$$

Features of a max cut approach – the design breadth metric

Continuing on the idea of max cut, we note that this method is extendable to hierarchical graphs. We also recognise that algorithm(s) can be applied recursively. The metric that it finally produces, the ***breadth***, is based on path count, and it is clearly indicative of the complexity of the graph (and hence the program or the design). It is similar to McCabe's cyclomatic complexity, discussed in sub-section 2.5.3, though it is not the same. Our metric is more practical because it gives a lower bound of how much testing is required. (Also refer to Corollary 5.3.3.)

Definition. The *breadth* of an HFSM is defined as the maximum cut of the flowgraph underlying the HFSM.

After the max cut is found?

Now that we have this path-count complexity metric, it would be ideal if we could also suggest some paths. At this point, we must acknowledge a traditional problem in software testing: certain paths might be infeasible, where the infeasibility is caused by transitions having conditions associated with them. Nevertheless, we would suggest:

- network flow algorithms for finding flows from a source to a sink

- circulation algorithms for finding flows inside a closed network

Specific cases

The following sections, from Section 3.3 onward, apply the ideas discussed above and examine how to do *counting* for different cases for the structure of subnodes (substates) in a single node (state). Section 3.7 discusses some suggestions for how possible paths might be found, assuming that there are no restrictions on which paths are feasible. First, some basic definitions follow.

3.2 Basic definitions and results from graph theory

A graph, consisting of a set of nodes and a set of edges, is said to be *directed* if each of the edges is directed; that is, each edge has a direction associated with it, indicating in which direction this edge should be traversed. A *(directed) path* can now be more formally defined as a sequence of (directed) edges where the terminating node of the i-th edge is also the starting node of the $(i + 1)$-th edge. A *cycle* is a closed directed path; that is, the ending point is the same as the starting point. A *directed acyclic graph* (DAG) is a directed graph that contains no cycles. A *source* node is a node at which all edges are directed outwards; a *sink* node is one where all edges are directed inwards. A cycle is also known as a *tour*. A *rural postman tour* (RPT) is a tour that covers a specified subset of all transitions at least once. A *Chinese postman tour* (CPT) is a minimum cost RPT [Kuan 62].

Now interpret a graph to be a network having a source node, a sink node, intermediate nodes, and flows on each edge. Let the nodes be divided into two sets, P and P' (where P' denotes the complement of P). A *cut* is the subset of all edges that originate in P and that terminate in P' — this cut would be denoted a (P, P') cut. The *capacity* of a cut is defined as the sum of the capacities of the edges in the cut. If we take "positive" flow to be

in the direction from P to P', then the capacity of any edge directed from P' to P would be subtracted from the total flow capacity [Tucker 80].

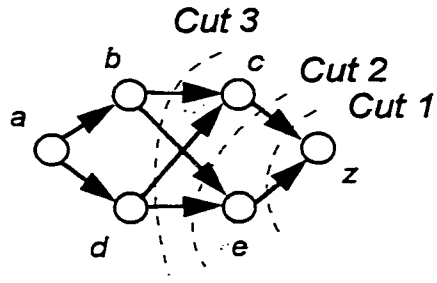
3.3 Breadth of directed acyclic graphs

Continuing from Section 3.1, we begin by showing how to do path-counting in the simple case of a flat HFSM in the form of a directed acyclic graph. (Hierarchy is addressed in Section 3.6.)

3.3.1 Maximum cut rule. In a directed acyclic graph (DAG), the minimum number of paths required to cover all transitions at least once is equal to the capacity of the maximum cut. (We treat the capacity on each edge to be 1 unit of flow.) ■

A proof of a variant of this rule can be found in [Lawler 76]. A more explanatory proof is given in Appendix 2.

Example.



The maximum cut in this figure is 4, occurring at Cut 3, which is $(\{a, b, d\}, \{c, e, z\})$. Therefore, by Rule 3.3.1 above, the minimum number of paths required to cover all transitions at least once is 4. ■

Fig. 3-6. *Finding different cuts.*

For small graphs such as the one in the preceding example, it is not difficult to find paths by inspection. However, for larger graphs, an algorithm can be applied. A discussion on path selection is given in Section 3.7.

3.4 Breadth of strongly connected graphs

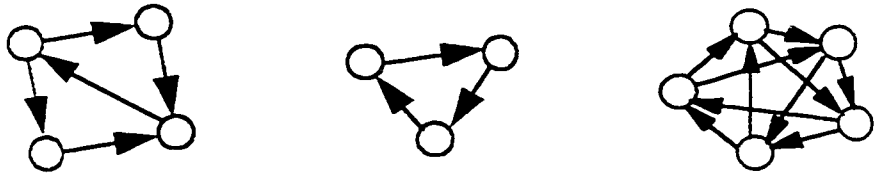


Fig. 3-7. *Examples of strongly connected graphs.*

3.4.1 Strong connection rule. In a strongly connected graph, the minimum number of paths required to cover all transitions at least once is one. ■

This rule is self-evident. By definition, a strongly connected graph implies that we can travel from any node to any other node by way of paths with cycles. Since there is never a need to terminate a path in order to start another one, only one path is required. Path selection is discussed in Section 3.7.

3.5 Breadth of graphs containing strongly connected subgraphs

Graphs containing strongly connected subgraphs may be dealt with by logically separating the strongly connected portion from the rest of the graph. An example is shown in Fig. 3-8. Here, the separated subgraph is depicted as a cloud. The resulting structure is that of a directed acyclic graph, and accordingly can be treated as such with regard to counting and selection of paths (Section 3.3).

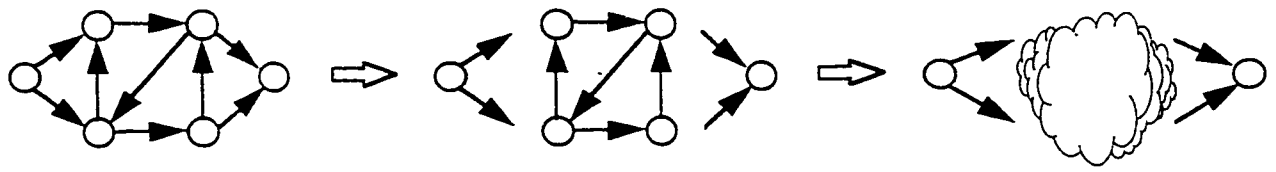


Fig. 3-8. *Break-out of a strongly connected subgraph. It becomes easy to see that only two paths are required.*

The understanding is that the traversal of the "cloud" is carried out in accordance with the procedure for strongly connected graphs, as described in Section 3.4.

In the event that there is more than one strongly connected subgraph within the graph, then each such subgraph can be treated as a cloud. The example in Fig. 3-9 shows three clouds in the same graph, which is now a directed acyclic graph.

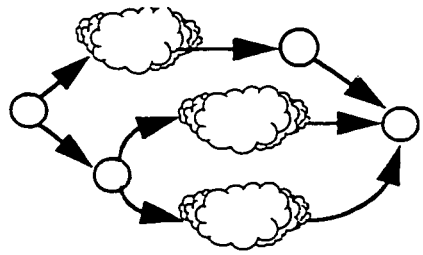


Fig. 3-9. *Multiple clouds are possible.*

It is also possible that the largest possible strongly-connected subgraph was not found the first time. Should this be the case, then clouds can be further merged with simple nodes or other clouds to form larger clouds. This process continues until the remaining graph of clouds and nodes is a DAG (refer to Fig. 3-10).

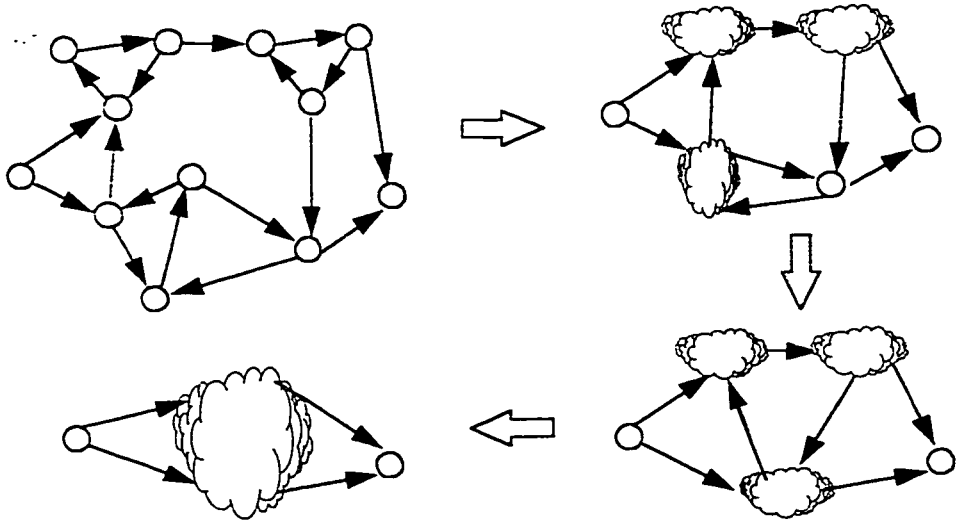


Fig. 3-10 (a), (b), (c), (d). *Successive mergers of clouds and nodes.*

3.5.1 Method for strongly connected subgraphs. In a graph containing strongly connected subgraphs, each of these subgraphs can be treated as a "cloud" having path-width

of 1. Clouds and nodes are merged until no portions are strongly connected. When no further merging can take place, the resultant graph of clouds and nodes is now a DAG, then Rule 3.3.1 for DAGs applies. ■

Alternatively, each cloud could be considered its own separate node, but this would introduce additional levels of hierarchy. This would be an unnecessary complication as we would lose the structure of a directed acyclic graph at this level.

3.6 Breadth of hierarchical graphs

Now that we have considered how to do counting in flat FSMs of various shapes, we must now consider how to do counting in a hierarchical graph. HFSMs can be handled by first considering the individual nodes at each level, as done in Sections 3.3 to 3.5. We saw that these sections established the minimum number of paths required to traverse a particular flat node. However, at any level, these single nodes will be linked to others in a larger node and in many different ways. Therefore,

> *we must now determine how many ways there are of traversing a hierarchical structure of large nodes containing smaller nodes.*

3.6.1 Counting at a single level

When individual nodes are part of a larger node, it is first necessary to determine how to traverse this larger node. We seek the minimum number of paths required to traverse the larger node while satisfying the specified minima at each of the smaller nodes.

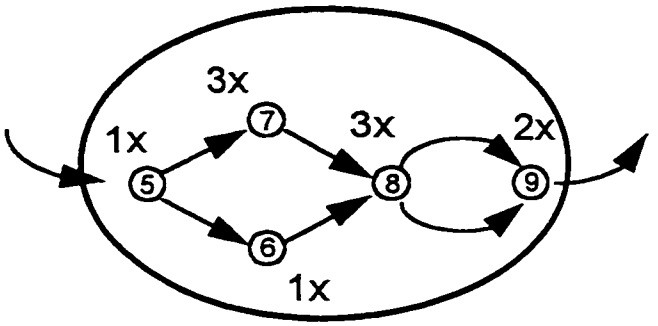


Fig. 3-11. *Node with minima labelled on subnodes.*
"3x" means that this node must be traversed 3 times.

In the example of Fig. 3-11, it is assumed that an initial analysis has been performed to reveal that some of the nodes need to be traversed more than once. The minimum requirement is indicated with the "x" notation.

Appendix 1 gives the algorithm for finding the maximum cut and minimum flow in a graph. This is used for counting the minimum number of test cases needed to cover a network. The algorithm can also be applied to networks having lower bounds on the edges; in the case of Fig. 3-11, we have lower bounds on the nodes. By applying the following simple transformation, we can make the graph suitable for the Appendix 1 algorithm.

3.6.1.1 Transformation.

Step 1. For each node that must be traversed 'k' times ($k > 1$), either

1a. replace it with a pair of nodes having k (one-way) transitions between them, or,

1b. *(alternate)* replace it with a pair of nodes joined by a single (one-way) transition labelled 'k'.

Step 2. Label all remaining transitions with a '1'. ■

A graphical example of Step 1 is shown in Fig. 3-12.

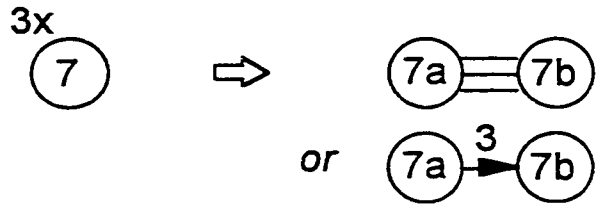


Fig. 3-12. *Transformation: Node 7 is split into two halves connected with a multi-edged bridge.*

After applying these two steps, Fig. 3-11 would now resemble Fig. 3-13.
Note that we do not have a tool to perform this node splitting, and certainly not graphically.

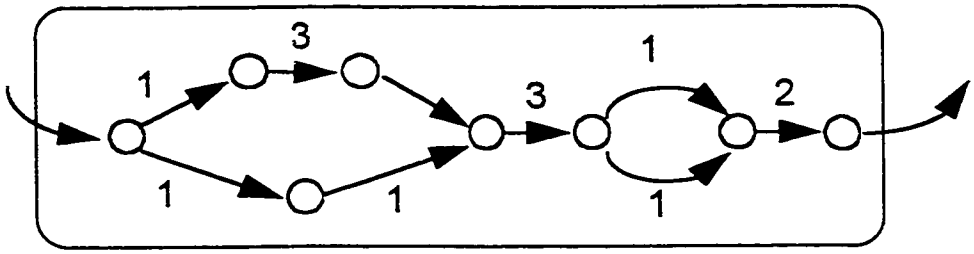


Fig. 3-13. *Re-stated version of problem in Fig. 3-11. Numbers indicate lower bounds on edge flows.*

3.6.1.2 Rule for multiple traversals of nodes. In a directed acyclic graph having nodes that need to be traversed more than once, the minimum number of paths needed to cover all transitions at least once and satisfy all the node restrictions is equal to the capacity of the maximum cut in the graph after applying Transformation 3.6.1.1 above. ■

This rule is evident in that the transformation merely forces us to have the required number of paths through the nodes. At this point, the problem becomes the same as for an ordinary DAG and handled as in Section 3.3.

3.6.2 Counting through the hierarchy

Now that we know how to count the paths through a single (large) node, we need a method of combining the path-counts in the hierarchical structure. The general idea is that the paths in a node can be counted only if the paths in all its subnodes have been counted.

And, we realise that at the lowest levels, the path-count of a leaf node is 1. Therefore the total number of paths required for the entire hierarchy can be derived "from the bottom, up." Appendix 3 gives a recursive algorithm for traversing an HFSM.

3.7 Application of breadth to test path selection

Using the methods and rules described in Sections 3.3 to 3.6, a path-count can be derived for every node. However, counting the paths does not guarantee that a path cover of this size exists. Obtaining a path cover consists of constructing a path cover of that size (which we do) and replacing infeasible paths with feasible ones (which we do not do).

3.7.1 In a DAG structure

For directed acyclic graphs with only a single point of entry and exit, the same algorithm that computes the path-count (Appendix 1) will also indicate the number of paths employing each edge. From this, we can determine which routes the paths actually follow. This comes from knowing that the number of paths entering a node must equal the number of paths exiting. Appendix 4 gives an algorithm for this.

Recall the example in Fig. 3-6. The number of paths was determined by finding the maximum cut. Application of the max-cut algorithm in Appendix 1 also finds the flow on each edge (Fig. 3-14).

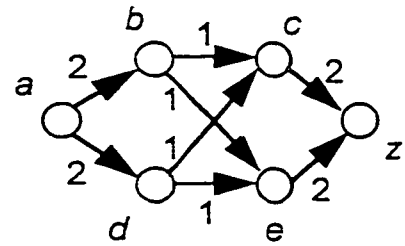


Fig. 3-14. *Looking at the flow on each edge for Fig. 3-6.*

By inspection, or by application of the algorithm in Appendix 4, we may conclude that the four paths are as shown in Fig. 3-15.

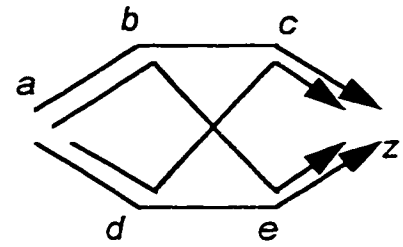


Fig. 3-15. *Inferring the routes of the 4 required paths for Fig. 3-6.*

3.7.2 In a strongly connected graph

We can use an algorithm for finding the *circulation* in a closed network to determine the number of times each edge must be traversed. A circulation is a network flow in which the sum of all inputs at each node is equal to the sum of all outputs at each node; there is no source or sink. We would expect a circulation at the top level of any HFSM that is modelling a real-time system that runs continually, that is, one in which there is no "final" state. Although there needs to be only one path, its route still must be selected. As a route, we may use a Chinese postman tour (CPT) (defined in Section 3.2).

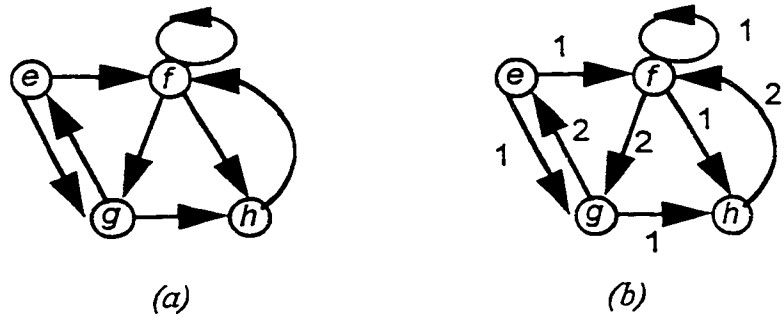


Fig. 3-16. *Obtaining the circulation in a closed network.*

Given the strongly connected graph in *(a)*, we can obtain the circulation shown in *(b)*. By inspection, a CPT of this graph is *e-f-g-e-g-h-f-h-f-f-g-e*. (The total path length is 11 transitions.)

3.7.3 In a graph containing a strongly connected subgraph

Intuitively, it should be possible to isolate the strongly connected subgraph from the rest of the graph. This isolation may or may not cause the rest of the graph to become disconnected. Within the subgraph, there may be a set of possible initial nodes and a set of possible final nodes instead of having a single starting node or ending node. Let the set of *initial nodes* be defined as those nodes of the strongly connected subgraph that are incident with non-subgraph nodes that can be reached from the beginning of the graph (the single point of entry). Similarly, the set of *final nodes* can be defined as those nodes of the strongly connected subgraph that are incident with non-subgraph nodes leading to the end of the graph (the single point of exit).

In selecting a path for the subgraph, we may use a Chinese postman (CP) path. Then we must find the shortest transfer sequence from the last node in the CP path to a final node. (If this is already a final node, simply continue in the rest of the graph.)

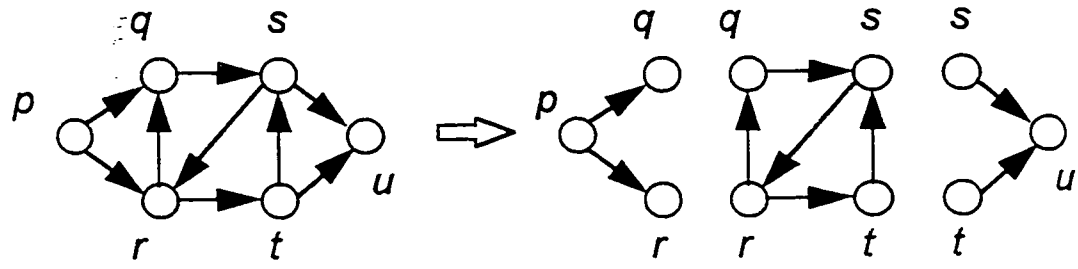


Fig. 3-17. *Graph containing a strongly connected subgraph.*

Example.

Consider the graph in Fig. 3-17. Initial nodes = $\{q, r\}$. Final nodes = $\{s, t\}$

An RCP path of the strongly connected subgraph is *r-q-s-r-t-s*. This can be prefixed by the *p-r* transition and suffixed by the *s-u* transition to form

Path 1 = *p-(r-q-s-r-t-s)-u*

This first path guarantees that we have covered all transitions inside the strongly connected portion. We now augment with additional paths to cover the remaining transitions outside. To cover edges *p-q* and *t-u*, we may use

Path 2 = *p-q-s-r-t-u*. ■

Summary of chapter.

In this chapter, we showed the difficulties behind our original requirement, which was to find a practical strategy for testing hierarchical finite-state machines. Flattening a hierarchy had the potential to lead very quickly to an explosion in the number of states. A small example re-inforced the magnitude of the problem, showing how expansion could be estimated. We investigated transition coverage of the underlying control-flow graph as a simple start. We recognised that this approach was dependent upon the "breadth" of the "fattest" node. Therefore, we came up with a derived requirement (sub-section 3.1.5) to find the maximum cut. All this work was based on the inherent graph structure, rather than on traditional FSM-based test methods.

We presented our approach, basically a very simple graph traversal strategy, for the counting and potential selection of paths in an HFSM, and we showed how it could be applied to various shapes of HFSM. The next chapter shows how our approach can be applied specifically to the ROOM extensions of HFSMs.

4

Pragmatic Considerations for ROOMcharts

The approach developed in Chapter 3 is based strictly upon the hierarchical structure of an HFSM. However, the ROOM modelling language supports extensions to the HFSM structure, largely based on Harel's statechart notation. As indicated in earlier statements regarding scope, we will not be addressing the structural aspect of ROOM, only the behavioural aspect (ROOMcharts). In this chapter, provision is made to handle some of the additional features of statecharts and ROOMcharts, specifically, group transitions, history, choicepoints, triggers, entry and exit code.

4.1 Transitions: group, history, and choicepoints

4.1.1 Group transitions

Directly related to the hierarchical structure are the concepts of *group transition* and *history*. Using an example from [Harel 87], a transition from grouped states enables us to model a statement such as, "In all airborne states, when yellow handle is pulled, the seat will be ejected." The hierarchical concept of a group state represents an exclusive-or (XOR) abstraction of all its contained states. The group transition represents an individual transition emanating identically from each of the smaller states (Fig. 4-1).

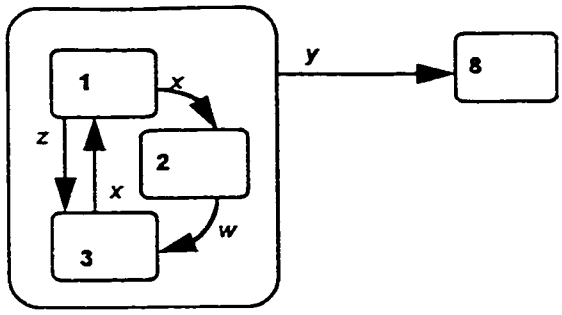


Fig. 4-1. *Example of a group transition, 'y'.*

The notation in Fig. 4-1 indicates that it is possible to go from state 1 to state 8 or from state 2 to state 8 or from state 3 to state 8, all by way of transition *y*; in other words, there are actually three transitions (1, 8), (2, 8), and (3, 8), but they are all transition *y*. This is illustrated in Fig. 4-2.

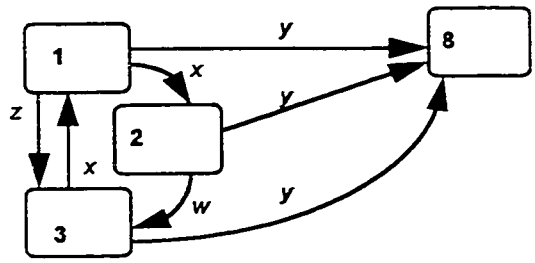


Fig. 4-2. *Expansion to the non-hierarchical equivalent of Fig. 4-1*

Our approach is that we will exercise the group transition only once, independent of what the originating substate was. This single test case is considered representative of all other transitions. A stronger coverage requirement would include exercising the transition from every substate, but this implicit expansion would not take any advantage of the hierarchy and would produce many more test cases.

Example. A traversal of the hierarchical (non-flattened) FSM in Fig. 3-3(a) would be *(x1) a2 (x2) a1 (x2) b (y1) b2 (y2) b1 (y1) a (x1).* Notice that we did not need to flatten the HFSM. ■

4.1.2 Top-level group transitions

There is another kind of group transition called a *top-level group transition* which is shown as being drawn from the border of a ROOMchart rather than from a specific substate (Fig. 4-3). This serves to emphasize that this transition can be triggered from any substate. In the case of the *timecheck* in Fig. 4-3(b), it also returns the system to the state it was in prior to execution, thus the action is handled without affecting the state of the system. The net effect is that of an interrupt-like activity. The convention of drawing them from the border is just 'short-hand' (a tidier representation), since the same action/transition can simply be drawn from every sub-state. Of course, this would result in a more cluttered diagram.

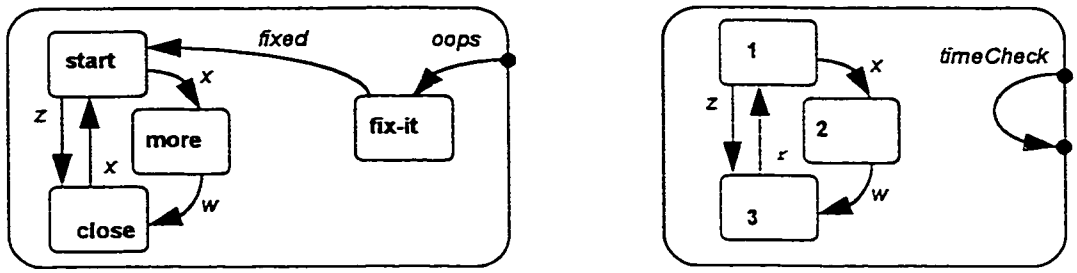


Fig. 4-3. *Two examples of a top-level group transition (drawn from the border) which deals with events, independent of the current state.*

Because a top-level group transition (for example, *oops* in Fig. 4-3(a)) is accessible from any state (substate), it is easy to design a path that includes it. Therefore, when counting the minimum number of ways through a state, a top-level group transition will likely not add to the total. If, however, such a transition forces a premature exit from the state (Fig. 4-1), then it will not be possible to continue the path, and a new one may have to be started. Intuitively, we would want to traverse as many other transitions as possible before taking an exiting transition.

It should be noted that the current algorithms do not account for top-level group transitions. The algorithms are based on taking a maximum cut in a connected graph, but top-level group transitions do not "appear" as connected to anything except the border. Manually, though, they are easy to accommodate in the counting and selection process.

4.1.3 Transitions to history

The "reverse" of a group transition is a transition to history. The *history* of a state is defined to be the substate that was last active upon the previous departure from this state. If the state has never been active before, then the history is not defined. Instead, the substate that is identified as the *initial state* is entered. (In Fig. 4-4, the small black circle indicates the initial state.) Harel uses a circled "H" to denote history; in ROOM, a transition to history is drawn terminating at the border (as in Fig. 4-3(b)) or Fig. 4-5).

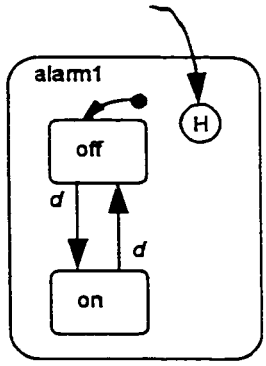


Fig. 4-4. *Example of transition to history in an alarm watch. (from [Harel 87])*

Again, using the principle of group transitions, it is considered sufficient to exercise this transition only once.

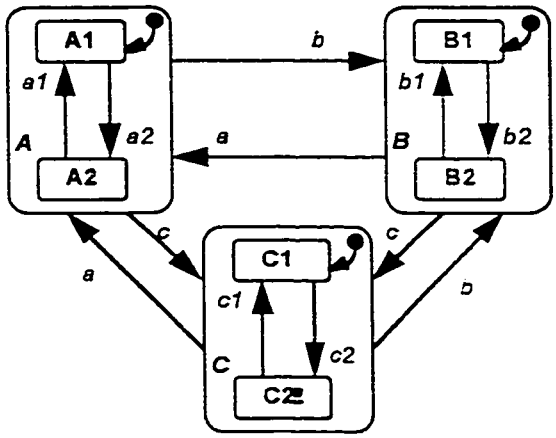


Fig. 4-5. *Non-combinatorial coverage. (from [Selic++ 94])*

Example

A transition tour of Fig. 4-5 would be the following transitions (and states listed in parentheses): *(A1) a2 (A2) b (B1) b2 (B2) c (C1) c2 (C2) a (A2) a1 (A1) c (C2) c1 (C1) b (B2) b1 (B1) a (A1)* for a total length of 12 transitions. Another possible transition tour is *a2.a1.b.b2.b1.c.c2.c1.a.c.b.a* which is also 12 transitions in total. This example is simple because each FSM contains a strongly-connected DAG and therefore requires only 1 path, but again, notice that the HFSM did not have to be flattened. ■

4.1.4 Transitions with choicepoints

ROOM permits a transition to split up into mutually exclusive branches terminating at different states, depending on some calculation within the transition. A *choicepoint* is situated in the middle of the transition.

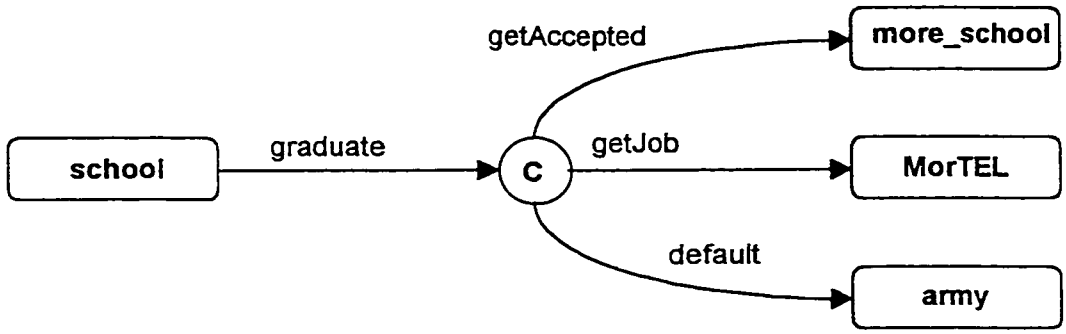


Fig. 4-6. *Choicepoint, 'c', with three choices, one of which is a default.*

One of the branches is required to be a default branch, to be taken in the event that all other branch conditions evaluate to "false." The counting and selection method should treat each branch as a separate transition to be executed at least once (see Fig. 4-6).

4.2 Triggers and signals

4.2.1 Multiple triggers

Special attention needs to be given to the transition triggers of ROOM. Recall from Section 2.2 that a *trigger* in the ROOM modelling language is simply the receipt of a message at a port. In ROOM, the format for a trigger definition is

```
t : {portName, signalName, optionalGuardName}.
```

Thus, a transition with its triggers, guards (conditions), and actions might be written as:

```
transition gotoErrorState:
  triggered by: {
        {deviceStatus, carrierDetect, (c2 == NONE)}
    or  {deviceStatus, buffer, full()} }
  action: {
        logError();
        displayError() }
```

A transition is taken whenever its trigger condition is satisfied.

ROOM permits the use of *multiple triggers* (as in the preceding example), each of which may have a guard associated with it. Therefore, when we say that we intend to exercise a transition, we still need to identify which trigger will be used. Each trigger for a transition may be considered a different path to that transition or may be considered a separate, but similar, transition.

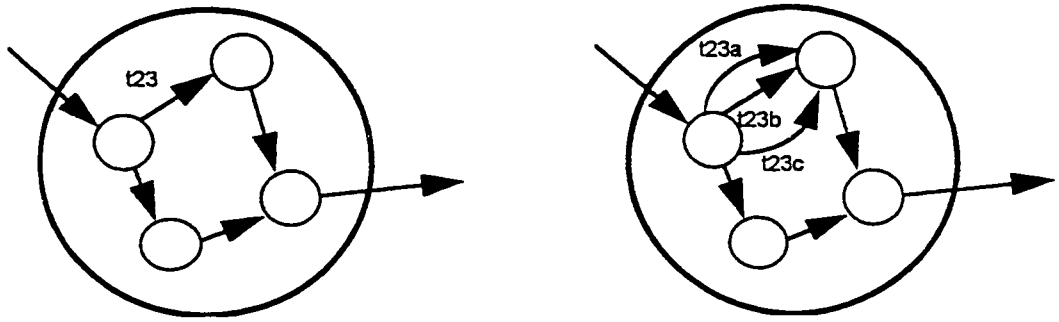


Fig. 4-7. *A transition with three triggers is treated as three parallel transitions.*

Similarly, if a single trigger has multiple conditions, possibly dealing with different signals on different ports, then each condition could be treated as a separate single-triggered transition. For example,

IF (RAIN .OR. SNOW .OR. HAIL) THEN TRANSITION23

could be treated as the following three

IF (RAIN) THEN TRANSITION23a

IF (SNOW) THEN TRANSITION23b

IF (HAIL) THEN TRANSITION23c

where all three transitions are actually the same as the one original transition. Fig. 4-7 shows a conceptual re-drawing of the graph.

It should be noted that a ROOMchart in ObjecTime need not actually be re-drawn; but, this is what the counting and selection algorithms would effectively do to perform their calculations. It is also acknowledged that combinations of conditions will not be attempted; as examples, (RAIN .AND. SNOW), or (HAIL .AND. RAIN), or all three at once.

If trigger conditions cannot be broken down into discrete values, for example,

IF (32.0 $\leq F \leq$ 212.0) THEN TRANSITION23

then we may apply the usual black-box strategies for handling a range of values, such as taking a value on each of the boundaries, just outside the boundaries, and one or more "representative" values inside.

Thus, we can think of a trigger as having a "width." Covering this width results in *coverage of all triggers (condition coverage)* throughout the HFSM.

4.2.2 Signals, ports, entry code, and exit code

Triggers and transitions are associated with specific ports. The following example shows the definition of a transition called paperHasBeenInput, from the behaviour specification of a controller module in a hypothetical fax machine.

```
TRANSITIONS
        {
        DEFINE paperHasBeenInput
                FROM STATE idle
                TO STATE waitForPhoneNumber
                TRIGGERS
                        {
                        DEFINE SIGNALS {PaperIn} ON {faxInputArea}
                        }
                ACTION
                        {
                        SEND displayPanel SIGNAL documentReadyMessage
                        ENDSEND
                        };
        };
```

The notation used above is called ROOM Linear Form, corresponding to the graphical form of ROOM, used by ObjecTime. Keywords are in capital letters. In this example, the signal PaperIn is expected on the port faxInputArea. When this occurs, the signal document-ReadyMessage will be sent on the port displayPanel.

Assuming that all input signals (all possible elements of the input 'alphabet') occur in some trigger somewhere, then *coverage of all transitions and triggers* will result in *coverage*

of all input signals. If, for some reason, there is an input signal that is defined but not used, it will not get tested.

In the preceding example, an action, specifically the transmission of an output signal, was specified in the ACTION block of the transition definition. However, actions can also occur upon entry to a state (independent of which transition was used to enter) and upon exit from a state. Since output signals, or *indicators*, are contained in actions, and all actions are located in a transition, in entry code, or in exit code, then *coverage of all states and transitions* will result in *coverage of all output signals*.

By default, all ports will end up being used at least once, unless there is a port with no signals defined. What will not be covered is all *combinations* of input and output signals.

4.3 Coverage achievable

The initial idea of starting with transition coverage actually attains considerably greater coverage than simple coverage of transitions – particularly when the ROOMchart is manipulated, conceptually or otherwise, to make itself more amenable to the general counting and selection algorithms. We summarize here.

The most direct result of *covering all transitions* is that *all states will be covered*. This comes from the reasonable assumption that all states are reachable, obviously by way of a transition. This assumption is reasonable because of the visual or graphic nature of the HFSM. Covering a state implies that *any entry code or exit code will also be covered*.

In the same way that some states contain more than one path through them, some transitions may also require more than one traversal, if we wish to cover all triggers. If each trigger is treated as a separate transition (to the same state), and if each trigger with more than one condition is treated as a separate transition (as discussed in Section 4.2.1), then a transition coverage algorithm will *cover all triggers*. In a reactive real-time system, all func-

tional inputs end up as triggers, therefore although not all possible inputs will be tested, all types or kinds of input (from a control-flow standpoint) are covered.

Because all transitions are covered, all action blocks are executed at least once. And, as pointed out above, all blocks of state-entry code or state-exit code are also executed at least once. At the black-box level, this will certainly include *all customer-visible outputs*, but only if the blocks of entry code or exit code are strictly linear in structure, that is, they contain no branching constructs. (This variation is discussed briefly in Section 6.1.1 in the chapter on future work.)

Finally, whereas it might seem reasonable or practical to cover only part of the hierarchy (to obtain partial coverage at lower cost and then estimate the amount remaining uncovered), the recursive algorithm provided in Appendix 3 covers all levels.

4.4 ROOM example

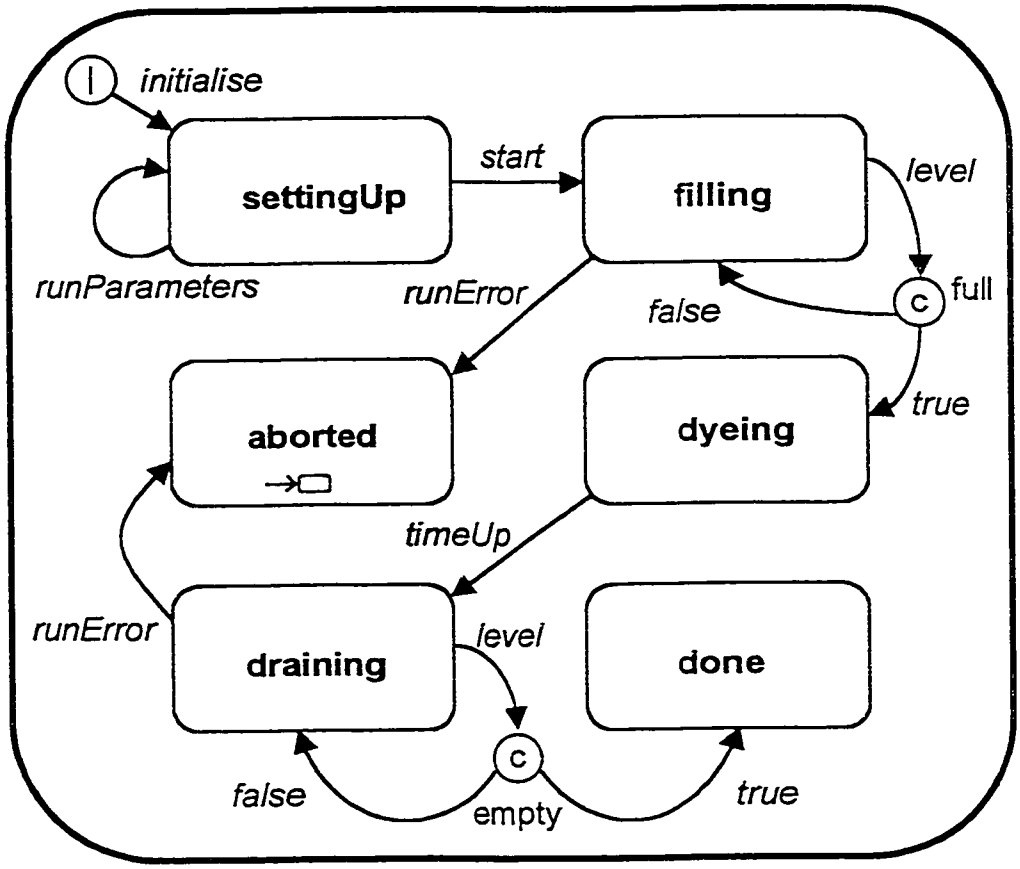


Fig. 4-8. *ROOMchart for a Dyeing Run Controller. (from [Ward 95])*

In the ROOM and ObjecTime references, the main instructional example used for modelling is that of a real-time "fabric dyeing control system." The basic idea is that there is a vat or tank in which there are many racks covered with cloth or fabric that is to be dyed. After setting some switches (such as a timer and temperature), the vat is filled with dye, and the timer is activated. After the requisite time has passed, the vat is drained. To control each run, a controlling device is used, referred to here simply as a "dyeing run controller."

Fig. 4-8 shows the top-level behaviour of a dyeing run controller. Additional explanation is required for some notation that was not covered in this thesis. In the top-left corner of the diagram, the vertical line (letter "T") in a circle is simply the initial transition point. The

small arrow and oval beneath the word "aborted" is ObjecTime notation means that there is entry code associated with this state (but not visible).

We can see that this network has one source ("I") but two sinks ("aborted" and "done"). We will not pretend that this is a very complete model, since there is no provision for errors while in the "dyeing" state, nor does there appear to be any way of recovering once the system goes into the "aborted" state. To convert this into a graph having a single point of entry and exit, we can add an artificial single end-state after both "aborted" and "done". Sections 5.6.2 and 6.3 will discuss multiple points of entry and exit.

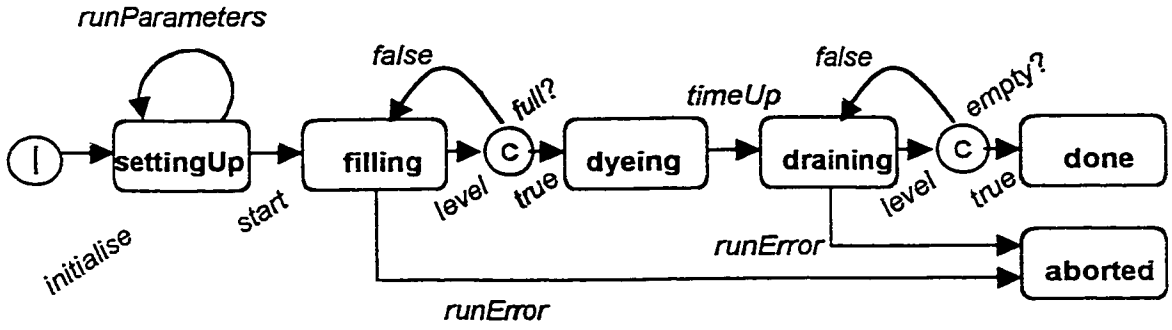


Fig. 4-9. *Fig. 4-8 re-drawn to show that the maximum cut is 3.*

From a quick inspection of Fig. 4-9, it can be seen that there are two end-states with three possible ways of entering; therefore, it will take at least three paths to cover this graph. One can confirm that the following three paths will indeed cover all transitions and states.

Path 1. *(I) initialise (settingUp) runParameters (settingUp) start (filling) level (c) false (filling) level (c) true (dyeing) timeUp (draining) level (c) false (draining) level (c) true (done)*

Path 2. *(I) initialise (settingUp) start (filling) runError (aborted)*

Path 3. *(I) initialise (settingUp) start (filling) level (c) true (dyeing) timeUp (draining) runError (aborted)*

This particular ROOMchart does not represent a very complicated HFSM, but shows the general idea.

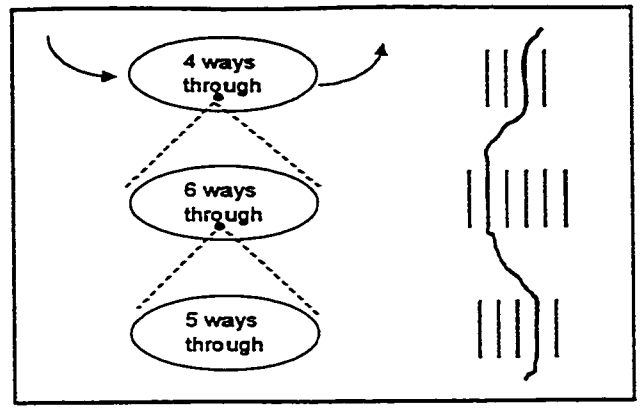
5

Assessment and Implementation of Approach

In the opening chapter, two requirements were given for any strategy to be useful in a commercial application of any significant size. These requirements focussed on a strategy's ability to scale up and its ability to be automated. This chapter explains to what degree our proposed approach strategy satisfies these requirements. Also included here is a general discussion on the overall assessment our approach, the ways in which it could be used, some limitations on its applicability, and a typical testing framework. Our particular implementation is also discussed.

5.1 Requirement R1: Scaleability

As seen in Section 3.1, flattening a hierarchical finite-state machine may result in a dramatic increase in the number of states. Even if it does not, the number of combinations of paths selected from each level of an HFSM grows multiplicatively with the number of levels. This greater-than-linear increase in the number of states or paths constitutes a significant practical obstacle to most formal methods. By comparison, the proposed method, which is dependent only upon the "thickest" portion of each node, can be applied to large applications without a disproportionate increase in testing cost. Recall Fig. 3-5(b), reproduced below.



Reproduction of **Fig. 3-5(b).** *An additional level does not necessarily increase the number of paths.*

We take further advantage of the hierarchy by exercising any *group transition* (recall Fig. 4-1) only once, not differentiating amongst cases for individual substates. For the purposes of a group transition, a transition from a single substate (represented by a single test case) is considered representative of transitions from all the other substates within that same state. Although a group transition could be repeatedly exercised from every substate, this implicit expansion would not take any advantage of the hierarchy and would produce more test cases. This treatment of group transitions further contributes to the scaleability of our approach.

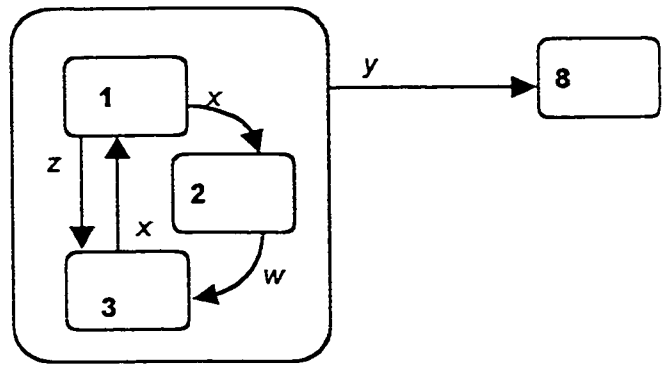


Fig. 5-1. *It is considered sufficient to exercise transition 'y' only once, as initiated from any of substates 1, 2, or 3; it is not necessary to repeat for each substate.*

Thus, with our strategy, it was shown (in Section 3.1) that as the number of states increases, the number of additional paths required for testing increases only linearly, at worst. Therefore, this strategy is considered scaleable and would be suitable for large-scale applications.

5.2 Requirement R2: Automatability

In order for any method or strategy to be practical, there must be potential for its ability to be automated, in full or in part. If an optimal solution can be found in automated fashion at reasonable cost, then it should be used. However, for practical purposes, it is more important to obtain a *cost-effective* implementation of our strategy. Therefore, a software tool may use an efficient heuristic if no optimal solution is readily available.

The proposed strategy is considered amenable to automation, as there is an algorithm for each of the steps. The algorithm for path counting is given in Appendix 1; Appendix 3 gives some of the pseudo-code for Appendix 1; Appendix 4 gives the algorithm for path selection; Appendix 6 gives source code in C.

The results of executing our strategy can also be made visible to the test designer or user. Because the ObjecTime toolset already displays graphs and paths, the effects of path counting and selection can be shown graphically. For example, a transition that is expected to be traversed a number of times could have that number displayed beside it, or it could be drawn proportionately thicker or coloured differently. A path that is *selected* could be highlighted in a stationary or animated fashion. Such visibility would fit in well with the functionality, look and feel of ObjecTime.

There are also occasions where the test designer or user could be prompted for guidance (graphically). One instance may be when encountering infeasible paths (which are discussed in a later subsection, 5.4.1). Of course, a user/designer may wish to have some sort of "override" capability to customize the generated test cases.

5.3 Measures and metrics

It is useful for any concept implemented as a tool to provide some measure(s) of what it accomplishes or covers. Depending on the needs of the end-user, a number of different

statistics can be made available. At the most basic level, the proposed method provides a lower bound, called the ***breadth***, on the number of paths (test cases) that would be required to cover all transitions, all states, all triggers (input signals), and all indicators (output signals) that are accessible in the basic hierarchical structure, under the assumptions listed in Section 4.3.

Although the implemented version of our tool (Appendix 6) computes only the breadth, the following additional outputs or measures could potentially be tracked and displayed:

a. the total length of the test cases (in terms of the number of transitions or states);

b. number of times each portion (transition, state, other) would be covered;

c. the routes of the suggested paths for each test case (assuming no infeasible paths);

d. an indication of which traversals are necessary to satisfy the minimum (at this level or below) and which were suggested to satisfy a requirement elsewhere along the same path (room for optimization here, if weights are supplied);

e. an estimated cost, if costs were supplied.

Knowing the minimum number of paths needed to cover all transitions gives an indication of the complexity of the design and consequently of the testing effort that would be required. The more paths there are, the more complex the design and the greater the effort required. Similarly, the lengths of each path as well as their overall total length also provide an indication of testing effort required. *Ours is a **lower bound.***

As a point of reference, this information can be compared to simpler measures such as the total number of states or transitions. For example, two designs (for two different applications) could have the approximately the same number of states or transitions (and therefore appear to be the same "size"), but one design may require significantly more paths, thereby revealing its greater internal complexity. Decreased complexity is therefore associated with increased testability.

By choosing to minimize the total number of paths rather than the overall total length, it is implicitly assumed that resetting the system to start a new path would be very costly compared to lengthening the path. Since resetting a system often involves manual intervention by an operator, path length has been subordinated to path count. However, it should be clear that if relative costs can be supplied, a more elaborate algorithm could be used to minimize the total cost. Weighted totals and other optimization ideas are discussed further in Chapter 6.

5.3.1 Quality and metrics

The reader will recall the discussion in Section 2.5 regarding *complexity*. It is worthwhile to note that this design attribute is sometimes referred to as the converse of simplicity (and therefore of maintainability and of testability) [Budgen 94]. Of particular interest is "cyclomatic complexity" because it too is based on path counts, like our own metric. Studies have found a strong correlation between McCabe's cyclomatic complexity index and the number of test defects [Kan 95], therefore path count is truly indicative of the design complexity.

Because cyclomatic complexity is a measure of control complexity, this metric is particularly significant for control-oriented applications (as opposed to data-oriented). Since control applications are what statecharts are typically used for, this metric is very relevant.

Our approach can also be complemented by instrumentation tools to assess coverage. (Recall that coverage was discussed in Section 4.3). In general, increased coverage is tied to increased reliability in software. It has been found that typical testing without code measurement will exercise only about 55 percent of the code. It has been found that it is not difficult to achieve 80 to 85 percent coverage for systems-level testing or integration testing when one uses a tool [Grady 92]. Ultimately, we are assessing design quality.

From the human factors standpoint, there is also the psychological effect of having access to metrics. Programmers will try to reduce the complexity of their revised designs or code if they can obtain relevant metrics easily. Our approach would be able to provide such a simple metric.

5.3.2 Comparison to McCabe

Whilst both our metric and McCabe's metric are based on path counts, there are some differences. As examples:

- Consider the FSM in Fig. 2-10 (repeated here below). The stated cyclomatic complexity is 2; however, by inspection we can see that it would take only one pass (one path) to cover the entire graph.

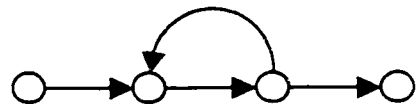


Fig. 2-10. *A control graph with cyclomatic complexity of 2 [After Beizer 90].*

- Consider the FSM in Fig. 2-11 (repeated here below). The McCabe metric is 5; however, we can see that it would take only three paths to cover this graph.

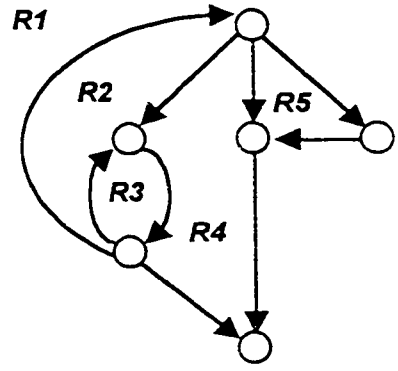


Fig. 2-11. *A control graph showing the number of regions (5). [After Pressman 92]*

By measuring essentially only the branching points, the McCabe metric clearly does not take into account a dynamic activity such as the fact that one can traverse self-loops and

cycles any number of times. Although our metric is also a static metric, it does take self-loops and cycles into consideration for path counting. Thus, the breadth metric is a true lower bound on the number of test paths based solely on FSM structure.

The reader will recall Beizer's comment in sub-section 2.5.5 (on metrics and testing) that the relationship between the McCabe metric and the number of paths required for branch coverage was "circumstantial". For our breadth metric, this relationship is exact, as shown below.

5.3.3 Corollary: Breadth = branch coverage

The theorem for maximum cut and minimum flow is provided in Appendix 2. As a corollory to this theorem, we can state that the breadth, as computed in our approach, provides us with branch coverage.

Corollary 5.3.3. The minimum number of paths required to obtain branch coverage is equal to the breadth.

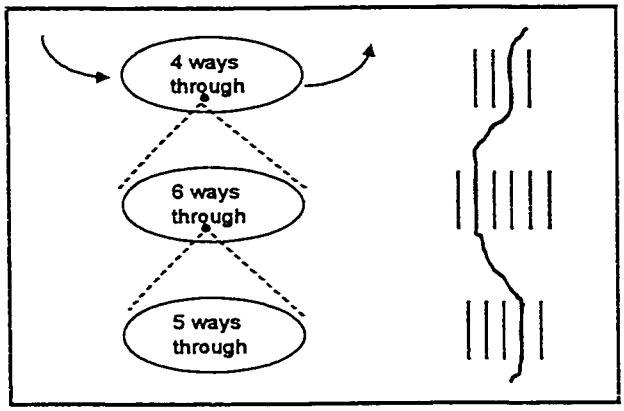
Proof.

Explanation of terms. Beizer defines "branch" as "a program point at which the control flow has two or more alternatives. A node with two or more outlinks." [Beizer 90]. In this thesis, branch is taken in the context of design. By "branch coverage", we mean that our testing is sufficient to cover 100 percent of all branches in the design. By "coverage", we mean that the item being covered is executed at least once.

Branch coverage = min flow with lower bound of 1 on all edges. Given that our designs are modelled as HFSMs which in turn are structured as directed graphs, and that networks are also directed graphs, we may treat the design as a network. Insisting that each branch of the design be covered at least once is the same as requiring that all edges in the corresponding network have at least some flow – in other words, that each edge have a flow of at least one

unit. This is, by definition, the "minimum flow" for a network where the lower bounds are set to one on each edge.

Breadth = max cut. As presented in Section 3.1.5, the *cut* in a graph is a partitioning of a graph into exactly two sets. The value of the cut is sum of the weights on the edges that cross between the two sets. The maximum cut is the cut where the sum of the weights is maximised. Now, if we take the graph to be the network discussed above and the weights on each edge to be exactly one unit, then we have found the partition with the greatest number of crossing edges – essentially the "fattest" part of the graph.



Reproduction of **Fig. 3-5(b)**

As alluded to graphically in the reproduction of Fig. 3-5(b) above, this fattest part is exactly what is computed in the breadth metric. In this fashion, we see that max cut is the same as breadth.

However, by the Maximum Cut Minimum Flow theorem, max cut equals min flow, therefore breadth equals (the minimum number of paths for) branch coverage. ■

5.4 Relationship to other approaches

In the preceding sections, we covered the three characteristic requirements for a useful test strategy. This section discusses at a more general level how the proposed strategy relates to other facets of software testing.

5.4.1 Grey-box aspects

Our approach is considered to be 'grey-box' strategy because it operates at the system's design level. In general terms (not specific to ROOM), variables that are associated with externally visible inputs are called *triggers*; those with outputs are called *indicators*. With our approach, we are testing all externally visible system inputs and outputs. By looking at communication signals, we are actually checking internal system interfaces. If system modules are difficult to analyse, it is a sign of potentially poor testability. The approach that we are taking is effectively, then, a grey-box method for communicating extended hierarchical finite-state machines.

5.4.2 Black-box aspects

Section 2.3.4 introduced black-box testing as functional testing, centred on the trying of varying inputs in accordance with the program's specifications and watching for the expected outputs. Our strategy retains some elements of black-box testing in its coverage of all inputs and the manner in which they can be covered. First, recall from Section 4.3 that a transition with multiple triggers can be re-written as separate transitions with individual triggers. In this fashion, transition coverage implies coverage of all triggers. Since this is where inputs are manifest, all inputs are covered.

In the cases where inputs are not drawn from a simple set of discrete values or where triggers have guard conditions associated with them, we may also apply traditional black-box techniques. *Equivalence partitioning* (or *input-domain partitioning*) consists of identifying

equivalence classes within the domain of all possible inputs and selecting representative test cases from each class [Myers 79]. *Boundary-value analysis* (or *boundary-interior analysis*) carries this approach one step further in that the input values for each equivalence class are selected specifically from the boundary values and from very close to the boundary [Myers 79]. Such test cases are observed to be more effective than with average or typical values.

By further analysing the input values, we may also determine and "drive" what output values are produced. Thorough analysis will produce *output-domain partitioning.* As our algorithm selects path segments associated with program outputs, guidance is required from a user/designer to derive the appropriate test data inputs to ensure that those path segments are actually covered.

5.4.3 White-box aspects

Although white-box techniques encompass both control flow and data flow coverage, we address only control flow measures because finite-state machines describe control flow. We have already noted that the traditional white-box technique of complete *path testing* is not practical for pursuing control flow coverage, because of the dramatic combinatorial increase in the number of tests required as the state machine increases in size. Nevertheless, the node coverage we have described earlier is comparable to traditional *statement coverage*, and our most basic transition coverage is comparable to traditional *branch coverage.*

By extending our approach to handle multiple triggers and trigger guards, we achieve *condition coverage* as well. The condition coverage is limited to coverage of the triggers that are externally visible; we do not achieve condition coverage of embedded code nor will we necessarily achieve multiple condition coverage, all combinations of all condition decisions, which is often associated with logic-based testing.

In terms of *loop testing*, our approach reflects the strategy or selecting simple paths through the looping structure. With a traditional single loop, the choice is to take it or not

take it. Because our approach advocates transition coverage, all loops will be taken at least once. However, if any node containing a loop needs to be traversed more than once, a cost-driven approach will normally dictate that the loop be bypassed on all subsequent traversals. (This is where some intelligent user guidance would be required in the selection of paths.)

Because of the visual representation of ROOMcharts, traditional static methods such as code inspections or walkthroughs are very appropriate, but these methods are not the subject of this thesis.

5.4.4 Reachability

For communicating FSMs, a common technique known as *reachability analysis* is often employed, especially for communication protocols modelled as CFSMs [Holzmann 91]. This analysis does not involve executing the implementation under test, but rather simulating all permutations of communications by the CFSMs involved. The concept of a *global state* is defined by taking the composite of the states and the message queues of all of the CFSMs together; thus, a global state depicts a "snapshot" of the execution of the CFSMs.

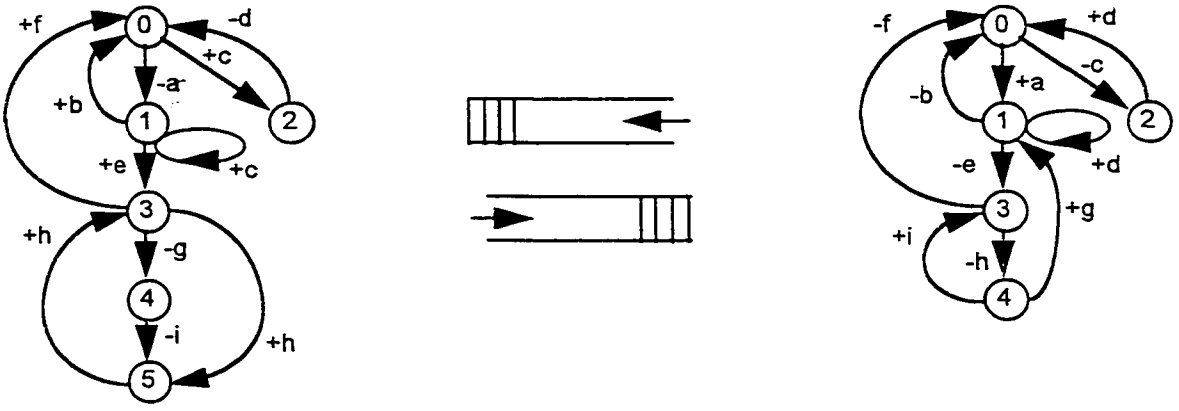


Fig. 5-2. *A global state is the composite of the individual states of the CFSMs along with the status (contents) of the communications channels between them.*

The purpose of reachability analysis is to verify the self-consistency of the model by trying to detect the existence of possible global states having certain undesirable conditions

such as *deadlock* (where no progress is made because the FSMs are both (or all) waiting for each other), *livelock* (where an endless cycle of execution is entered), or *unspecified reception* (which is the receipt of a signal for which there is no defined response).

The main practical drawback of reachability analysis is that the number of permutations or global states to examine for even two small CFSMs is extremely large. In the literature, this is commonly known as the *state space explosion problem.*

Our proposed approach does not require (or necessarily benefit from) reachability analysis. ROOM does not retain the statechart concept of global or orthogonal states (which were introduced in Section 2.2.1), since orthogonal states are modelled by separate ROOM actors, each having its own ROOMchart to describe its behaviour. Consequently, our approach examines only one state machine at a time. Whereas statecharts assume that transitions are instantaneous and that communications are reliable, ROOMcharts recognise that such modelling assumptions may result in specifications that are not feasible to implement. But, because ROOMcharts are already executable models (HFSMs) in themselves, designs modelled in ROOM can be tried out early in the development stages to detect and circumvent undesirable conditions.

5.4.5 Practicality

Our method consists primarily of counting and, to a lesser extent (depending on the prevalence of infeasible paths), selection of paths to be used for testing. The act of counting paths is actually an estimation of the testing effort required; the selection of paths is equivalent to test case generation. These are two very practical methods for the testing of software modelled as hierarchical finite-state machines.

As with other structure-based testing methods, our approach covers tests that a customer would not necessarily think of specifying. The difference here is that because the structure is at the design-level, the tests may be more relevant to the functionality.

Industry has criticized developmental test techniques for communication software, particularly finite-state machine testing methods, as being too focussed on the development of theories and not being amenable to practising engineers without a background in formal methods [Lai+ 95]. Because of the graphical nature of ROOMcharts and the ObjecTime toolset, any tool implementing our approach will make the path-count metrics easily displayed and understood, and should make the path selection/suggestion (test case generation) run well in a semi-automated environment with user input.

In fact, our metric represents the lower bound on the number of probes that one would require to instrument the system (assuming one probe per path, rather than k–1 for a k-way branch.)

Finally, we have already seen at the beginning of this chapter how our approach embodies practicality in three characteristics: ability to be scaled-up for large systems, ability to be automated, and ability to provide useful measures.

5.5 Application of approach

Seeing how our approach can be applied helps provide us with an assessment of our approach. As discussed in Section 5.3, the path count indicates the minimum number of test cases that would be required to cover all transitions, all states, all triggers, and all indicators. The count is considered a minimum, because certain paths may be unachievable. Therefore, more paths may be required to cover off the same number of transitions. (Section 5.6.1 discusses infeasible paths.)

As an example of its simplest contribution, suppose that a test designer has created a suite of test cases. Should the test suite contain fewer test cases than the calculated minimum number (according to our strategy), then it will be plainly evident that some parts of the design are not being covered at all. From here, a test designer or test manager can take further decisions and action.

Having such a metric enables a designer to immediately estimate the complexity of the design space. It is also useful for observing the effects of modifications (seeing whether complexity has gone up or down and by how much). Analysing the complexity of individual modules can be useful in estimating how much effort may be required to integrate them. This is especially useful when applied to externally supplied modules.

5.5.1 Using a cut-off depth

When dealing with a very large testing space or analysis space, it is reasonable to decide to cover only part of it. Although our proposed approach recursively covers the entire hierarchical FSM, it may be useful to apply some notion of a 'cut-off' depth, beyond which, analysis is not conducted. 'Pruning the state space' is a technique that is used with reach-ability analysis (which was introduced in Section 5.4.4). For this, we would need a simple counter to track and control the level of recursion. Such a capability may make our approach more suitable to industry use and probably more consistent with industry practice.

Therefore, although paths can easily be *counted* down to all depths, varying notions of cost-effectiveness may render it impractical to actually generate paths (test cases) that differ only at the deepest levels; in other words, we might be satisfied that *any* traversal of the lower-level nodes is considered sufficiently representative. To this end, a two-column table can be drawn up, relating the cut-off depth to the path count; the deeper one delves, the more different paths there are to analyse. By comparing these path counts to the total path count (for all levels), we also obtain percentages of coverage for each cut-off depth.

Depending on whether a module is new, old, mission-critical, or low-risk, it may even be desirable to adjust the cut-off depth in different portions of the HFSM. In areas where strong coverage is desired, we can remove any limits on recursion and apply our analysis down to the lowest-level nodes. In areas that we have already tested or where the observed yield is deemed not to be worth the effort, we may content ourselves with 'weak' coverage.

In the end, a risk-based, resource management decision must be made as to how much testing will be done and where.

5.5.2 Calculating costs or weights

As alluded to in Section 5.3 on measures provided, the 'cost' we have considered thus far is measured in the total number of paths. This may or may not be an accurate way of determining cost or effort. If the implementation details are known (or can be estimated), *costs* or *weights* can be associated with each link to represent computing time, relative effort, execution time, relative importance, criticality, or whatever we seek to measure or estimate.

We have seen in Chapter 4 that a path can also be described listing its constituent transitions (rather than its nodes) in the order in which they occur. In Fig. 5-3, there are three nodes *(63, 64, 65)* in the main node connected by six transitions labelled *t, u, v, w, x,* and *y*.

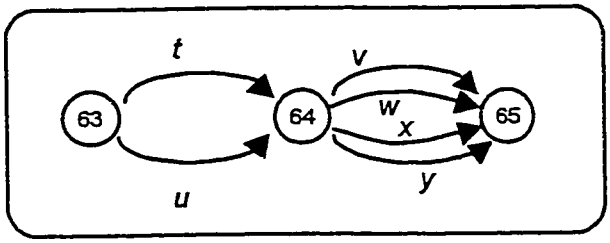


Fig. 5-3. *Costs may be associated with each transition.*

If the associated costs for *t, u, v, w, x, y* are *ct, cu, cv, cw, cx, cy* respectively, then the cost of traversing the large node by way of path segment *tx* is simply *ct + cx*. Since certain transitions may have to be traversed many additional times in order to satisfy minimum coverage requirements in nearby nodes, it will not matter which transitions are selected, after the first traversal. In such "don't care" situations, having such costs available to the test designer can let the designer assist any tool in selecting the most cost-effective paths. If the

respective path counts were *kt, ku, ... , ky,* then the total cost would be *kt·ct + ku·cu.+ ...+ ky·cy.*

A more advanced technique dealing with path expressions and regular expressions is dealt with in Section 6.2.

5.6 Limitations on applicability

5.6.1 Infeasible paths

One of the main limitations on the strategy presented is that some of the paths suggested by the method may not be feasible. The issue of path infeasibility arises regardless of whether the number of path combinations is computed hierarchically or not, because we do not take into account any restrictions that may be imposed by data values. That is, there may be a transition whose execution is dependent on a constant or variable that was set elsewhere. In ROOM, this can occur when a transition has one or more triggers with conditions (guards) that preclude its being executed unless certain data variables (that may have been set in previously executed FSM nodes) have particular values. Re-entrant states may or may not present such problems, depending on the *breadth* of the associated HFSM.

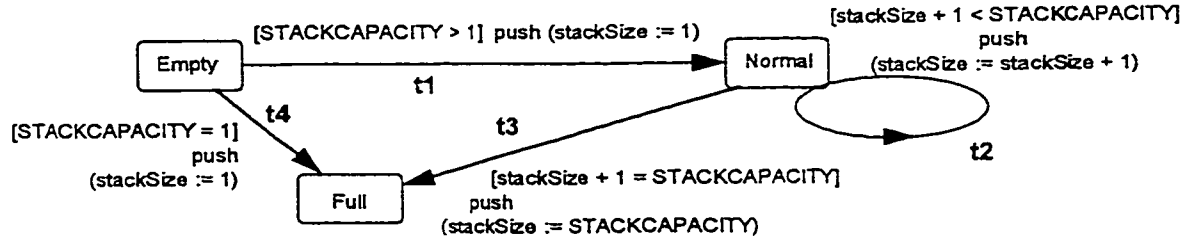


Fig. 5-4. *Example of an infeasible path: t1.t2.t3 is infeasible when STACKCAPACITY = 2.*

Fig. 5-4 shows a very simple example of a single node (perimeter not drawn) consisting of three subnodes. (The notation was explained in Section 2.1.2.) Note that there is not even any hierarchy here. It can be seen that the path *t1.t2.t3* (three successive "push" operations onto a stack) is infeasible if STACKCAPACITY is only 2. Thus, the feasibility of the described path is dependent upon the values of the program variables or constants. Such a situation could easily arise in ROOM, and would not be automatically detectable.

The next example, though not drawn as an FSM, shows how a path might be always infeasible because of the program logic. Note that it is not the trigger guards (the conditions) in themselves that present a difficulty, since it was shown (in Section 4.2) that they can be accommodated (counted and selected) individually at the design level. Rather, it is the combination of conditions. This example in Fig. 5-5 is taken from [Probert 95].

```
path-a
if ( (A > 1) .AND. (B = 0))
   then path-b
   else path-c
endif
path-d
if ( (A = 1) .AND. (B = 1))
   then path-e
   else path-f
endif
path-g
```

Fig. 5-5. *Sample flowgraph containing an infeasible path.*

Path *abdeg* is infeasible. This path represents the (hypothetical) case where both of the 'if' statements evaluate to "true." By inspection, we see that the conditions contained are such that this can never be the case – we can never have *B=0* and *B=1* at the same time. Therefore, any path containing them both is not achievable. This, too, is not detectable by looking at only the structure of the flow. As a postscript, it should be noted that it *is* possible to cover the code with two paths; however one of them will not be *abdeg*. (Example A=2 and B=0 will cover True and False; then A=1 and B=1 will cover False and True.)

In some cases, the existence of infeasible paths is indicative of program logic that can be rationalized (so that there are no infeasible paths). This commonly occurs when flags or switches or correlated between one part of code and another; in essence, the second switch is merely a deferral of the predicate from the first switch. In such a case, it is recommended that the code be re-written so as to avoid having the second branch point. This results in code with fewer paths, but all of which are achievable, and is therefore more testable.

There will be occasions where we must legitimately contend with infeasible paths. This is another reason why the path count obtained from our automated approach is only a minimum. To cover the transitions that could not be covered with the infeasible path, we will have to be augment our path collection by manually creating additional paths.

Trying to build paths under the restriction that certain statement pairs are impossible is difficult to do in an efficient automated fashion. In fact, it has been shown that the problem of determining the existence of an "impossible-pairs" constrained program path is actually NP-complete [Gabow++ 76]. One workaround strategy is to generate all the test paths first, see which ones are not feasible, and then apply an "adaptive" testing strategy [Probert 95].

5.6.2 Multiple points of entry or exit

The basic model for counting and selection assumes that each node has only a single point of entry and exit. This assumption corresponds with the assumption that the execution of paths is independent between sibling nodes (substates within the same state) and between parent- and offspring-nodes. Such independence makes calculations easier.

One way of dealing with multiple entry points is to create a new special sub-node at the 'front end' of all entry points to 'collect' all the entry paths, so that the remainder of the node sees all entries as coming from a single sub-node. A special 'back end' sub-node can also be created to collect multiple exits.

Another way of dealing with multiple points of exit is shown in the next chapter on future work (node-splitting).

Finally, it could also be argued that modules having more than a single point of entry or exit are indicative of poor program structure (for example, entering a module to re-use only a small portion of code and then exiting directly). In this case, it is recommended that the code be re-written to avoid this situation, thereby improving testability.

5.6.3 Main limitation

The most obvious limitation of HGS is that it applies only to that portion of the control-flow structure that is visible. For software that is truly modelled as a hierarchical finite-state machine, this is not a problem. However, ROOM permits actions to occur (code to be executed) upon entry to a state or upon exit from the state. Such structures are not visible in a ROOMchart. Additionally, some of the code on transitions may conceal branching structures that are also not detected by our approach. Proper white-box testing methods are in order here. The next chapter discusses some of these.

Although our testing strategy of attempting to cover all inputs and outputs is black-box in style, if any of these (especially output statements) are amongst the 'hidden code', they risk being overlooked. In the specific case of transitions having multiple triggers, it was shown in Section 4.2.1 that our method could be extended. But, in general, unless other measures can be devised to make hidden code more visible, our strategy cannot be applied fully to ROOM. Overall, our approach is still beneficial, because it makes a great deal of coverage available quickly and easily. One must simply recognise the limiting factors.

5.7 A testing framework

Although not a critical part of our strategy, it is useful to consider a testing framework in which our strategy might be employed. By 'testing framework' is simply meant

a description of the overall environment in which testing is conducted, including process or cycle, a tool, system interfaces, iterative steps, etc.

Step 1. A software designer prepares the desired software using ROOM and a toolset, such as ObjecTime, that implements the ROOM methodology.

Step 2. A software tester applies the first part of our technique (basic counting), available as an integrated menu item on the toolbar of an existing toolset or as a completely separate standalone tool, in an automated fashion to count the number of test cases required to achieve transition coverage. *Our metric is produced.* This metric is indicative of how many test cases (what level of testing effort) would be required to achieve the indicated coverage. This is a lower bound.

Step 3. The tester must decide if this appears to be a suitable number of test cases.

a. If the decision is to use fewer test cases than recommended, then it must be acknowledged that there will be untested portions and that this must be consciously accepted as "business risk" by management personnel. To reduce the number of test cases, a 'cut-off' depth can be applied; if one has already been applied, then it can be made more drastic. Go back to Step 2.

b. Otherwise, if the metric produced is an acceptable number of test cases, we proceed to Step 4.

Step 4. The second part of our technique (path selection) will generate a skeleton suite of tests showing what needs to be covered. This is done in "semi-automated" fashion; that is to say, the test case designer will be prompted for inputs. In this manner, the designer/user provides guidance to the tool with regard to the feasibility of paths or the relative costs of different paths.

Optional *Step 5*. As an alternative to Step 2, or as a fifth step afterwards, a more advanced version of the tool could be invoked to handle structural details that are not in the visible portion of the structure. As an example, it could check all branches within transitions. (Go back to Step 3; else proceed to Step 6.)

Step 6. The test cases thus far derived are now executed. To verify that the coverage is as claimed, probes can easily be inserted into the executable ROOM model using appropriate windows in the ObjecTime toolset. If for some reason, the coverage is not what was expected, we return to Step 4 to modify the test suite. If the functionality is not as expected, then we return to Step 1 to modify the software.

5.8 Application and implementation

Appendix 6 provides C code that implements some of the ideas being discussed here. As there are numerous variants on the HFSM concept, it is necessary to state the subset of the HFSM universe that is being treated in Appendix 6.

5.8.1 Subset implemented

For the purposes of Appendix 6, the following constraints apply:

- an HFSM is a set of one or more graphs and the parent-offspring relationships that connect the graphs. Graphs must be connected. (Note: if there is no hierarchy, then it is really only a simple FSM; the program in Appendix 6 will still find the minimum number of paths)
- each graph consists of nodes and edges (representing states connected by transitions)
- each edge has a minimum capacity denoted by an integer greater than or equal to one (1)
- each edge is directed
- each graph has a source node and a sink node
- program execution is assumed to start at the source node upon entry to that graph
- there is only one "root" graph at the "top"

- except for the top-level graph, each graph must have exactly one parent node (thereby creating a partial ordering amongst graphs)
- any node can be expanded into a separate "offspring" graph (which may contain further nodes for expansion); thus, any node can have zero or more offspring nodes collected into a single offsping graph
- each graph must have at least two nodes, because a graph that has only one node is really only the same as its parent graph
- there is only one thread of execution; there is no concurrent activity
- the graph cannot contain any "self-loops," transitions that loop back to the node that it started from
- the graph cannot contain any cycles; that is, for every node, there is no path that will bring lead back to this node
- details of each transition (such as input alphabets, outputs, guards, probability of execution, etc.) need not be described, since we are dealing with only the structure of the transitions and not the content

Coding conventions. For actual use of the program in Appendix 6, some coding conventions apply for the input data.

Connectivity. The program uses an $n \times n$ matrix to represent a graph of n-nodes. It is a variant of the simple *adjacency matrix*, in that the matrix entry represents the minimum desired flow rather than just 0/1 indicating "connected"/ "not connected".

matrix $(x, y) = k$, where k is the minimum desired flow from node x to node y,
$= 0$, otherwise. (That is, there is no transition between x and y.)

Offspring. Whether or not node x has any offspring is indicated by a vector ("array")

offspring $(x) = 1$, if node x can be expanded into a graph;
$= 0$, otherwise. (That is, node x is a "leaf node".)

Name. For reference in printouts, for ease of composing input datasets, and for reference during program debugging sessions, each graph has a name (40 characters, no spaces)

Input file format. The input file is an ASCII text file with the following information:

COMMENT (free text, any printable character, maximum 100 characters)

NAME OF GRAPH 1 (free text, maximum 40 characters, no spaces)
NUMBER OF NODES IN GRAPH 1 (integer, maximum 10)
MATRIX FOR GRAPH 1 (all integers)
OFFSPRING ARRAY FOR GRAPH 1 (each array element is either 1 or 0)

NAME OF GRAPH 2
NUMBER OF NODES IN GRAPH 2
MATRIX FOR GRAPH 2
OFFSPRING ARRAY FOR GRAPH 2

:

: *etc.*

:

NAME OF GRAPH *n*
NUMBER OF NODES IN GRAPH *n*
MATRIX FOR GRAPH *n*
OFFSPRING ARRAY FOR GRAPH *n*

Fig. 5-6. *Format of the input data file for the program in Appendix 6.*

Example input:

```
This graph has 7 main nodes
ORION
7
0 1 0 0 1 0 0
0 0 1 0 0 0 0
0 0 0 1 0 1 0
0 0 0 0 1 0 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 0 0 0 0 0 0
1 0 0 0 1 0 1
BETELGEUSE
2
0 1
0 0
0 0
ALNITAK
3
0 1 1
0 0 1
```

```
0 0 0
0 0 0
RIGEL
2
0 3
0 0
0 0
```

Fig. 5-7. *Sample input data file for the program in Appendix 6.*

Appendix 6 gives some actual print runs with both inputs and outputs.

5.8.2 Application and tests of implementation.

The implemented subset can be made to accommodate more kinds of HFSMs by applying some of the following interpretations:

- Multiple transitions. This term refers to the case where '*m*' number of transitions all start at the same node *x* and all terminate at the same node *y*. (For example, the refer to the graph in Fig. 5-3.) In these situations, we can let *matrix (x, y) = m* in the input file. This will force the algorithm to assign at least *m* units of flow between *x* and *y*.
- Transitions that stop at the boundary can be treated as though they enter the first state
- Models that comprise more than one "actor" (models that have concurrent components) can be analysed by assessing each component individually, though the concurrent aspect will be lost
- Cycles. Some kinds of cycles can be accommodated by manually "breaking" them first for the input file

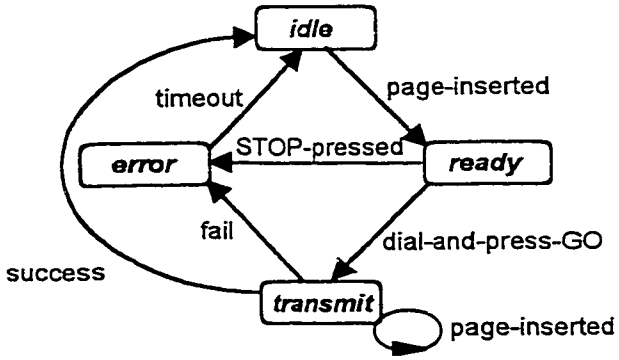


Fig. 5-8 (a). *FSM containing cycle and self-loop.*
This FSM represents the transmit portion of a simple fax machine. [After Probert 94]

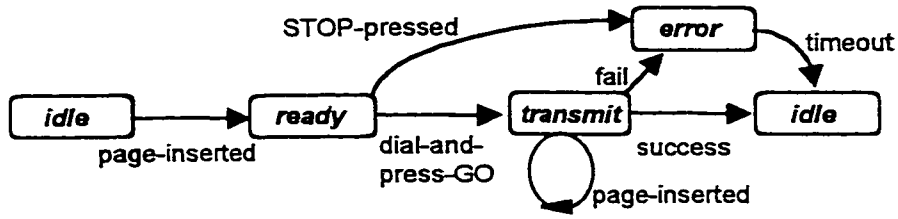


Fig. 5-8 (b). *The cycle is broken by "splitting" one of the nodes ("idle")*

Section 3.5 discussed strategies for breaking up graphs containing strongly connected subgraphs. This is an important consideration, given that many real-time systems do not have a fixed start and stop – most operate continually, but they typically have certain states that are "less active" than others, in which they spend a lot of time, and to which they return after being active. Examples: for a car, we might use the "gear in Neutral" (or "Park"); for an elevator, we might use "stopped at ground floor"; for a telephone, we might use "On hook and bell not ringing".

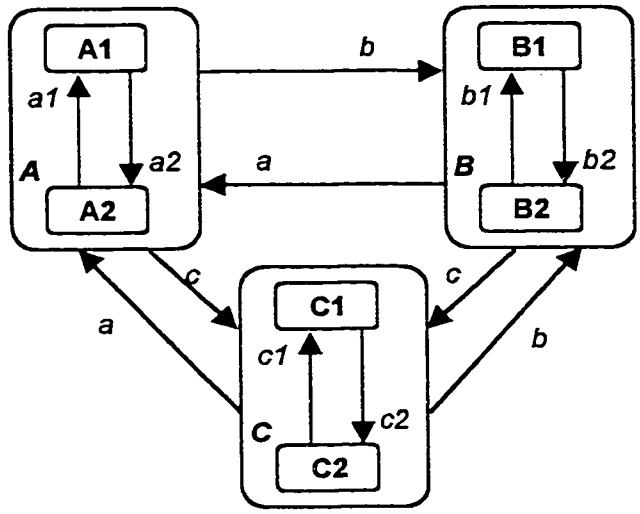


Fig. 5-9.. *Graphs with cycles still have an intuitive "maxcut" or "width." Figure from [Selic++ 94]*

In the example of Fig. 5-9, we may wish to use the designated "initial state" as the one to split. Fig. 5-10 shows the cycle in one of the inner states being opened up.

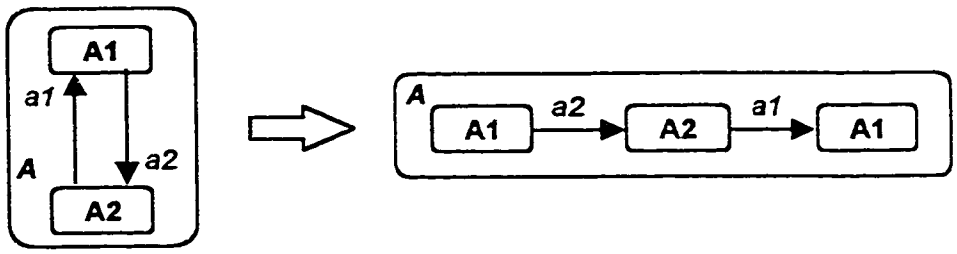


Fig. 5-10. *A cycle can be "opened up" at one of the nodes in the cycle by splitting it to create a source and a sink.*

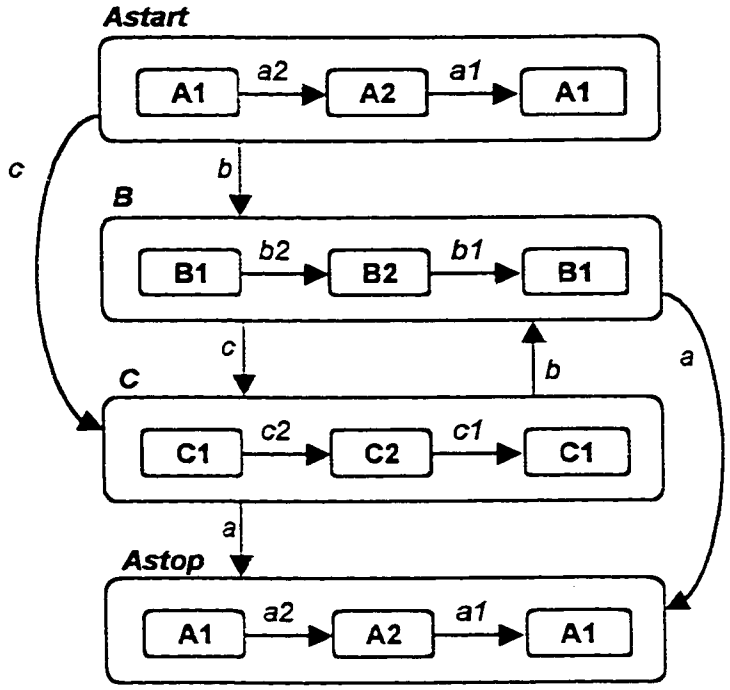


Fig. 5-11. *With each of the smaller cycles split, the large cycle can also be split. (Compare with Fig. 5-9)*

5.8.3 Subset not implemented.

Graphs with cycles still have an intuitive "maxcut" or number of "paths" necessary to cover. Although Fig. 5-11 still contains a cycle in the centre (between nodes B and C), it is not difficult to see that all states and transitions could be covered with two paths:

Path 1:

Astart	A1 – A2 – A1	transition *b* (in the middle)
B	B1 – B2 – B1	transition *c*
C	C1 – C2 – C1	transition *a*
Astop		

and Path 2:

Astart	A1 – A2 – A1	transition *c* (along left-hand outside)
C	C1 – C2 – C1	transition *b* (up the middle right)
B	B1 – B2 – B1	transition *a* (down the right-hand outside)
Astop		

The program in Appendix 6 is not clever enough to handle this aspect of cycles, but this strategy is part of what is proposed.

Self-loops are also part of our strategy. It is clear that the existence of a self-loop does not generally increase the number of paths required for coverage. To use our computer program, one would simply not enter the self-loops into the adjacency/flow matrix; that is, *matrix (j, j)* = 0, for all *j*. However, if there were a large minimum coverage requirement for a particular self-loop (for example, *matrix* (3,3) = 140), then we would want to take this into account (which our program does not do). In Section 6.2, under "Huang's Theorem," we discuss why it may be beneficial to execute all loops at least twice. This is just a matter of setting the appropriate *matrix (k, k)* value to 2 or greater, either manually or in automated fashion.

Our strategy includes graphs that contain strongly connected subgraphs, but, as indicated above, this feature is not implemented.

5.9 Best use

The strategy that is proposed in this thesis addresses the hierarchical and FSM aspects of ROOMcharts for testing; it is not intended to particularly address real-time issues or

object-oriented issues. Furthermore, it does not replace other test case generation strategies. It is designed to be complementary to other software testing techniques.

It should be emphasized that the testing contemplated here is at a very basic level: the design level. The disadvantage of this is that many of the details, such as the actual code in the transitions, are not directly accessible for testing; we can assert only that the transition will indeed be executed. On the other hand, the advantage of testing early at the design level is that any flaws revealed should result in the developer fixing the design, not just a portion of the code.

The strategy of counting and selection is best combined with other well-known strategies for deriving test cases. In a conservative approach, the results of counting and selection should be seen as a minimum. Additional test cases should be generated by other means. For example, as a result of a risk analysis, we may wish to add test cases for certain high-risk scenarios. In a more aggressive and time-saving approach, we may wish to ignore test cases that exercise low-risk or previously tested portions of the design, dropping below the minimum levels if necessary. An example of the results from counting and selection is given in Appendix 5.

In HFSMs, as with normal FSMs, the phases of program execution often correspond well with the modelled states. This suggests that a phase-directed strategy [Boyce+ 96] may also be appropriate, particularly when drawn from a suitably high level of ROOMchart. Test cases drawn from specific scenarios will also prove useful.

Most FSM-based formal test methods focus their efforts on identifying states within a black-box implementation, which is not the actual problem at hand. And, where they are applicable, they tend to result in fairly lengthy test sequences (even for small FSMs) which would be costly to implement. By comparison, the relative simplicity of our graph-traversal based strategy makes it inexpensive to implement. In this light, it is suggested that, despite its limitations, the benefits would outweigh the cost, and would provide us with a cost-

effective means of obtaining a base set of test cases – a set that could be later augmented according to scenario type (for example, exceptional cases) or risk. Finally, the metric provided by the path counting can also be used in some sort of company quality improvement programme and could easily form part of any collection of project cost estimation tools.

6

Extensions and Future Work

In the preceding chapter, a number of limitations were described on the use of the strategy for counting and selection. This chapter considers how to extend the basic strategy to address some of the limitations and be generally more effective. In addition to these shorter-term ideas, there are longer-term plans that need to be considered in the development of any overall testing framework for commercial modelling languages such as ROOM. A brief discussion is given of some of these issues.

6.1 White-box (code-based) testing extensions

6.1.1 Code contained in transitions and states

Because the suggested strategy addresses only the overall design of the modelled system, it will not examine any of the structure within the manually coded portions of the program. Such code is "hidden" in the ROOMchart depiction. Nevertheless, it is reasonable to extend the idea of path counting and selection to accommodate the structure within that code contained in transitions and states.

If the contained code is fairly linear (that is, if most of the structure of the program is in the visible portion of the ROOMchart) then the path counts will not increase significantly. There would be *no* increase if all contained code were "straight-line." If however the contained code has many branching or looping constructs, then separate paths will have to be found. [Beizer 90] and [Beizer 95] discuss different ways of handling or counting paths in loops. (And, naturally, these ways are also applicable to the visible paths as well.)

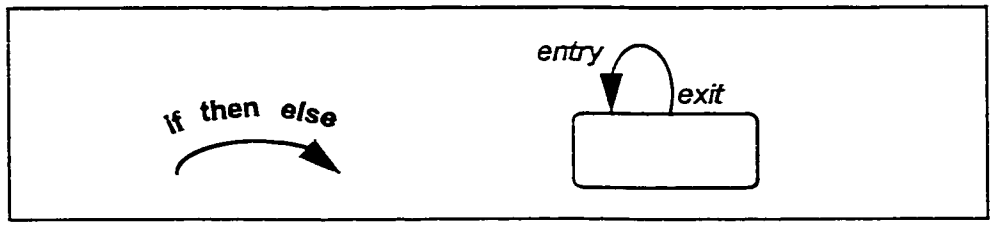


Fig. 6-1. *(a) Code contained in a transition. (b) Exit and Entry code incurred.*

Simple example. Case: we are given a transition that has more than one way of going through it. Suppose that there are exactly two ways; that is, it takes two paths to cover all statements in the transition code. Then, we can mark this transition as being "2 wide", and we know that we will have to devise two test cases to cover this transition. ■

Another example. Case: there exists code within a state. There are potentially three places for code within a state: upon entry, upon exit, and along the transitions belonging to any substates contained. The first two are not shown in a ROOMchart, and our methods in Chapters 3 and 4 do not allow for the entry code or exit code to be more than "1 wide."

One way to make such code visible is to model transitions explicitly to capture any complexity in entry code and exit code. Additional states may also have to be created to collect such transitions. This would give us a small FSM in front of every entry to the original state and at the end of every exit, if more than one. From here, we could then apply previous methods.

A simpler way of dealing with code within a state is to increase the required path count for (or 'width' of) this entire state, commensurate with the number of different ways of entering or exiting. For example, if there were 3 ways of executing the state-entry code, then the path count for the state would simply be increased by 2. The disadvantage of this simpler method is that greater human assistance is required in generating the actual paths in order to ensure that the same path is not being taken each of the 3 times. ■

Just as we forced ourselves to go through certain states a minimum number of times, we must do the same for certain transitions. And, we must force ourselves to traverse a state a sufficient number of times to cover all manners of entry, all manners of exit, and all ways of going through. As an example, suppose for a given state, that there are 3 different ways of executing the entry code, 5 different ways of traversing the middle, and 6 different ways of executing the exit code. Then, this state must be traversed at least $\max\{3, 5, 6\} = 6$ times.

In summary, these "portions" of the HFSM (the transitions, the state entrances, the state exits, and the states themselves) may all be tagged with a minimum traversal requirement or explicitly re-modelled so that they become amenable to automated methods.

6.1.2 Data-flow methods

Under the general category of strategies for testing software modelled as hierarchical FSMs, it is certainly reasonable to consider data-flow methods. Introduced in Section 2.3.5 and discussed with extended FSMs, such methods are applicable to all software in general, provided that a flowgraph can be generated, and are therefore applicable to HFSMs. Although separate data-flowgraphs can be generated, the methods are normally applied to ordinary control flowgraphs which are then annotated with the appropriate data information. HFSMs are therefore very amenable to such methods. However applicable they may be, data-flow methods are not the subject of this thesis, but because data-flow anomalies are partially detectable when writing regular (path) expressions, we discuss these in the next section.

6.2 Paths and path expressions

The analysis of paths in a flowgraph can be considered 'black-box' if we are looking at the functional transaction flow (regardless of how the program code implements this) and can be considered 'white-box' if the techniques are applied directly to the code. In any case, such methods are highly applicable to HFSMs and ROOMcharts because the paths are so visible and related to both functionality and structure.

We have seen that a path can be described listing its constituent transitions in the order in which they occur. In the flowgraphs in this section, we concentrate on the order of the links rather than the nodes. We see in Fig. 6-2 that there are four paths through the main node.

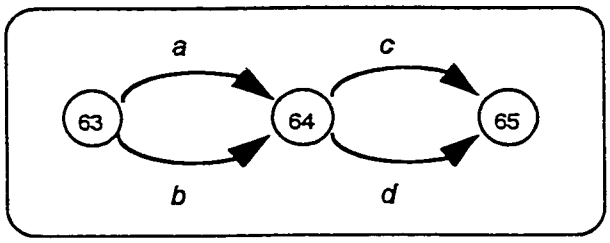


Fig. 6-2. *Describing paths using a path expression.*

To traverse this node, we have the choice of taking *a* or *b*, and then *c* or *d*. Parentheses can be used to group these choices. A *path expression* can then be formed by taking "addition" as meaning "or" and "multiplication" as meaning "concatenation" or "and." Therefore, a path expression for this graph is

$$(a + b)\,(c + d).$$

Using conventional arithmetic, this can be expanded to

$$ac + ad + bc + bd$$

which then confirms that there are indeed four different paths through this node, one for each arithmetic 'term.'

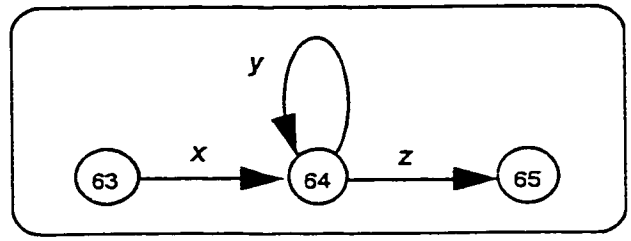


Fig. 6-3. *Graph with a loop.*

The path expression for the graph in Fig. 6-3 is *xy*z*, where an asterisk (*) beside an element indicates "zero or more" occurrences of that element. For the graph in Fig. 6-4, the path expression is *t1.t2.(t3.t4.t5.t2)*.t6.* Thus, any path with a loop in it will contain an asterisk in its path expression.

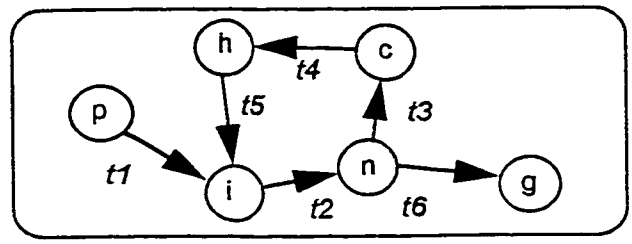


Fig. 6-4. *Graph containing a cycle.*

General procedures are given in [Linz 90] for creating and simplifying the long expressions that would be obtained from much larger graphs.

It was shown in Section 5.5.2 that path expressions can have some very practical applications. First of all, a weight or cost can be assigned to each transition. This "cost" may represent an actual cost of testing, the length of time needed, personnel cost, amount of computer resources required, or other statistic. Thus, the path expression may be able to provide the test manager with useful information.

Probabilities may also be assigned to the different branches at each branch point. These assignments of weight will help produce a more likely or realistic total. If the consequence of error is placed as a value on each branch, then the path expression provides an estimate of the risk associated with this path.

Regular expressions

Path expressions can be subjected to certain kinds of analyses reserved for so-called regular expressions to determine whether certain properties hold. As **definition**, we first take the empty set, Ø, the null string, λ, and any member of our input alphabet, say, *a*, to be *primitive regular expressions.* Then we take the rule that if *r1* and *r2* are regular expressions, then so are *r1 + r2, r1.r2*, and *r1**. Finally, a string is a *regular expression* if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rule above. [Linz 90].

Using the transition names (rather than node names) to describe paths emphasizes the order in which actions are being done (since actions generally occur on the transitions). Therefore, we can also detect invalid sequences of events, such as a variable being used before it has been defined. A theorem, due to Huang, describes the relationship between regular expressions and whether a particular sequence can ever occur. [Huang 79].

Huang's Theorem. Let *A*, *B*, and *C* be non-empty sets of character sequences whose smallest string is at least one character long. Let *T* be a two-character string of characters. Now, if *T* is a substring of AB^nC then it is also a substring of AB^2C ■.
[Huang 79], [Beizer 90].

The result of this theorem is that a lot of information (or assurance) can be gained simply by executing a loop twice – but it must be executed at least twice. Thus, there is potential in examining path expressions to help generate test cases for software modelled as HFSMs or modelled using ROOMcharts.

Our proposed strategy, as currently described, assures us of coverage of all transitions, and therefore of all loops, at least once. But, based on Huang's result above, we can simply change our strategy to cover all loops at least twice. Since some portions of the HFSM have to be covered more than once anyway, the additional cost is really only at those loops that would have been covered only once. Making this arbitrary extension to our strategy is obviously easier than deriving path expressions and may return significant gains. (Also see Section 5.8.3 regarding how this would be applied.)

6.3 Multiple points of exit

In Section 5.6.2, it was suggested that a simple way of dealing with multiple points of exit would be to create a large 'back-end' node that could collect the multiple exits. However, an additional variable or switch would then be required to keep track of what the next state should be. The following is another way of handling multiple points of exit. This method consists of splitting the original node according to the respective exits, so that each of the newly split nodes has a single clean exit. Fig. 6-5 shows an example in which the original node has two exits.

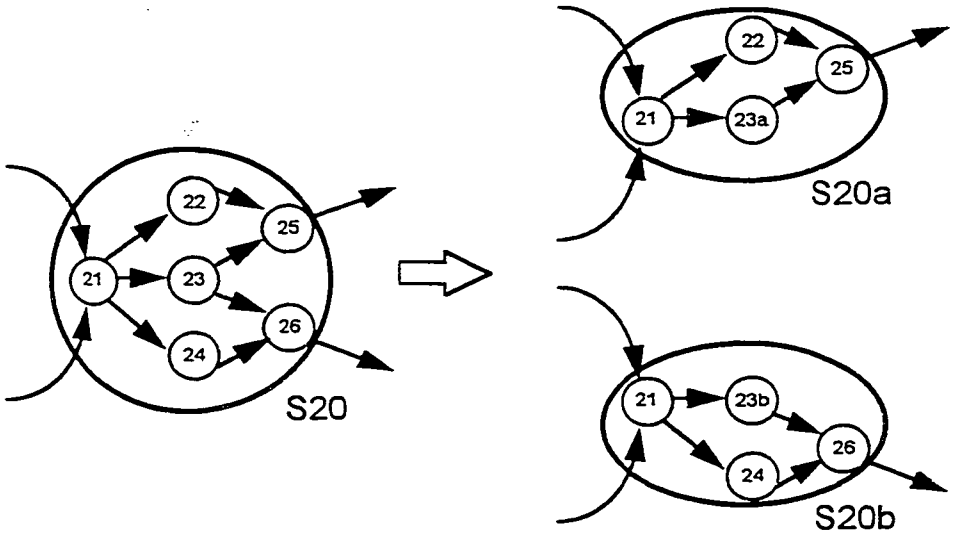


Fig. 6-5. *Subnode S23 becomes S23a and S23b.*

Such a remodelling entails more work on behalf of the designer, but improves the testability of the overall model. The implication here is that the program or HFSM should have originally been designed so as not to have multiple exits. [Beizer 90]

6.4 Optimization

In our discussions thus far regarding the selection of path routes (the generation of test cases or test sequences), no mention has been made as to the optimality of the solution. In general, we have been satisfied to have useful and practical heuristics; however, it is obvious that there is room for optimization. Further study could be undertaken in the area of minimizing path lengths or total path lengths. If weights are added, we could try to minimize the total cost, rather than the total length. In fact, one could attempt to minimize any of the measures.

In the case of "don't care" routing selections, such as in the main example of Appendix 5, it has been suggested that the minimum cost route could be selected wherever there is a choice. However, it may be more effective to select varying *combinations* of path segments (regardless of cost), thereby covering more of the testing space. If it is risk that we wish to minimize, there may be no simple way (for example, by giving a 'risk weighting' to each link) to do so in an automated fashion; this may require a more complex overall strategy, combining numerous methods.

6.5 Implementation issues

Future work should also consider how to extend and improve our prototype implementation. Appendix 3 uses pseudo-code to suggest how some of the algorithms might be realised, and Appendix 5, the results displayed. Appendix 6 gives C-program source code for traversing an HFSM.

The most logical place from which to consider implementation issues is the ObjecTime toolset, which currently does implement the ROOM modelling methodology. For example, we may wish to consider how best to integrate our counting and selection activities with the animated on-screen presentation of ObjecTime (or any other toolset). This on-screen interaction would be most important in deciding upon which paths to choose in order to avoid infeasible paths or in order to reduce cost or risk. The on-screen displays can be used to monitor probes, verify the current state, see which lower-level paths were taken, (potentially the number of times that each was taken), and determine generally which parts of the ROOMchart were covered and which were not.

The tables in Appendix 5 show a tally of counts for the minimum coverage and the implicitly "induced" coverage as well as lists for the suggested paths. The actual production and display of such lists may be a bit cumbersome for large HFSMs. Again, a graphical display that integrates with a toolset would be more desirable.

7

Concluding Remarks

In the opening chapter, two requirements were given for any approach to be useful in a commercial application of any significant size. They focussed on its ability to scale up and ability to be automated. In Chapter 5, we saw to what degree our proposed approach satisfied these requirements and how it related to standard software testing techniques. In this chapter, we summarize the results from the other chapters and give some concluding remarks.

7.1 Conclusions

Objective

Our objective was to derive a metric that could be used to direct testing of a software modelling system that used, in part, hierarchical finite-state machines to model behaviour. General software testing methods were briefly reviewed as well as certain formal test generation methods that are specific to finite-state machines. The former, being as general as they are, are always applicable in some manner; the latter were found to address a slightly different situation (in that the usual problem of state identification did not arise) and were deemed

more appropriate to later stages of software development. A review of the literature revealed that there were no particular testing methods found for HFSMs comparable to the abundant and formal methods available for testing traditional FSMs (for example, machine identification, checking sequences, etc.).

The underlying graph structure of the HFSM was found to be amenable to solution by graph-oriented methods. Unfortunately, there remained the problem of combinatorial growth in numbers of paths and potentially in the global state space as well.

Results

This thesis proposes a simple but effective method based on the traversal of the underlying graph for counting the number of test paths required to cover all transitions in an HFSM. This also results in the coverage of all states. It is shown in Section 4.3 that when applied appropriately, the method can cover all triggers (inputs influencing execution flow), trigger conditions, signals (message types), ports, and some of the outputs (indications) depending on whether there is any branching in the embedded code of ROOMcharts.

The path count is obtained by applying the max-cut min-flow theorem to the directed graph underlying the HFSM, with lower bounds of one on the transitions. The resultant output is a lower bound on the number of paths (test cases) needed to provide the above-mentioned coverage, as shown in Chapter 5. Subtotals are also available for each level of node. The selection of possible paths is obtained by finding a set of actual sequences of nodes and transitions which, if feasible, will achieve complete coverage of the transitions and therefore all the coverage mentioned above.

Advantages and Limitations of Current Approach

The main advantage of our approach is that it is scaleable. The computation for large models is only linearly greater than that for smaller ones. Also, the strategy is directly applic-

able to "industrial-strength" CASE design tools such as ObjecTime, which was considered in this thesis. Our approach also lends itself to automation or, at least, to partial automation functioning with user input or guidance. It is able to provide some measure of its results, and can list what it has covered and how many times.

A cut-off depth, as described in Section 5.5.1, is potentially available, should the suggested minimum already be too much to test. As part of future enhancement, measures could be assigned weights to reflect cost, risk, or whatever we wish to measure.

Our approach is very practical approach for measuring the lower bound on the testing required for programs modelled in ROOM/ObjecTime, or any other modelling method based on HFSMs. A test designer who produces a test suite comprising fewer than the suggested minimum number of test cases will be well aware that many portions of the model will go untested. The counting method actually identifies the number of probes that would have to be inserted if the system were to be monitored.

The counting portion alone can provide a test manager with an indication of the complexity of the software design and therefore potentially on longer-term issues such as life-cycle maintenance. The minimum path count for such a control-oriented design is, in itself, a metric that is very similar to but more exact as a lower bound than McCabe's cyclomatic complexity (Chapter 5 refers). It provides a software designer or modeller with an (inverse) indication of the predicted reliability or quality of the design. Appendix 5 shows how it operates for a simple HFSM model.

Unfortunately, our strategy does not provide coverage of *combinations* of paths, nor can it identify infeasibility of paths without human assistance. However, this is a drawback that is common to all methods based on static analysis. Techniques such as symbolic execution [Beizer 90], for example, would help identify unachievable paths. One approach is to enumerate impossible predicate pairs, that is, predicates *pi* and *pj*, such that $pi \wedge pj$ is not satisfiable. However, it has been noted that building a path that satisfies restrictions on

impossible predicate pairs is an NP-complete problem [Gabow++ 76]. One inexpensive workaround is to detect the infeasible pairs *after* having selected paths, rather than try to avoid infeasible pairs in advance.

Ultimately, it is suggested that the relative simplicity of our approach outweighs its limitations. The impact of any method is further enhanced with tool support. This kind of support is possible because the method lends itself to at least partial automation. Any implementation of our approach should have much appeal to industrial and commercial concerns which use FSM-based design models and tools.

7.2 Recommendations for Future R&D

In the short term, additional tool features for research and development (R&D) could include some ability to influence the generation of test cases, for example, by minimizing a cost measure or some combination of measures. Or, the tool might even solicit information from the user. A test designer might appreciate being prompted for guidance, for example around infeasible paths, or being allowed to override suggestions for testing, perhaps in low-risk areas. If weights or measures can be input, then it should be possible to generate statistics other than cost, depending on the ingenuity of the user. Extension to white-box testing methods can include extensive examination of embedded code and application of data-flow methods. And, as always, a good user interface is important to any tool.

In the mid-term, it is envisioned that our approach should become part of an overall testing framework in ROOM or other HFSM-based commercial toolset. This grey-box or design-structure approach is complementary to scenario-based or customer-directed approaches (for example, comparison of message sequence charts or pure functional testing). The notation used for the specification of test cases can be standardized, based on the internationally recognised tree and tabular combined notation (TTCN).

- o -

References

[Aho++ 88] AHO, Alfred V.., DAHBURA, Anton T., LEE, David., and UMIT UYAR, M. "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese Postman tours," *Proc. Symp. Protocol Specification, Verification, and Testing,* North-Holland, 1988

[Beizer 90] BEIZER, Boris. *Software Testing Techniques, 2nd Edition*, International Thompson Computer Press, Boston, 1990

[Beizer 95] BEIZER, Boris. *Black Box Testing,* John Wiley and Sons, Inc., New York, 1995

[Boni Bangari 97] BONI BANGARI, Anandra. "A Use-Case Based Validation Framework," Master's Thesis, University of Ottawa, Ottawa, 1997

[Boyce+ 96] BOYCE, T.T, and PROBERT, Robert L. "Protocol decomposition as an aid in generating meaningful test cases," 1996

[Budgen 95] BUDGEN, David. *Software Design,* Addison-Wesley, Wokingham, 1995

[Burer+ 98] BURER, Samuel and MONTEIRO, Renato D.C. "An Efficient Algorithm for Solving the MAXCUT SDP Relaxation," Georgia Institute of Technology, Atlanta, Dec 1998 [in publication]

[Chan++ 89] CHAN, Wendy Y.L., VUONG, Son T., and ITO, M. Robert. "An Improved Protocol Test Generation Procedure Based on UIOs," *Proceedings of ACM SIGCOMM '89,* 1989, pp. 283-294

[Chen++ 90] CHEN, Mon-Song, CHOI, Yanghee, and KERSHENBAUM, Aaron. "Approaches Utilizing Segment Overlap to Minimize Test Sequences," *Proc. Protocol Specification, Testing, and Verification,* Vol. X, North-Holland, 1990, pp. 85-98

[Chen+ 95] CHEN, Wen-Huei, and URAL, Hasan. "Synchronizable Test Sequences Based on Multiple UIO Sequences," *IEEE/ACM Transactions on Networking*, Vol. 3, No. 2, April 1995, pp. 152-157

[Cheston++ 79] CHESTON, G.A., PROBERT, R.L., and SAXTON, L.V. "Graph grammars and constrained path covers," *Proc. 1979 Conference on Information Sciences and Systems*, John Hopkins University, Baltimore, 1979

[Chow 78] CHOW, T. "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, 1978, pp. 178-187

[Dahbura++ 90] DAHBURA, A.T., SABNANI, K.K., and UYAR, M.U., "An optimization technique for protocol conformance testing using multiple UIO sequences," *Proc. IEEE*, Vol. 78, 1990, pp. 1317-1325

[Day 93] DAY, Nancy. "A Model Checker for Statecharts (Linking CASE tools with Formal Methods)," Technical Report 93-25, University of British Columbia, Vancouver, 1993

[Fenton 94] FENTON, Norman. "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering*, Vol. 20, No. 3, March 1994, pp. 199-206

[Ford+ 64] FORD, L, and FULKERSON, D. *Flows in Networks*, Princeton University Press, Princeton, 1964

[Fujiwara++ 91] FUJIWARA, S., v.BOCHMANN, G., KHENDECK, F, AMALOU, M., and GHEDAMSI, A. "Test selection based on finite-state models," *IEEE Transactions on Software Engineering*, Vol. 17, No. 6, 1991, pp. 591-603

[Gabow++ 76] GABOW, Harold N., MAHESHWARI, Shachindra N., and OSTERWEIL, Leon J. "On two problems in the generation of program test paths," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3 (Sept. 1976), pp. 227-231

[Garey+ 79] GAREY, M.R. and JOHNSON, D.S. *Computers and Intractability — A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., 1979

[Gonenc 70] GONENC, G. "A model for the design of fault detection experiments," *IEEE Transactions on Computers*, Vol. C-19, No. 6, June 1970, pp. 551-558

[Grady 92] GRADY, Robert B. *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, 1992

[Hadlock 70] HADLOCK, F. "Finding a maximum cut of a planar graph in polynomial time," *SIAM Journal on Computing*, Vol. 4, No. 3, September 1975, pp. 221-225

[Halstead 77] HALSTEAD, Maurice H. *Elements of Software Science*, Elsevier North-Holland, New York, 1977

[Harel 87] HAREL, David. "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, Vol. 8 (July 1987), pp. 231-274

[Harel 92] HAREL, David. *Algorithmics: The Spirit of Computing*, 2/ed., Addison-Wesley, Reading, 1992

[Harel+ 96] HAREL, David, and NAAMAD, Amnon. "The STATEMATE Semantics of Statecharts," *ACM Trans. Soft. Eng. Meth.*, Vol. 5, No. 4 (Oct. 1996)

[Harel+ 97] HAREL, David, and GERY, Eran. "Executable Object Modeling with Statecharts," *IEEE Computer*, Vol. 30, No. 7 (July 1997), pp. 31-42

[Heimdahl+ 96] HEIMDAHL, Mats P.E., and LEVESON, Nancy G. "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Transactions on Software Engineering*, Vol. 22, No. 6 (June 1996), pp. 363-377

[Holzmann 91] HOLZMANN, Gerard J. *Design and Validation of Computer Protocols*, Prentice-Hall Inc., Englewood Cliffs, 1991

[Hopcroft+ 79] HOPCROFT, J. and ULLMAN, J. *Introduction to Automata Theory*, Addison-Wesley, Reading, 1979

[Horrocks 99] HORROCKS, Ian. *Constructing the User Interface with Statecharts*, Addison-Wesley, Harlow, 1999

[Huang 79] HUANG, J.C. "Detection of data flow anomaly through program instrumentation," *IEEE Transactions on Software Engineering*, Vol. 5 (1979), pp. 226-236

[Kan 95] KAN, Stephen H. *Metrics and Models in Software Quality Engineering*, Addison-Wesley, Reading, 1995

[Karolak 96] KAROLAK, N. *Software Engineering Risk Management*, IEEE Press, Piscataway (USA), 1996

[Kohavi 78] KOHAVI, Zvi. *Switching and Finite Automata* Theory, McGraw-Hill, 1978

[Kuan 62] KUAN, Mei-ko. "Graphic Programming Using Odd and Even Points," *Chinese Math*, Vol. 1 (1962), pp. 273-277

[Lai+ 95] LAI, Richard, and LEUNG, Wilfred. "Industrial and academic protocol testing: the gap and the means of convergence," *Computer Networks and ISDN Systems*, Vol. 27, 1995, pp. 537-547

[Lawler 76] LAWLER, Eugene. *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, Dallas, 1976

[Levi 97] LEVI, Francesca. "Compositional Verification of Timed Statecharts," *Proceedings of ICTL'97*, 1997

[Linz 90] LINZ, Peter. *An Introduction to Formal Languages and Automata*, D.C. Heath and Company, Lexington (USA), 1990

[Lu 96] LU, Hsueh-I. "Efficient Approximation Algorithms for Some Semidefinite Programs," Technical Report CS-96-33, Brown University, Providence, 1996

[Maggiolo-Schettini++ 96] MAGGIOLO-SCHETTINI, Andrea, PERON, Adriano, and TINI, Simone. "Equivalences of Statecharts," *Springer Lecture Notes in Computer Science*, Vol. 1119, 1996, pp. 687-702

[Maggiolo-Schettini+ 97] MAGGIOLO-SCHETTINI, Andrea, and MERRO, Massimo. "Priorities in Statecharts," *Springer Lecture Notes in Computer Science*, Vol. 1192, 1997, pp. 388-403

[McCabe 76] McCABE, Thomas J. "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, Dec 1976, pp. 308-320

[McCabe 82] McCABE, Thomas J. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, National Bureau of Standards Special Publication 500-99, Dec 1982

[McCabe+ 94] McCABE, Thomas J. and WATSON, Arthur H. "Software Complexity," *CrossTalk*, Dec 1994, Software Technology Support Center, Hill Air Force Base, Utah

[MillerR+ 93] MILLER, Raymond E and PAUL, Sanjoy. "On the generation of minimal-length conformance tests for communication protocols," *IEEE/ACM Transactions on Networking*, Vol. 1, No. 1, Feb 1993

[MillerS+ 95] MILLER, Steven P. and SRIVAS, Mandayam. "Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods," *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*

[Myers 79] MYERS, Glenford J. *The Art of Software Testing*, John Wiley & Sons, New York, 1979

[Naito+ 81] NAITO, S. and TSUNOYAMA, M. "Fault detection for sequential machines by transition-tours," *Proc. Fault Tolerant Computer Systems*, 1981, pp. 238-243

[Peron 95] PERON, Adriano. "Statecharts, Transition Structures and Transformations," *Springer Lecture Notes in Computer Science*, Vol. 915, 1995, pp. 454–468

[Perry 95] PERRY, William. *Effective Methods for Software Testing*, John Wiley & Sons, New York, 1995

[Pressman 92] PRESSMAN, Roger S. *Software Engineering: A Practitioner's Approach*, 3/ed., McGraw-Hill, New York, 1992

[Probert 95] PROBERT, Robert L. *Software Quality Engineering*, course notes, University of Ottawa, Ottawa, 1995

[Sabnani+ 92] SABNANI, K.K. and DAHBURA, A.T. "A Protocol Test Generation Procedure," *Computer Networks*, Vol. 15, No. 4, 1988, pp. 285-297

[Selic 92] SELIC, Bran. "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems," *Proceedings, Fifth International Workshop on Computer-Aided Software Engineering (CASE '92)*, July 1992

[Selic++ 94] SELIC, Bran, GULLEKSON, Garth, and WARD, Paul T. *Real-Time Object-Oriented Modeling*, John Wiley & Sons, Toronto, 1994

[Sidhu 90] SIDHU, Deepinder P. "Protocol Testing: The First Ten Years, The Next Ten Years," *Protocol Specification, Testing and Verification, X*, [ed.] L. Logrippo, R.L. Probert, H. Ural, Elsevier Science Publishers B.V., North-Holland, IFIP, 1990, pp. 47-68

[Srivas+ 95] SRIVAS, Mandayam K. and MILLER, Steven P. *Formal Verification of an Avionics Microprocessor*, Technical Report SRI-CSL-95-4, SRI International Computer Science Laboratory, Menlo Park (USA), 1995

[Tucker 80] TUCKER, Alan. *Applied Combinatorics*, John Wiley & Sons, New York, 1980

[Ural 94] URAL, Hasan. Course notes, University of Ottawa, Ottawa, 1994

[Ural++ 97] URAL, Hasan, WU, Xiaolin, and ZHANG, Fan. "On Minimizing the Lengths of Checking Sequences," *IEEE Transactions on Computers*, Vol. 46, No. 1, January 1997, pp. 93-99

[Vuong++ 94] VUONG, Son T., Loureiro, A.A.F., and CHANSON, Samuel T. "A framework for the design for testability of communication protocols," *Protocol Test Systems, VI*, [ed.] O. Rafiq, Elsevier Science Publishers B.V., North-Holland, 1994, pp. 89-108

[Ward 95] WARD, Paul T. "The ObjecTime Curriculum: New User Workshop: Basic Modeling," Training manual, Paul Ward Associates, New York, 1995

[Weyuker 88] WEYUKER, E.J. "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988, pp. 1357-1365

[Withrow 90] WITHROW, C. "Error Density and Size in Ada Software," *IEEE Software*, January 1990, pp. 26-30

[Zave 91] ZAVE, Pamela. "An Insider's Evaluation of PAISLey," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 212-225.

APPENDIX 1

Algorithm for Finding Maximum Cut

This algorithm is derived from a suggested exercise in [Tucker 80].

Given: a directed graph representing a network and labels on each edge showing the *lower bound* for flow on that edge. The source node is labelled *a*; the destination node is labelled *z*.

Find: the minimum total flows throughout the network that are required to satisfy the lower bounds specified on each edge of the network.

Step 1. Assign initial flows. By inspection, assign enough individual flows from the source to the destination until the totals on each edge satisfy all lower bounds. Unless the optimum flow assignment happens to have been chosen, this step will result in significant surpluses on many edges.

*Steps 2 - 6 constitute a **Decreasing Flow algorithm.***

The main idea of this algorithm is to start at the destination end of the network and label "successive" (preceding) nodes with the amount of flow that can be removed along the edge from the preceding node(s). In other words, we start "downstream" and work our way back "upstream."

Convention for node labels. A node label has the form

nodeName (prevNode, slackUpTillNow)

where *nodeName* is the name of the node (in this appendix, a letter of the alphabet),

prevNode is the previous node in the path, and

slackUpTillNow is the amount of flow available to be reduced along the entire path from *z*, the destination node, until the current node.

When the node labelling procedure reaches the source end of the network, it means that the amount of flow calculated can be safely removed along the entire path found. This amount is then removed end-to-end along that path. All labels are erased, and the procedure is repeated. During one of the repetitions (iterations), when the labelling procedure fails to reach the other end, then the nodes labelled up to this point constitute "half" the nodes (one side) in a maximal *cut* — the remaining nodes constitute the other side of the cut. The *capacity* of this cut is equal to the minimum amount of flow required to satisfy the lower bounds on all edges.

Step 2. Begin with node *z*, the destination. Label it *z*(-, ∞). The dash (-) shows that there are no further nodes downstream. The infinity symbol (∞) shows that initially there is no limit on how much surplus flow is available to be removed. Let *currentNode* denote the node currently being considered. Thus, initially

currentNode = *z*

slackUpTillNow = ∞

Step 3. (a) Check each *outgoing* edge from *currentNode*. If the flow is greater than zero and the downstream node is unlabelled, then label it as

downstreamNode (currentNode, minSlack).

where *minSlack* := min (*slackUpTillNow*, the actual flow on that edge)

slackUpTillNow := minimum amount of slack amongst all the edges that are along the path from *z* to the current node.

Note that in Step 3(a), "slack" is actually the amount of flow to be *added*.

(b) Check each *incoming* edge from *upstreamNode* to *currentNode*. Call this edge an upstream edge. Repeat the rest of this Step for each upstream edge.

Define the *slack* on this edge as the current flow minus the lower bound.

slack(upstreamEdge) := *flow(upstreamEdge) - lowerBound(upstreamEdge)*

If the slack is greater than zero and the *upstreamNode* is not yet labelled,

then label it as

***upstreamNode* (*currentNode, minSlack*),**

where *minSlack* := min (*slack, slackUpTillNow*)

If the slack is zero, then no labelling is performed.

Step 4. If node *a* has not yet been labelled,

choose another labelled node to be the *currentNode* and go to Step 3.

Else if node *a* has been labelled now,

then we have found an end-to-end path with some slack. Go to Step 5.

Else if there are no more labelled nodes to try, go to Step 6.

Step 5. Remove some surplus flow. Starting from *a*, follow the *prevNode* labels to back-track to node *z*, removing *slackUpTillNow* from each edge. Erase labels from all nodes. Go to Step 2.

Step 6. The max cut is found. Let P' be the set of nodes currently labelled. Then (P, P') is a *maximum cut.* The capacity of this cut is equal to the minimum amount of flow required to satisfy the lower bounds on all edges. ■

Example.

Given: the directed graph, Fig. A1-1, with labels on each edge showing the lower bound for flow on that edge. The source node is labelled *a*; the destination node is labelled *z*.

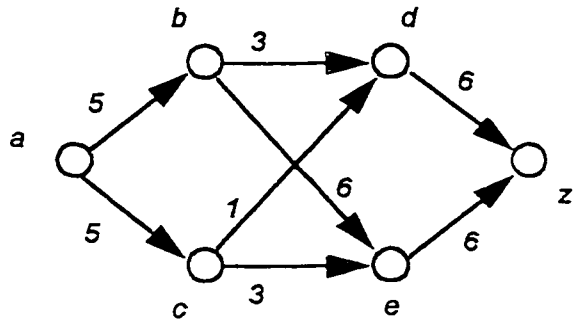


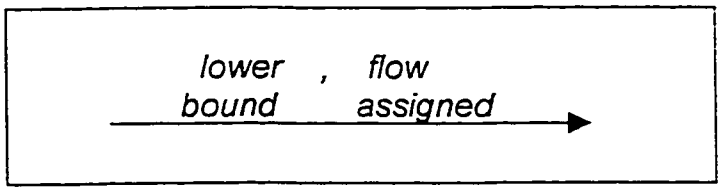
Fig. A1-1. *Looking for total a-z flow that satisfies minimum flows on all edges.*

Find: the minimum total flows throughout the network required to satisfy the lower bounds specified on all edges.

Solution:

(Step 1.) By inspection, we assign some initial flows that will cover all the edges.

Convention for edge labels. In the following figures, each edge is labelled with first the lower bound, followed by a comma, and then the flow assigned.



Try Flow 1 := 6 units along *a - b - e - z* to cover edges *b-e* and *e-z*
Flow 2 := 6 units along *a - c- d - z* to cover edge *d-z*
Flow 3 := 3 units along *a - b - d - z* to cover edge *b-d*
Flow 4 := 3 units along *a - c - e - z* to cover edge *c-e*

Total of all flows = 18 units. (See Fig. A1-2.)

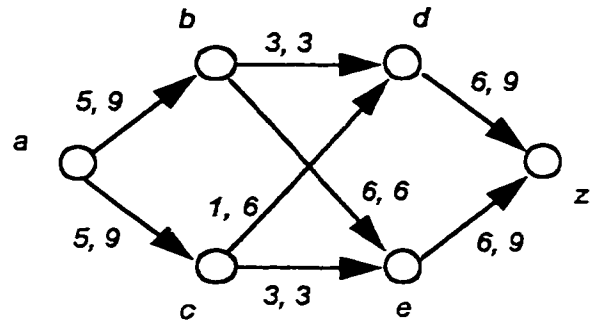


Fig. A1-2. *Initial flows assigned across the network. (lower bound, initial flow)*

We will now try to reduce this total flow (18 units) to the smallest flow that still satisfies the lower bounds on all edges. We proceed with Steps 2 - 5. We start with node *z*.

(Step 2.) Label *z*(-, ∞)
currentNode = *z*
(Step 3a.) There are no outgoing edges from *z* to unlabelled nodes.
(Step 3b.) There are 2 incoming edges: from *d* and from *e*
Consider *d*. Edge *d*-*z* has slack of 9 - 6 = 3. Min (3, ∞) = 3.
Label *d* (*z*, 3)
Consider *e*. Similarly, label *e* (*z*, 3)
(Step 4.) Since node *a* has not been labelled yet, choose another labelled node to be the *currentNode*.

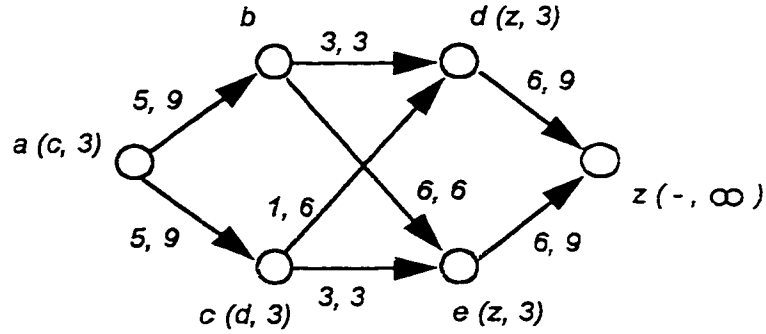


Fig. A1-3. *Labelling the nodes by identifying surplus (slack) flows.*

(Step 4.) *currentNode* = *d*
(Step 3b.) There are no outgoing edges from *d* to unlabelled nodes.
There are 2 incoming edges: from *b* and *c*
Consider *b*. slack (edge *b*-*d*) = 3-3 = 0.
There is no slack, so it (node *b*) cannot be labelled.
Consider c. slack (edge *c*-*d*) = min (6-1, 3) = 3. Label *c* (*d*, 3)

(Step 4.) *currentNode* = *e*
(Step 3b.) There are two incoming edges: from *b* and *c*
Consider *b*. Edge *b-e* has no slack. Cannot be labelled.
Consider *c*. Already labelled.

(Step 4.) *currentNode* = *c*
(Step 3b.) Check the incoming edges.
Consider *a*. Label *a* (*c*, min(9-5, 3)). Label *a* (*c*, 3)
Now that *a* is labelled, proceed to Step 5.

(Step 5) We see that there is slack of 3 units all the way from *z* to *a* along path *z - d - c - a*.
Therefore, we remove 3 units from each edge along this path.
(See the revised flow values in Fig. A1-4.)
Erase all labels from the nodes. Go back to Step 2.

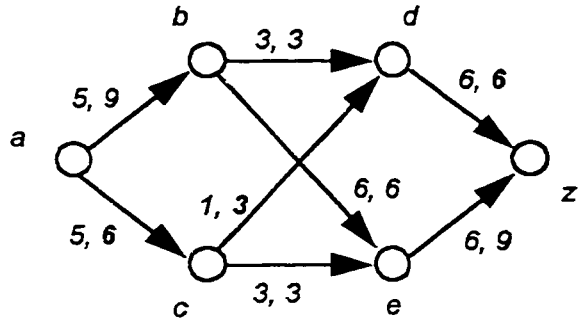


Fig. A1-4. *Flows are revised after first iteration.*

(Step 2.) Label *z*(-, ∞)
currentNode = *z*
(Step 3b.) Check incoming edges:
Consider *d*. Slack (edge *d-z*) = 0. Cannot label.
Consider *e*. Slack (edge *e-z*) = 9-6 = 3. Label *e* (*z*, 3)

(Step 4.) *currentNode* = *e*
(Step 3b.) Check incoming edges:
Consider *b*. Slack = 3-3 = 0. Cannot label.
Consider *c*. Slack = 6-6 = 0. Cannot label.

(Step 4.) *currentNode* = (none).
There are no more labelled nodes to try. We proceed to Step 6.
(Step 6) The max cut has been found.
P' is the set of nodes currently labelled. *P'* = {*z*, *e*}.
Therefore *P* = {*a*, *b*, *c*, *d*} and (*P*, *P'*) is a maximum cut (see Fig. A1-5.)

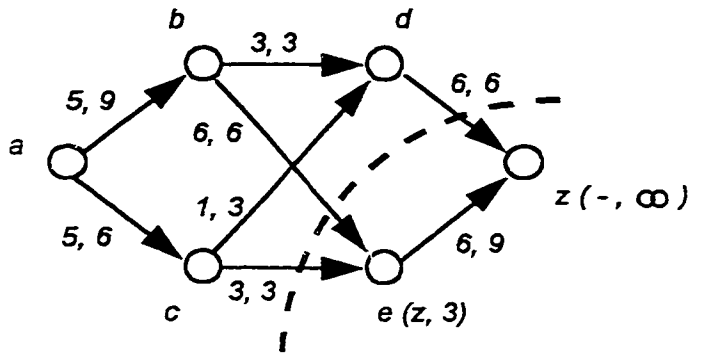


Fig. A1-5. *The maximum cut has been found. Minimum flow* = 3 + 6 + 6 = 15.

In conclusion, it can be seen that we were able to reduce the total flow required from 18 units to 15 units. The actual flow on each edge is shown (after the lower bound). The max cut is shown; the cut set is ({*a, b, c, d*}, {*e, z*}). The capacity of the cut is 15 units of flow. ■

Application.

When the max-cut algorithm is run with all lower bounds being set equal to 1, the capacity of the cut is also equal to the number of paths required to traverse each edge at least once. *This result forms the counting strategy in this thesis.*

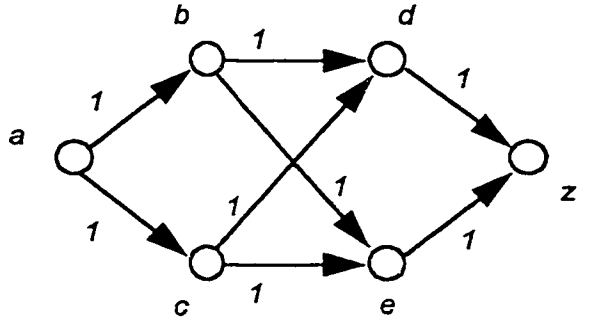


Fig. A1-6. *The lower bounds are all set equal to one unit of flow.*

Repeating the example with the lower bounds set to 1, we would find that the cut set is ({*a, b, c*}, {*d, e, f*}), and that the capacity of the cut is 4. Therefore, four paths would be required to cover the edges in this graph. ■

Note: The iterative portion of this algorithm is known to have an execution time on the order of $O(N^{5/3} T^{2/3})$ where N is the number of nodes and T is the number of transitions. [Cheston++ 79]

APPENDIX 2

Maximum Cut Minimum Flow Theorem

This appendix is derived from Minimum Cut Maximum Flow material in [Tucker 90] and [Lawler 76]

Definitions and notation.

a is the source vertex,

z is the sink vertex,

$k(e)$ is the *minimum* capacity (minimum requirement) of edge e,

$f(e)$ is the flow along edge e,

$\text{In}(x)$ is the set of inward-directed edges at node x,

$\text{Out}(x)$ is the set of outward-directed edges at node x.

An *a-z flow*, f, in a network is defined to be an integer-valued function that is defined on each edge, e, satisfying the following three conditions:

Flow Condition 1. On any edge, the flow must meet or exceed the minimum capacity
$f(e) \geq k(e)$

Flow Condition 2. There are no flows into the source or out of the sink.
$f(e) = 0$ if $e \in \text{In}(a)$ or $e \in \text{In}(a)$

Flow Condition 3. For any node x other than the source, a, or sink, z, the incoming flow is equal to the outgoing flow.

$$\sum_{e \in In(x)} f(e) = \sum_{e \in Out(x)} f(e)$$

Let c be the chain of transitions that was found by one iteration of the decreasing flow algorithm, with the value of the decrement being m.

Let fc be the decreasing a-z "unit flow chain" along c.

We can build upon the Decreasing Flow algorithm, presented in Appendix 1, to produce the following theorems. It is important to recall from Appendix 1 that the initial flows that were assigned always *exceed* the given "capacity" because the capacities are actually minimum requirements for traversal.

Theorem A2-1. For any a-z flow f and any a-z cut (P, P') in a network, the *value* of the flow f is $|f| \geq k(P, P')$, where $k(P, P')$ is the *minimum capacity requirement* of the cut (P, P')

Proof. This proof involves making a construction: we will add a new source node aa to the network such that aa feeds the old source, a, by way of a very small-requirement edge, ee. In the same way that $z \in P$ and $a \in P'$, we also have that $aa \in P'$. We assign a flow value of $|f|$ to ee. Now, f is an aa-z flow in this new expanded network. Applying Flow Condition 3 (with aa instead of a) to P, we get that the flow into P is equal to the flow out of P.
But the flow into P is (at most) $|f|$; that is,

$$|f| \geq \sum_{e \in (P', P)} f(e)$$

And, from Flow Condition 3 (modified), flow into P is equal to the flow out of P:

$$\sum_{e \in (P', P)} f(e) = \sum_{e \in (P, P')} f(e)$$

And, from Flow Condition 1, the flow out of P is greater than or equal to the corresponding minimum requirements out of P:

$$\sum_{e \in (P, P')} f(e) \geq \sum_{e \in (P, P')} k(e)$$

which, by definition, $= k(P, P')$.

By following the chain of the these four equalities or inequalities, we obtain

$$|f| \geq k(P, P'),$$

which is what we wanted to prove. ■

Corollary A2-2. For any a-z flow f and any a-z cut (P, P') in a network, $|f| = k(P, P')$ if and only if

(i) for each edge $e \in (P', P)$, $f(e) = 0$; and

(ii) for each edge $e \in (P, P')$, $f(e) = k(e)$.

And, when $|f| = k(P, P')$, then f is the minimal flow and (P, P') is a cut of maximal requirement.

Proof. The flow from P' into P in the expanded network equals $|f|$ if condition (i) holds; otherwise the flow into P is greater than $|f|$. The flow out of P equals $k(P, P')$ if condition (ii) holds; otherwise it is less. Equality in the set of four inequalities and equalities on the previous page holds if and only if conditions (i) and (ii) above are both true. As for the last sentence, since the flow always meets or exceeds the minimum requirement on each edge, $|f| \geq k(P, P')$, it is obvious that in the case where the flow it is actually *equal*, it is a ***minimum*** (that still satisfied the entire graph). And, satisfying the graph means satisfying the maximal requirement, therefore the cut that is found in this case is maximal. ■

Theorem A2-3. For any given flow, f, that meets or exceeds all minimum flow requirements on all edges, a finite number of applications of the Decreasing Flow Algorithm will result in a minimal flow (that still satisfies all edges). If P is the set of vertices labelled by the final application of the Decreasing Flow algorithm, and P' is the set of remaining (unlabelled) vertices, then (P, P') is a maximum cut set.

Proof. We need to show that application of the Decreasing Flow algorithm results in a "legal" flow, in accordance with the three flow conditions; that is, show that $f - m \cdot fc$ satisfies the three flow conditions.

Since individually f and $m \cdot fc$ each satisfy Flow Conditions 2 and 3 above and are integer-valued, then $f - m \cdot fc$ also satisfies Flow Conditions 2 and 3 and is integer-valued. Because the algorithm is designed to look for slack > 0, we can also state that $f - m \cdot fc$ satisfies Flow Condition 1; that is, $0 \leq f(e) - m \cdot fc(e) \leq k(e)$.

Since m is a positive integer, each new flow is smaller by an integral amount. The capacities and number of edges is finite, therefore the algorithm must eventually halt. At this point in the algorithm, we are unable to find any chain of slack between z and a, and therefore node a remains unlabelled by this iteration of our algorithm. Let P be the set of edges that are labelled at the time that the algorithm halts. Then (P, P') is an a-z cut since z is labelled and a is not. There can be no edge with slack in between any vertex x in P and vertex y in P', otherwise y would have been labelled from x. Therefore, both conditions in Corollary A2-2 hold. Therefore, the final flow is $k(P, P')$ and is minimal; (P, P') is a maximum-requirement cut. ■

Corollory A2-4. In any directed flow network, the value of a minimal a-z flow is equal to the capacity of a maximal-requirement a-z cut.

This corollary arises directly from the previous theorem and is the Maximum Cut Minimum Flow Theorem. ■

APPENDIX 3

Algorithm for Traversing HFSM

The following is a partial pseudo-code listing for the algorithm in Appendix 1, which is for counting paths in a hierarchical finite-state machine (HFSM). C-code is Appendix 6.

```
procedure countPaths (node)
    if (this node contains no subnodes)
    then return (1)
    else
        begin
/*          for each subnode, obtain the number of paths through it */
/*          and label it with its path count                        */
            for (each subnode) do
                temp := countPaths (subnode)
                pathCount [subnode] := temp
            endfor
/*          so now each of the subnodes is labelled with its path count */
            Based on the path counts at the subnodes,
            figure out the minimum number of paths needed
            to traverse this node.  Call this number 'nWays'

            Note: if this node is strongly connected, nWays := 1
                (careful: still must do a lot of traversals
                          during path selection)
            Note: if all its subnodes are leaf-nodes (that is, tagged
                    with pathCount of 1), nWays  := maxCut ()
                    Also record the paths found.
            Note: if some of the tags are greater than 1, (say, k)
                    convert tagged nodes to k-parallel pairs,
                    nWays := maxCut ()
                    Also record the paths found
            return (nWays)
        endelse
endprocedure


procedure maxCut ()
/*  We are looking for the smallest number of paths (flows)
    to cover all the edges.    */
    assign each edge the label 1
    (1 is the lower bound on the flow assigned to that edge)
    (this equals the minimum number of times that this edge must be
traversed)

    fillEdges ()

/*  Now apply a flow-reduction algorithm to find optimal flow  */

    flowReduce ()
```

```
/*  If the edge lower-bounds were all 1, then the value of the final
flow is equal to the size of the max cut and is equal to the
number of paths required to "cover" this graph.  */


procedure fillEdges ()
/*  When done by hand, an intial (non-optimal) assignment of flows
    to edges can be done by inspection. This algorithm, however,
    starts all edges with zero flow and gradually adds flow paths
    until all edges have at least some flow.

    In this process, many edges may receive more than the optimal
    amount (and far more than the required minimum  */

    while (there is still an edge that is has a no flow) do
        select any unfilled edge from the list of edges
        (call this edge 'selectedEdge')
        flow [selectedEdge] := 1  /* assign a flow of 1 to this edge */
        currentEdge := selectedEdge

/*      Increment the flow on *successor* edges continuing forward
        along ONE path until the END of the graph.  */

        nextEdge := aSuccessor (currentEdge)
/*      if there is more than one successor edge,
            the function should choose one that has no flow yet;
        if they all have some flow, it picks any one ,say the least. */
        while (nextEdge != END) do
            flow [nextEdge] := flow [nextEdge] + 1
            /* increment flow on successor */
            currentEdge := nextEdge
            nextEdge := aSuccessor (currentEdge)
        endwhile

/*      Increment the flow on *predecessor* edges along ONE path
        continuing backwards until the beginning of the graph.  */

        currentEdge := selectedEdge
        previousEdge := aPredecessor (currentEdge)
/*      if there is more than one predecessor edge,
            the function should choose one that has no flow yet;
        if they all have some flow, it picks any one, say the least. */
        while (previousEdge != BEGINNING) do
            flow [previousEdge] := flow [previousEdge] + 1
            /* increment flow on predecessor */
            currentEdge := previousEdge
            previousEdge := aPredecessor (currentEdge)
        endwhile

    endwhile

/*  Now, all edges have a flow of at least one (1) unit  */
endprocedure
------------------- /* not coded */ -------
procedure aSuccessor (edge)
procedure aPredecessor (edge)
procedure flowReduce()
```

APPENDIX 4

Algorithm for Finding Paths from Edge Flows

The algorithm in this appendix is taken from [Tucker 80]
It is included here for completeness only.

A4.1 Introduction

In previous appendices, we showed how to find the maximum cut through a graph which would then tell us how many paths (and therefore, how many test cases) would be required to cover this graph. In the activity of determining the maximum cut through a graph, not only have we determined the number of paths, we have also found the routes of the paths. In this instance, the material in this appendix is not required.

If we are provided only with the flows on each edge, it becomes necessary to determine the actual routing. Just like charting the flows of water in a network of tunnels or of electrical current in a circuit, this can often be done by inspection for small graphs. However, the algorithm described below provides a systematic way of obtaining the correct result. This algorithm, from [Tucker 80], assumes that we are given only the *maximum* flows on each edge, not the actual flows. Our case of knowing the *actual* flows is simply a special case in which *all* of the edges become saturated the moment the maximum cut is saturated.

The basic idea is to start with no flows at all. Initially, some simple paths are added by inspection. Then an "augmenting flow algorithm" is applied, which will not only augment some flows but may decrease or reverse others where necessary for optimization.

A4.2 Algorithm

Given: a directed graph representing a network and labels on each edge showing the *upper bound* for flow on that edge. The source node is labelled *a*; the destination node is labelled *z*.

Find: the maximum total flow throughout the network, subject to the constraints imposed by the upper bounds specified on each edge of the network.

Convention for node labels. As before, a node label has the form

nodeName (prevNode, slackUpTillNow)

where *nodeName* is the name of the node (in this appendix, a letter of the alphabet),

prevNode is the previous node in the path, and

slackUpTillNow is the amount of flow available to be reduced along the entire path from *z*, the destination node, until the current node.

Procedure. Assign initial flows. By inspection, assign individual flows from the source to the destination. Unless the optimum flow assignment happens to have been chosen, this step will result in significant surpluses on many edges.

From [Tucker 80]

> The algorithm assigns two labels to a vertex q: $(p\pm, \Delta(q))$, where p is the previous vertex on a flow chain from a to q, the superscript of p is + if the last edge of the chain is [directed] (p, q), and is – if the last edge is [directed] (q, p), and $\Delta(q)$ is the minimum slack among the edges of the chain from a to q. On a backwardly directed edge e, the slack is the amount of flow that can be removed, namely $\varphi(e)$
>
> **Augmenting Flow algorithm**
>
> (An algorithm for a network with a given a-z flow φ.)
>
> 1. Give vertex a the labels $(-, \infty)$. Let a be the first vertex to be scanned.
> 2. Call the vertex being scanned p with second label $\Delta(p)$.

(a) Check each incoming edge $e = (q \dashrightarrow p)$. If $\varphi(e)>0$ and q is unlabelled, then label q with $(p-, \Delta(q))$, where $\Delta(q) = \min(\Delta(p), \varphi(e))$.

(b) Check each outgoing edge $e = (p \dashrightarrow q)$. If $s(e) = k(e)-\varphi(e) > 0$ and q is unlabelled, then label q with $(p+, \Delta(q))$, where $\Delta(q) = \min(\Delta(p), s(e))$.

3. If z has been labelled, go to Step 4. Otherwise choose another labelled vertex to be scanned (which was not previously scanned) and go to Step 2. If there are no more labelled vertices to scan, let P be the set of labelled vertices and now (P, P') is a saturated a-z cut. Moreover, $|\varphi| = k(P, P')$, and thus φ is maximal.

4. Find an a-z chain K of slack edges by backtracking from z as in the shortest path algorithm. Then an a-z flow chain $\varphi(K)$ along K of $\Delta(z)$ is the desired augmenting flow.

Like the shortest path algorithm, the flow algorithm extends partial-flow chains from currently unlabelled vertices to adjacent unlabelled vertices. ■

In the above section, $s(\ )$ is the slack, $k(\ )$ is the capacity, $\varphi(\ )$ is the flow on an edge, and K is a chain of links (transitions).

APPENDIX 5

Illustration of Approach

Sample HFSM and Outputs

This appendix provides an example of a hierarchical finite-state machine (HFSM) with the expected outputs from applying the counting and selection method.

Given: Suppose that we are given the HFSM described by Figures A5-1, A5-2, and A5-3.

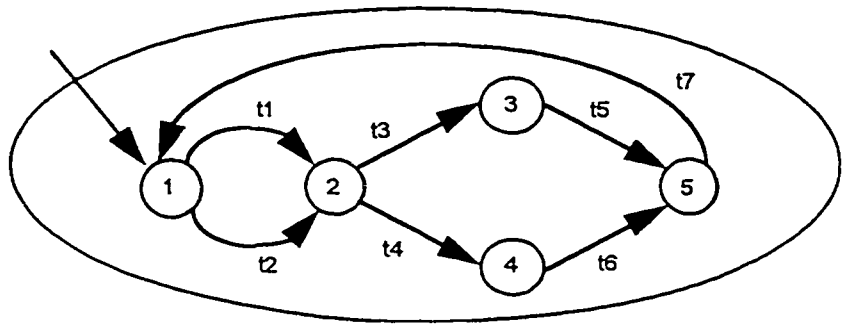


Fig. A5-1. *Top-level node of HFSM, showing Level 1 nodes inside*

The numbering convention that we will use is that names of subnodes are formed by adding digits to the right-hand side. Example: subnodes of node 2 would be called 21, 22, 23, etc. Subnodes of node 42 would be called 421, 422, 423, etc. There is no particular order for numbering within a node.

Objective: The objective is to determine the *minimum number* of paths required to cover this HFSM, and which paths they could be.

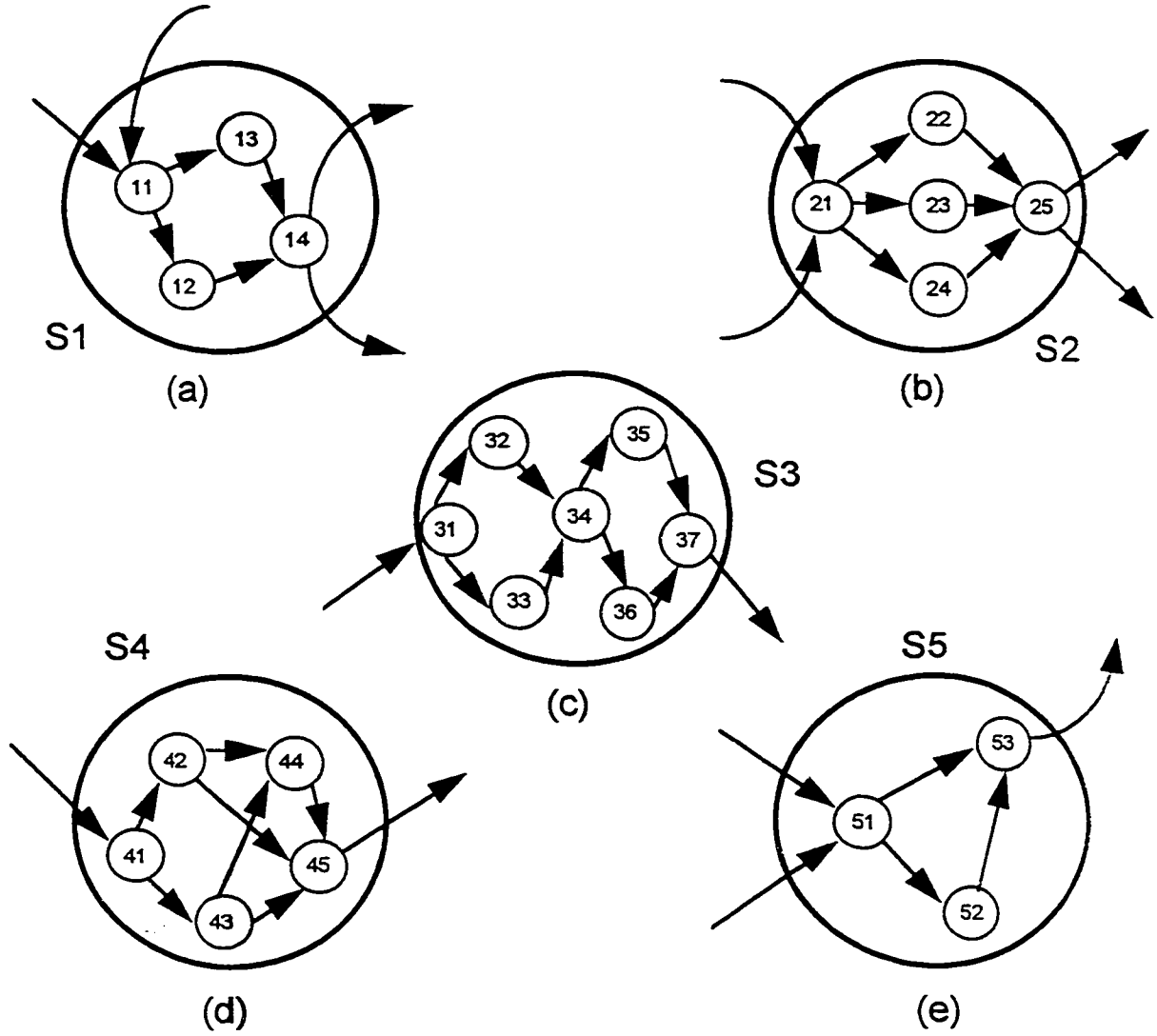


Fig A5-2. *Level 1 nodes showing Level 2 nodes within.*

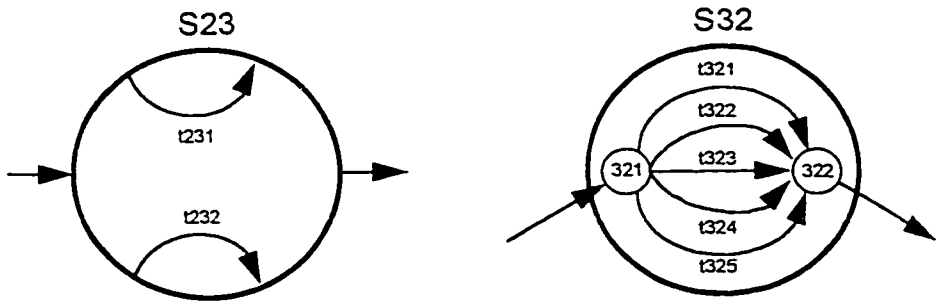


Fig. A5-3. *Level 2 nodes showing Level 3 nodes (or simply transitions) within.*

Step 1. Counting within individual nodes

The first step in counting paths is to count the number of ways through each subnode. This can be done in any order; examples: recursively by way of breadth-first traversal; recursively by way of depth-first traversal; or simply (linearly) the order in which the nodes are 'physically' stored in the toolset. This counting step must also include the top-level node, 'level zero.' The resultant information is shown in the adjoining table.

These are *non-cumulative* totals; they do not take into account paths within subnodes. For this step, subnodes are temporarily considered atomic (non-decomposable).

In counting the number of ways through a given node, it is not necessary to decide on the exact route of each way through; it is only necessary to obtain the "thickness" of each node, that is, the *maximal cut* through the node.

State /Node	Number of ways through
Level 0 node (top level)	
+0	2
Level 1 nodes (1 digit)	
+1	2
+2	4
+3	6
+4	4
+5	2
Level 2 nodes (2 digits)	
11	1
12	1
13	1
14	1
21	1
22	1
+23	2
24	1
25	1
31	1
+32	5
33	1
34	1
35	1
36	1
37	1
41	1
42	1
43	1
44	1
45	1
51	1
52	1
53	1
Level 3 nodes (3 digits)	
321	1
322	1

"+" plus-sign indicates that the node is expandable

Table A5-4. *Step 1.*

Step 2. Obtaining cumulative totals for "one level."

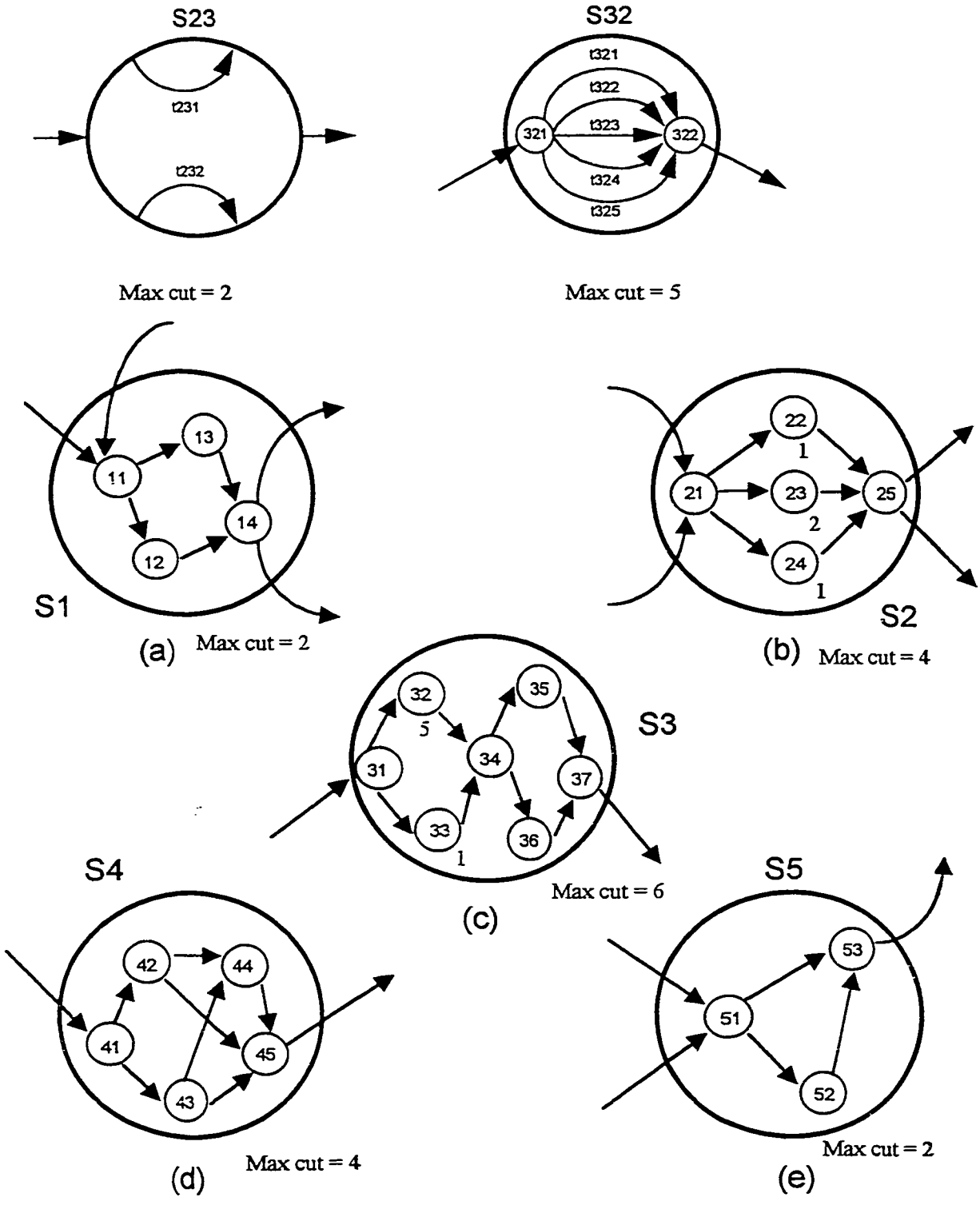


Fig. A5-5. *We annotate the original figures with the maximum cut for each state.*

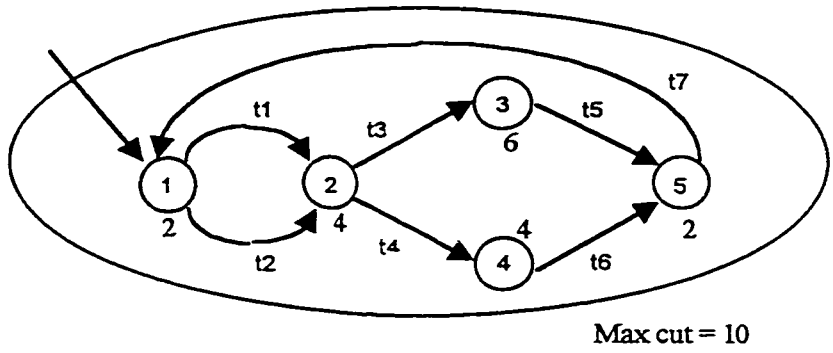


Fig. A5-5 (continued). *Finally, the top level is annotated with the maximum cut (based on the results from all the lower levels).*

Step 2 is to obtain cumulative totals for each node and its subnodes just one level down. Mechanically, this can be done working from the lower levels, upwards (as displayed in the adjoining table). By computer, this can be done recursively during the same traversal that was used for Step 1; that is, as the individual nodes are being counted, the cumulative counts can be tallied as well. (See Table A5-6 on the following page.)

The third column is derived by considering the number of ways through adjoining nodes and the number of transitions incoming or outgoing. For example, node 14 is atomic (only one way through), but there are two different nodes that feed it, therefore it will have to be traversed at least twice. As another example, node 321 is atomic and so is its lone neighbour, but there are five transitions between the two, therefore we already know that node 321 will have to be traversed at least 5 times. Note that the third column is *not yet* the final count of the number of times the node will be traversed; this number reflects the analysis at only one level above. The final count is done in a later step.

Again, it is not yet necessary to know the precise routes for each way through. In many of the cases where the precise route has not yet been determined, a variable is used. For example, Table A5-6 shows that the number of traversals of node 35 and node 36 must add up to 6, it does not matter (for the total count) how they are split. Therefore, node 35 is assigned *n35* and node 36 is assigned (6 - *n35*).

Step 3. Deriving cumulative totals for the entire graph (all levels).

By the end of Step 2, we have determined the maximal cut for the entire graph and hence the derived minimum number of times that this graph must be traversed. What we have *not* done is updated the required number of traversals for each of the lower levels, based on the totals at the higher levels.

Because the requirements at the higher levels may impose a number of seemingly unnecessary traversals (to satisfy other nodes upstream or downstream), it will not matter academically which routes are chosen for repetition. However, this is a practical consideration and an opportunity for user-driven input to make some decisions on what is the best for test coverage or what is the least expensive path for testing.

The results of Step 3 tabulation are shown in Table A5-7.

State /Node	NWT = Number of ways through	# of paths that it is on (at level of its parent node)
Level 3 nodes (3 digits)		
321	1	5
322	1	5
Level 2 nodes (2 digits)		
11	1	2
12	1	1
13	1	1
14	1	2
21	1	4
22	1	1
+23	2	2
24	1	1
25	1	4
31	1	6
+32	5	5
33	1	1
34	1	6
35	1	*n35* (*where* $0 \leq n35 \leq 6$)
36	1	6-*n35*
37	1	6
41	1	4
42	1	2
43	1	2
44	1	2
45	1	4
51	1	2
52	1	1
53	1	2
Level 1 nodes (1 digit)		
+1	2	10
+2	4	10
+3	6	*n3* (*where* $6 \leq n3 \leq 10$)
+4	4	10-*n3*
+5	2	10
Level 0 node (top level)		
+0	2	10

Table A5-6. *Step 2.*

State /Node	Number of ways through	Derived cumulative totals (all levels)
Level 1 nodes (1 digit)		
+1	2	10
+2	4	10
+3	6	*n3* ($6 \leq n3 \leq 10$)
+4	4	10-*n3*
+5	2	10
Level 2 nodes (2 digits)		
11	1	10
12	1	*n12*
13	1	*10-n12* ($1 \leq n12 \leq 10$)
14	1	10
21	1	10
22	1	*n22* ($1 \leq n22 \leq 7$)
+23	2	*n23* ($2 \leq n23 \leq 10-n22-1$)
24	1	10-*n22-n23*
25	1	10
31	1	6
+32	5	5
33	1	1
34	1	6
35	1	*n35* ($1 \leq n35 \leq 6$)
36	1	6-*n35*
37	1	6
41	1	4
42	1	2
43	1	2
44	1	2
45	1	4
51	1	10
52	1	*n52* ($1 \leq n52 \leq 10$)
53	1	10
Level 3 nodes (3 digits)		
321	1	5
322	1	5

Table A5-7. *Step 3. Figures for upper levels are pushed down to the lower levels.*

Step 4. Selection of paths.

Now that we have finished counting paths, we can start selecting them. A path selection tool should be able to provide a list of paths, citing name of node and transition to be followed.

Notation.

- *In the lists that follow, the name of the transition is not given for lower-level nodes whenever there is only one way of proceeding from one state to another.*
- *Optional parentheses and boldface fonts have been added to highlight the regions in which variations occur from one path to the next*

By inspection, we can find two top-level cycles:

Cycle 1. S1-t1-S2-t3-S3-t5-S5-t7-S1 *(the "upper" route)*
Cycle 2. S1-t2-S2-t4-S4-t6-S5-t7-S1 *(the "lower" route)*

Upper-paths:

Path 1. S1-(S11-**S13**-S14)-t1-S2-(S21-**S22**-S25)-t3-S3-S31-S32-(S321-**t321**-S322)-S34-S35-S37-t5-S5-(S51-S53)-t7-S1

Path 2. S1-S11-S13-S14-t1-S2-(S21-**S23-t231**-S25)-t3-S3-S31-S32-(S321-**t322**-S322)-S34-S35-S37-t5-S5-(S51-**S52**-S53)-t7-S1

Path 3. S1-(S11-S13-S14)-t1-S2-(S21-S23-**t232**-S25)-t3-S3-S31-S32-(S321-**t323**-S322)-S34-S35-S37-t5-S5-(S51-S53)-t7-S1

Path 4. S1-(S11-**S12**-S14)-t1-S2-(S21-**S24**-S25)-t3-S3-S31-S32-(S321-**t324**-S322)-S34-S35-S37-t5-S5-(S51-S52-S53)-t7-S1

Path 5. S1-S11-S12-S14-t1-S2-S21-S24-S25-t3-S3-S31-S32-(S321-**t325**-S322)-S34-S35-S37-t5-S5-(S51-S53)-t7-S1

Path 6. S1-S11-S12-S14-t1-S2-(S21-S22-S25)-t3-S3-S31-**S33**-(S321-t325-S322)-S34-S35-S37-t5-S5-(S51-S52-S53)-t7-S1

Lower paths:

Path 7. S1-(S11-**S13**-S14)-**t2**-S2-(S21-S22-S25)-**t4**-S4-(S41-**S42-S44**-S45)-**t6**-S5-(S51-S53)-t7-S1

Path 8. S1-(S11-S13-S14)-t2-S2-(S21-**S23-t231**-S25)-t4-S4-(S41-**S42**-S45)-t6-S5-(S51-**S52**-S53)-t7-S1

Path 9. S1-(S11-**S12**-S14)-t2-S2-(S21-S23-**t232**-S25)-t4-S4-(S41-**S43-S44**-S45)-t6-S5-(S51-S53)-t7-S1

Path 10. S1-(S11-S12-S14)-t2-S2-(S21-**S24**-S25)-t4-S4-(S41-**S43**-S45)-t6-S5-(S51-S52-S53)-t7-S1

The distribution of routes has been fairly uniform in the selection of these paths. In other words, if there were only two possible ways through a node that had to be traversed ten times, then each route was taken five times. The changes were made in such a manner so as to increase the number of different pairings or combinations of changes; for example: Given a node with $\{a$ .or. $b\}$, followed by 1 of 6 possible transitions, followed by a node with $\{x$ .or. $y\}$, the path selection would be:

a-t1-x

a-t2-y

a-t3-x

b-t4-y

b-t5-x

b-t6-y

Recap.

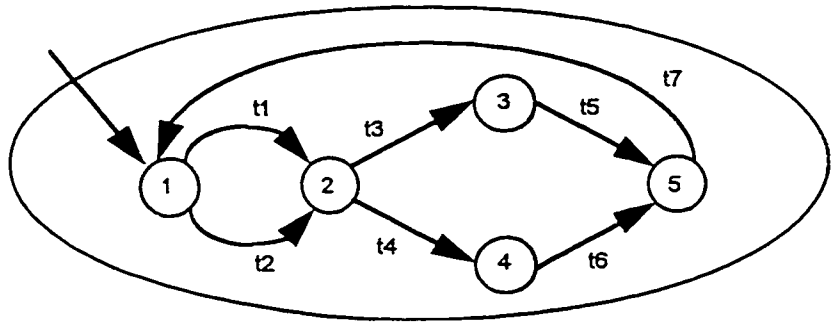


Fig. A5-1 (repeated). *Top-level node of HFSM, showing Level 1 nodes inside*

We have seen that the top-level HFSM appears to require only two paths to test it (assuming that they are both feasible). As we delve deeper into the hierarchy, more sub-paths become visible. If we have a tool that permits the setting of a "cut-off depth" (Section 5.5.1), we can adjust the number of test paths generated. We determine that covering *all* levels in this example requires ten paths (shown in Step 2). In doing so, many nodes will be covered more than their minimum number of times; the final counts are shown in Step 3. The paths actually suggested (including transitions) are listed in Step 4. ■

APPENDIX 6

C-program source code for traversing HFSM

The following is a source code listing for a program that finds the maximum "cut" in a hierarchical finite-state machine (HFSM). It calculates the maximum cut by traversing the HFSM and putting unit flows onto each transition, and then summing the flows entering the sink node. This program incorporates the algorithms described in the other appendices.

This computer program was written and developed using *Borland C++ Builder*, Version 4, "Standard," (Windows 98/95/NT), from Inprise Corporation. Execution was performed on an Intel-based microcomputer.

```
// ASTRO2d.CPP  - Recursive traversal of hierarchical graph

//*************************************************************
//  This console program calls a recursive traversal subroutine
//    to obtain the maximum cut (MAXCUT) of a flowgraph
//
//   Last update: 2000-03-07 20:15
//*************************************************************
#pragma hdrstop
#include <condefs.h>
#include <fstream.h>    // needed for file manipulation: ifstream
#include <iostream.h>   // needed for cout, endl (cin?)
#include <conio.h>      // needed for getch()
#define  TRUE   1
#define  FALSE  0
#define  YES    1
#define  NO     0
//---------------------------------------------------------------------------
#pragma argsused
int const MAXFILENAMELENGTH = 80, MAXNAME = 40, MAXNODES = 10;
int readMatrix (int[MAXNODES][MAXNODES], int&, int[MAXNODES], char[MAXNAME]);
void traverse (int[MAXNODES][MAXNODES], int&, int[MAXNODES],char[MAXNAME],int&);
int inEdges (int&, int[MAXNODES][MAXNODES], int&, int&);
int outEdges (int&, int[MAXNODES][MAXNODES], int&, int&);
void copylb2flow (int[MAXNODES][MAXNODES], int&, int[MAXNODES][MAXNODES]);
void printMatrix (int[MAXNODES][MAXNODES], int&);
void fillEdges (int[MAXNODES][MAXNODES], int[MAXNODES][MAXNODES], int&);
void decrFlow (int[MAXNODES][MAXNODES],int&, int[MAXNODES][MAXNODES],int&,int&);
ifstream file5;    // file5 is the "handle" for our input file

//===================================
int main(int argc, char* argv[])
{
      int child[MAXNODES], graphSize, maxcut,
           myGraphLB[MAXNODES][MAXNODES];
      char comment[100], filename[MAXFILENAMELENGTH], gName[MAXNAME];

      cout << "astro2d.exe" << endl;

//  Prompt for and Open the input data file
      cout << "Please enter input-filename: ";
      cin.getline(filename, sizeof(filename)-1);
      file5.open(filename);       // try to open file
      if (!file5) {               // if cannot open file
        printf("Error Opening File\n");
        cout << endl << "Press any key to continue ...";
        getch();
        return 0;
      }
      else {  // file is opened okay

//  Read the Comment line from the input file
        file5.getline(comment, sizeof(comment)-1);
        cout << "Comment: " << comment << endl;

        readMatrix(myGraphLB, graphSize, child, gName);

        traverse (myGraphLB, graphSize, child, gName, maxcut);

        cout << endl << "In main()";
        cout << "\nMAXCUT OVERALL = " << maxcut << endl;

//  Close the input data file
        file5.close();
```

```
        cout << endl << "Normal termination.";
        cout << endl << "Press any key to continue ...";
        getch();
        }  //endelse

        return 0;
}
//-------------------------------------------------------------
void traverse(int lb[MAXNODES][MAXNODES], int& nNodes,
              int offspring[MAXNODES], char gName[MAXNAME], int& minFlow)
{
//  This recursive subprogram traverses a node (graph) that may or may not
//     contain sub-nodes (sub-graphs).
//
//   Accepts: lb        - matrix representation of the graph, where
//            lb[x][y] = f, a positive integer representing the min flow
//                       ("lower bound") from node x to y;
//                     = 0, otherwise.
//            nNodes    - number of nodes (vertices) in the graph
//            offspring[j] = 1, if node j has any subnodes
//                         = 0, otherwise (that is, j is a "leaf node")
//            gName     - name of the graph (for easy reference)
//   Returns: minFlow = the minimum flow (= maximum cut) across this graph
//            lb       - updates lb, the lower bound on egde-flows, according
//                       to node capacities that are derived from the hierarchy
//
//   Last update:  2000-03-07 22:25
//-------------------------------
      int capacity[MAXNODES], delta, entry, firstNode,
          grandchild[MAXNODES], in, lastNode, min, mincol, minFlow2,
          minrow, myGraph[MAXNODES][MAXNODES], node, out, row,
          sink, sinkFound, source, sourceFound,
          subGraphLB[MAXNODES][MAXNODES], subgraphSize, sum;
      char subGName[MAXNAME];

//    myGraph - is a working copy of the flow matrix in which
//    myGraph[x][y] = 0, if there is a transition from node x to y;
//                  = -1, if there is no transition from x to y.
//    We will add and remove flows from myGraph while respecting lower bounds

       cout << endl << endl << "In traverse()";
       cout << "\t\t\t\tPress any key to continue ...";
       getch();

       firstNode = 0;
       lastNode = nNodes - 1;

// Step 1. Create a copy of the flow matrix from the lower-bounds matrix.
//          (This will be our working copy to adjust the flows.)

       copylb2flow (lb, nNodes, myGraph);
       cout << endl << "  The corresponding Flow matrix (working copy):";
       printMatrix (myGraph, nNodes);

// Step 2. Determine (recursively) the "value" of each node in this graph
//          (This is dependent on the values of its subnodes, if any)
       cout << endl << "  Analysis of Lower Bounds matrix above:";
//     For each node in this graph:
       for (node=firstNode; node <=lastNode; node++) {
         cout << endl << "  Analysing node: " << node;
         cout << "\t\t\t\tPress any key to continue ...";
         getch();
         if (offspring[node] == YES) {
           cout <<endl<<"  offspring["<<node<<"] = YES (Read next subgraph:)\n";
           readMatrix(subGraphLB, subgraphSize, grandchild, subGName);
```

```
          traverse (subGraphLB, subgraphSize, grandchild, subGName, minFlow2);
//        ** recursion occurs here **

          capacity[node] = minFlow2;
        }
        else { // there are no subnodes
          capacity[node] = 1;
        }
      cout<<"\n  capacity of node "<<node<<" is "<<capacity[node]<<endl;
      }  //endfor

// Step 3. Find source node and sink node.
//   Find Source node (no predecessors => entire column = -1)
        sourceFound = FALSE;
        source = 0;       // try first node first
        while (sourceFound == FALSE && source <= lastNode) {
          sourceFound = TRUE;
          for (int i=0; i<=lastNode; i++)
            if (myGraph[i][source] != -1) sourceFound = FALSE;
          if (sourceFound == FALSE) source++;
        }
        cout << endl <<"  Conclusion: source node = " << source << endl;

//    Find Sink node (no successors => entire row = -1)
        sinkFound = FALSE;
        sink = lastNode;      // try last node first
        while (sinkFound == FALSE && sink >= 0) {
          sinkFound = TRUE;
          for (int j=0; j<=lastNode; j++)
            if (myGraph[sink][j] != -1) sinkFound = FALSE;
          if (sinkFound == FALSE) sink--;
        }
        cout << "  Conclusion:   sink node = " << sink << endl;
//  Source and sink node have now been established

// Step 4.  From Step 2, we now have the capacity of all subnodes in the graph
//    All subnodes have been traversed. Now update lower bounds accordingly.
//    Apply cap[] to lb[][] by increasing "lowerBound" up to "capacity"
//       (if not already that high)

// Step 4a.  Handle non-sink, non-source nodes. Increase low bound if required.
      for (node = firstNode; node <= lastNode; node++){
        if (node != source && node != sink) {
          if ((inEdges(node, lb, nNodes, in) == 1) && // If only 1 incoming
edge,
                (capacity[node] > lb[in][node])) {  //and flow < cap[node],
            lb[in][node] = capacity[node]; //then load that edge.
          }
          if ((outEdges(node, lb, nNodes, out) == 1) // If only 1 outgoing edge,
               && (capacity[node] > lb[node][out])) {// and its flow < capnode,
              lb[node][out] = capacity[node];   // then load that edge.
          }
        }
      }

      cout << endl << "  Updated Lower Bounds matrix: " << gName;
      printMatrix (lb, nNodes);
      cout << endl;

// Step 4b. Fill the edges.
//    Assign initial end-to-end flows, based on lower bounds (lb).
//    (Typically, this results in overfilling many edges.)

      fillEdges(lb, myGraph, nNodes);

// Step 4c. Reduce flow.
```

```
      decrFlow (myGraph, nNodes, lb, source, sink);
//      printMatrix(myGraph, nNodes);

// Step 5.  Determine if source and sink nodes increase the lower bounds.
// Step 5a. Handle source node. Increase lb[][] if required.
      sum = 0;           //Find total outflow from source
      for (int j=0; j<=lastNode; j++) sum = sum + lb[source][j];
      cout << "source node: sum = " << sum;
      delta = capacity[source] - sum;
      cout << "      delta = " << delta << "    " << gName << endl;

      if (delta > 0) {   // If source capacity > total outflow
//      Find smallest positive number in this row
        min = 99999;
        mincol = 0;
        for (int col=0; col<=lastNode; col++) {
          entry = lb[source][col];
          if (entry < min && entry > 0) {
            min = entry;
            mincol = col;
          }
        } //smallest non-negative entry is found
        lb[source][mincol] += delta; // top-up the smallest outgoing edge

        // Step 5c. Top-up the edges.
        //    Assign additional end-to-end flows, based on revised lower bounds
        fillEdges(lb, myGraph, nNodes);

        // Step 5d. Reduce flow.
        decrFlow (myGraph, nNodes, lb, source, sink);
      } //endif

// Step 6a. Handle sink node
      sum = 0;           //Find total inward to sink
      for (int i=0; i<=lastNode; i++) sum = sum + lb[i][sink];
      cout << "  sink node: sum = " << sum;
      delta = capacity[sink] - sum;
      cout << "      delta = " << delta << endl;
      if (delta > 0) {   // If sink capacity > total inward flow
//      Find smallest positive number in this column
        min = 99999;
        minrow = 0;
        for (int row=0; row<=lastNode; row++) {
          entry = lb[row][sink];
          if (entry < min && entry > 0) {
            min = entry;
            minrow = row;
          }
        } //smallest non-negative entry is found
        lb[minrow][sink] += delta; // top-up the smallest inward edge
// Step 6b. Top-up the other edges, if lower bound was changed
        fillEdges(lb, myGraph, nNodes);
// Step 6c. Reduce flow.
        decrFlow (myGraph, nNodes, lb, source, sink);
      }

// Step 7. Calculate minimum flow.
//    minFlow equal the sum of all (non-negative) flows into the sink
//    INCLUDING all sub-graphs in that graph
      minFlow = 0;
      for (int row=0; row<=lastNode; row++) {
        if (myGraph[row][sink] > 0)
          minFlow = minFlow + myGraph[row][sink];
      }
      cout << "*** minFlow of " << gName << " is " << minFlow;
      cout << "       (= sum of flows into sink) " << endl;
```

```
      cout << endl << "Final flow matrix for " << gName;
      printMatrix (myGraph, nNodes);

      cout << "Out traverse()" << endl;

      return;    // return value minFlow; end traverse()
}
//-----------------------------------------------------
int inEdges (int& node, int lb[MAXNODES][MAXNODES], int& nNodes, int& whichOne)
{
//  This subprogram returns the number of incoming edges to the node
//   listed as the first parameter in the calling list.
//  If there is only 1 incoming edge, it is saved as 'whichOne'.
//                                          Called by traverse()
//  Accepts:  node, lb[][], nNodes
//  Returns:  inEdges(), whichOne
//  Last update: 2000-03-07 00:08
//---------------
// Check if there is only 1 incoming edge to this node
//   Count number of non-zero entries in this node's column
      int count, i, lastNode;
      count = 0;
      lastNode = nNodes - 1;
      for (i=0; i<=lastNode; i++) {
        if (lb[i][node] != 0) {
          count ++;
          whichOne = i;
        }
      }
      return count;   // called by traverse()
}
//-----------------------------------------------------
int outEdges (int& node, int lb[MAXNODES][MAXNODES], int& nNodes, int& whichOne)
{
//  This subprogram returns the number of outgoing edges from the node
//   listed as the first parameter in the calling list.
//  If there is only 1 outgoing edge, it is saved as 'whichOne'
//                                          Called by traverse()
//  Accepts:  node, lb[][], nNodes
//  Returns:  outEdges(), whichOne
//  Last update: 2000-03-07 00:10
//---------------
// Check if there is only 1 outgoing edge from this node
//   Count number of non-zero entries in this node's row
      int count, j, lastNode;
      lastNode = nNodes - 1;
      count = 0;
      for (j=0; j<=lastNode; j++) {
        if (lb[node][j] != 0) {
          count ++;
          whichOne = j;
        }
      }
      return count;   // called by traverse()
}
//-------------------------------------------------------------
vcid decrFlow(int myMatrix[MAXNODES][MAXNODES], int& nNodes,
              int lb[MAXNODES][MAXNODES], int& sourceNode, int& sinkNode)
{
//  This subprogram accepts a flow graph and reduces the flows
//    until reaching the minimum amount that will still cover the graph
//    with at least [lower bound] units of flow on every edge.
//   This subprogram incorporates a "decreasing flow algorithm"
//    that uses a node-labelling techique to find surplus flows
//
//   Accepts: myMatrix[x][y] = f, where f is a suggested flow (f >= 1)
```

```
//                                from node x to node y,
//                              = -1 when there exists no edge (therefore no flow
//                  nNodes = number of nodes in myMatrix
//                lb[x][y] = lower bound (desired flow) on transition (x,y)
//              sourceNode = an integer (0..nNodes) indicating the origin
//                sinkNode = an integer (0..nNodes) indicating the destination
//    Returns: myMatrix[x][y] = f, where f is the reduced flow from x to y,
//                              = -1 when there exists no edge
//
//      Last updated: 2000-03-06 21:30
//-------------------------------------

        int c, currentNode, entry, foundFlag, i, j, labelling, lastNode,
            maxcut, r, reduction, surplusOnThisEdge;
        int arrivedViaNode [MAXNODES], hadATurn[MAXNODES],
            slackUpTillNow [MAXNODES];
        // For each node, we use an array to keep track of:
        //      which other node this node was labelled from,
        //      the min amt of slack there is along the path from sink to here
        //      whether or not this node has had a turn at being "current node"

        int CHECKED = 1,
            INFINITY = 99999,
            NOT = -1,               NOT_UNDETERMINED = -2,
            POSSIBLE = 1,           UNDETERMINED = -1;

        cout << "In decrFlow() " << endl;
        lastNode = nNodes - 1;

// Initialise arrays
        for (int k=0; k<=lastNode; k++) {
          arrivedViaNode[k] = UNDETERMINED;
          slackUpTillNow[k] = INFINITY;
          hadATurn[k] = NO;
          }

// "Step 2": Start the labelling process with sink node
        currentNode = sinkNode;
        arrivedViaNode[sinkNode] = NOT_UNDETERMINED;
        slackUpTillNow[sinkNode] = INFINITY;
        labelling = POSSIBLE;

        while (labelling == POSSIBLE) {          // Main Loop
          cout << "Current node = " << currentNode << endl;

//      Label nodes on outgoing edges, if any
//        "Step 3a": Check each outgoing edge from currentNode
          for (c=0; c<=lastNode; c++) {  // for each downstream node
            if(myMatrix[currentNode][c]>=1 && arrivedViaNode[c]==UNDETERMINED) {
              // then case: c is a downstream node from current node
              //                  and c is yet unlabelled
              arrivedViaNode[c] = currentNode;       // label node 'c'
              slackUpTillNow[c] = min (slackUpTillNow[currentNode],
                                       myMatrix[currentNode][c]);
            }  // endif
          }   // endfor

//      Label nodes on incoming edges, if any
//        "Step 3b": Check each incoming edge to currentNode
          for (r=0; r<=lastNode; r++) {    // for each upstream node
            surplusOnThisEdge = myMatrix[r][currentNode] - lb[r][currentNode];
            if (surplusOnThisEdge > 0 && arrivedViaNode[r] == UNDETERMINED) {
              arrivedViaNode[r] = currentNode;
              cout << "  Potential surplus on Edge(" << r << ", "
                   << currentNode << ") = " << surplusOnThisEdge << endl;
              slackUpTillNow[r] = min (slackUpTillNow[currentNode],
```

```
                                        surplusOnThisEdge);
          } // endif
        } //endfor

        hadATurn[currentNode] = CHECKED;

        cout << "\t\t\t\t\t Press any key to continue..";
        getch();
        cout << endl;
//      Check the results of this iteration of labelling:
//      "Step 4": If we reached and labelled source node
        if (arrivedViaNode[sourceNode] != UNDETERMINED) {

//        "Step 5": Remove surplus flow from end to end
          cout << "Removing flow"  << endl;
          reduction = slackUpTillNow[sourceNode];
          i = sourceNode;
          j = arrivedViaNode[sourceNode];
          while (i != sinkNode) {     // traverse end-to-end flow
            myMatrix[i][j] = myMatrix[i][j] - reduction;
            cout << "  Flow(" << i << ", " << j << ") is decr by "
                 << reduction << endl;
            i = j;
            j = arrivedViaNode[i];
          }
//        erase all labels
          for (int k=0; k<=lastNode; k++) {
            arrivedViaNode[k] = UNDETERMINED;
            slackUpTillNow[k] = INFINITY;
            hadATurn[k] = NO;
          }
          printMatrix(myMatrix, nNodes);
          currentNode = sinkNode;    // "Step 2" begins again
          cout << endl << "---------------------";
          cout << endl << "Re-start at sink node" << endl;
        } // endif

        else {     // source node is not yet reached.
//            If there are labelled nodes we haven't tried yet
//                    currentNode = some other node
          foundFlag = FALSE;
          for (int k=lastNode; k>=0; k--) { // look for untried labelled node
            if (arrivedViaNode[k] != UNDETERMINED &&
                    hadATurn[k] == NO) {
              foundFlag = TRUE;
              currentNode = k;
            }
          } // endfor

          if (foundFlag != TRUE) labelling = NOT * POSSIBLE;
//          If there are no more labelled nodes to try,
//          Then labelling is not possible; Set Flag; Exit to "Step 6"
//          Else try the new-found node; Go back to "Step 3a"

        } //endelse

      } // endwhile

//  "Step 6": The MAX CUT is found.  All surpluses have been removed.
      cout << endl << "No further surplus flows to remove." << endl;

//    maxcut = 0;
//    for (int k=0; k<=lastNode; k++) {
//      entry = myMatrix[k][sinkNode];
//      if (entry != -1) maxcut = maxcut + entry;
      // convention: -1 means no edge
```

```
//      }
//      cout << "decrFlow(): Min Flow = " << maxcut << endl;
        return;    // end decrFlow(); return to traverse()
}
//-----------------------------------------------------------------
void fillEdges(int lowerBound[MAXNODES][MAXNODES],
               int myMatrix[MAXNODES][MAXNODES], int& nNodes)
{
//  This subprogram fills a graph with end-to-end flows until the flow on
//     each edge is at least the "lower bound" specified by matrix lowerBound
//
//   Accepts: lowerBound[x][y] = k, if there is a transition from node x to y;
//                             = 0, otherwise;
//            myMatrix[x][y] = 0, if there is a transition from node x to y;
//                             = -1, otherwise;
//                     nNodes = the number of nodes in the graph
//   Returns: myMatrix[x][y] = f, where f is a suggested flow from x to y,
//                             = -1 when there exists no edge (therefore no flow)
//
//      Last updated: 2000-03-06 22:30
//-------------------------------------

        int entry, i, ifound, ix, j, jfound, jx, lastNode, matrixFilled, min,
            mincol, minrow, noPredecessors, noSuccessors, shortfall, sum;
        cout << "In fillEdges() " << endl;
        lastNode = nNodes - 1;

// +--------------
// |
// | Repeat the following until there are no more edges that have less than LB:
// |
// +--------------
//  Find an edge that has less flow than Lower Bound flow
        matrixFilled = TRUE;
        for (int r=0; r<=lastNode && matrixFilled == TRUE; r++) {
          for (int c=0; c<=lastNode && matrixFilled == TRUE; c++) {

            if (myMatrix[r][c] != -1) {      // skip non-existent edges
              shortfall = lowerBound[r][c] - myMatrix[r][c];
              if (shortfall > 0) {
                matrixFilled = FALSE;
                i = r;
                j = c;
              }
            }

          }
        }

    while (matrixFilled == FALSE) {    //-------(Main Loop)-------

//  Call this the "Found Edge."  Put 'shortfall' unit(s) of flow on it.
        ifound = i;
        jfound = j;
        myMatrix[ifound][jfound] = myMatrix[ifound][jfound] + shortfall;
        //current edge = (ifound, jfound)
        cout << endl << "----------------------" << endl;
        cout << "Current edge = ("<<i<< ", "<<j<<")" << endl;
        cout << "  Adding " <<shortfall<< " unit(s) of flow to current edge";

//  Add one unit of flow to all edges from Found edge back to Source node

//   +----------
//   |   First, check for predecessor(s)
//   |    Technique: the column for that node will be all '-1's
//   +----------
```

```
        sum = 0;
        for (int row=0; row<=lastNode; row++)
          sum = sum + myMatrix[row][ifound];
        if (sum != -nNodes) {                  // Process predecessors
//      then
          noPredecessors = FALSE;
          cout << endl << "  There are predecessors" << endl;
//        start filling edges backwards to Sink node
          ix = ifound;
          jx = jfound;  //current edge = (ix, jx)

          while (noPredecessors == FALSE) {
//          Find smallest non-negative number in this column
            min = 99999;
            minrow = 0;
            for (int row=0; row<=lastNode; row++) {
              entry = myMatrix[row][ix];
              if (entry < min && entry >= 0) {
                min = entry;
                minrow = row;
              }
            }  //smallest non-negative entry is found
            // current edge is now (minrow, ix)
            jx = ix;
            ix = minrow;  //current edge is now (ix, jx)
            myMatrix[ix][jx] = myMatrix[ix][jx] + shortfall;  //add flow
            cout << "  Adding " << shortfall
                 << " unit(s) of flow to predecessor edge = ("
                 << ix << ", " << jx << ")";

            cout << "\t\t\t\tPress any key to continue ...";
            getch();

//          Check for additional predecessor(s)
            sum = 0;
            for (int row=0; row<=lastNode; row++)
              sum = sum + myMatrix[row][ix];
            if (sum == -nNodes) noPredecessors = TRUE;  // set flag and exit
          }    //end-while
        }  //end-then  (A path has been drawn from Found node to Source)

//    +----------
//    |  Next, check for successor(s)
//    |    Technique: the row for that node will be all '-1's
//    +----------
        sum = 0;
        for (int col=0; col<=lastNode; col++)
          sum = sum + myMatrix[jfound][col];

        if (sum != -nNodes) {                  // Process successors
//      then
          noSuccessors = FALSE;
          cout << endl << "  There are successors" << endl;
//        start filling edges forwards to Sink node
          ix = ifound;
          jx = jfound;  //current edge = (ix, jx)
          while (noSuccessors == FALSE) {
//          Find smallest non-negative number in this row
            min = 99999;
            mincol = 0;
            for (int col=0; col<=lastNode; col++) {
              entry = myMatrix[jx][col];
              if (entry < min && entry >= 0) {
                min = entry;
                mincol = col;
              }
```

```
            }  //smallest non-negative entry is found
            // current edge is now (jx, mincol);
            ix = jx;
            jx = mincol;  //current edge is now (ix, jx)
            cout << "  Adding " << shortfall
                 << " unit(s) of flow to successor edge = ("
                 << ix << ", " << jx << ")";
            myMatrix[ix][jx] = myMatrix[ix][jx] + shortfall;  //add flow

//          Check for additional successor(s)
            sum = 0;
            for (int col=0; col<=lastNode; col++)
              sum = sum + myMatrix[jx][col];
            if (sum == -nNodes) noSuccessors = TRUE;
          }    //end-while
        }  //end-then  (A path has been drawn from Found node to Sink)

      printMatrix (myMatrix, nNodes);
      cout << "\t\t\t\t\t Press any key to continue..";
      getch();
      cout << endl;

//  Check to see if there are any more empty edges (zero entries)
      matrixFilled = TRUE;
      for (int r=0; r<=lastNode && matrixFilled == TRUE; r++) {
        for (int c=0; c<=lastNode && matrixFilled == TRUE; c++) {

          if (myMatrix[r][c] != -1) {      // skip non-existent edges
            shortfall = lowerBound[r][c] - myMatrix[r][c];
            if (shortfall > 0) {
              matrixFilled = FALSE;
              i = r;
              j = c;
            }
          }

        }
      }

   }  // end-while  (End of Main Loop)

//  If there are no more empty edges, then we are finished filling.

        return;  // return to traverse()
}
//-------------------------------------------------------------
void copylb2flow(int lowerBound[MAXNODES][MAXNODES], int& nNodes,
                 int flow[MAXNODES][MAXNODES])
{
//  This subprogram takes an adjacency-matrix representation of a graph
//      in which the non-zero entries are lower bounds on flows and
//      copies it to a matrix in which non-adjacent nodes get a flow
//      of "-1" and adjacent nodes are initialised to zero flow.
//     This is our "working matrix" for adding and subtracting flows.
//
//   Accepts: lowerBound - matrix representation of the graph:
//            lowerBound[x][y] = k, the minimum desired flow from mode x to y;
//                             = 0, if there is no transition from node x to y;
//            nNodes    - number of nodes (vertices) in the graph
//   Returns: flow - a working copy of the matrix in which
//            flow[x][y] = 0,  if there is a transition from node x to y;
//                       = -1, if there is no transition from x to y.
//   Last update: 2000-03-05 15:28
//---------------------------------

        int lastNode, low;
```

```
//      cout << endl << endl << "In copylb2flow()";
        lastNode = nNodes -1;
        for (int r=0; r <= lastNode; r++) {
          for (int c=0; c<= lastNode; c++) {
            low = lowerBound[r][c];
            if (low <= 0)
              flow[r][c] = -1;
            else
              flow[r][c] = 0;
          }   //endfor
        }   //endfor
//      cout << endl << "Out copylb2flow()";
        return;
}
//---------------------------------------------------------------
void printMatrix(int myMatrix[MAXNODES][MAXNODES], int& nNodes)
{
//  This subprogram prints an adjacency matrix.
//   Accepts: nNodes   - number of nodes (vertices) in the graph
//            myMatrix - matrix representation of the graph:
//            myMatrix[x][y] = -1, if there is no transition from node x to y;
//                           or = f, the flow on the transition from x to y.
//   Last update: 2000-02-15 21:24
//--------------------------------

//  Print each row of the adjacency matrix
        cout << endl;
        for (int i=0; i<=nNodes-1; i++) {
          cout << "         ";
          for (int j=0; j<=nNodes-1; j++) {
            cout << myMatrix[i][j] << "   ";
          }
          cout << endl;
        }
        return;
}
//---------------------------------------------------------------
int readMatrix(int lb[MAXNODES][MAXNODES], int& nNodes,
               int offspring[MAXNODES], char graphName[MAXNAME])
{
//  This subprogram reads in an adjacency matrix representing
//   a graph with nodes and edges.
//   Global:  file5    - the "handle" of the file being used for input
//   Returns: nNodes   - number of nodes (vertices) in the graph
//            lb       - matrix representation of the graph:
//            lb[x][y] = f, a positive integer representing the min flow
//                       ("lower bound") from node x to y;
//                     = 0, otherwise.
//            offspring[j] = 1, if node j has any subnodes
//                         = 0, otherwise (that is, j is a "leaf node")
//          graphName - name of the graph
//   Last update:  2000-03-06 21:45
//--------------------------------

//      cout << endl << "In readMatrix()";
//      cout << "\t\tPress any key to continue ..."; getch();

//  Read the Comment line or graph name from the input file
        file5 >> graphName;
//      file5.getline(graphName, sizeof(graphName)-1);   <-- cannot do this
        cout << endl << "Graph name: " << graphName << endl;

//  Read in the number of nodes in this graph
        file5 >> nNodes;
        // The extraction operator (<<) reads a number from file
        // and assigns it to nNodes, doing the implied type-conversion
```

```
        cout << "Number of nodes = " << nNodes << endl;

//  Read and print each row of the adjacency matrix
        cout << "Lower Bounds matrix: (minimum flows)" << endl;
        for (int i=0; i<=nNodes-1; i++) {
          cout << "  Row " << i << " is ";
          for (int j=0; j<=nNodes-1; j++) {
            file5 >> lb[i][j];
            cout << lb[i][j] << "  ";
          }
          cout << endl;
        }
//  Read the offspring vector -- whether each node has any subnodes
        cout << "Offspring vector: ";
        for (int i=0; i<=nNodes-1; i++) {
          file5 >> offspring[i];
          cout << offspring[i] << "  ";
        }
        cout << "\t\t\tPress any key to continue ...";
        getch();
        return 1;
}
```

APPENDIX 7

Execution of Approach on HFSM Test Cases

This appendix provides printed listings of the input data files and gives corresponding printouts created from the computer program listed in Appendix 6.

HEXAGON

Characteristics: No hierarchy. Flat graph only. 6 nodes. Shows basic algorithm working.

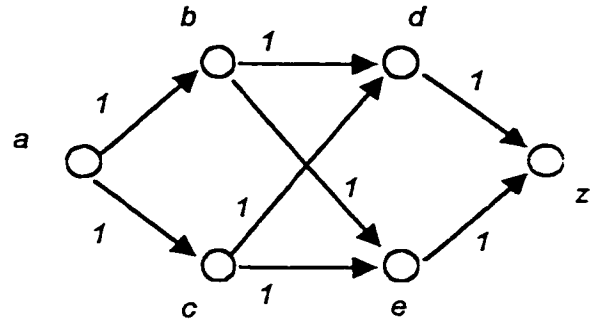


Fig. A7-1. *HEXAGON. Expected MAXCUT = 4.*

Hexagon. Run #1: Unit flows.

```
Input:  hexagon1.txt

This run is for HEXAGON1 (Fig. A1-6, 6 main nodes and unit flows only)
HEXAGON1
6
0 1 1 0 0 0
0 0 0 1 1 0
0 0 0 1 1 0
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0
0 0 0 0 0 0
```

```
Output:

astro2d.exe
Please enter input-filename: hexagon1.txt
Comment: This run is for HEXAGON1 (Fig. A1-6, 6 main nodes and unit flows only)

Graph name: HEXAGON1
Number of nodes = 6
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1  0  0  0
  Row 1 is 0  0  0  1  1  0
  Row 2 is 0  0  0  1  1  0
  Row 3 is 0  0  0  0  0  1
  Row 4 is 0  0  0  0  0  1
  Row 5 is 0  0  0  0  0  0
Offspring vector: 0  0  0  0  0  0              Press any key to continue ...

In traverse()                    Press any key to continue ...
  The corresponding Flow matrix (working copy):
        -1  0  0  -1  -1  -1
        -1  -1  -1  0  0  -1          Notation: -1 = no edge present (and therefore no flow)
        -1  -1  -1  0  0  -1                     0 = edge present, but no flow on it
```

```
        -1  -1  -1  -1  -1  0
        -1  -1  -1  -1  -1  0
        -1  -1  -1  -1  -1  -1

 Analysis of Lower Bounds matrix above:
 Analysing node: 0                Press any key to continue ...
 capacity of node 0 is 1

 Analysing node: 1                Press any key to continue ...
 capacity of node 1 is 1

 Analysing node: 2                Press any key to continue ...
 capacity of node 2 is 1

 Analysing node: 3                Press any key to continue ...
 capacity of node 3 is 1

 Analysing node: 4                Press any key to continue ...
 capacity of node 4 is 1

 Analysing node: 5                Press any key to continue ...
 capacity of node 5 is 1

 Conclusion: source node = 0
 Conclusion:   sink node = 5

 Updated Lower Bounds matrix: HEXAGON1
        0  1  1  0  0  0
        0  0  0  1  1  0
        0  0  0  1  1  0
        0  0  0  0  0  1
        0  0  0  0  0  1
        0  0  0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current edge
  There are successors
  Adding 1 unit(s) of flow to successor edge = (1, 3)  Adding 1 unit(s) of flow
to successor edge = (3, 5)
        -1  1  0  -1  -1  -1
        -1  -1  -1  1  0  -1
        -1  -1  -1  0  0  -1
        -1  -1  -1  -1  -1  1
        -1  -1  -1  -1  -1  0
        -1  -1  -1  -1  -1  -1
                  Press any key to continue..

----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current edge
  There are successors
  Adding 1 unit(s) of flow to successor edge = (2, 3)  Adding 1 unit(s) of flow
to successor edge = (3, 5)
        -1  1  1  -1  -1  -1
        -1  -1  -1  1  0  -1
        -1  -1  -1  1  0  -1
        -1  -1  -1  -1  -1  2
        -1  -1  -1  -1  -1  0
        -1  -1  -1  -1  -1  -1
                  Press any key to continue..

----------------------
Current edge = (1, 4)
  Adding 1 unit(s) of flow to current edge
```

```
  There are predecessors
  Adding 1 unit(s) of flow to predecessor edge = (0, 1)          Press any key
to continue ...
  There are successors
  Adding 1 unit(s) of flow to successor edge = (4, 5)
         -1  2  2  -1  -1  -1
         -1  -1  -1  1  1  -1
         -1  -1  -1  1  0  -1
         -1  -1  -1  -1  -1  2
         -1  -1  -1  -1  -1  1
         -1  -1  -1  -1  -1  -1
                       Press any key to continue..

----------------------
Current edge = (2, 4)
  Adding 1 unit(s) of flow to current edge
  There are predecessors
  Adding 1 unit(s) of flow to predecessor edge = (0, 2)              Press any
key to continue ...
  There are successors
  Adding 1 unit(s) of flow to successor edge = (4, 5)
         -1  2  2  -1  -1  -1
         -1  -1  -1  1  1  -1
         -1  -1  -1  1  1  -1
         -1  -1  -1  -1  -1  2
         -1  -1  -1  -1  -1  2
         -1  -1  -1  -1  -1  -1
                       Press any key to continue..
In decrFlow()
Current node = 5
  Potential surplus on Edge(3, 5) = 1
  Potential surplus on Edge(4, 5) = 1
                       Press any key to continue..
Current node = 3
                       Press any key to continue..
Current node = 4
                       Press any key to continue..

No further surplus flows to remove.
source node: sum = 2     delta = -1   HEXAGON1
  sink node: sum = 2     delta = -1
*** minFlow of HEXAGON1 is 4      (= sum of flows into sink)

Final flow matrix for HEXAGON1
         -1  2  2  -1  -1  -1
         -1  -1  -1  1  1  -1
         -1  -1  -1  1  1  -1
         -1  -1  -1  -1  -1  2
         -1  -1  -1  -1  -1  2
         -1  -1  -1  -1  -1  -1
Out traverse()

In main()
MAXCUT OVERALL = 4

Normal termination.
Press any key to continue ...
```

Hexagon. Run #2: Flows (>1) are assigned to each edge.

(From Fig. A1-1.)

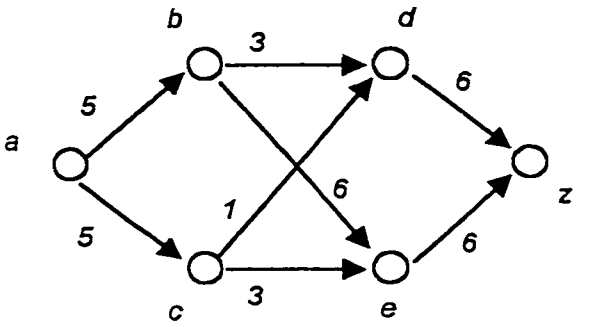


Fig. A7-2. *HEXAGON. Expected MAXCUT = 15.*

```
Input:  hexagon-h.txt

This run is for HEXAGON-H (Fig. A1-1, 6 main nodes and heavy flows)
HEXAGON-H
6
0 5 5 0 0 0
0 0 0 3 6 0
0 0 0 1 3 0
0 0 0 0 0 6
0 0 0 0 0 6
0 0 0 0 0 0
0 0 0 0 0 0


Output:

astro2d.exe
Please enter input-filename: hexagon-h.txt
Comment: This run is for HEXAGON-H (Fig. A1-1, 6 main nodes and heavy flows)

Graph name: HEXAGON-H
Number of nodes = 6
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  5  5  0  0  0
  Row 1 is 0  0  0  3  6  0
  Row 2 is 0  0  0  1  3  0
  Row 3 is 0  0  0  0  0  6
  Row 4 is 0  0  0  0  0  6
  Row 5 is 0  0  0  0  0  0
Offspring vector: 0  0  0  0  0  0          Press any key to continue ...

In traverse()                    Press any key to continue ...
  The corresponding Flow matrix (working copy):
        -1  0  0  -1  -1  -1
        -1  -1  -1  0  0  -1
        -1  -1  -1  0  0  -1
        -1  -1  -1  -1  -1  0
        -1  -1  -1  -1  -1  0
        -1  -1  -1  -1  -1  -1

  Analysis of Lower Bounds matrix above:
  Analysing node: 0                Press any key to continue ...
  capacity of node 0 is 1
```

```
 Analysing node: 1                Press any key to continue ...
 capacity of node 1 is 1

 Analysing node: 2                Press any key to continue ...
 capacity of node 2 is 1

 Analysing node: 3                Press any key to continue ...
 capacity of node 3 is 1

 Analysing node: 4                Press any key to continue ...
 capacity of node 4 is 1

 Analysing node: 5                Press any key to continue ...
 capacity of node 5 is 1

 Conclusion: source node = 0
 Conclusion:   sink node = 5

 Updated Lower Bounds matrix: HEXAGON-H
        0  5  5  0  0  0
        0  0  0  3  6  0
        0  0  0  1  3  0
        0  0  0  0  0  6
        0  0  0  0  0  6
        0  0  0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 5 unit(s) of flow to current edge
  There are successors
  Adding 5 unit(s) of flow to successor edge = (1, 3)  Adding 5 unit(s) of flow
to successor edge = (3, 5)
         -1  5  0  -1  -1  -1
         -1  -1  -1  5  0  -1
         -1  -1  -1  0  0  -1
         -1  -1  -1  -1  -1  5
         -1  -1  -1  -1  -1  0
         -1  -1  -1  -1  -1  -1
                    Press any key to continue..

----------------------
Current edge = (0, 2)
  Adding 5 unit(s) of flow to current edge
  There are successors
  Adding 5 unit(s) of flow to successor edge = (2, 3)  Adding 5 unit(s) of flow
to successor edge = (3, 5)
         -1  5  5  -1  -1  -1
         -1  -1  -1  5  0  -1
         -1  -1  -1  5  0  -1
         -1  -1  -1  -1  -1  10
         -1  -1  -1  -1  -1  0
         -1  -1  -1  -1  -1  -1
                    Press any key to continue..

----------------------
Current edge = (1, 4)
  Adding 6 unit(s) of flow to current edge
  There are predecessors
  Adding 6 unit(s) of flow to predecessor edge = (0, 1)              Press any
key to continue ...
  There are successors
  Adding 6 unit(s) of flow to successor edge = (4, 5)
         -1  11  5  -1  -1  -1
         -1  -1  -1  5  6  -1
```

```
        -1  -1  -1  5  0  -1
        -1  -1  -1  -1  -1  10
        -1  -1  -1  -1  -1  6
        -1  -1  -1  -1  -1  -1
                    Press any key to continue..

----------------------
Current edge = (2, 4)
  Adding 3 unit(s) of flow to current edge
  There are predecessors
  Adding 3 unit(s) of flow to predecessor edge = (0, 2)              Press any
key to continue ...
  There are successors
  Adding 3 unit(s) of flow to successor edge = (4, 5)
        -1  11  8  -1  -1  -1
        -1  -1  -1  5  6  -1
        -1  -1  -1  5  3  -1
        -1  -1  -1  -1  -1  10
        -1  -1  -1  -1  -1  9
        -1  -1  -1  -1  -1  -1
                    Press any key to continue..
In decrFlow()
Current node = 5
  Potential surplus on Edge(3, 5) = 4
  Potential surplus on Edge(4, 5) = 3
                    Press any key to continue..
Current node = 3
  Potential surplus on Edge(1, 3) = 2
  Potential surplus on Edge(2, 3) = 4
                    Press any key to continue..
Current node = 1
  Potential surplus on Edge(0, 1) = 6
                    Press any key to continue..
Removing flow
  Flow(0, 1) is decr by 2
  Flow(1, 3) is decr by 2
  Flow(3, 5) is decr by 2

        -1  9  8  -1  -1  -1
        -1  -1  -1  3  6  -1
        -1  -1  -1  5  3  -1
        -1  -1  -1  -1  -1  8
        -1  -1  -1  -1  -1  9
        -1  -1  -1  -1  -1  -1

--------------------
Re-start at sink node
Current node = 5
  Potential surplus on Edge(3, 5) = 2
  Potential surplus on Edge(4, 5) = 3
                    Press any key to continue..
Current node = 3
  Potential surplus on Edge(2, 3) = 4
                    Press any key to continue..
Current node = 2
  Potential surplus on Edge(0, 2) = 3
                    Press any key to continue..
Removing flow
  Flow(0, 2) is decr by 2
  Flow(2, 3) is decr by 2
  Flow(3, 5) is decr by 2

        -1  9  6  -1  -1  -1
        -1  -1  -1  3  6  -1
        -1  -1  -1  3  3  -1
        -1  -1  -1  -1  -1  6
        -1  -1  -1  -1  -1  9
```

```
         -1  -1  -1  -1  -1  -1

---------------------
Re-start at sink node
Current node = 5
  Potential surplus on Edge(4, 5) = 3
                       Press any key to continue..
Current node = 4
                       Press any key to continue..

No further surplus flows to remove.
source node: sum = 10     delta = -9   HEXAGON-H
  sink node: sum = 12     delta = -11
*** minFlow of HEXAGON-H is 15      (= sum of flows into sink)

Final flow matrix for HEXAGON-H
         -1  9  6  -1  -1  -1
         -1  -1  -1  3  6  -1
         -1  -1  -1  3  3  -1
         -1  -1  -1  -1  -1  6
         -1  -1  -1  -1  -1  9
         -1  -1  -1  -1  -1  -1
Out traverse()

In main()
MAXCUT OVERALL = 15

Normal termination.
Press any key to continue ...
```

LIBRA

Characteristics: Has hierarchy. 4 nodes. 1 child node.

Libra Run #1. Child node is hanging off the *sink* node. Child node has a doubled edge on transition (0, 1). This run illustrates that the MAXCUT of the child (2 + 1 = 3) is not simply added to the MAXCUT of the parent (= 2) which would = 5.

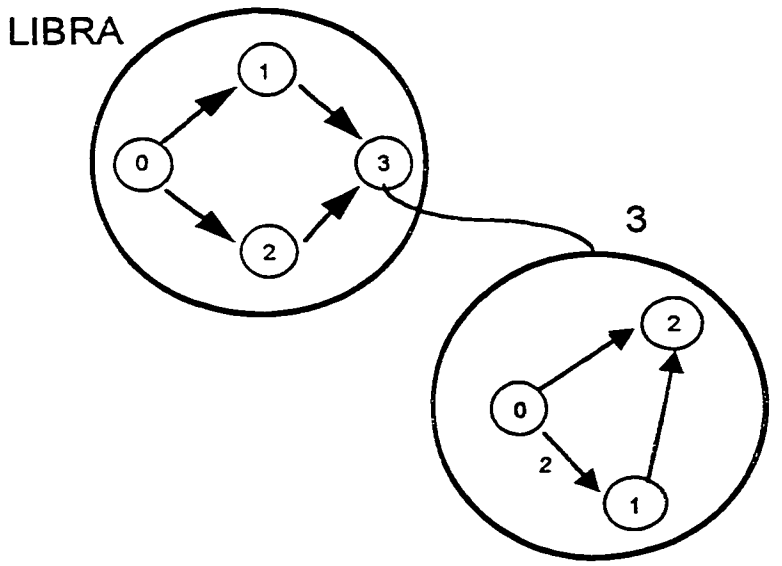


Fig. A7-3. *LIBRA. Expected MAXCUT = 3.*

```
Input: libra2snk.txt

This run is for LIBRA2-SNK (a graph with 4 main nodes and one child node)
LIBRA
4
0 1 0 1
0 0 1 0
0 0 0 1
0 0 0 0
0 0 0 1
LIBRA-ALPHA
3
0 2 1
0 0 1
0 0 0
0 0 0


Output:

astro2d.exe
Please enter input-filename: libra2snk.txt
Comment: This run is for LIBRA2-SNK (a graph with 4 main nodes and one child
node)
```

```
Graph name: LIBRA
Number of nodes = 4
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  0  1
  Row 1 is 0  0  1  0
  Row 2 is 0  0  0  1
  Row 3 is 0  0  0  0
Offspring vector: 0  0  0  1          Press any key to continue ...

In traverse()                    Press any key to continue ...
  The corresponding Flow matrix (working copy):
        -1  0  -1  0
        -1  -1  0  -1
        -1  -1  -1  0
        -1  -1  -1  -1

  Analysis of Lower Bounds matrix above:
  Analysing node: 0                Press any key to continue ...
  capacity of node 0 is 1

  Analysing node: 1                Press any key to continue ...
  capacity of node 1 is 1

  Analysing node: 2                Press any key to continue ...
  capacity of node 2 is 1

  Analysing node: 3                Press any key to continue ...
  offspring[3] = YES (Read next subgraph:)

Graph name: LIBRA-ALPHA
Number of nodes = 3
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  2  1
  Row 1 is 0  0  1
  Row 2 is 0  0  0
Offspring vector: 0  0  0             Press any key to continue ...

In traverse()                    Press any key to continue ...
  The corresponding Flow matrix (working copy):
        -1  0  0
        -1  -1  0
        -1  -1  -1

  Analysis of Lower Bounds matrix above:
  Analysing node: 0                Press any key to continue ...
  capacity of node 0 is 1

  Analysing node: 1                Press any key to continue ...
  capacity of node 1 is 1

  Analysing node: 2                Press any key to continue ...
  capacity of node 2 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 2

  Updated Lower Bounds matrix: LIBRA-ALPHA
        0  2  1
        0  0  1
        0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 2 unit(s) of flow to current edge
  There are successors
```

```
  Adding 2 unit(s) of flow to successor edge = (1, 2)
         -1   2   0
         -1   -1   2
         -1   -1   -1
                        Press any key to continue..

----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current edge
         -1   2   1
         -1   -1   2
         -1   -1   -1
                        Press any key to continue..
In decrFlow()
Current node = 2
  Potential surplus on Edge(1, 2) = 1
                        Press any key to continue..
Current node = 1
                        Press any key to continue..

No further surplus flows to remove.
source node: sum = 3      delta = -2    LIBRA-ALPHA
  sink node: sum = 2      delta = -1
*** minFlow of LIBRA-ALPHA is 3       (= sum of flows into sink)

Final flow matrix for LIBRA-ALPHA
          -1   2   1
          -1   -1   2
          -1   -1   -1
Out traverse()

  capacity of node 3 is 3

  Conclusion: source node = 0
  Conclusion:   sink node = 3

  Updated Lower Bounds matrix: LIBRA
          0   1   0   1
          0   0   1   0
          0   0   0   1
          0   0   0   0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current edge
  There are successors
  Adding 1 unit(s) of flow to successor edge = (1, 2)  Adding 1 unit(s) of flow
to successor edge = (2, 3)
          -1   1   -1   0
          -1   -1   1   -1
          -1   -1   -1   1
          -1   -1   -1   -1
                         Press any key to continue..

----------------------
Current edge = (0, 3)
  Adding 1 unit(s) of flow to current edge
          -1   1   -1   1
          -1   -1   1   -1
          -1   -1   -1   1
          -1   -1   -1   -1
                         Press any key to continue..
In decrFlow()
Current node = 3
                         Press any key to continue..
```

```
No further surplus flows to remove.
source node: sum = 2      delta = -1   LIBRA
  sink node: sum = 2      delta = 1
In fillEdges()

----------------------
Current edge = (0, 3)
  Adding 1 unit(s) of flow to current edge
         -1  1  -1  2
         -1  -1  1  -1
         -1  -1  -1  1
         -1  -1  -1  -1
                      Press any key to continue..
In decrFlow()
Current node = 3
                      Press any key to continue..

No further surplus flows to remove.
*** minFlow of LIBRA is 3       (= sum of flows into sink

Final flow matrix for LIBRA
         -1  1  -1  2
         -1  -1  1  -1
         -1  -1  -1  1
         -1  -1  -1  -1
Out traverse()

In main()
MAXCUT OVERALL = 3

Normal termination.
Press any key to continue ...
```

Libra Run #2. Child node is hanging off the *source* node.

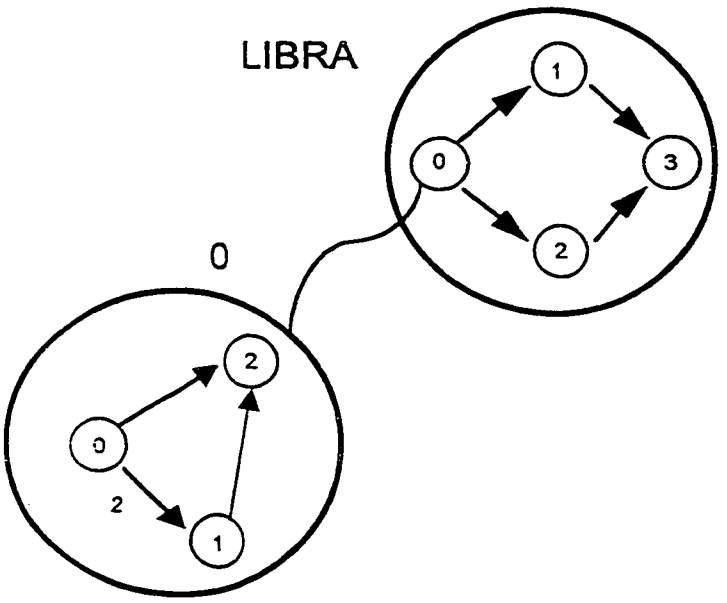


Fig. A7-4. *LIBRA. Expected MAXCUT = 3.*

```
Input:  libra2src.txt

This run is for LIBRA2-SRC (a graph with 4 main nodes and one child node)
LIBRA
4
0 1 0 1
0 0 1 0
0 0 0 1
0 0 0 0
1 0 0 0
LIBRA-ALPHA
3
0 2 1
0 0 1
0 0 0
0 0 0


Output:

astro2d.exe
Please enter input-filename: libra2src.txt
Comment: This run is for LIBRA2 (a graph with 4 main nodes and one child node)

Graph name: LIBRA
Number of nodes = 4
Lower Bounds matrix: (minimum flows)
  Row 0 is 0   1   0   1
  Row 1 is 0   0   1   0
  Row 2 is 0   0   0   1
  Row 3 is 0   0   0   0
Offspring vector: 1   0   0   0           Press any key to continue ...

In traverse()                    Press any key to continue ...
  The corresponding Flow matrix (working copy):
        -1   0   -1   0
```

```
        -1  -1  0  -1
        -1  -1  -1  0
        -1  -1  -1  -1

  Analysis of Lower Bounds matrix above:
  Analysing node: 0                Press any key to continue ...
  offspring[0] = YES (Read next subgraph:)

Graph name: LIBRA-ALPHA
Number of nodes = 3
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  2  1
  Row 1 is 0  0  1
  Row 2 is 0  0  0
Offspring vector: 0  0  0              Press any key to continue ...

In traverse()                    Press any key to continue ...
  The corresponding Flow matrix (working copy):
        -1  0  0
        -1  -1  0
        -1  -1  -1

  Analysis of Lower Bounds matrix above:
  Analysing node: 0                Press any key to continue ...
  capacity of node 0 is 1

  Analysing node: 1                Press any key to continue ...
  capacity of node 1 is 1

  Analysing node: 2                Press any key to continue ...
  capacity of node 2 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 2

  Updated Lower Bounds matrix: LIBRA-ALPHA
        0  2  1
        0  0  1
        0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 2 unit(s) of flow to current edge
  There are successors
  Adding 2 unit(s) of flow to successor edge = (1, 2)
        -1  2  0
        -1  -1  2
        -1  -1  -1
                        Press any key to continue..

----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current edge
        -1  2  1
        -1  -1  2
        -1  -1  -1
                        Press any key to continue..
In decrFlow()
Current node = 2
  Potential surplus on Edge(1, 2) = 1
                        Press any key to continue..
Current node = 1
                        Press any key to continue..

No further surplus flows to remove.
```

```
source node: sum = 3      delta = -2    LIBRA-ALPHA
  sink node: sum = 2      delta = -1
*** minFlow of LIBRA-ALPHA is 3        (= sum of flows into sink)

Final flow matrix for LIBRA-ALPHA
         -1  2  1
         -1  -1  2
         -1  -1  -1
Out traverse()

  capacity of node 0 is 3

  Analysing node: 1                Press any key to continue ...
  capacity of node 1 is 1

  Analysing node: 2                Press any key to continue ...
  capacity of node 2 is 1

  Analysing node: 3                Press any key to continue ...
  capacity of node 3 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 3

  Updated Lower Bounds matrix: LIBRA
         0  1  0  1
         0  0  1  0
         0  0  0  1
         0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current edge
  There are successors
  Adding 1 unit(s) of flow to successor edge = (1, 2)  Adding 1 unit(s) of flow
to successor edge = (2, 3)
         -1  1  -1  0
         -1  -1  1  -1
         -1  -1  -1  1
         -1  -1  -1  -1
                    Press any key to continue..

----------------------
Current edge = (0, 3)
  Adding 1 unit(s) of flow to current edge
         -1  1  -1  1
         -1  -1  1  -1
         -1  -1  -1  1
         -1  -1  -1  -1
                    Press any key to continue..
In decrFlow()
Current node = 3
                    Press any key to continue..

No further surplus flows to remove.
source node: sum = 2      delta = 1    LIBRA
In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current edge
  There are successors
  Adding 1 unit(s) of flow to successor edge = (1, 2)  Adding 1 unit(s) of flow
to successor edge = (2, 3)
         -1  2  -1  1
```

```
         -1  -1  2  -1
         -1  -1  -1  2
         -1  -1  -1  -1
                        Press any key to continue..
In decrFlow()
Current node = 3
  Potential surplus on Edge(2, 3) = 1
                        Press any key to continue..
Current node = 2
  Potential surplus on Edge(1, 2) = 1
                        Press any key to continue..
Current node = 1
                        Press any key to continue..

No further surplus flows to remove.
  sink node: sum = 2     delta = -1
*** minFlow of LIBRA is 3       (= sum of flows into sink)

Final flow matrix for LIBRA
         -1  2  -1  1
         -1  -1  2  -1
         -1  -1  -1  2
         -1  -1  -1  -1
Out traverse()

In main()
MAXCUT OVERALL = 3

Normal termination.
Press any key to continue ...
```

TRIANGULUM

Characteristics: Has 3 levels of hierarchy.
There are 3 main nodes, 2 children, and 2 grandchildren

Triangulum Run #1: Subnodes hang off a sink node and a middle node.
Includes special cases of having only 2 nodes. One such case has a doubled edge (which is equivalent to a single edge of minimum capacity 2)

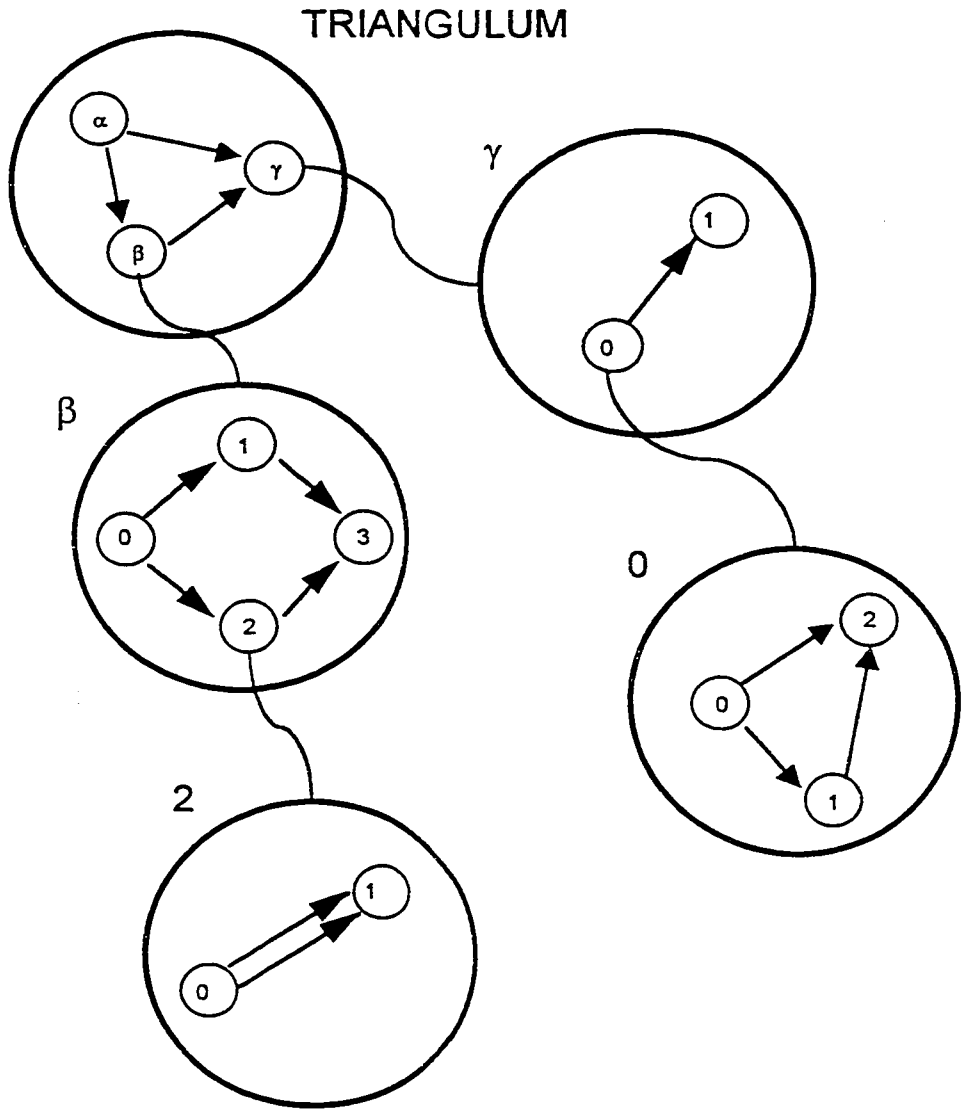


Fig. A7-5. *TRIANGULUM. Expected MAXCUT = 4.*

Input: triangulum5.txt

```
This run is TRIANGULUM5 (3 main
nodes,
 2 children, 2 grandchildren)
TRIANGULUM
3
0 1 1
0 0 1
0 0 0
0 1 1
TRIANGULUM:BETA
4
0 1 1 0
0 0 0 1
0 0 0 1
0 0 0 0
0 0 1 0
TRIANGULUM:BETA-2
2
0 2
0 0
0 0
TRIANGULUM:GAMMA
2
0 1
0 0
1 0
TRIANGULUM:GAMMA-0
3
0 1 2
0 0 1
0 0 0
0 0 0
```

Output:

```
astro2d.exe
Please enter input-filename:
triangulum5.txt
Comment: This run is TRIANGULUM5 (3
main nodes, 2 children, 2
grandchildren)

Graph name: TRIANGULUM
Number of nodes = 3
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1
  Row 1 is 0  0  1
  Row 2 is 0  0  0
Offspring vector: 0  1  1

In traverse()
  Corresponding Flow matrix (working
copy):
         -1  0  0
         -1  -1  0
         -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  offspring[1] = YES (Read next
subgraph:)

Graph name: TRIANGULUM:BETA
Number of nodes = 4
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1  0
  Row 1 is 0  0  0  1
  Row 2 is 0  0  0  1
  Row 3 is 0  0  0  0
Offspring vector: 0  0  1  0

In traverse()
  Corresponding Flow matrix (working
copy):
         -1  0  0  -1
         -1  -1  -1  0
         -1  -1  -1  0
         -1  -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  offspring[2] = YES (Read next
subgraph:)

Graph name: TRIANGULUM:BETA-2
Number of nodes = 2
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  2
  Row 1 is 0  0
Offspring vector: 0  0

In traverse()
  Corresponding Flow matrix (working
copy):
         -1  0
         -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 1

  traverse()- In Step 4: updating
lower bounds
  Updated Lower Bounds matrix:
TRIANGULUM:BETA-2
         0  2
         0  0

In fillEdges()

----------------------
Current edge = (0, 1)
```

```
  Adding 2 unit(s) of flow to current
edge
         -1  2
         -1  -1

In decrFlow()
Current node = 1


No further surplus flows to remove.
source node: sum = 2      delta = -1
TRIANGULUM:BETA-2
  sink node: sum = 2      delta = -1
In fillEdges()
In decrFlow()
Current node = 1


No further surplus flows to remove.
*** minFlow of TRIANGULUM:BETA-2 = 2
                (= sum of flows into
sink)

Final flow matrix: TRIANGULUM:BETA-2
         -1  2
         -1  -1
Out traverse()

  capacity of node 2 is 2

  Analysing node: 3
  capacity of node 3 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 3

  traverse()- In Step 4: updating
lower bounds
  Updated Lower Bounds matrix:
TRIANGULUM:BETA
         0  1  2  0
         0  0  0  1
         0  0  0  2
         0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 3)
         -1  1  0  -1
         -1  -1  -1  1
         -1  -1  -1  0
         -1  -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 2 unit(s) of flow to current
edge
  There are successors
  Adding 2 unit(s) of flow to
successor edge = (2, 3)
         -1  1  2  -1
         -1  -1  -1  1
         -1  -1  -1  2
         -1  -1  -1  -1

In decrFlow()
Current node = 3


No further surplus flows to remove.
source node: sum = 3      delta = -2
TRIANGULUM:BETA
  sink node: sum = 3      delta = -2
In fillEdges()
In decrFlow()
Current node = 3


No further surplus flows to remove.
*** minFlow of TRIANGULUM:BETA = 3
                (= sum of flows into
sink)

Final flow matrix: TRIANGULUM:BETA
         -1  1  2  -1
         -1  -1  -1  1
         -1  -1  -1  2
         -1  -1  -1  -1
Out traverse()

  capacity of node 1 is 3

  Analysing node: 2
  offspring[2] = YES (Read next
subgraph:)

Graph name: TRIANGULUM:GAMMA
Number of nodes = 2
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1
  Row 1 is 0  0
Offspring vector: 1  0

In traverse()
  Corresponding Flow matrix (working
copy):
         -1  0
         -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  offspring[0] = YES (Read next
subgraph:)

Graph name: TRIANGULUM:GAMMA-0
Number of nodes = 3
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  2
  Row 1 is 0  0  1
  Row 2 is 0  0  0
Offspring vector: 0  0  0

In traverse()
  Corresponding Flow matrix (working
copy):
         -1  0  0
```

```
         -1  -1  0
         -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  capacity of node 2 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 2

  traverse()- In Step 4: updating
lower bounds
  Updated Lower Bounds matrix:
TRIANGULUM:GAMMA-0
         0  1  2
         0  0  1
         0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 2)
         -1  1  0
         -1  -1  1
         -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 2 unit(s) of flow to current
edge
         -1  1  2
         -1  -1  1
         -1  -1  -1

In decrFlow()
Current node = 2


No further surplus flows to remove.
source node: sum = 3     delta = -2
TRIANGULUM:GAMMA-0
  sink node: sum = 3     delta = -2
In fillEdges()
In decrFlow()
Current node = 2


No further surplus flows to remove.
*** minFlow of TRIANGULUM:GAMMA-0 = 3
              (= sum of flows into
sink)

Final flow matrix: TRIANGULUM:GAMMA-0
         -1  1  2
         -1  -1  1
         -1  -1  -1
Out traverse()

  capacity of node 0 is 3

  Analysing node: 1
  capacity of node 1 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 1

  traverse()- In Step 4: updating
lower bounds
  Updated Lower Bounds matrix:
TRIANGULUM:GAMMA
         0  1
         0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
         -1  1
         -1  -1

In decrFlow()
Current node = 1


No further surplus flows to remove.
source node: sum = 1     delta = 2
TRIANGULUM:GAMMA
  sink node: sum = 3     delta = -2
In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 2 unit(s) of flow to current
edge
         -1  3
         -1  -1

In decrFlow()
Current node = 1


No further surplus flows to remove.
*** minFlow of TRIANGULUM:GAMMA = 3
              (= sum of flows into
sink)

Final flow matrix: TRIANGULUM:GAMMA
         -1  3
         -1  -1
Out traverse()

  capacity of node 2 is 3

  Conclusion: source node = 0
  Conclusion:   sink node = 2

  traverse()- In Step 4: updating
lower bounds
```

```
  Updated Lower Bounds matrix:
TRIANGULUM
          0   3   1
          0   0   3
          0   0   0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 3 unit(s) of flow to current
edge
  There are successors
  Adding 3 unit(s) of flow to
successor edge = (1, 2)
          -1   3   0
          -1   -1   3
          -1   -1   -1


----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current
edge
          -1   3   1
          -1   -1   3
          -1   -1   -1

In decrFlow()
Current node = 2


No further surplus flows to remove.
source node: sum = 4      delta = -3
TRIANGULUM
  sink node: sum = 4      delta = -1
In fillEdges()
In decrFlow()
Current node = 2


No further surplus flows to remove.
*** minFlow of TRIANGULUM = 4
                (= sum of flows into
sink)

Final flow matrix: TRIANGULUM
          -1   3   1
          -1   -1   3
          -1   -1   -1
Out traverse()

In main()
MAXCUT OVERALL = 4

Normal termination.
```

"MAIN EXAMPLE"

Characteristics: Has 3 levels of hierarchy. Node 1 has been split into a starting portion and a finishing portion (1 and 1'), and transition *t7* which originally formed a cycle has been re-routed to the finishing portion of Node 1 in order to break the cycle. (Refer Fig. A5-1.)

There are now 6 nodes in main, 5 children, and 2 grandchildren

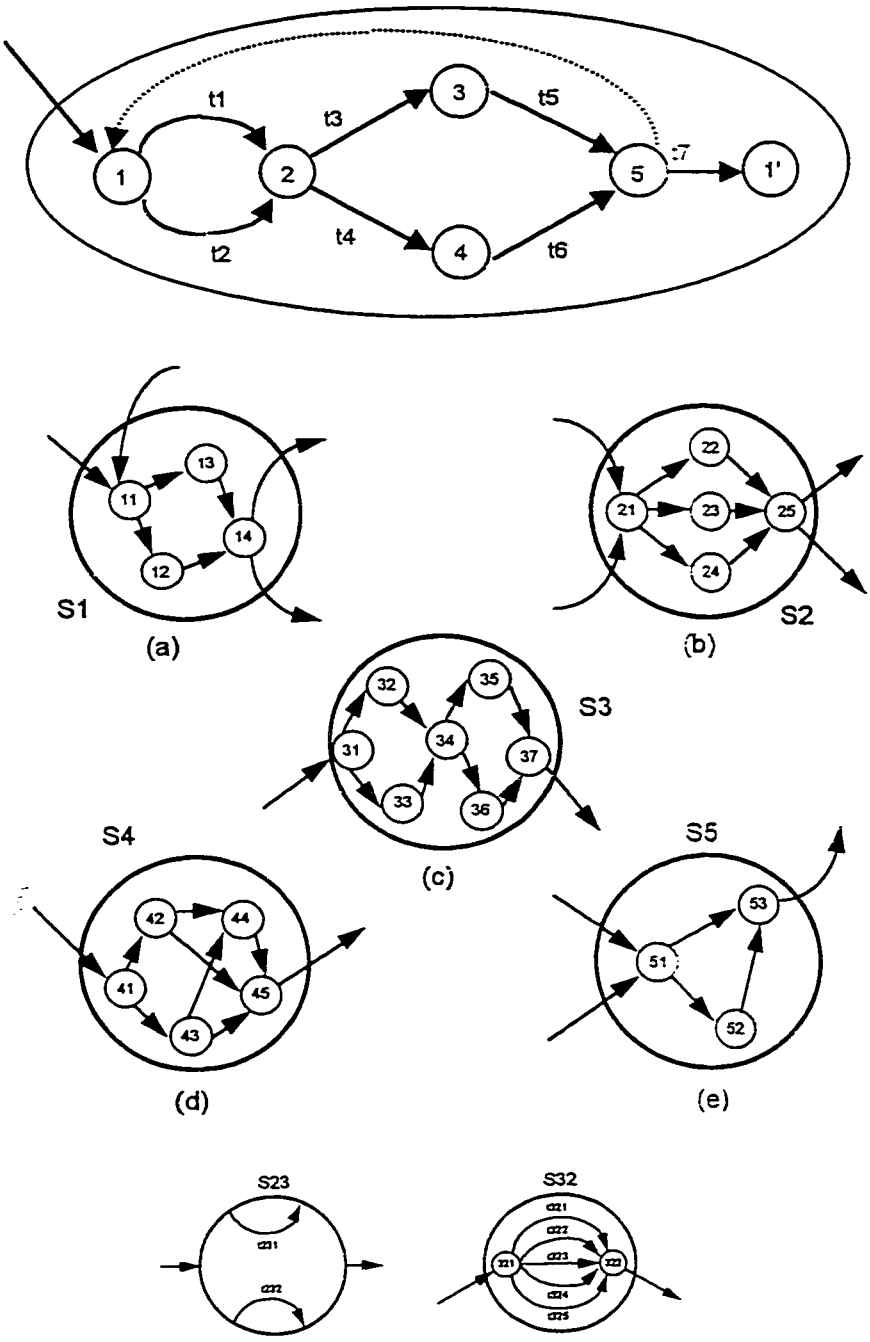


Fig. A7-6. *"MAIN EXAMPLE." Expected MAXCUT = 10. (as per Appendix 5)*

Main Example - Run #1: Because transitions *t1* and *t2* start from the same node and terminate at the same node, they have been represented in the datafile as a single transition, but with minimum capacity of 2.

```
Input:  main-example.txt

This run is for MAIN EXAMPLE (Fig.
A5-1, 5+1=6 main nodes)
MAIN-EXAMPLE
6
0 2 0 0 0 0
0 0 1 1 0 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 0 2
0 0 0 0 0 0
1 1 1 1 1 1
S1
4
0 1 1 0
0 0 0 1
0 0 0 1
0 0 0 0
0 0 0 0
S2
5
0 1 1 1 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0
0 0 1 0 0
S23
2
0 2
0 0
0 0
S3
7
0 1 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 0 0 0 0 0 0
0 1 0 0 0 0 0
S32
2
0 5
0 0
0 0
S4
5
0 1 1 0 0
0 0 0 1 1
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
0 0 0 0 0
S5
3
0 1 1
0 0 1
0 0 0
0 0 0

Output:

astro2d.exe
Please enter input-filename:
Comment: This run is for MAIN EXAMPLE
(Fig. A5-1, 5+1=6 main nodes)

Graph name: MAIN-EXAMPLE
Number of nodes = 6
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  2  0  0  0  0
  Row 1 is 0  0  1  1  0  0
  Row 2 is 0  0  0  0  1  0
  Row 3 is 0  0  0  0  1  0
  Row 4 is 0  0  0  0  0  2
  Row 5 is 0  0  0  0  0  0
Offspring vector: 1  1  1  1  1  1

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  -1  -1  -1  -1
         -1  -1  0  0  -1  -1
         -1  -1  -1  -1  0  -1
         -1  -1  -1  -1  0  -1
         -1  -1  -1  -1  -1  0
         -1  -1  -1  -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  offspring[0] = YES (Read next
subgraph:)

Graph name: S1
Number of nodes = 4
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1  0
  Row 1 is 0  0  0  1
  Row 2 is 0  0  0  1
  Row 3 is 0  0  0  0
Offspring vector: 0  0  0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  0  -1
         -1  -1  -1  0
         -1  -1  -1  0
         -1  -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1
```

```
  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  capacity of node 2 is 1

  Analysing node: 3
  capacity of node 3 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 3

  Updated Lower Bounds matrix: S1
         0  1  1  0
         0  0  0  1
         0  0  0  1
         0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 3)
         -1  1  0  -1
         -1  -1  -1  1
         -1  -1  -1  0
         -1  -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (2, 3)
         -1  1  1  -1
         -1  -1  -1  1
         -1  -1  -1  1
         -1  -1  -1  -1

In decrFlow()
Current node = 3


No further surplus flows to remove.
source node: sum = 2     delta = -1
S1
  sink node: sum = 2     delta = -1
*** minFlow of S1 is 2       (= sum of
flows into sink)

Final flow matrix for S1
         -1  1  1  -1
         -1  -1  -1  1
         -1  -1  -1  1
         -1  -1  -1  -1
Out traverse()

  capacity of node 0 is 2

  Analysing node: 1
  offspring[1] = YES (Read next
subgraph:)

Graph name: S2
Number of nodes = 5
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1  1  0
  Row 1 is 0  0  0  0  1
  Row 2 is 0  0  0  0  1
  Row 3 is 0  0  0  0  1
  Row 4 is 0  0  0  0  0
Offspring vector: 0  0  1  0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  0  0  -1
         -1  -1  -1  -1  0
         -1  -1  -1  -1  0
         -1  -1  -1  -1  0
         -1  -1  -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  offspring[2] = YES (Read next
subgraph:)

Graph name: S23
Number of nodes = 2
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  2
  Row 1 is 0  0
Offspring vector: 0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0
         -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 1

  Updated Lower Bounds matrix: S23
         0  2
         0  0

In fillEdges()

----------------------
Current edge = (0, 1)
```

```
  Adding 2 unit(s) of flow to current
edge
         -1  2
         -1  -1

In decrFlow()
Current node = 1


No further surplus flows to remove.
source node: sum = 2      delta = -1
S23
  sink node: sum = 2      delta = -1
*** minFlow of S23 is 2       (= sum
of flows into sink)

Final flow matrix for S23
         -1  2
         -1  -1
Out traverse()

  capacity of node 2 is 2

  Analysing node: 3
  capacity of node 3 is 1

  Analysing node: 4
  capacity of node 4 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 4

  Updated Lower Bounds matrix: S2
         0  1  2  1  0
         0  0  0  0  1
         0  0  0  0  2
         0  0  0  0  1
         0  0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 4)
         -1  1  0  0  -1
         -1  -1  -1  -1  1
         -1  -1  -1  -1  0
         -1  -1  -1  -1  0
         -1  -1  -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 2 unit(s) of flow to current
edge
  There are successors
  Adding 2 unit(s) of flow to
successor edge = (2, 4)
         -1  1  2  0  -1
         -1  -1  -1  -1  1
         -1  -1  -1  -1  2
         -1  -1  -1  -1  0
         -1  -1  -1  -1  -1

----------------------
Current edge = (0, 3)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (3, 4)
         -1  1  2  1  -1
         -1  -1  -1  -1  1
         -1  -1  -1  -1  2
         -1  -1  -1  -1  1
         -1  -1  -1  -1  -1

In decrFlow()
Current node = 4


No further surplus flows to remove.
source node: sum = 4      delta = -3
S2
  sink node: sum = 4      delta = -3
*** minFlow of S2 is 4       (= sum of
flows into sink)

Final flow matrix for S2
         -1  1  2  1  -1
         -1  -1  -1  -1  1
         -1  -1  -1  -1  2
         -1  -1  -1  -1  1
         -1  -1  -1  -1  -1
Out traverse()

  capacity of node 1 is 4

  Analysing node: 2
  offspring[2] = YES (Read next
subgraph:)

Graph name: S3
Number of nodes = 7
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1  0  0  0  0
  Row 1 is 0  0  0  1  0  0  0
  Row 2 is 0  0  0  1  0  0  0
  Row 3 is 0  0  0  0  1  1  0
  Row 4 is 0  0  0  0  0  0  1
  Row 5 is 0  0  0  0  0  0  1
  Row 6 is 0  0  0  0  0  0  0
Offspring vector: 0  1  0  0  0  1  0


In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  0  -1  -1  -1  -1
         -1  -1  -1  0  -1  -1  -1
         -1  -1  -1  0  -1  -1  -1
         -1  -1  -1  -1  0  0  -1
         -1  -1  -1  -1  -1  -1  0
         -1  -1  -1  -1  -1  -1  0
         -1  -1  -1  -1  -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
```

```
  capacity of node 0 is 1

  Analysing node: 1
  offspring[1] = YES (Read next
subgraph:)

Graph name: S32
Number of nodes = 2
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  5
  Row 1 is 0  0
Offspring vector: 0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0
         -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 1

  Updated Lower Bounds matrix: S32
        0  5
        0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 5 unit(s) of flow to current
edge
         -1  5
         -1  -1

In decrFlow()
Current node = 1


No further surplus flows to remove.
source node: sum = 5     delta = -4
S32
  sink node: sum = 5     delta = -4
*** minFlow of S32 is 5      (= sum
of flows into sink)

Final flow matrix for S32
         -1  5
         -1  -1
Out traverse()

  capacity of node 1 is 5

  Analysing node: 2
  capacity of node 2 is 1

  Analysing node: 3
  capacity of node 3 is 1

  Analysing node: 4
  capacity of node 4 is 1

  Analysing node: 5
  capacity of node 5 is 1

  Analysing node: 6
  capacity of node 6 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 6

  Updated Lower Bounds matrix: S3
        0  5  1  0  0  0  0
        0  0  0  5  0  0  0
        0  0  0  1  0  0  0
        0  0  0  0  1  1  0
        0  0  0  0  0  0  1
        0  0  0  0  0  0  1
        0  0  0  0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 5 unit(s) of flow to current
edge
  There are successors
  Adding 5 unit(s) of flow to
successor edge = (1, 3)  Adding 5
unit(s) of flow to successor edge =
(3, 4)  Adding 5 unit(s) of flow to
successor edge = (4, 6)
         -1  5  0  -1  -1  -1  -1
         -1  -1  -1  5  -1  -1  -1
         -1  -1  -1  0  -1  -1  -1
         -1  -1  -1  -1  5  0  -1
         -1  -1  -1  -1  -1  -1  5
         -1  -1  -1  -1  -1  -1  0
         -1  -1  -1  -1  -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (2, 3)  Adding 1
unit(s) of flow to successor edge =
(3, 5)  Adding 1 unit(s) of flow to
successor edge = (5, 6)
         -1  5  1  -1  -1  -1  -1
         -1  -1  -1  5  -1  -1  -1
         -1  -1  -1  1  -1  -1  -1
         -1  -1  -1  -1  5  1  -1
         -1  -1  -1  -1  -1  -1  5
         -1  -1  -1  -1  -1  -1  1
         -1  -1  -1  -1  -1  -1  -1

In decrFlow()
Current node = 6
  Potential surplus on Edge(4, 6) = 5

Current node = 4
  Potential surplus on Edge(3, 4) = 5
```

```
Current node = 3

Current node = 5


No further surplus flows to remove.
source node: sum = 6      delta = -5
S3
  sink node: sum = 2      delta = -1
*** minFlow of S3 is 6       (= sum of
flows into sink)

Final flow matrix for S3
         -1  5  1  -1  -1  -1  -1
         -1  -1  -1  5  -1  -1  -1
         -1  -1  -1  1  -1  -1  -1
         -1  -1  -1  -1  5  1  -1
         -1  -1  -1  -1  -1  -1  5
         -1  -1  -1  -1  -1  -1  1
         -1  -1  -1  -1  -1  -1  -1
Out traverse()

  capacity of node 2 is 6

  Analysing node: 3
  offspring[3] = YES (Read next
subgraph:)

Graph name: S4
Number of nodes = 5
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1  0  0
  Row 1 is 0  0  0  1  1
  Row 2 is 0  0  0  1  1
  Row 3 is 0  0  0  0  1
  Row 4 is 0  0  0  0  0
Offspring vector: 0  0  0  0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  0  -1  -1
         -1  -1  -1  0  0
         -1  -1  -1  0  0
         -1  -1  -1  -1  0
         -1  -1  -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  capacity of node 2 is 1

  Analysing node: 3
  capacity of node 3 is 1

  Analysing node: 4
  capacity of node 4 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 4

 Updated Lower Bounds matrix: S4
        0  1  1  0  0
        0  0  0  1  1
        0  0  0  1  1
        0  0  0  0  1
        0  0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 3)  Adding 1
unit(s) of flow to successor edge =
(3, 4)
         -1  1  0  -1  -1
         -1  -1  -1  1  0
         -1  -1  -1  0  0
         -1  -1  -1  -1  1
         -1  -1  -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (2, 3)  Adding 1
unit(s) of flow to successor edge =
(3, 4)
         -1  1  1  -1  -1
         -1  -1  -1  1  0
         -1  -1  -1  1  0
         -1  -1  -1  -1  2
         -1  -1  -1  -1  -1


----------------------
Current edge = (1, 4)
  Adding 1 unit(s) of flow to current
edge
  There are predecessors
  Adding 1 unit(s) of flow to
predecessor edge = (0, 1)
         -1  2  1  -1  -1
         -1  -1  -1  1  1
         -1  -1  -1  1  0
         -1  -1  -1  -1  2
         -1  -1  -1  -1  -1


----------------------
Current edge = (2, 4)
  Adding 1 unit(s) of flow to current
edge
  There are predecessors
  Adding 1 unit(s) of flow to
predecessor edge = (0, 2)
         -1  2  2  -1  -1
         -1  -1  -1  1  1
         -1  -1  -1  1  1
         -1  -1  -1  -1  2
         -1  -1  -1  -1  -1
```

```
In decrFlow()
Current node = 4
  Potential surplus on Edge(3, 4) = 1

Current node = 3


No further surplus flows to remove.
source node: sum = 2      delta = -1
S4
  sink node: sum = 3      delta = -2
*** minFlow of S4 is 4       (= sum of
flows into sink)

Final flow matrix for S4
         -1  2  2  -1  -1
         -1  -1  -1  1  1
         -1  -1  -1  1  1
         -1  -1  -1  -1  2
         -1  -1  -1  -1  -1
Out traverse()

  capacity of node 3 is 4

  Analysing node: 4
  offspring[4] = YES (Read next
subgraph:)

Graph name: S5
Number of nodes = 3
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1
  Row 1 is 0  0  1
  Row 2 is 0  0  0
Offspring vector: 0  0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  0
         -1  -1  0
         -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  capacity of node 2 is 1

  Conclusion: source node = 0
  Conclusion:   sink node = 2

  Updated Lower Bounds matrix: S5
         0  1  1
         0  0  1
         0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 2)
         -1  1  0
         -1  -1  1
         -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current
edge
         -1  1  1
         -1  -1  1
         -1  -1  -1

In decrFlow()
Current node = 2


No further surplus flows to remove.
source node: sum = 2      delta = -1
S5
  sink node: sum = 2      delta = -1
*** minFlow of S5 is 2       (= sum of
flows into sink)

Final flow matrix for S5
         -1  1  1
         -1  -1  1
         -1  -1  -1
Out traverse()

  capacity of node 4 is 2

  Analysing node: 5
  offspring[5] = YES (Read next
subgraph:)

Graph name: S5
Number of nodes = 3
Lower Bounds matrix: (minimum flows)
  Row 0 is 0  1  1
  Row 1 is 0  0  1
  Row 2 is 0  0  0
Offspring vector: 0  0  0

In traverse()
  The corresponding Flow matrix
(working copy):
         -1  0  0
         -1  -1  0
         -1  -1  -1

  Analysis of Lower Bounds matrix
above:
  Analysing node: 0
  capacity of node 0 is 1

  Analysing node: 1
  capacity of node 1 is 1

  Analysing node: 2
  capacity of node 2 is 1
```

```
  Conclusion: source node = 0
  Conclusion:   sink node = 2

  Updated Lower Bounds matrix: S5
         0  1  1
         0  0  1
         0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 1 unit(s) of flow to current
edge
  There are successors
  Adding 1 unit(s) of flow to
successor edge = (1, 2)
         -1  1  0
         -1  -1  1
         -1  -1  -1


----------------------
Current edge = (0, 2)
  Adding 1 unit(s) of flow to current
edge
         -1  1  1
         -1  -1  1
         -1  -1  -1

In decrFlow()
Current node = 2


No further surplus flows to remove.
source node: sum = 2     delta = -1
S5
  sink node: sum = 2     delta = -1
*** minFlow of S5 is 2       (= sum of
flows into sink)

Final flow matrix for S5
         -1  1  1
         -1  -1  1
         -1  -1  -1
Out traverse()

  capacity of node 5 is 2

  Conclusion: source node = 0
  Conclusion:   sink node = 5

  Updated Lower Bounds matrix: MAIN-
EXAMPLE
         0  4  0  0  0  0
         0  0  6  4  0  0
         0  0  0  0  6  0
         0  0  0  0  4  0
         0  0  0  0  0  2
         0  0  0  0  0  0

In fillEdges()

----------------------
Current edge = (0, 1)
  Adding 4 unit(s) of flow to current
edge
  There are successors
  Adding 4 unit(s) of flow to
successor edge = (1, 2)  Adding 4
unit(s) of flow to successor edge =
(2, 4)  Adding 4 unit(s) of flow to
successor edge = (4, 5)
         -1  4  -1  -1  -1  -1
         -1  -1  4  0  -1  -1
         -1  -1  -1  -1  4  -1
         -1  -1  -1  -1  0  -1
         -1  -1  -1  -1  -1  4
         -1  -1  -1  -1  -1  -1


----------------------
Current edge = (1, 2)
  Adding 2 unit(s) of flow to current
edge
  There are predecessors
  Adding 2 unit(s) of flow to
predecessor edge = (0, 1)
  There are successors
  Adding 2 unit(s) of flow to
successor edge = (2, 4)  Adding 2
unit(s) of flow to successor edge =
(4, 5)
         -1  6  -1  -1  -1  -1
         -1  -1  6  0  -1  -1
         -1  -1  -1  -1  6  -1
         -1  -1  -1  -1  0  -1
         -1  -1  -1  -1  -1  6
         -1  -1  -1  -1  -1  -1


----------------------
Current edge = (1, 3)
  Adding 4 unit(s) of flow to current
edge
  There are predecessors
  Adding 4 unit(s) of flow to
predecessor edge = (0, 1)
  There are successors
  Adding 4 unit(s) of flow to
successor edge = (3, 4)  Adding 4
unit(s) of flow to successor edge =
(4, 5)
         -1  10  -1  -1  -1  -1
         -1  -1  6  4  -1  -1
         -1  -1  -1  -1  6  -1
         -1  -1  -1  -1  4  -1
         -1  -1  -1  -1  -1  10
         -1  -1  -1  -1  -1  -1

In decrFlow()
Current node = 5
  Potential surplus on Edge(4, 5) = 8

Current node = 4


No further surplus flows to remove.
source node: sum = 4     delta = -2
MAIN-EXAMPLE
  sink node: sum = 2     delta = 0
*** minFlow of MAIN-EXAMPLE is 10
(= sum of flows into sink)
```

```
Final flow matrix for MAIN-EXAMPLE
        -1  10  -1  -1  -1  -1
        -1  -1   6   4  -1  -1
        -1  -1  -1  -1   6  -1
        -1  -1  -1  -1   4  -1
        -1  -1  -1  -1  -1  10
        -1  -1  -1  -1  -1  -1
Out traverse()
```

```
In main()
MAXCUT OVERALL = 10

Normal termination.
```

■

APPENDIX 8

Precise Definition of HFSM Subset

In Chapter 2, Background and Related Work, we saw background descriptions of finite-state machines (FSMs) and some of their common variants: extended FSM (EFSM), communicating FSM (CFSM), and the hierarchical FSM (HFSM), the latter being the basis of study for this thesis. Section 2.2 discussed two variants on HFSMs: statecharts [Harel 87] and ROOMcharts [Selic++ 94]. Both of these are considered visual formalisms, and both are expressive and powerful with their respective extensions to the basic HFSM concept. The work in this paper is limited to a subset of HFSMs, as characterised by the description in Section 5.8, Application and Implementation. This appendix provides a more formal depiction of the subset under consideration.

Definition of HFSM

As alluded to in the discussion on research in HFSMs and statecharts (Section 2.2.4), many scholars have attempted to make the statechart notation more rigorous. The definitions used in [Heimdahl+ 96] for the Requirements State Machine Language (RSML) and in [Harel+ 96] for statecharts are essentially the same with respect to their hierarchical and parallelism properties. In our work, we do treat hierarchy, but, as indicated in Section 5.8, we do not consider the concurrent threads of execution possible with parallel components.

Our HFSM, M, can be described by the following tuple:

$$M = (S, H, I, O, F, s0)$$

where

S is a finite set of symbols denoting states

H is a hierarchy function mapping a state onto a set of states,

and is defined by

$$H0(s) = \{s\}$$

$$H1(s) = \{\textit{the "children" of } s\}$$

$$H2(s) = \{\textit{the "grandchildren" of } s\}$$

$$H(i+1)(s) = \mathbf{UNION}_{t \in H(i)(s)} \; H1(t)$$

(= the union of all *children* of all nodes at level i)

and we can write

$$H^*(s) = \mathbf{UNION}_{i=0}^{\infty} \; H(i)(s)$$

I is a set of symbols denoting the possible inputs

O is a set of symbols denoting the possible outputs

F is a relation defining the global state changes

and, in essence, describes the *behaviour* of M.

A *global state*, in the case of HFSMs without parallelism, is simply the set of states that constitute the current configuration of the machine. F maps one global state onto another and accounts for different inputs and outputs.

$s0$ is the initial global state of the machine; $s0 \in$ powerset $2^{**}S$.

As an aside, [Heimdahl+ 96] and [Harel+ 96] also include a *partitioning* function, P, for parallelism (or "concurrency"), to partition the descendants of a state into mutually exclusive subsets. Again, parallelism is not part of the subset under our investigation.

HFSM Subset

For the subset that was actually implemented in C code, there were additional restrictions and requirements placed on the definition above. (Ref: Section 5.8)

1. Restriction: Each graph has a source and a sink. There are no cycles.

   flow(e) = 0, whenever e ∈ *InwardEdge(source)*

   flow(e) = 0, whenever e ∈ *OutwardEdge(sink)*

2. Requirement: There must exist a state, $x \in S$, known as the *root*, satisfying

$$\exists \text{ unique } x \in S \ : \ x = lcp(S)$$

   where *lcp* is the *least common parent* of the states in the set $X \in$ powerset 2 ** S,

   $lcp(X) = y$, and is defined as

$$[\, X \subseteq H^*(y) \,] \ \wedge \ [\, \forall s \in S : (X \subseteq H^*(s)) \Rightarrow (y \in H^*(s)) \,]$$ [Heimdahl+ 96]

3. Restriction: There is no parallelism in the HFSM structure. Only a single thread of execution is supported by this implementation.

Example.

Consider the graph, TRIANGULUM, as seen in Fig. A6-5 (which is repeated here for the reader's convenience).

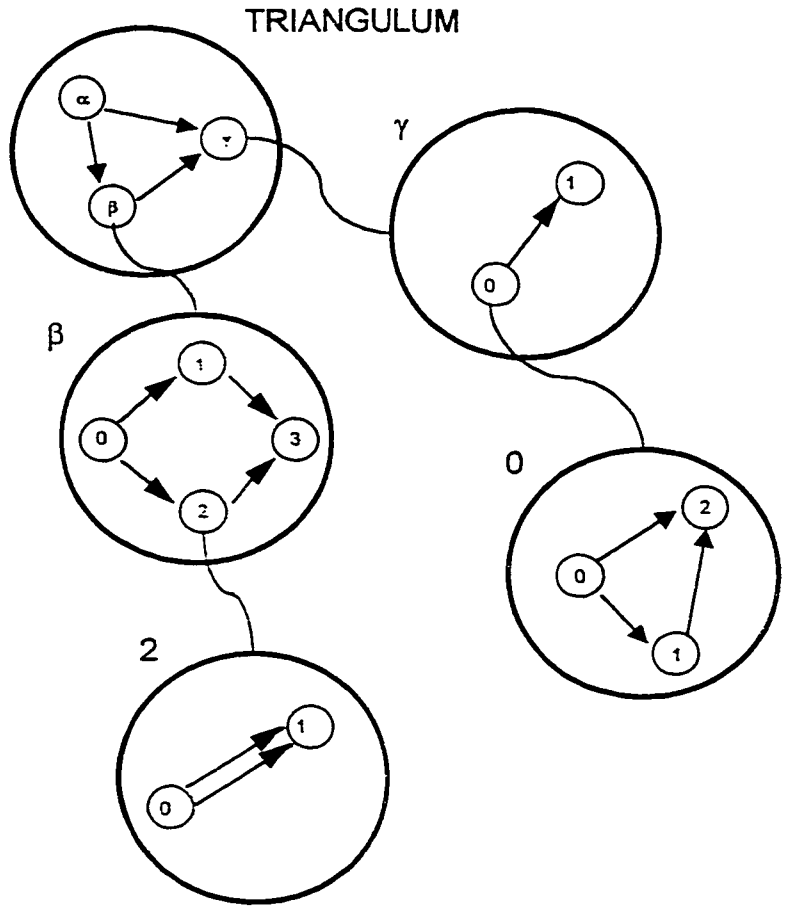


Fig. A6-5. *TRIANGULUM* (repeated)

We see the following values for our defined terms, *S, H, I, O, F,* and *s0.*

S = the set of possible states =

{α, β, γ,

β\0, β\1, β\2, β\3,

β\2\0, β\2\1,

γ\0, γ\1,

γ\0\0, γ\0\1, γ\0\2

}

A note on notation: It happens that in Triangulum the state-names are not unique across the HFSM, therefore the full "pathname" has been included, using backslashes ("\") to separate the levels of hierarchy.

H is the hierarchy function exactly as defined in the preceding pages. Therefore,

H0(Triangulum) = {Triangulum}

H1(Triangulum) = {α, β, γ}

H2(Triangulum) = {β\0, β\1, β\2, β\3, γ\0, γ\1}

H3(Triangulum) = {β\2\0, β\2\1, γ\0\0, γ\0\1, γ\0\2}

I, the set of inputs, is not shown for Triangulum.

O, the set of outputs, is not shown for Triangulum.

Possible global states:

{α},
{β, β\0},
{β, β\1},
{β, β\2, β\2\0},
{β, β\2, β\2\1},
{β, β\3},
{β, β\1},
{γ, γ\0, γ\0\0},
{γ, γ\0, γ\0\1},
{γ, γ\0, γ\0\2},
{γ, γ\1}.

F, the behaviour relation, is illustrated graphically by the transitions (the directed edges) in the HFSM diagram for Triangulum. Two transitions are presented in text format here.

For example,

$F(\{\beta, \beta\backslash 1\}) = \{\beta, \beta\backslash 3\}$

Another example,

$F(\{\gamma, \gamma\backslash 0, \gamma\backslash 0\backslash 0\}) =$ either $\{\gamma, \gamma\backslash 0, \gamma\backslash 0\backslash 1\}$

or $\{\gamma, \gamma\backslash 0, \gamma\backslash 0\backslash 2\}$, depending on the input (not shown)

s0, the initial global state, = $\{\alpha\}$

Thus, we have displayed some of the values of the tuple (*S, H, I, O, F, s0)* describing Triangulum. ■

Evidence of the useability of this exact HFSM is shown in the sample run for Triangulum in Appendix 6 (pages A6-17 to A6-22).