NASA-CR-192176

145576
p-534

EXPERT SYSTEM VERIFICATION AND VALIDATION STUDY

WORKSHOP & PRESENTATION MATERIAL

Scott W. French
David Hamilton

International Business Machines Corporation

August 1992

(NASA-CR-192176) EXPERT SYSTEM VERIFICATION AND VALIDATION STUDY: WORKSHOP AND PRESENTATION MATERIAL (Research Inst. for Computing and Information Systems) 534 p

N93-21661

Unclas

G3/61 0145576

Cooperative Agreement NCC 9-16
Research Activity No. AI.16

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

ricis

Research Institute for Computing and Information Systems
University of Houston-Clear Lake

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

EXPERT SYSTEM VERIFICATION AND VALIDATION STUDY

WORKSHOP & PRESENTATION MATERIAL

Scott W. French
David Hamilton

International Business Machines Corporation

August 1992

Cooperative Agreement NCC 9-16
Research Activity No. AI.16

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

EXPERT SYSTEM VERIFICATION AND VALIDATION STUDY

WORKSHOP & PRESENTATION MATERIAL

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Scott W. French and David Hamilton of the International Business Machines Corporation. Dr. T. F. Leibfried, Jr. served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Christopher Culbert, Chief, Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

Workshop on Verification and Validation of Expert Systems

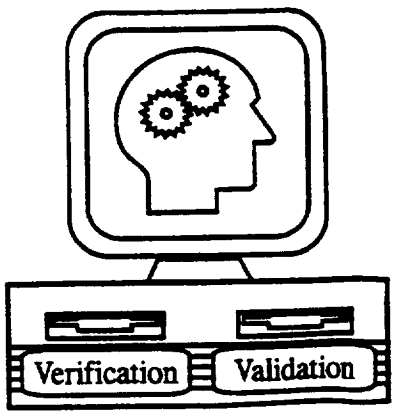
Introduction

Authors:

Scott W. French
FRENCHS@HOUVMSCC.VNET.IBM.COM

David Hamilton
HAMILTON@HOUVMSCC.VNET.IBM.COM

IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058



Welcome

Welcome to the Workshop on Verification and Validation (V&V) of Expert Systems

This introduction will tell you

- Where we have been with respect to V&V of ES
- Where we are headed with this workshop
- What you, the student, will learn

Where We Have Been

Significant work has been done in KBS V&V

- Development of conceptual approaches
- Proposing various techniques

No significant case studies or field demonstrations

- Many conjectures have been made
  - » No requirements,
  - » Few ES subjected to the same level of V&V as conventional software
  - » . . .
- Many problems discussed
  - » Test Coverage
  - » Unpredictability (rule interaction, run-time performance, etc.)
  - » . . .

Where We Have Been ...

A survey was performed to assess the state-of-the-practice in ES V&V

- Determine the real issues in V&V of ES

- Assess the accuracy of the many conjectures

- Determine the course of future work in V&V of ES

Where We Have Been ...

60+ projects were asked questions such as

- What V&V activities are done, not done?
- What issues occur in practice?
- To what extent does V&V play a part in these issues?
- How satisfied are the users with the quality and reliability of the ES
  - » NOTE: The survey did not attempt to evaluate the quality of a specific ES

<u>Where We Have Been ...</u>

Caveats

- Results are not statistically valid since responses were voluntary

- Responses were not validated since they reflected the responder's opinion

Given these caveats, the survey results point to recommendations

- <u>Direct recommendations</u>: those derived directly from survey responses

- <u>Inferred recommendations</u>: those supported only by the data (i.e., the responder did not list it as an issue)

<u>Where We Have Been ...</u>

Address the most frequently cited issues (direct)

- Test coverage determination (63%)
- Knowledge Validation (60%)
- Problem Complexity (40%)

Recommend a Life-Cycle (direct)

- 22% indicated that no life-cycle model was followed
- 43% indicated that the resulting ES was taken directly from a prototype (an *operational* prototype)

Where We Have Been ...

Develop guidelines for ES V&V (direct)

- Ad-hoc application of techniques
- ES evaluation difficult (27% of developers vs. 100% of users)
- Expected ES to be at least as accurate as expert (79% users and developers)
  - » System did not meet expectations (49% of developers and 100% of users)
  - » System was less accurate than expert (44% of developers vs. 80% of users)
- 57% of operational development efforts wrote no requirements
- 52% used only one technique

Where We Have Been ...

Address understandability and modularity (inferred)

- 85% indicated test coverage was a problem
- 83% indicated problem complexity as a problem
- Yet, modularity and understandability were not specifically addressed

Investigate potential configuration management issues (inferred)

- Only 14% cited CM as an issue
- Yet, interviews indicated it was more of a concern than the numbers reflected

Where We Have Been ...

Investigate analysis tools to aid the expert (inferred)

- 59% relied on the expert to analyze knowledge structures

- 61% relied on the expert for requirements

Develop criteria to classify systems by intended use (inferred)

- e.g., Expert clone, Expert assistant, Autonomous, etc.

- Interviews indicated a need for tailorable guidelines based, not only on criticality, but on intended use

Where We Are Headed

This workshop was developed in response to the recommendations found in the survey

The purpose of the workshop is to positively impact the state-of-the-practice in ES V&V

- Encourage the systematic application of V&V techniques and approaches
  - » Ease problems in managing ES projects
  - » Reduce re-work
  - » Reduce long-term costs (i.e., make maintenance easier)
- Provide tailorable guidelines
  - » Give developers help in being "systematic"

Where We Are Headed...

Day 1 Basic Concepts

- Morning
  - » Introduction
  - » Presentation of background on Verification and Validation (V&V) concepts
  - » Demo
  - » Presentation of common misconceptions concerning both AI and V&V
- Afternoon
  - » Presentation of the Apollo 11 Scenario
  - » Presentation covering differences between ES and procedural systems and how those differences impact V&V
  - » Demo

Where We Are Headed...

Day 2 Techniques

- Morning
  - » Review of Day 1 topics
  - » Present class discussion problem and begin team exercises
  - » Demo
  - » Presentation on the importance of Planning, Problem Analysis, and Re-Engineering
- Afternoon
  - » Present Verification techniques and exercises

<u>Where We Are Headed...</u>

<u>Day 3</u> Techniques ...

- Morning
  - » Review of Day 2 Topics
  - » Present more Verification techniques and exercises
  - » Demo
- Afternoon
  - » Exchange verification approaches
  - » Present Validation techniques and exercises

Where We Are Headed...

Day 4 Guidelines

- Morning
  - » Present guidelines for applying V&V approaches
  - » Prepare presentation of exercises
- Afternoon
  - » Team presentations of exercise solutions

How We Will Get There

The following student material has been provided in the notebook at your desk

1. Copy of all presentation material
   - Introduction (tab)
   - Basic Concepts (tab)
   - Techniques (tab)
   - Guidelines (tab)
2. Handouts (tab)
   - Material to be periodically referenced
   - Used to support presentation material
   - Contains exercises and some possible solutions

How We Will Get There ...

Student materials ...

3. TLC Solutions (tab)
   - Presents different approaches to building functionally correct solutions to the class problem
4. Exercises (tab)
   - A collection of problems to be worked in teams
5. Worksheets
   - Provides quick reference information and examples for use in applying key techniques
6. References
   - Collection of optional but suggested reading

How We Will Get There ...

Questions encouraged during lectures

Class discussion questions will be posed (informal roundtable discussion)

Will be divided into teams for some exercises

- Results discussed informally for all but final exercise
- Results of final exercise presented before class
- Exercises are NOT a test. Ask questions.

What You Should Learn

- What is V&V and why it is important.
  - Problem Complexity and Understandability (i.e., Modularity)
  - Life-Cycle Issues
  - Configuration Management
- Differences between conventional and ES V&V
- Conventional and ES V&V techniques
  - Test coverage
  - Knowledge validation
- Some key V&V rules of thumb
- How to make V&V easier
  - Easing analysis burden for the Expert
- A suggested approach to V&V
  - Guidelines for ES V&V

Workshop On Verification and Validation of Expert Systems

Basic Concepts

Authors:

Scott W. French
FRENCHS@HOUVMSCC.VNET.IBM.COM

David Hamilton
HAMILTON@HOUVMSCC.VNET.IBM.COM

IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058

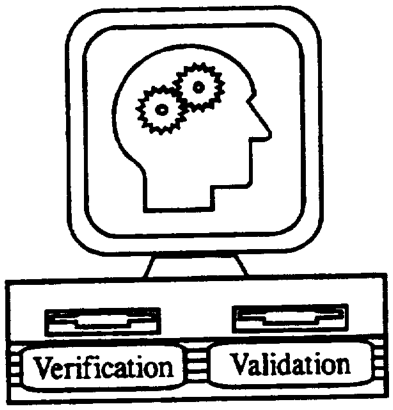


Table of Contents

Introduction

Overview

Purpose

- Review conventional V&V concepts
- Dispel myths concerning AI and Software Engineering
- Clarify the difference between ES and conventional software and how those differences impact V&V
- <u>In short</u>, justify the need for doing V&V

PRECEDING PAGE BLANK NOT FILMED

Overview

Self-imposed Constraints

- Discuss concepts independent of a specific life-cycle model
- Do not assume a particular development methodology
- Separate the description of V&V from the similar description of designing a software system

Overview ...

Notes

- Our focus will be on V&V, not on how the system is developed.

- We will not assume a background in V&V or conventional software development.

Overview ...

Key Tenants

- A full understanding of the problem is never initially possible but must be developed incrementally along with the system.

- Correctness can never be practically proved and a system will always have errors.

- To develop test cases, one needs to understand the problem being solved.

- The earlier an error is discovered, the more cheaply it can be corrected.

Goals

To show that V&V should be done

- Verification helps a developer implement the system more efficiently and cost-effectively

- Validation ensures the system solves the customers problem in a reliable, predictable, and user-friendly manner.

Goals ...

To show that V&V works best when performed as the system is developed

- This will be done as we review the major V&V tasks.

- For a V&V task, we will look at the inputs required from a corresponding development task.

To show that the system can be developed so as to make V&V easier

- We look to see how V&V might be done more easily and cheaply by doing some tasks earlier in the development process.

The Verification Puzzle

There are many *pieces* to *The Verification Puzzle*

- *Functional Correctness:* A correct response for every stimulus to the system, during installation and checkout as well as operational use

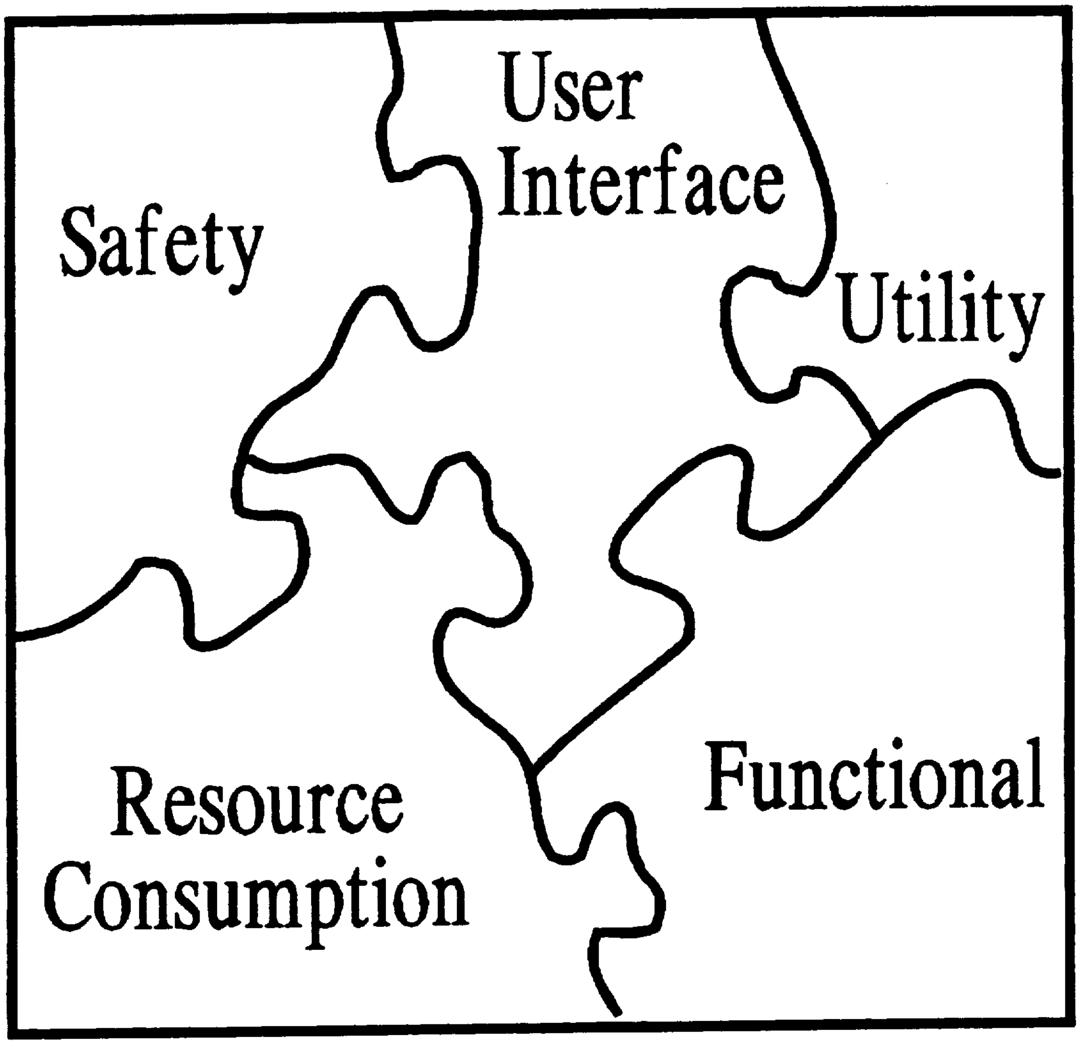
- *User-Interface Correctness:* Responses intended for human view are clear; expected stimulus does not put excessive burden on the user

The Verification Puzzle ...

Pieces to *The Verification Puzzle ...*

- *Safety Correctness:* **Will never generate a response that will cause harm to anyone or anything**
- *Resource Consumption Correctness:* **No more processor time, storage, bandwidth, etc. are used than is allowed**
- *Utility Correctness:* **The system (sufficiently) satisfies the user's needs.**

The Verification Puzzle



<u>The Verification Puzzle ...</u>

Three aspects to demonstrating system correctness - <u>consistency</u>, <u>completeness</u> and <u>termination</u>.

1. Consistency

- The system is both externally and internally consistent
  - » <u>External</u> - correct outputs and actions (e.g., hitting ESC from any window produces the same result)
  - » <u>Internal</u> - all internal items are consistent (e.g., integer variables are only assigned integer values)

The Verification Puzzle ...

Aspects to demonstrating system correctness ...

2. Completeness

- The system does all it should
  - » Accepts all required inputs
  - » Performs all required actions
  - » Creates all required outputs
  - » Maintains all required data
- More difficult than checking consistency

The Verification Puzzle ...

Aspects to showing correctness ...

3. Termination

- correct programs produce the right output for all possible inputs
- consistency and completeness show that all outputs are correct
- termination shows that output is always generated

The Verification Puzzle ...

- Demonstrating system correctness depends on the type of software system being developed

There are many different types of software

- Large software systems vs. smaller self-contained problem solvers
- Highly complex vs. less complex software
- Critical software vs. noncritical software
- Expert system vs. a traditional software problem; that can be conveniently solved using expert system techniques

The Verification Puzzle ...

Demonstrating system correctness also depends on how the system is represented

Representation relates to type of software

Many kinds of system representations

- text, code, flow charts, etc.

Organization is more important than "kind"

Easiest to V&V when the "what" and "how" of a system representation are separated

Three views of a system are helpful in building this kind of representation[35]

- "Object"/Data View
- Control View
- Function View

The Verification Puzzle ...

Demonstrating system correctness also depends on how the system is represented ...

"Object"/Data View

- View of the domain
- Foundation for the other views

Control View

- "Problem-solving Method"
- How elements of the "object" view are used to solve the problem

Function View

- Defines methods the "control" view may use
- Best when linked to elements of the "object" view

The Verification Puzzle ...

Many V&V techniques have been developed to address these aspects of demonstrating system correctness

- Some are more suitable for certain classes of correctness than others.
- Some are more suitable for certain types, sizes and/or complexities of software.

The key to solving the Verification puzzle is to use the right techniques in the right situations.

The Verification Puzzle ...

- A systematic approach exists for applying correctness techniques (i.e., solving the Verification Puzzle)

This approach can be broken down into three parts

- ***System Testing:*** Dynamic testing of all classes of correctness of an overall software system
- ***Unit/Integration Testing:*** Dynamic testing of small self-contained pieces of an overall system, focusing on certain classes of correctness
- ***Static Testing:*** Analysis (desk checking) of software specifications (requirements, design) at different levels of abstraction, focusing on certain classes of correctness

The Verification Puzzle ...

Each of these steps (or test phases and will be discussed separately

- A breakup of these phases into an ordered sequence of *tasks* is part of the development *life cycle*.

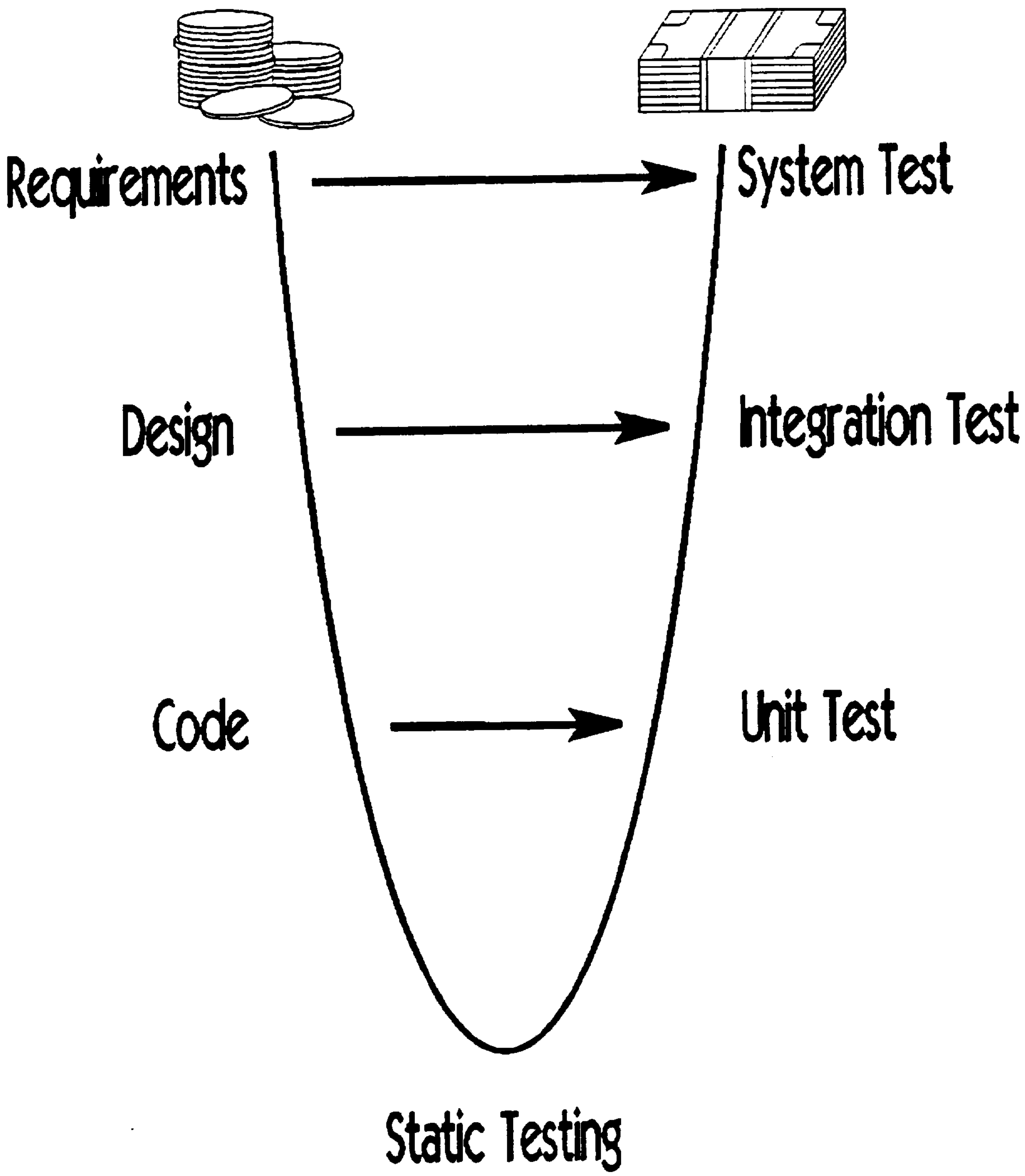
- We will not restrict our discussion to any specific life-cycle.

The Verification Puzzle ...

There is a *testing phase* for each major development phase

- System testing tests overall system requirements.
- Integration and unit testing test the units and subsystems created during system construction
- Static testing can be used to check all representations of a system
  - » design, code, requirements, etc.
- There is an implied order to these testing phases
  - » has cost implications
  - » implies earlier phases support later phases

Phases of Correctness



Overview of Test Phases ...

Each phase will be examined based on:

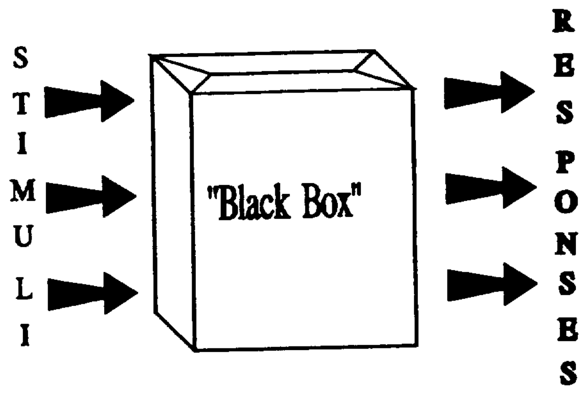
- ***Characteristics***: An overall description of the test phase
- ***Inputs***: Each phase requires certain information before it can be applied.
- ***Implications***: How the required inputs can be acquired from other development or testing phases

Testing Phases

System Testing

Characteristics

- *Black box*: Ignores implementation details
  - » Required and observed behavior
  - » Sometimes called the "function" view



- *Behavior*: Described in terms of stimulus/response pairs
  - » Defines an abstract "control" view
  - » Maps to detailed internal "control"

- *Validation*: Checks that the system will satisfy the users' needs

System Testing ...

There is a difference between verification and validation

Verification: "Am I building the product right ?"

- Best when performed during system development
- Emphasize showing correct implementation of requirements

Validation: "Am I building the right product?"

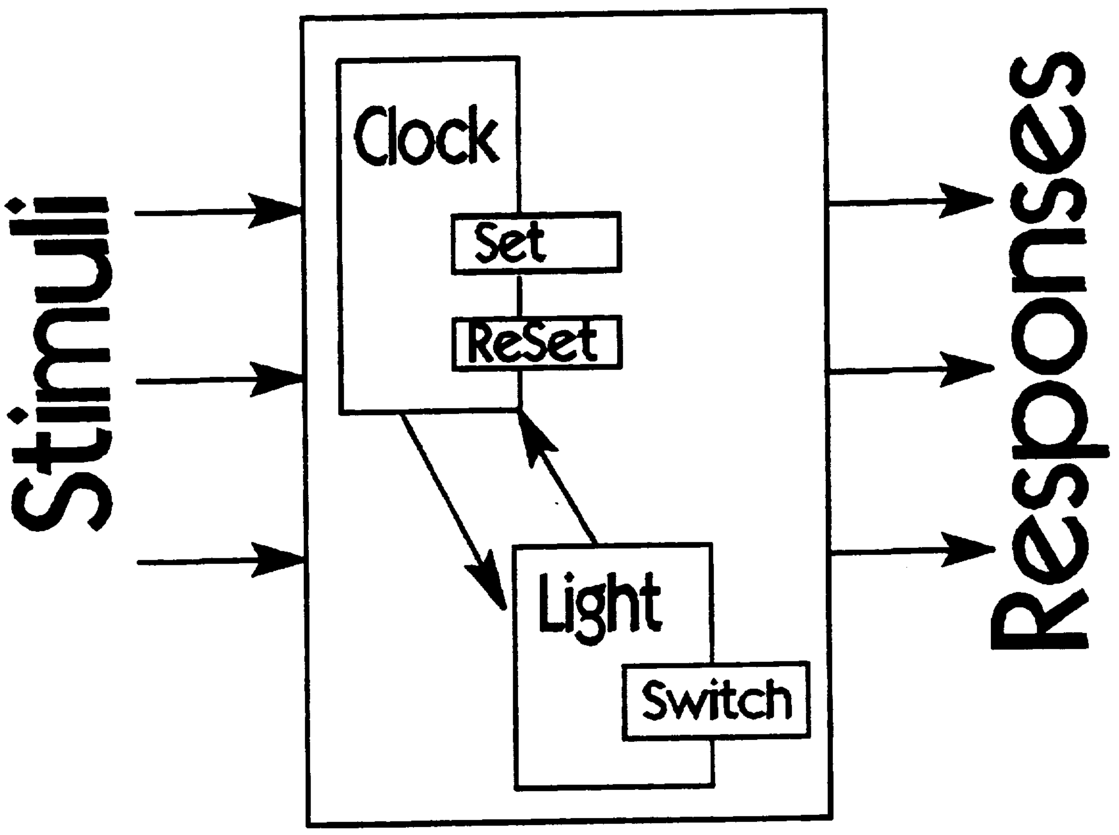
- Best performed when the system is complete
- Can be partially done early via prototyping
- Emphasis is on ensuring the requirements are correct

System Testing ...

Inputs

- The software system itself.
- Ideally, for each possible stimulus:
  - » Description of the required response
  - » Indication of criticality (i.e., safety implications of the response)
  - » Indication of response time allowed (if constrained)
  - » Description of user interface for the stimulus/response
  - » Indication of resources allowed for generating the response
- In reality, impractical for all possible stimuli
- Stimulus sequences can further be described in terms of operational scenarios

System Testing ...



System Testing ...

Implications

- Specify requirements as operational scenarios (i.e., documenting expected use)
- Classes of stimulus/response pairs correspond to self-contained units "inside" the system
  - » Stimuli form *classes* or *groups*
  - » Classes or groups are units[1].
  - » Units have subunits
  - » Subunits exhibit the same characteristic views (object, function and control)
- Overcome system test impracticalities by testing underlying units

1 This makes testing easier. This can be done regardless of how the system is actually implemented. For example, the Space Shuttle Flight Software (FSW) is tested by principal function even though this may not directly correspond to how the FSW is implemented.

Unit/Integration Testing

Characteristics

- *White Box*: Does "look inside the system" to see how it was implemented
  - » Tests exercise internals units
- *Behavior:* Stimulus history can be described in terms of internal software states (e.g., sets of variable values) and expected transitions between states.
  - » "Control" view becomes more explicit
- *Interfaces*: Much of the testing may focus on how well the separately developed units (subsystems) interface with each other (i.e., does the system "hang together").

<u>Unit/Integration Testing ...</u>

O—π Software can be modeled based on <u>state</u>

- Any program can be represented as an <u>automaton</u>

  "a machine or control mechanism designed to follow, automatically, a predetermined sequence of operations or respond to encoded instructions."Webster

- State refers to the behavior of an automaton at a given point in time as determined by its environment
  - » i.e., a "snapshot" of the system
- State determines the future course the automaton will take
  - » i.e., determines the next <u>state transition</u>

Unit/Integration Testing ...

Inputs

- The software units themselves.
- Stimulus/response behavior for each unit
- Identification of subsystems (collections of units) along with their required behavior
- Scenarios (e.g., operational scenarios) that indicate how the units and subsystems will be used

Unit/Integration Testing ...

Implications

Use of modularity directly benefits Unit/Integration testing

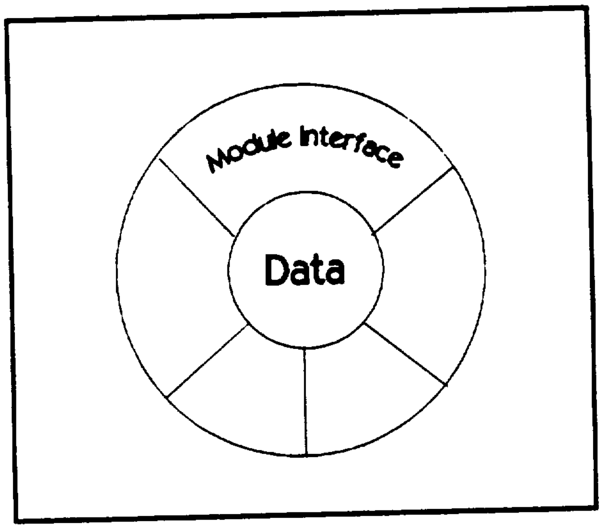
- Reduces a system complexity
- The "object"/data view of the system
- Aids overall system understanding
  - » Design structure becomes explicit
  - » "... the designer can spend more time understanding and deciding (about the design) - rather than gathering the information on which to base the decision."14

Unit/Integration Testing ...

Implications ...

What are the modules or parts of a system?

- Modules can be defined in many ways
  - » A program procedure that captures some common task is an example of a module
  - » The best modules capture, not only a common task, but common data as well



Unit/Integration Testing ...

Implications ...

What are the modules or parts of a system ...

- Criteria for identifying modules
  - » Best choice is to capture state within a module
  - » Capture complicated design decisions
  - » Capture collections of common "services"

Unit/Integration Testing ...

Implications ...

So, what are the benefits of modularity?

- Separate development and test
  - » Provides a framework for reuse
- Framework for information hiding
  - » Hiding unimportant implementation details from module users
- Enforces standard methods of access (encapsulation)
  - » Data access can only happen through the module interface
- Incremental development(build a little, test a little).

Unit/Integration Testing ...

Implications ...

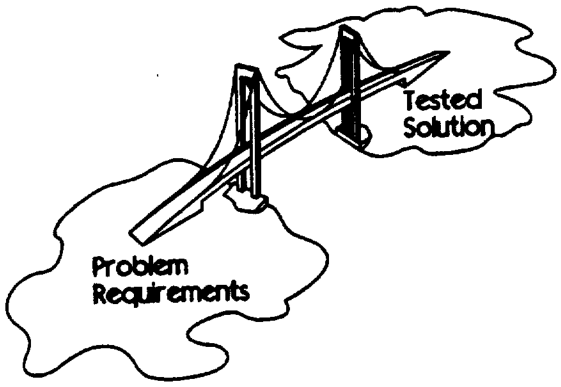
Benefits of Modularity ...

- Reduces re-verification burden
  - » Changes are localized to specific modules
  - » Stable interface minimizes impacts to the outside world
- Eases project management
  - » One module = One unit of work
  - » One unit of work = One programmer

Unit/Integration Testing ...

Implications ...

Design "bridges the gap"



System testing becomes easier

- Internal units need not be re-tested

However, Exhaustive testing is still impractical

- Human analysis of design/code can find many errors relatively cheaply
- Static testing addresses this

Static Testing

Characteristic

- ***Analysis:*** **Software is not dynamically executed; instead it is analyzed statically (e.g., inspection).**
- ***Specifications:*** **Can take many different forms but are generally different from stimulus/response behavior.**
- ***General:*** **Can be performed on software, design, requirements, test cases, etc.**
- ***Abstraction:*** **Whereas dynamic testing is on different sizes of software (units, subsystems), static testing is on different levels of abstraction (requirements through detailed implementation).**

Static Testing ...

Complementary to Dynamic Testing

- Dynamic testing is needed because:
  - » Humans can not execute software in their head very fast.
  - » Humans have difficulty managing large numbers of small details.
- Static testing is needed because:
  - » Comprehensive dynamic testing is impossible.
  - » Humans can perform more comprehensive analysis than the checking of individual stimulus/response pairs.
  - » Humans can analyze abstract descriptions (unlike computers).

Static Testing ...

Abstraction and refinement increases human effectiveness in finding errors

- Abstraction
  - » Simplifies system descriptions
  - » Suppresses less important details
  - » Only consider important actions
  - » Consider similar objects identical

Refinement

- » Is the incremental use of abstraction
- » Creates nested levels of description
- » Eases development and comprehension of the three system views

Static Testing ...

Inputs

- Description of the problem to be solved (can be very high level)
- Description of requirements (safety, user interface, etc.)
- Specifications of the item to be statically tested

Implications

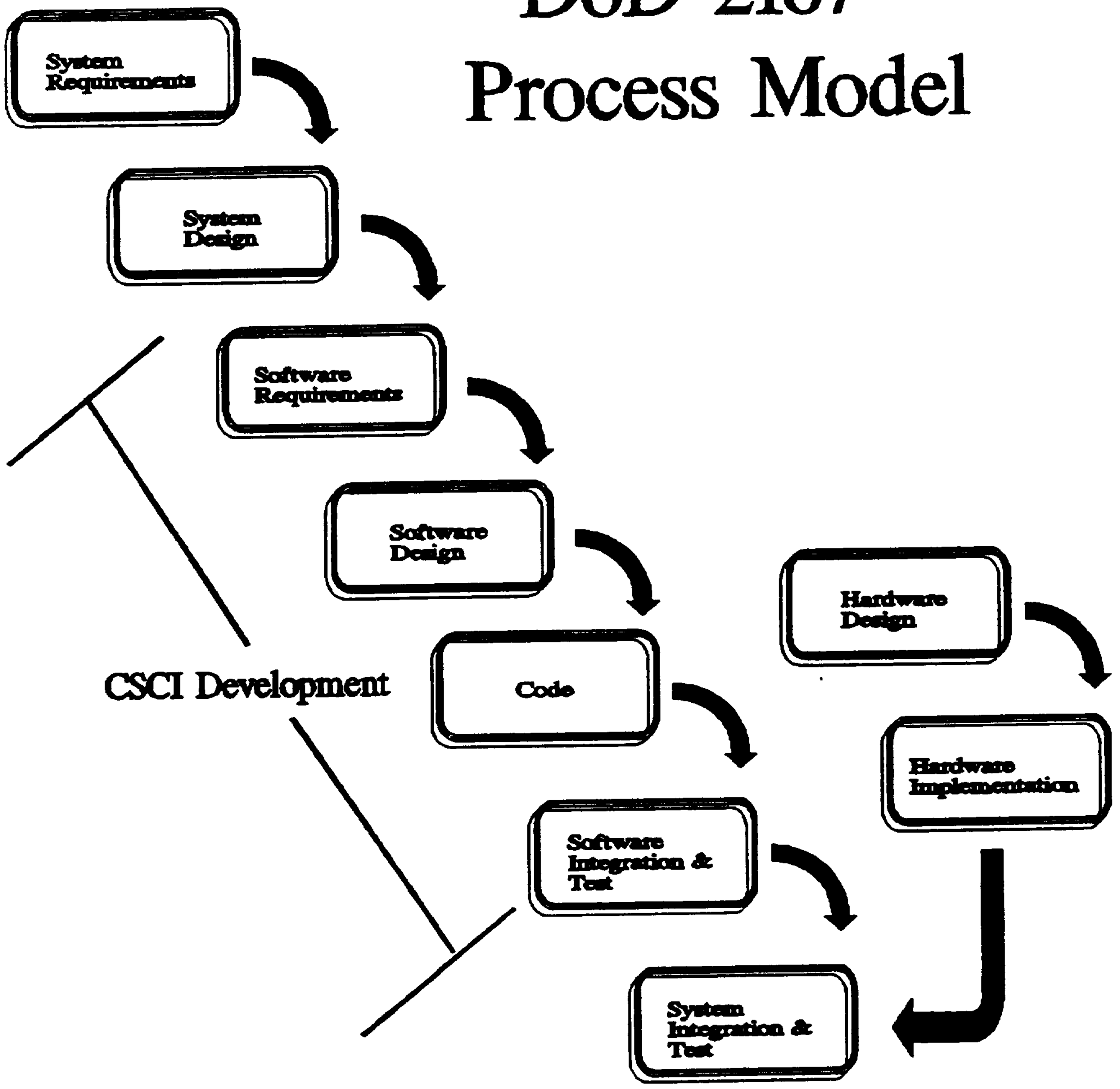
- Can be done *hand-in-hand* with development; this decreases cost.
  - » Not dependant on specific representations of the system
- Natural precursor activity for unit / integration testing.

Life-Cycle Models

- The testing phases are compatible with many standard, well-defined life-cycle models.

Example model : DoD 2167

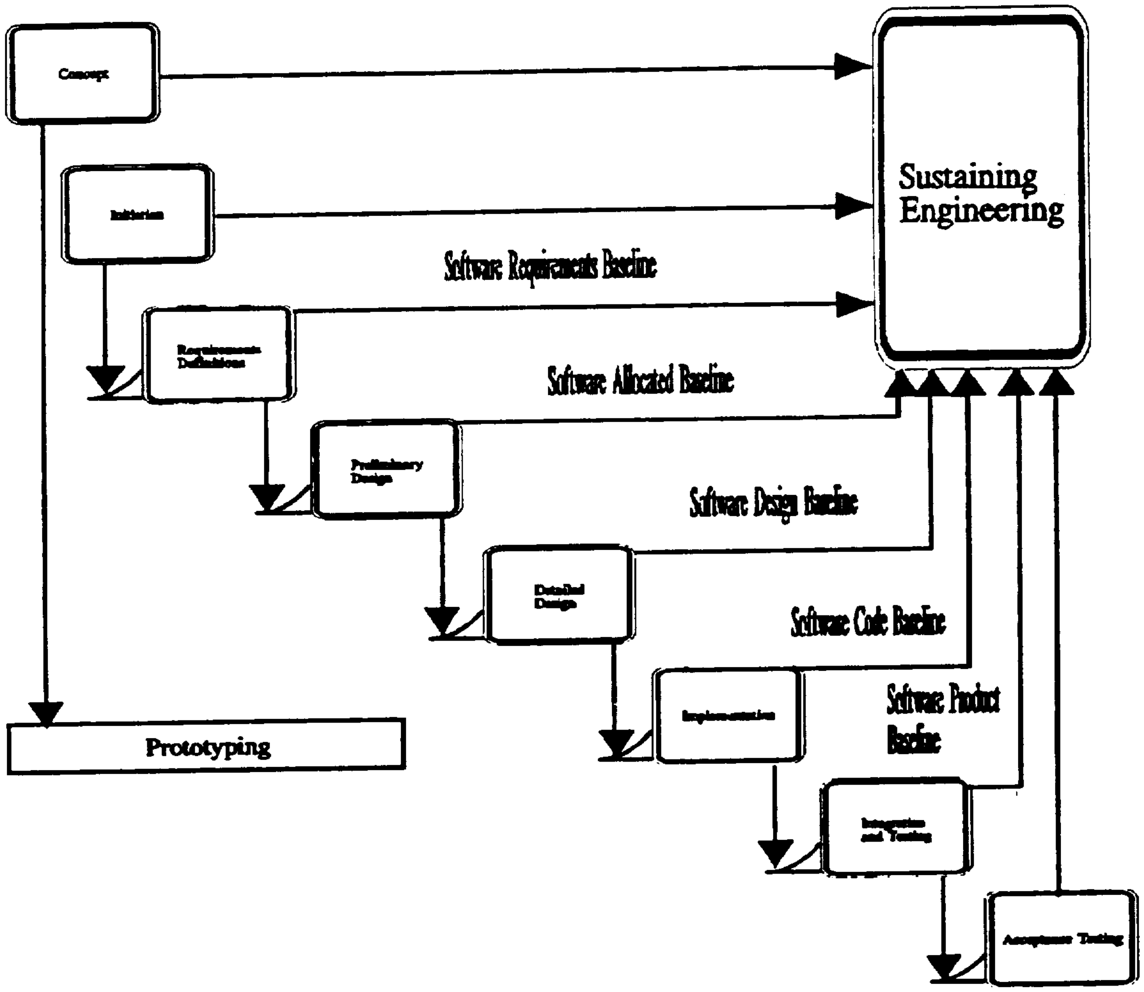
DoD 2167 Process Model



Life Cycle Models ...

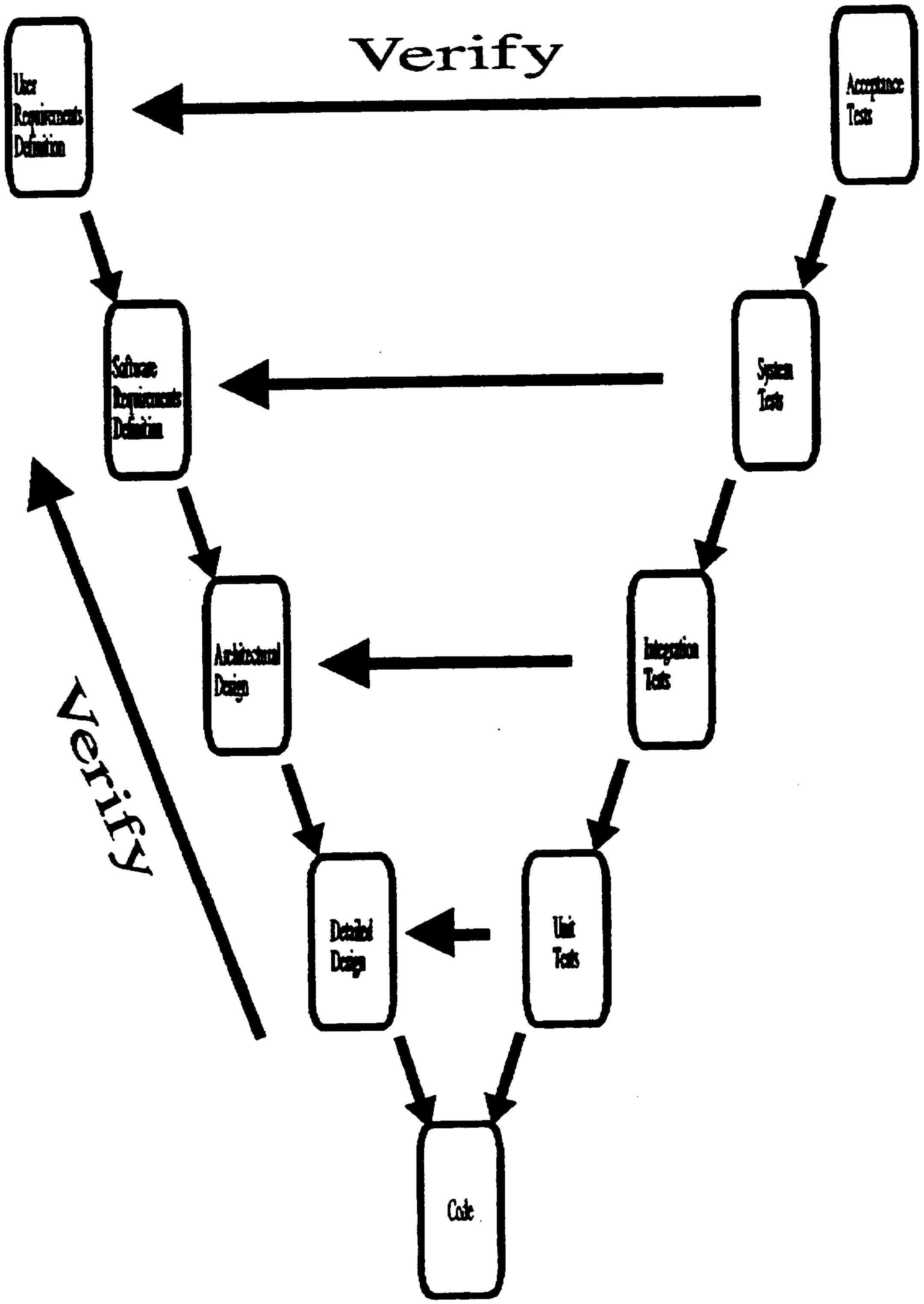
Example model: NASA Model

NASA Life-Cycle Model



Life Cycle Models ...

Example model: European Space Agency Model


Verify
User Requirements Definition
Acceptance Tests
Software Requirements Definition
System Tests
Verify
Architectural Design
Integration Tests
Detailed Design
Unit Tests
Code

Common Misconceptions

Overview

The theoretical foundation of V&V has been presented

Does this foundation still apply for Expert Systems?

There are many misconceptions of V&V

- Some for software in general
- Many for Expert Systems

These misconceptions have impacted the application of V&V

Software in general:

Misconception: The only important deliverable of a software project is the executable version of the program.

Facts:

- Software must be understood by its users.
- Software must be understood by its maintainers.
- Software must be re-tested as it is changed.
- Therefore software should be well-documented and V&V work products (e.g., test cases) should be saved1

Software in general ...

Misconception: Small Prototypes can be scaled up into full-scale solutions.

Facts:

- "The heart of the problem is whether the problem solving method used in a prototype - which solves only a small portion of the problem - will scale up to solve the entire problem"[11]

- "Building large programs is NOT like building small ones and software engineering is different from most other engineering disciplines."[12]

<u>Software in general ...</u>

<u>Misconception</u>: Methodical examination of software is too costly.

<u>Facts</u>:

- Don't confuse rigor with formality
- "... by understanding what would be involved in constructing a formal argument, a programmer can do a far better job constructing a rigorous informal one"[12]

<u>Misconception</u>: Software can be proved correct

<u>Facts</u>:

- One can prove certain properties about software (e.g., the algorithm never results in deadlock)
- One can not prove ***<u>all</u>*** aspects of correctness.

Expert Systems/AI in particular:

Misconception: Expert Systems are Magic (i.e., they are quick and easy to build)

Facts:

- "AI entails massive software engineering."36

- "Software engineering is harder than you think: I can not emphasize strongly enough how true this statement is."36

Expert Systems/AI in particular ...

Misconception: All "expert systems" are expert systems

Facts:

- Just because a program is written in an "expert system language" does not make it (fully) an expert system.
- Just because a program is written in a "conventional language" does not prevent it from being an expert system

Misconception: Expert Systems are all "Expert" Systems.

Facts:

- Most Expert Systems have a significant amount of conventional code/function (survey results indicate at least 45% of the developed system is conventional[16]).

Expert Systems/AI in particular ...

Misconception: The heuristic nature of Expert Systems make them inherently unreliable.

Facts:

- They are still predictable.

- They should be as effective as the heuristic

- They should be safe (i.e., be relied upon not to create a hazard)

Expert Systems/AI in particular ...

Misconception: Learning an Expert System *shell* is all we need to know about Expert Systems.

Facts:

- Knowledge representation (i.e., language) is key to expert systems and V&V of them
- Knowledge acquisition, reasoning paradigms, and software engineering are also needed skills
  - » Domain engineer: knowledge centered
  - » System engineer: computer centered

Expert Systems Differences

Overview

Common software misconceptions impacting V&V have been discussed

Having "cleared the air", we can begin to examine V&V of Expert Systems

There are similarities and differences between Expert Systems and other kinds of software

These similarities and differences impact the V&V approach

To assess this impact, these differences and similarities need to be understood

- Different implementation languages
- Different problem types

Building a foundation for "ES-specific" techniques to be discussed later

Experts systems are software

Expert systems are:

- Computer programs
- Written using a programming language
- Executed in a (deterministic) computer

A program may not be easily classified as conventional or expert system.

- May include some but not all characteristics
- May be part expert system, part conventional

Problems that look expert system may be easily (or better) solved with a conventional solution .

Expert System Implementation Differences

Often uses some type of "AI language", e.g.:

- Forward and/or backward chaining rules
- Frames
- "AI language" characteristics
  - » Declarative (what) instead of imperative (how)
  - » Separation of control and data (i.e., execution sequence is not obvious)
  - » Language semantics unclear or complex (works by "magic", e.g., conflict resolution)

Expert System Implementation Differences ...

Often developed iteratively

- Especially if design by knowledge acquisition
- Especially if it is unclear whether the solution will work satisfactorily

No explicit algorithm is used, e.g.,

```
While . . . Loop
     If . . .
     Then Call . . .
               Exit Loop
      End If
End Loop
```

Expert System Problem Differences

Often solve problems requiring human expertise

- Solution already exists (in someone's head) and is translated to a different form
- e.g., Capturing the "rules of thumb" of an expert and mechanically applying them
- Often called "shallow" or "surface level" reasoning systems
  - » As opposed to model-based (or "deep" reasoning)
  - » Sometimes called "design by knowledge acquisition" as opposed to "design by analysis"

Expert System Problem Differences

...

Expert Systems often solve problems that have been difficult to solve with conventional software approaches

Sometimes rely on human judgment for correctness of solutions (i.e., are "fuzzy")

May replace or just augment human expert

Two Traffic Controller Problems

Overview

A lot has been presented and analyzed

- V&V concepts
- Software misconceptions
- How ES V&V is different

Time to consider an example problem

- Help focus our understanding

Two problems presented

- Simple traffic controller
- "Expert" traffic controller

Later discussion of techniques will refer to these problems

A Simple Traffic Controller Problem

Consider the following problem:

A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every 2 minutes. With a car or pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Exercises

1. Read the problem description.

2. Consider the "testability" of the description

3. Identify key terms from the problem description

4. Construct a black-box view of this system

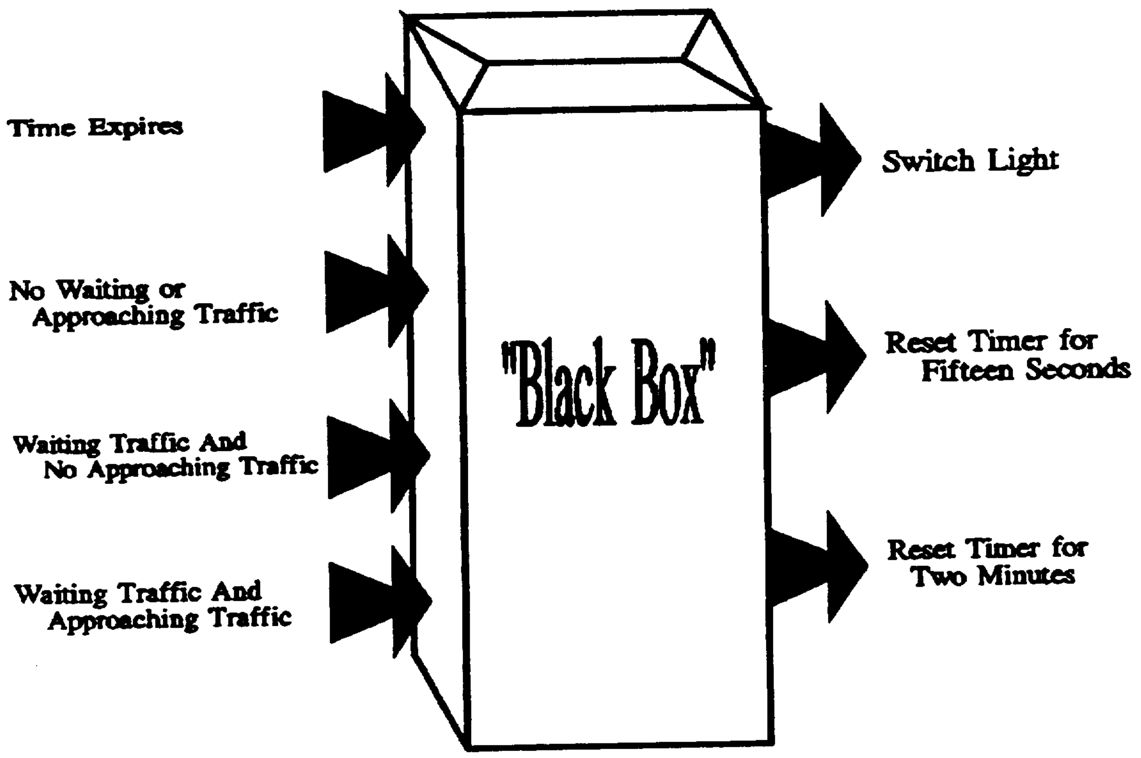
5. Now, compare the "testability" of the problem description to that of the "black-box".

6. Predict the number of scenarios required to comprehensively test the system. Can this be reduced?

7. Exchange your results with a neighbor. How "testable" is their view of the problem?

Black Box View

Initial *black box* view of system testing


Time Expires
No Waiting or Approaching Traffic
Waiting Traffic And No Approaching Traffic
Waiting Traffic And Approaching Traffic
"Black Box"
Switch Light
Reset Timer for Fifteen Seconds
Reset Timer for Two Minutes

Refinement

Refine Requirements based on further understanding of the problem

- State becomes evident
  - » What is the color of the light in a given direction?
  - » How long has the controller waited to switch the light?
- State helps identify and classify stimulus/response histories.
- The state remaining constant might imply testing one scenario verifies the other scenario as well.

Continuing this refinement will lead to a more organized test approach.

- Operational scenarios can be constructed/selected.

Testing "Black Box" Scenarios

Test case scenarios can developed by looking at the "black box"

Consider the following definitions:

switch	traffic light changes
approaching	controller detects traffic in the direction of the green light
waiting	controller detects a waiting auto or pedestrian

Scenarios are defined as ordered pairs:

(t, event)	ordered pair linking elapsed time, t, and an event

NOTE: This is one possible representation of scenarios. Pick one of your own and stick to it!

Testing "Black Box" Scenarios ...

The following scenarios can be generated:

1. (2 minutes, switch)
   (2 minutes, switch) ...
2. (t: t < 2 minutes, approaching)
   (2 minutes, switch)
3. (t: t < 2 minutes, waiting)
   (15 seconds, switch)
4. (t: t < 2 minutes, approaching)
   (t: t < 2 minutes, waiting)
   (15 seconds, switch)
5. ...

List is NOT exhaustive

- Inifinite possibilities

Classes of Scenarios

The notion of testing "classes" reduces the number of scenarios

- Different test cases that exhibit common characteristics
- One test case represents the class

Many options to identifying "classes"

- Based on scenarios
- Based on state

Consider using the scenarios defined

- Each scenario defines a class
- <u>Example</u>: Consider scenario #3

  (t: t < 2 minutes, waiting)
  (15 seconds, switch)

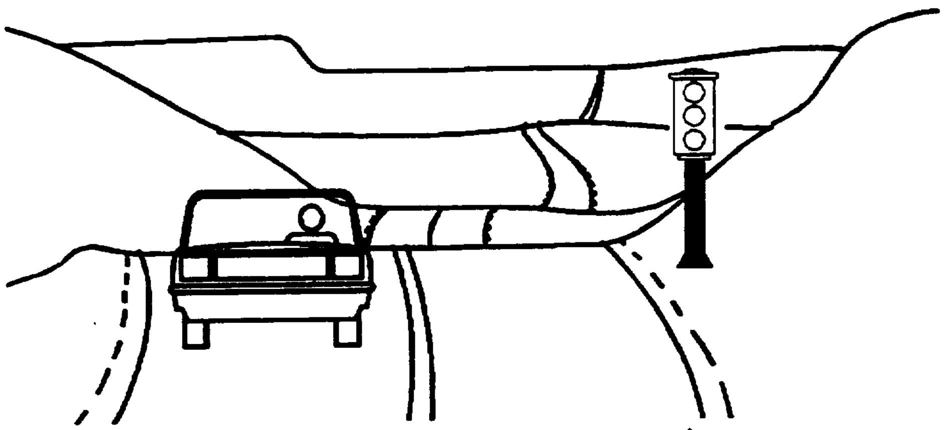
  » Infinite number of values for t

  » Yet, picking any one should be sufficient to test them all

Verification Validation

Classes of Scenarios ...

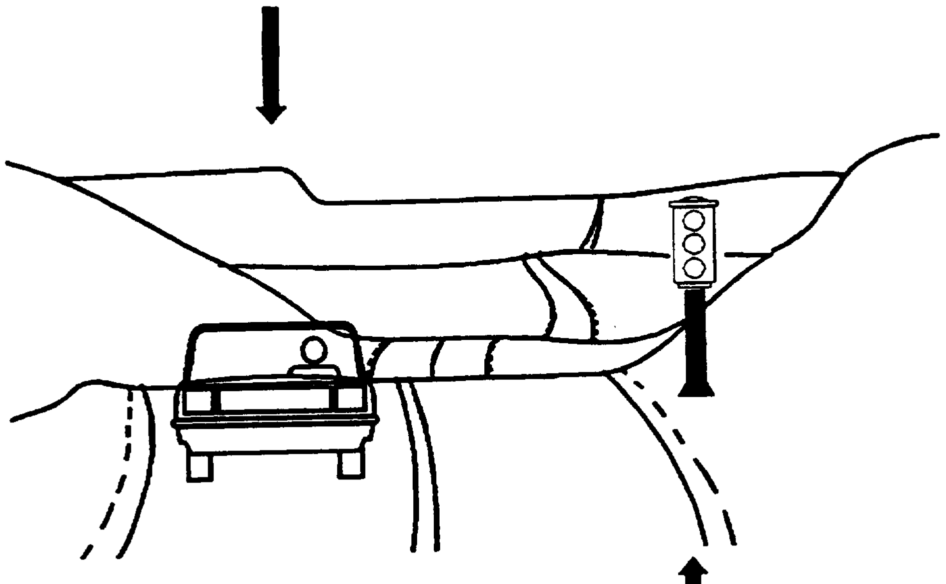
- Car Arrives from the West
- No North-South Traffic for 15 seconds following last signal change



- Switch West-East light to Green

Identification of State

• Car Arrives from the West • No North-South Traffic for 15 seconds following last signal change.	• Pedestrian Arrives from the West • No North-South Traffic for 15 seconds following last signal change



• Switch West-East light to Green

An "Expert" Traffic Controller Problem

Consider the following problem:

At certain times of the day an intersection becomes congested, the electronic traffic light controller becomes inadequate and a policeman is used to direct the traffic. The same policeman has been directing traffic at this intersection for a number of years and there are much fewer complaints from citizens about having to wait at this intersection (than there were several years ago). It is now desirable to make the electronic system "smarter" so it can handle the same amount of flow as the policeman while being as fair as the policeman (i.e., he doesn't force any one direction to wait for a longer time than another direction).

An "Expert" Traffic Controller Problem ...

The new system will function as before when traffic is "light" and will switch to "smart mode" when the traffic becomes heavy. In "smart mode", the system will look at

- the length of traffic in each direction (new sensors will be installed to provide this information)

- the number of people waiting to turn left as opposed to going straight (new sensors will be installed to indicate how many people are waiting in the left turn lane)

- the speed of traffic going through the intersection (new sensors will be installed to provide this information)

Exercises

1. Read the problem description.

2. What are the differences in the two traffic controller problems? Predict the impact to the V&V of the traffic controller.

3. Consider the "testability" of the description

4. Identify key terms from the problem description

5. Construct a black-box view of this system

6. Now, re-consider the "testability" of the problem description.

7. Exchange your results with a neighbor. How "testable" is their view of the problem?

8. Compare this description with that of the first traffic controller? What are the differences? Were they what you expected?

Knowledge Acquisition Results

Initial knowledge acquisition from the policeman reveals the following:

- the policeman walks a beat a few blocks from the intersection and when he hears several horn honks close together, he goes to the intersection to help clear the traffic

- if the line is so long in any direction that he can't see the end of it then he lets those directions (including turning left) go for about three minutes before changing

- otherwise, he lets each direction go for about two minutes, except for turning left which he allows for about one minute

Knowledge Acquisition Results

Initial Knowledge Acquisition ...

- **He lets the longest direction go about half a minute longer than the other directions**
- **If the line waiting to turn left is small when compared to the opposing direction, he will skip them for one cycle (i.e., let each other direction go once more)**
- **If the line waiting to go straight is small, compared to the perpendicular direction, let it go for half a minute less**
- **If you can notice a car that has been waiting for three cycles and has not gone, let that direction go half a minute longer (that line is just moving slow; this roughly corresponds to less than 20 cars per cycle for 3 cycles).**

Exercise

Analyze these high level results

- Look for conflicting statements

- Identify some test scenarios that will determine if this solution seems to satisfy the goals

- Think of some scenarios that this solution does not seem to cover.

- Discuss whether this is an expert system problem or not

Problem Features

Is the solution being created for the first time or does it already exist in someone's head ?

Is it a shallow or deep reasoning solution?

Would this be difficult to solve with conventional software?

Does it rely on human judgment?

Will it replace or augment a human expert?

Two Implementations

Two different traffic controller problems have been analyzed

Expert System problem differences have been studied

What about Expert System implementation differences?

Three implementations of the simple TLC will highlight these differences

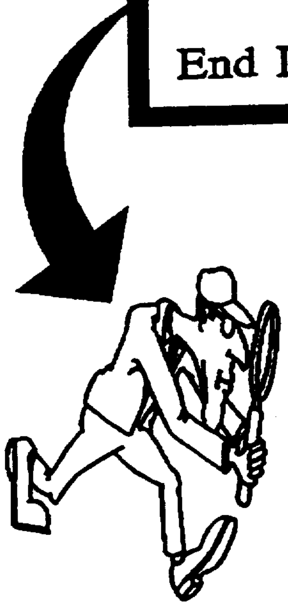
- Two Expert System implementations using a "pseudo" rule-base language
  - » One is well-structured
  - » One is not
- A procedural implementation in a "pseudo" procedural language

Exercises

1. Study the procedural implementation shown in handout #2
   - Consider the state diagram of Handout #1 for this implementation
2. Study the unstructured implementation of Handout #3
3. Study the structured ES implementation shown in Handout #4
4. Define an approach for doing V&V on each solution.
5. Describe how the implementation impacted the V&V approach.

Conventional Implementation

```
Loop
     Case State Is
          When S1 and Time Expires =>
                    State := S1;
           When S1 and (Approaching Or
                Light Changes) =>
                    State := S1;

          •  •  •

       End Case;
End Loop;
```

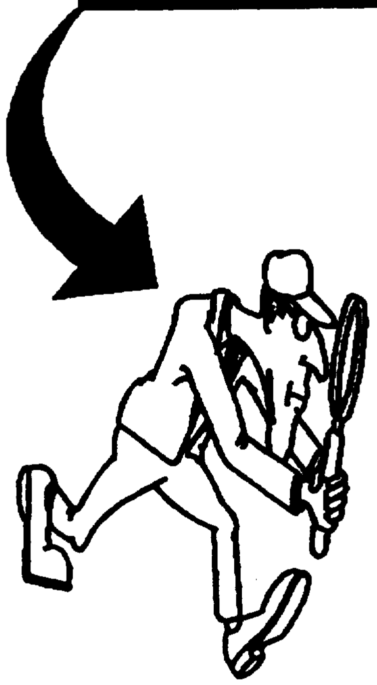


```
Loop
     Case State Is
          When S1 and (NO approaching and
                         NO Waiting) =>
                    State := S2;
           When S1 and (Approaching Or
                Light Changes) =>
                    State := S1;
                     Reset 2 Minute Timer;

          •  •  •

           When S2 and NOT Timer Expired =>
                     State := S2;
            When S2 and Timer Expired =>
                     State := S6;
                      Reset Timer;
                       Switch Light;

          •  •  •

       End Case;
 End Loop;
```

Expert System Implementation

If time expires Then switch light
If in S1 and approaching Then start S1
If in S1 and waiting Then start S2

• • •



If timer expires Then
  switch light and retract timers
If light changes or approaching Then
  Set long timer
If waiting Then
  Set short and medium timers

• • •

Comparison and V&V Implications

Expert System approach turned out to be easier/shorter.

- Production rules directly map to state transitions
  - » if (old state) then (new state) (and action)
- Pattern matching simplified the rules
  - » (3-4 times the number of "whens" as rules).
- Procedural approach wound up implementing a crude inference engine.
  - » A loop with a big nested case statement in it.

Therefore V&V should be easier on expert system implementation, right?

Comparison and V&V Implications ...

Procedural approach has fewer and simpler internal interactions.

- **Execution order of comparisons in procedural approach is very explicit**
  - » ***whens*** **"executed" exactly once per "cycle"**
  - » **as opposed to use of priorities to control execution**
- ***Pure*** **functions (no side effects)**
  - » **Procedure "Change_Light" affects several rules**
- **No "garbage collection" concerns**
  - » **Rule-base implementation must retract old facts**

Therefore, because there are more subtle things to be tested in the expert system approach, it should be harder to V&V, right?

Comparison and V&V Implications ...

Each implementation approach has different V&V concerns.

Procedural concerns

- More decisions to test (more code)
- Overall control structures (e.g., loop termination)

Expert System concerns

- Test correct garbage collection
- Test for invalid rule interactions.
- Must test function side-effects.
- Test that rule patterns are not too broad
- Test that rules only fire at the right time

Comparison and V&V Implications ...

Different concerns need different test approaches/techniques

Both must show a correct solution

Emphasis is different

- e.g., ES must demonstrate no undesirable side-effects

Different view of expert system V&V

- ES failures relate to a different computation model

Kinds of errors humans make:

- Slips/Lapses:(overlapping rule sequences)
- New exceptions:(LHS too broad)
- Erroneous beliefs: (bad rules)

Testing Good and Bad Rule Based Designs

The design of expert systems can greatly simplify the new testing concerns.

The unstructured version (Handout #3):

- Fewer rules
- More complex rules
- Less modular
- More rule interactions
- Has a subtle problem (can you spot it ?)

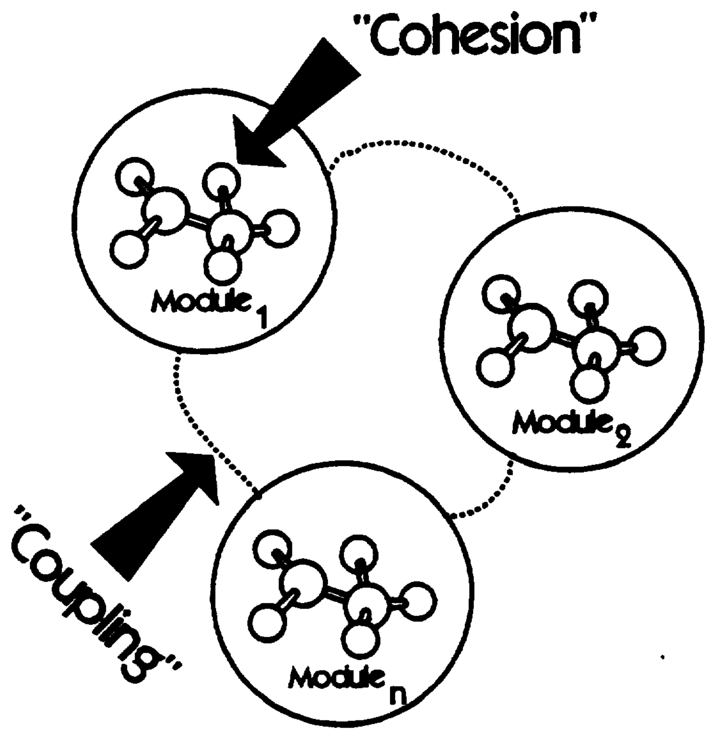
The shorter version is harder to analyze (and thus to verify).

The longer version can be tested in pieces.

Cohesion and Coupling

Cohesion: Connections within a module

Coupling: Connections between modules



Loose coupling reduces interface problems

Strongly cohesive means modules are "atomic" or "primitive"

These are the easiest to V&V

Planning for V&V

Overview

As we will see later, there are many V&V techniques

Ad-hoc application of techniques will make correctness more difficult to assess

We need a plan

- Planning directly <u>impacts</u> V&V
- Planning <u>serves as the framework</u> for the systematic application of V&V techniques
- <u>Therefore</u>, poor planning increases the likelihood that V&V will be ineffective

Before discussing how to plan, let's consider some issues related to planning ...

Issues in Planning for V&V

The following issues represent common pitfalls that can result from poor planning

These issues relate to development of software in general and to the misconceptions previously discussed

However, Expert Systems may be more sensitive to poor planning

- Heavy reliance on experts[16]
- Problems are often ill-defined[4]
- So many projects are only viewed as prototypes (yet, they often become "operational")[16]

<u>Issues in Planning for V&V ...</u>

"Operational" prototypes

- This prototype looks so good, why can't we use it now?
- Unfortunately, small scale solutions rarely <u>scale-up</u> to complete solutions
- Need a defensible development plan

Performing V&V at the end of the development cycle

- Combined Black/White Box testing
- "It is not uncommon to spend 30 to 50 percent of the ... cost ... for the verification effort when using the after-the-fact approach"[15]
- "Testing should be integrated into the development-application cycle"[23]
- Case study #2 resulted from this
- Need a plan for doing V&V "as you go"

Issues in Planning for V&V ...

Unavailability of resources impacting testing

- e.g., special hardware, simulators, expert analysts, etc.
- Without a plan, resources are initially assumed to be available on demand and affordable
- From experience we know they rarely are[3]
- Need a plan for capturing availability and cost of resources

Issues in Planning for V&V ...

Inconsistent/Incomplete/Missing work products[1]

- Estimated 2:1 cost ratio between development and maintenance[5]
  - » Missing work products must be re-created (Reverse Engineering[7])
- Documenting the wrong information
- Inconsistent use of information (conceptual integrity[6])
- Planning should focus on building maintainable systems
  - » kinds of work products, format of products, intended users

Issues in Planning for V&V ...

Implementation approach does not match the problem

- Makes V&V more difficult
- Problem determines the approach
- "Many problems that occur ... are the result of ... generating code without thinking about the design"15
- Plan to follow a logical sequence

Even small design changes result in significant amounts of re-work

- Typical of non-modular systems
- Want to build similar applications from existing "verified" ones
- Plan to minimize re-work (maximize re-use)

Issues in Planning for V&V ...

Inordinately large costs incurred at the end of the development cycle

- "Pay me now or pay me later"
- Difficult to predict end cost
- Maintenance costs can also increase
- Plan to:
  - » Find and correct errors early
  - » Define when to stop testing[37]

Building the wrong user interface

- "... there is now less excuse than ever for not involving users early on ..."[3]
- "The only question is whether you or your customer will discover them (user interface errors)."[31]
- Plan for early user involvement

Issues in Planning for V&V ...

Ineffective testing

- Even worse, minimal insight into why testing was ineffective
- Can result from vague system objectives
- Can result in higher testing costs
- Compounds problem when testing is left to the very end
- Planning will help focus test objectives which drive test selection
  - » "A comprehensive test-management approach recognizes the differences in objectives and strategies of different types of testing"[39]
  - » Define testability[40]

<u>Issues in Planning for V&V ...</u>

We have looked at some key issues related to planning

These issues can help guide us in building a plan

Any V&V plan should consider the following

- <u>In theory</u>, every project has sufficient time and resources to do a competent level of V&V
- <u>In reality</u>, most projects do not achieve this level of V&V because time and resources are constrained
- Planning for V&V can bridge the gap between reality and theory.

Issues in Planning for V&V ...

In summary, a good plan needs to satisfy two goals

- Finding an approach to the problem
  - » Representation vs. Problem
  - » Situation vs. Technique
  - » Technique vs. Representation
- Deciding what you need to do the job
  - » Availability of resources
  - » Realistic schedules and cost
  - » V&V is part of your job
  - » Do not forget maintenance!

With this in mind let's consider a framework for V&V planning

Framework for V&V Planning

Involve user's early in development

- Use prototyping
  - » Provide some results early
  - » Develops problem understanding
  - » Discussed in the "Techniques" section
- Helps define validation testing

Pick a life-cycle model and follow it

- Include the 3 test phases discussed
- Guides the application of techniques
- Helps decide what "work products" to generate

Framework for V&V Planning ...

Plan for an approach that minimizes re-work (i.e., maximizes re-use)

- Decide on an approach for applying modularity

Plan for an approach that matches the problem

- Remember, the goal is to build something that correctly solves a problem
- The goal should never be to build an Expert System

Define what correctness means

- Vague objectives are satisfied by any implementation

Framework for V&V Planning ...

Prioritize the kinds of correctness you wish to demonstrate

- Many kinds of criteria to consider
  - » Complexity, Criticality of the software system
  - » Type of problem to be solved
- Test at the highest levels of priority and work your way down
  - » Framework for applying resources

Framework for V&V Planning ...

Identify areas of risk and a plan to respond to those risks

- Many risks in software development
  - » Changing requirements
  - » Availability of resources
- Assess risks and impacts early

Plan for doing "smarter" testing

- Focus on finding errors early
- Match testing techniques to desired correctness
  - » Will help identify required resources
- Record the plan

Exercise

1. Reconsider the part that planning played in the Apollo 11 scenario

2. Suggest how better planning could have possibly prevented this situation from ever happening

3. Develop your own plan for V&V'ing the Apollo 11 software

4. Develop a plan for Verifying and Validating your team exercise.

Summary of Basic Concepts

Key Points

1. There is a difference between Verification and Validation
   - *Verification*: building the system right
   - *Validation*: building the right system

2. Three important phases to testing software
   - Static: desk checking/code reviews
   - Unit/Integration: testing in pieces
   - System: Overall V&V

3. Test phases have an implied order that can aid in applying V&V
   - Focus on phases that find errors early
   - Pick a life-cycle and follow it

Key Points ...

4. Three main aspects to demonstrating correctness
   - <u>Completeness</u>: Does all it should
   - <u>Consistency</u>: Does it correctly
   - <u>Termination</u>: Output will always be generated for any given input

5. Using abstraction and refinement aids in human analysis of software
   - <u>Abstraction</u>: Suppress details
   - <u>Refinement</u>: Incremental abstraction

Key Points ...

6. **Modularity has many positive benefits for analysis/development of software**

- **Divide and Conquer**
- **Simplifies system comprehension**
- **Aids work-load management**

Common Software Misconceptions

Many misconceptions presented and analyzed

Two categories

- Software in general
  - » Development work products
  - » Use of formality and proofs
- AI/Expert Systems in particular
  - » Expert Systems are "magic"
  - » What constitutes and Expert System
  - » Heuristics

All have negatively impacted V&V

Should not be a roadblock anymore

Expert Systems Differences

Many similarities and differences between Expert Systems and other kinds of software were presented

Similarities:

- Experts systems are software
- Difficult to classify software as conventional or expert system.
- Problems that look expert system may be easily (or better) solved with a conventional solution

Expert Systems Differences ...

Differences fall into two categories

- Implementation
  - » Uses some type of "AI language"
  - » Developed iteratively
  - » No explicit algorithm
- Problem
  - » Often solve problems requiring human expertise
  - » Often solve problems that have been difficult to solve with other approaches
  - » Often rely on human judgement
  - » Focus on replacing or augmenting a human expert

Planning for V&V

The need for planning was discussed

- Planning directly impacts V&V

- Planning serves as the framework for the systematic application of V&V techniques

- Therefore, poor planning increases the likelihood that V&V will be ineffective

Planning for V&V

Examined many issues that can help focus our planning

- "Operational" prototypes
- Performing V&V at the end of the development cycle
- Unavailability of resources impacting testing
- Inconsistent/Incomplete/Missing work products[1]
- Implementation approach does not match the problem
- Even small design changes result in significant amounts of re-work
- Inordinately large costs incurred at the end of the development cycle
- Building the wrong user interface
- Ineffective testing

Planning for V&V

Based on these issues a good plan needs to satisfy two goals

- **Finding an approach to the problem**
  - » **Situation vs. Technique**
  - » **Representation vs. Problem**
- **Deciding what you need to do the job**
  - » **Availability of resources**
  - » **Realistic schedules and cost**
  - » **V&V is part of <u>your</u> job**
  - » **<u>Do not forget maintenance!</u>**

Two Traffic Controller Problems

Two problems presented

- Simple traffic controller
- "Expert" traffic controller

Focus our thoughts on topics discussed

- V&V concepts
- Software misperceptions
- How ES V&V is different

Discussion of techniques will refer to these problems

Appendix A: References

References

1. Baxter, I.D.. "Design Maintenance Systems". Communications of the ACM. April 1992.

2. Beckman, F.S.. *Mathematical Foundations of Programming*. Addison-Wesley Publishing, 1980.

   A complete book that explores the mathematical basis of programming. Issues such computational complexity, grammars, effective procedures, Turing machines, etc. are discussed in some depth. Recommended reading for someone desiring a better understanding of the theory behind programming. This theory helps support many of the approaches to V&V.

3. Behrendt, W., Lambert, S., and Ringland, G.. "An Outline Model for Reasoning about KBS Projects and Development Risks". *Heuristics*. Volume 4 Number 4, pp. 30-38, Winter 1991.

   A short article that lists some interesting things to consider when planning a KBS project. These considerations apply to software in general.

References ...

4. Bell, M.Z.. "Why Expert Systems Fail". *Journal of the Operational Research Society*. Volume 36 Number 7, pp. 613-619, 1985.

5. Boehm, B.. "Industrial Software Metrics Top 10 List". *IEEE Software*. Volume 4 Number 5, pp. 84-85, September 1987.

6. Brooks, F., *The Mythical Man Month*, Addison-Wesley, 1975

   The classic book on software engineering. It is a collection of personal observations on software development. Although the book is many years old, the observations are just as true today as they were 15 years ago. This book is very highly recommended reading.

7. Chikofsky, E.J., and Cross, J.H.. "Reverse Engineering and Design Recovery: A Taxonomy". *IEEE Software*. Volume 7 Number 1, pp. 13-17, January 1990.

   A very good article that explains some the issues involved with trying reverse engineer a system. Highly recommended reading.

References ...

8. Davis, J.S.. "Effect of Modularity on Maintainability of Rule-Based Systems". *International Journal of Man-Machine Studies*. pp. 439-447, 1990.

9. Downs, T. "Reliability Problems in Software Engineering - A Review." *IEEE Software* Volume 2 No. 3 pp. 131-147, July 1987.

10. European Space Agency. *Software Verification and Validation.* Document No. PSS-05-0 Issue 2 p. 2-22, February 1991.

Excerpt from a European Space Agency document outlining their approach to V&V of space software. Input from the Europeans is good because, in many respects, they are ahead of the U.S. in applying V&V approaches.

11. Fox, M.S., "AI and Expert System Myths, Legends, and Facts", *IEEE Expert*, Feb. 1990

Contains personal observations by the author that help explain some causes of ineffective AI applications; many are due to a misunderstanding of AI technology.

References ...

12. Guttag, J.V., "Why Programming is Too Hard and What to Do About It", *Research Directions in Computer Science: An MIT Perspective,* MIT Press, 1991

Contains personal observations by the author on the difficulties in software programs. The author, a respected professor and researcher in software development techniques, offers some very candid opinions in this paper.

13. Hall, A., "Seven Myths of Formal Methods", *IEEE Software*, September, 1990

14. Kamel, R.F.. "Effect of Modularity on System Evolution". *IEEE Software.* pp. 48-54, January 1987.

15. Kemmerer, R.A.. "Integrating Formal Methods into the Development Process". *IEEE Software.* pp. 37-50, September 1990.

16. "KBS V&V - State of the Practice and Implications for V&V Standards"

This paper is included in the references section. It summarizes a survey that was performed of 60 expert system projects to determine what techniques were currently being used to V&V expert systems and what difficulties were being encountered.

References ...

17. Laufmann, S.C., DeVaney, D.M., and Whiting, A.. "A Methodology for Evaluating Potential KBS Applications". *IEEE Expert*. pp. 43-62, December 1990.

This paper provides a detailed checklist for evaluating whether a given application has potential as a KBS.

18. Leveson, N.G.. "Safety." *Aerospace Software Engineering: A Collection of Concepts*. Ed. Christine Anderson and Merlin Dorfman. Volume 136 pp. 319-336, American Institute of Aeronautics and Astronautics, Publisher. 1991.

This and other articles by Leveson, et.al. are easy to read, informative articles discussing, at a high level, issues in demonstrating safety correctness in software.

19. Linger, R.C., Mills H.D. and Witt, E.I.. *Structured Programming: Theory and Practice*. Addison-Wesley Publishing Company 1979.

A text book describing the foundations of structured programming. This book, not only covers the theory behind structured programming, but provides the information needed to apply structured programming.

References ...

20. Leveson, N.G.. "Software Safety in Embedded Computer Systems." *Communications of the ACM*. Volume 34 No. 2, February 1991.

21. Liskov, B. and Guttag, J.. *Abstraction and Specification in Program Development*. McGraw-Hill Book Company 1986.

A complete text book on the use of abstraction and refinement to help in program development. Recommended reading for those who want a thorough understanding of how to use abstraction and refinement as a tool for specifying program behavior.

22. Maibor, D.S.. "The DoD Life Cycle Model." *Aerospace Software Engineering: A Collection of Concepts*. Ed. Christine Anderson and Merlin Dorfman. Volume 136 p. 34, American Institute of Aeronautics and Astronautics, Publisher. 1991.

23. Marcot, Bruce. "Testing Your Knowledge Base." *AI Expert*, July 1987

This article offers some practical advice for testing knowledge bases by listing some very general guidelines. It also has a good detailed list of types of correctness.

References ...

24. Miller, L.A.. "Dynamic Testing of Knowledge Based Systems Using the Heuristic Testing Approach". *Expert Systems with Applications*, Volume 1 Number 3, 1990.

A good article that describes the prioritization approach to planning a test approach.

25. Mills, H.D.. "Structured Programming: Retrospect and Prospect." *IEEE Software* Volume 3 No. 6, November 1986.

The Mills, Myers, and Parnas references provide a thorough understanding of the use of modularity in program development. Not only as a tool for easing development, but also as a foundation for demonstrating program correctness. These are **classics** in the Software Engineering field.

26. Mills, H.D., Linger, R.C. and Hevner, A.R.. "Box Structured Information Systems." *IBM Systems Journal* Volume 26 No. 4, 1987.

27. Mills, H.D., Linger, R.C. and Hevner, A.R.. *Principles of Information Systems Analysis and Design.* Academic Press, Inc. 1986.

References ...

28. Myers, G.J.. *Software Reliability Principles and Practices.* John Wiley & Sons, Publishing 1976.

29. Myers, G.J.. *Reliable Software Through Composite Design.* Mason/Charter Publishers 1975.

30. Myers, G.J.. *Composite/Structured Design.* Litton Educational Publishing 1978.

31. Nielsen, J.. "Big Paybacks from 'Discount' Usability Engineering". IEEE Software. Volume 7 Number 3, pp. 107-108, May, 1990.

32. Parnas, D.. *Software Engineering Principles.* Department of Computer Science, University of Victoria. Report No. DCS-29-IR, February 1983.

This reference and others by Parnas represent classic work done by this respected practitioner of software engineering. These are highly recommended reading.

33. Parnas, D.. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM* Volume 15 No. 12, pp. 1053-1058, December 1972.

References ...

34. Parnas, D.L., Clements, P.C., "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering*, Feb. 1986

Describes why one would wish to document a product as if it were designed according to an idealized development process/methodology, even if was developed in a very ad-hoc manner. Also includes suggestions on what the documentation of a product should contain.

35. Rumbaugh, J.., *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

36. Schank, R.C., "Where's the AI ?", *AI Magazine*, Winter 1991

A very readable description of some personal observations by the author on some difficulties in developing truly intelligent systems. This article is highly recommended reading.

37. Sherer, S.A.. "A Cost-Effective Approach to Testing". *IEEE Software*. pp. 34-40, March 1991.

References ...

38. Stevens, W.P. and Myers, G.J. and Constantine, L.L.. "Structured Design". *IBM Systems Journal* Number 2 pp. 115-139, 1974.

The classic paper on modularity.

39. Wallace, D.R. and Fujii, R.U.. "Software Verification and Validation." *IEEE Software* Volume 6 No. 3 pp. 10-17, May 1989.

A very good article stating high level objectives and techniques for verifying and validating conventional software.

40. Voas, J., Morell, L. and Miller, K.. "Predicting Where Faults Can Hide from Testing". *IEEE Software*. pp. 41-48, March 1991.

41. Wilson, W.M.. "NASA Life Cycle Model." *Aerospace Software Engineering: A Collection of Concepts*. Ed. Christine Anderson and Merlin Dorfman. Volume 136 pp. 319-336, American Institute of Aeronautics and Astronautics, Publisher. 1991.

Workshop On Verification and Validation of Expert Systems

Techniques

Authors:

Scott W. French
FRENCHS@HOUVMSCC.VNET.IBM.COM

David Hamilton
HAMILTON@HOUVMSCC.VNET.IBM.COM

IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058

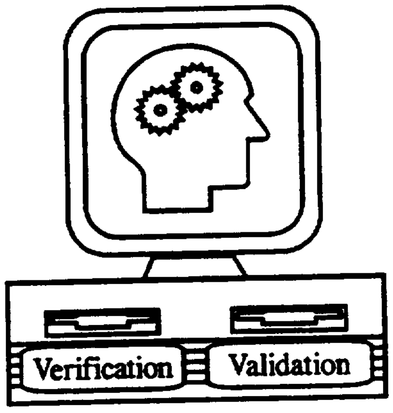


Table of Contents

Table of Contents ...

Introduction

Overview

This section will summarize some key techniques

- There are others
- Those presented are some of the best
- Applicability to ES and/or Conventional software addressed

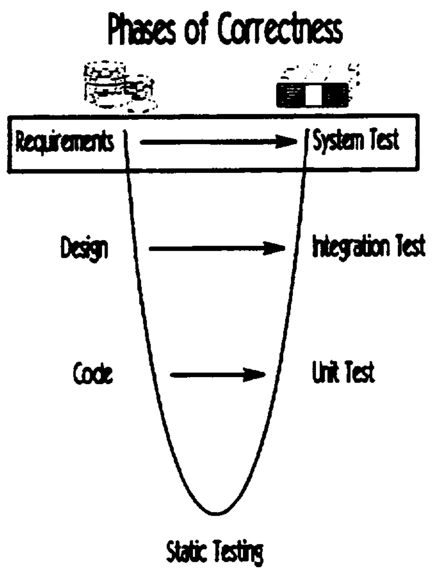
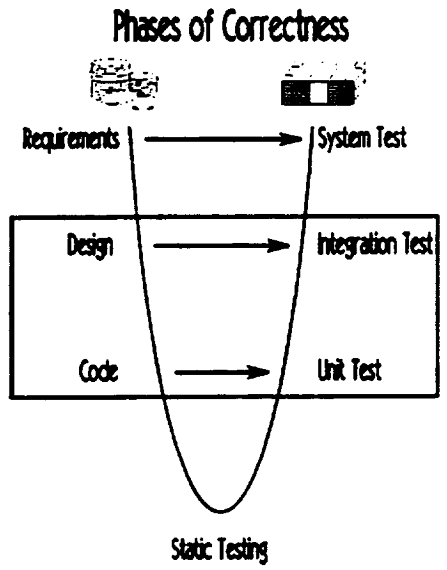
Each technique will be discussed in terms of

- Over all description
- Implementation
- Error detection capability
- Available tools
- Examples based on the Traffic Light problem

Overview ...

Techniques will be grouped by test phases where they apply

- One exception: some important techniques are applicable to many phases
- These are categorized as General Techniques

System Testing	Unit / Integration Testing
 Phases of Correctness Requirements System Test Design Integration Test Code Unit Test Static Testing	 Phases of Correctness Requirements System Test Design Integration Test Code Unit Test Static Testing

General Techniques

General Techniques

Regression Testing

Typically a maintenance activity

Requires some process for capturing and retrieving test cases

Example: Change the controller so waiting traffic can wait up to 1.5 minutes

- Scenarios with no waiting traffic for 2 minutes should work as before.

Prototyping

Develop a working model to test aspects of requirements or design

E.g., prototyping might reveal the need for a yellow light.

General Techniques ...

Competing Designs

Define multiple *design* teams

Each team designs a solution

Select the best or merge solutions

Independent V&V

Pick an independent team to perform V&V on the software

Independence avoids potential bias

Applicable anywhere in the process

- Commonly applied at System Test

General Techniques ...

Inspections

Review of work products

Formal/Informal (or walkthrough)

- Follows a set of rules governing review
- Many roles

Continuous inspections

- Frequent review of smaller items
- Best approach when applying stepwise refinement

Major advance in the practice of V&V

- Creates "active verification frame of mind"
- An estimated 60% of errors can be found during inspection[4]

General Techniques ...

Decision Tables

Very popular in the early and mid '70s

Originally considered a complete development methodology

Really is a specification approach

Very similar to rule-based programming

» Left side := *condition* columns

» Right side := *action* columns

» A row is called a rule

Has some differences from rule-based programming

» No pattern matching or unification

» No chain of inference

General Techniques ...

Decision Tables ...

Completeness checking

- **Figure total number of rules**
  - » **Product of number of possible entries in each column**
- **Ensure each rule is considered**

Consistency checking similar to rule consistency checking

- **Redundancy, overlapping rules**
- **Contradictory rules**
- **etc.**

General Techniques ...

Decision Tables ...

Example: Complete TLC solution ($2^5*6=192$ rules; see handout #1)

Approaching Vehicle	Waiting Vehicle	2 Min Timer Expires	1 Min Timer Expires	15 Sec Timer Expires	Current State	New State	Change Light
0	0	0	0	0	1	1	0
1	0	0	0	0	1	1	0
0	1	0	0	0	1	3	0
...	...	...	...	...	...	...	...

Example: Abstract TLC solution ($2^5*2=64$ rules; see handout #1)

Approaching Vehicle	Waiting Vehicle	2 Min Timer Expires	1 Min Timer Expires	15 Sec Timer Expires	Current State	New State	Change Light
0	0	0	0	0	1	0	1
1	0	0	0	0	1	1	0
0	1	0	0	0	1	2	0
...	...	...	...	...	...	...	...

General Techniques ...

Decision Tables ...

Practical and effective if used on small modules

Example: Timer module ($2^3 = 8$ rules)

Set for 99999	Expired	Error	Expires= True	Set Time	Print Message
0	0	0	0	0	0
1	0	0	0	1	0
0	1	0	1	0	0
1	1	0	?	?	?
0	0	1	0	0	1
1	0	1	0	0	1
0	1	1	?	?	?
1	1	1	?	?	?

Class Exercise: Answer the following

What action do you think should be in the "question mark" rule entries?

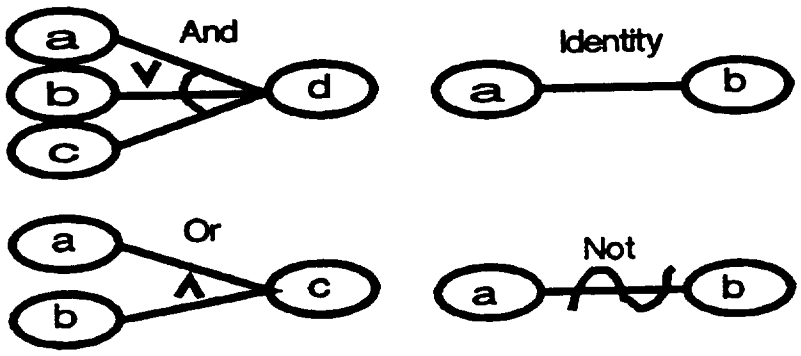
- **What does the Timer module really do?**

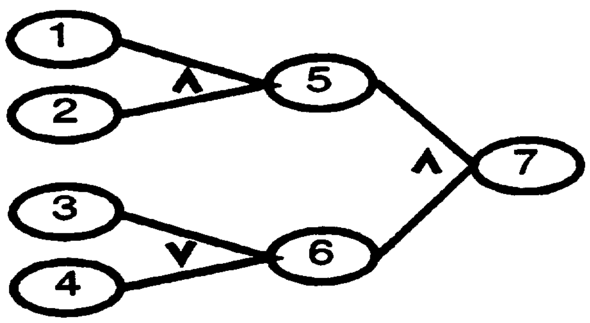
General Techniques ...

Cause-Effect Graphing

Technique for selecting tests that exercise combinations of causes

Highlights *interesting* cases





Highlights useful abstractions

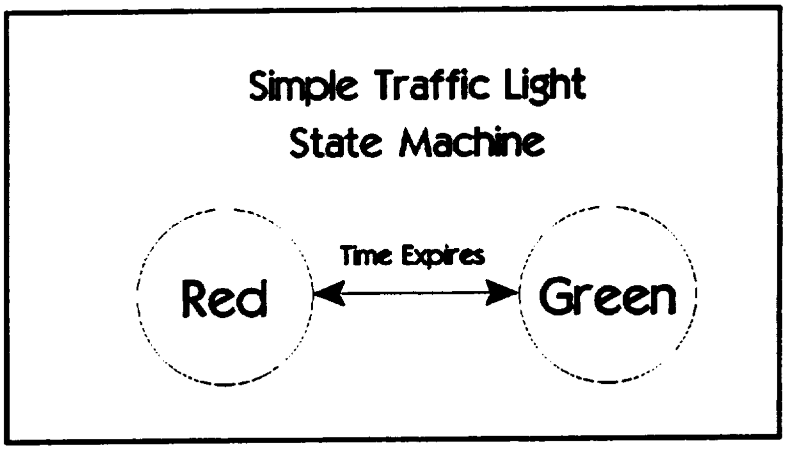
04/21/92

General Techniques ...

State Diagrams

Views a system as state and transitions

States are "nodes" and transitions are "arcs"



Transitions map to causing conditions

NOTE: Helps analyze abstract system behavior during system test

General Techniques ...

State Diagrams ...

Translates to a matrix

Place a 1 at each (S_i, S_j) where S_i -> S_j

	Red	Green
Red	0	1
Green	1	0

Highlights interesting system properties

Sensitivity Analysis

Assess system sensitivity to change

"Graphing" techniques are helpful

Primarily an analysis technique as opposed error finding technique

Supported only by a research tool

Directly benefits classification problems

General Techniques ...

Testability Analysis

Estimating the presence of "hidden" faults

- "If the presence of faults in programs guaranteed program failure, every program would be highly testable."49

Three main parts to the estimation

- Execution Analysis: Probability a given component is executed
- Infection Analysis: Probability that a component is sensitive to errors
- Propagation Analysis: Probability that an "infected" component will affect "what the user sees"

Low numbers imply low testability

- Infer larger dynamic test costs
- Static testing would be beneficial

General Techniques ...

Testability Analysis ...

Approach to Execution analysis

- Run random cases
- Count component executions
- Example: Consider rules Update_Time and Del_Old_Changes from Handout #3
  - » Update_Time has a ratio of 1
  - » Del_Old_Changes has a ratio of 0
  - » Which is more "testable"?

General Techniques ...

Testability Analysis ...

Approach to Infection analysis

- Build component mutations
- Apply to mutant and non-mutant
- Count # of different results
- Example: Consider mutations of Update_Time and Del_Old_Changes
  - » Del_Old_Changes ratio is near 0 (why?)
  - » Update_Time ratio is near 1 (why?)

General Techniques ...

Testability Analysis ...

Approach to Propagation Analysis

- "Break" after component executes
- Change data state and continue
- Look for cases with different results
- Example:
  - » Set breakpoints after Update_Time and Del_Old_Changes
  - » Perturb the fact list
  - » What effects would you expect in the result?

Exercises

1. Define the "black box" view for your team exercise.
2. Identify key terms from the problem description.
3. Which of the following techniques would you use? Explain your answer.
    - Prototyping
    - Competing Designs
    - Independent V&V
    - Inspections
4. Do a very high level specification for your system using one of the following techniques
    - Decision Table
    - Cause-Effect Graph
    - State Diagram

System Testing Techniques

System Testing Techniques

Realistic Testing

Focus on those functions used the most.

Realistically, more autos wait than pedestrians.

- Therefore, select scenarios that involve waiting autos.

Attribute-based Test Case Selection

Choose test cases based on an attributes

- Complexity, Criticality, Reliability, ...

Tests can be chosen according to

- Statistical Record-keeping
- Random
- Error Guessing

Example: More complex scenarios include both waiting and approaching traffic.

System Testing Techniques

Boundary-Value Testing

Identifies cases at the boundaries of each stimulus/response class

Example, Approaching traffic is detected at the same instant a timer expires

- Exercises the *boundary* value of when the timer should expire

Stress Testing

Choose "*off-nominal*" tests to test safe operation in stressful/critical situations.

Examples:

- Pedestrian repeatedly hits the *change signal* button?
- Power surge occurs when a car trips the *change signal* button?

System Testing Techniques ...

Active Interface Testing

Test the interface to an external agent (e.g., a person)

Examples:

- **Auto weight required to trip signal**
- **Pedestrian signal button sticks**

Performance Testing

Choosing tests that "push the envelope" (speed, accuracy, etc.)

Examples:

- **Effect of hardware delays**
  - » **e.g.,Signal tripped at t_0+14.999**

System Testing Techniques ...

Knowledge Acquisition Correctness Checking

Looking for inconsistencies and "holes" in knowledge acquired from the expert.

Similar to analyzing system requirements.

Made easier by representing the knowledge in a consistently structured form.

Example: How does the expert traffic controller know when to stop and go back to conventional mode?

System Testing Techniques ...

Knowledge Acquisition Correctness Checking ...

Consider the following approach for finding inconsistencies and "holes":

- Only do things that "make sense"
- Ensure proper sequence
- Example: Checking both timers when only one should be checked

Step 1: Verify no conflicting sequences

- Rules, questions, facts, etc.
- Build a matrix mapping these (e.g., a rule-to-rule matrix)
- Mark possible conflicts with an "X"
  - » Checking both timers can result in changing the light at the wrong time

System Testing Techniques ...

Knowledge Acquisition Correctness Checking ...

Step 2: Establish "master"/"subject" relationships to resolve conflicts

- E.g., determine what should happen first
- Example: Do not use the short timer unless traffic is waiting

Step 3: Compare "legal" values of "master" to "utilized" values of subject

- "Legal" values = all possible values
- "Utilized" values = values used
- "Subject" vs. legal values of "master"
  - » A matrix with an "X" for each conflict

Helps build "sequence" expressions

System Testing Techniques ...

Minimum Competency Testing

Certifying the competency of an expert system

- "Test" as would a human expert

Certification exams exist for many types of human experts.

- CPA, MD, PE

Assumes the ES will make same errors as the expert

Expert can be asked to identify abilities of a novice, advanced beginner, etc.

Similar to statistical testing (exam is a representative sample)

Discussion: Develop a certification test for the "expert" traffic controller.

System Testing Techniques ...

Disaster Testing

Identify scenarios that indicate potential disaster (during knowledge acquisition)

- **Experts are often good at recognizing potential disasters**
- **Many disaster situations are "common sense"**

Generate tests to check that the system responds to potential disasters

- **Use with specification-directed verification (disaster = specification).**
- **Example TLC disaster: Light is red in all directions**

System Testing Techniques ...

Expert Review

Some answers can only be judged correct by the expert.

Experts can check test scenarios/results

Expert may not understand implementation details

With minimal training, an expert can check

- Acquired knowledge
  - » Miscommunication
  - » Gaps in the knowledge
- Knowledge base design
  - » Correct approach
  - » Correct interpretations

Format the review material so the expert can easily understand it.

System Testing Techniques ...

Explicit Modelling

Different kinds of models:

- Set of equations
- Small scale replica (e.g., toy airplane model)
- Metaphor (i.e., making analogy)
- Any simplified representation of a system

"Instead of having no models in a KBS, there are often a multitude of unexpressed models;"2

System Testing Techniques ...

Explicit Modelling ...

Different people may each have a different model for the same system (but should all be consistent)

- Client (e.g., traffic control system)
- User (e.g., traffic light switching system)
- Developer (e.g., state machine)

Helps with V&V by facilitating abstraction

Leads into model-based reasoning[50]

System Testing Techniques ...

Explicit Modelling ...

The concept of modelling is straightforward, practice can be difficult

- Identifying a suitable model
- Mapping the model to the system
- Reasoning about the model

However difficult, it is usually worthwhile

- Models are always created[14]. They are often implicit (not documented).
- An explicit model can make the system easier to understand; this helps all aspects of development and use.

System Testing Techniques ...

Explicit Modelling ...

Example: Timer module

- Timers are countdown clocks with alarms
- Asserting a timer creates a new clock which begins to count down to zero
- Alarm goes off when the clock counts down to zero

Example: CLIPS inference engine

- There are 2 lists of rules: KB and agenda.
- There is a list of facts.
  - Each cycle, the inference engine goes through the KB list and the fact list, picking rules to put on the agenda.

Exercises

1. Define 1 or more "realistic" test cases for your team exercise
2. Define some attributes of your system. Define 1 or more test cases based on the attributes you defined.
3. Define 1 or more test cases that do "boundary value" testing.
4. Define 1 or more test cases that "stress" the system.
5. Define the external interfaces to your system. Define 1 or more test cases to test those interfaces.
6. Define 1 or more test cases to test the system's performance.
7. For each question, indicate how the results of each test case will be analyzed (i.e., how you will know the answer is correct).

Exercises

8. Did the problem description provide enough detail to adequately perform the tests from questions 1-6.

9. Develop a certification test for your system.

10. Identify system "disasters" (i.e., things that should not happen). Explain how you will test your system for these "disasters".

11. Will your project need the aid of an expert (provide rationale)? If so, indicate the kind of expert required and the type of analysis to be performed.

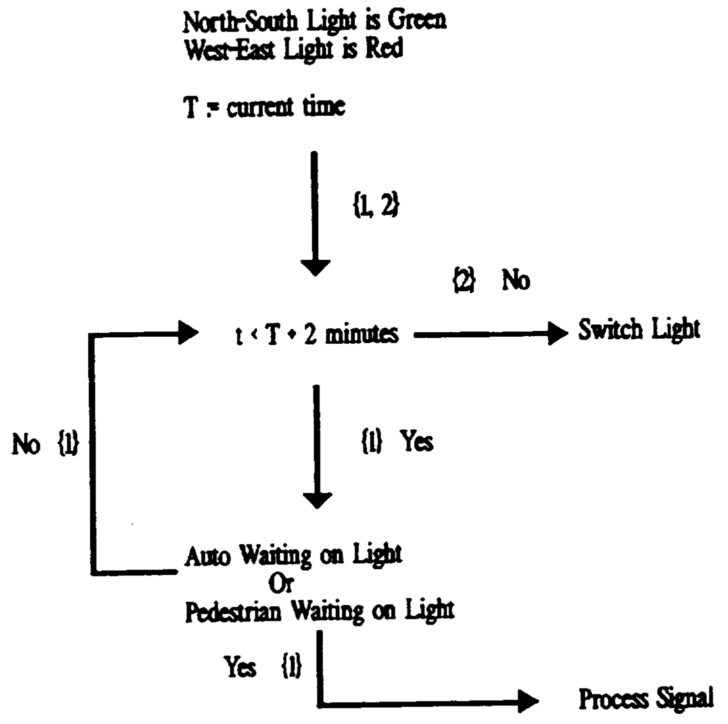
12. Define 1 or more models to aid in your understanding of the system. Document each model.

Unit/Integration Testing Techniques

Unit/Integration Testing Techniques

Branch Coverage

Choosing tests that will cover all possible outcomes of each internal logical decision (e.g., if-then-else)

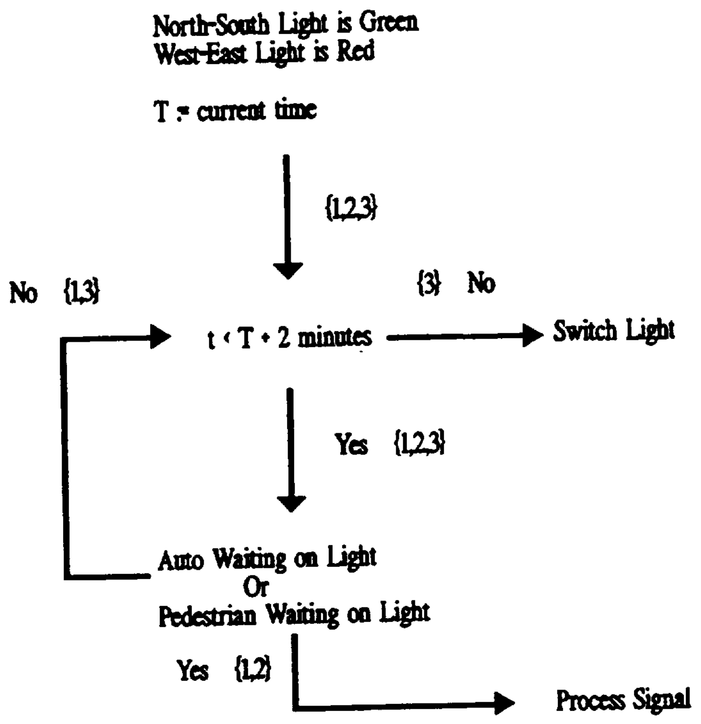


Coverage techniques assume a different meaning for Expert Systems

Unit/Integration Testing Techniques...

Path Coverage

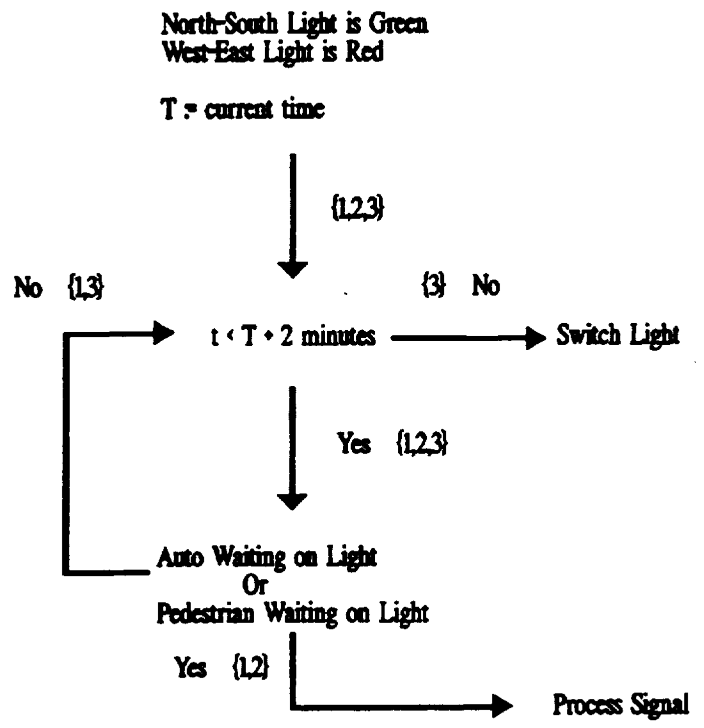
Choosing tests that will cover all possible combinations of outcomes of each internal logical decision



Unit/Integration Testing Techniques...

Condition Coverage

Choosing tests that will cover all possible situations that could lead to an internal logical decision choice



Unit/Integration Testing Techniques...

InterProcedural Dataflow Testing

Focuses on coverage testing for areas where units interact

- Look at Global data and Passed Parameters

Involves Building a Definition/Use Table

- Identifies pairs of statements for each variable based on definition and use

Can be complex to build without some automated assistance

Unit/Integration Testing Techniques...

InterProcedural Dataflow Testing ...

Assume the following procedure for handling the timing when traffic is waiting

```
1.    Procedure Process_Signal(Switch_At) Is
2.          Switch_At := Clock.Current+15
3.          Time_Limit := Clock.Current+60
4.    Begin
5.          While Clock.Current < Switch_At Loop
6.                If Approaching_Signal Then
7.                      If Clock.Current+15>Time_Limit
                        Then
8.                            Switch_At := Time_Limit;
                        Else
9.                            Switch_At := Clock.Current+15;
10.                     End If;
11.               End If;
12.               Clock.Tick;
13.   End Process_Signal;
```

Unit/Integration Testing Techniques...

Step 1: Find interface and global variables

- Clock.Current, Clock.Tick query and pulse the clock
- Approaching_Signal senses approaching traffic
- Switch_At

Step 2: Build a Definition/Use table for these items

Definition/Use Table for Process_Signal		
Variable	**Definition**	**Use**
Approaching_Signal		6
Clock.Tick		12
Clock.Current		2,3,5,7,9
Switch_At	2	5,8,9

Step 3: Select test cases that exercise these statements

Unit/Integration Testing Techniques...

Flavor Analysis

Attempts to find errors of omission

Documents:

- **expected sequences of actions**
- **assertions about the effects of a piece of code**

Methods:

- **Data Comments: documents abstractions used in program construction**
- **Operator Comments: documents a legal "ordering" of operators**

Goal: Compare actual execution against expectations

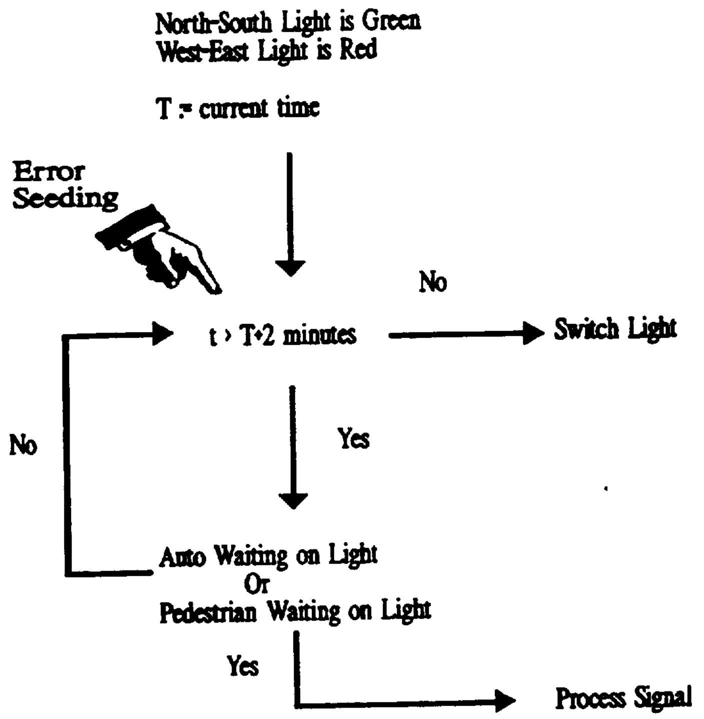
(

Unit/Integration Testing Techniques...

Mutation Testing

"Seed" a program with errors

Evaluates effectiveness of test cases



Unit/Integration Testing Techniques...

Reliability Testing

Identify structures that could adversely affect system reliability if they fail

- **Are not necessarily error-prone**

For example, the system *clock*.

Prototype Evaluation

Test the user-interface pieces of the system early

Involves either *stubbing out* some pieces of the system or developing a simulation

For example: simulate signal hardware so the traffic light software can be prototyped.

Unit/Integration Testing Techniques...

Structural Testing

Goal: Comprehensive testing by executing all parts of a knowledge base

Adaptable to cover any ES representation

Commercial tools available but are not widely used (e.g., Expert/Measure)

Exercise: generate test cases for modular TLC solution that cover:

- each rule
- each path from update_time to timer_expires
- an assertion and a retraction of at least one instance of each fact template

Exercises

1. Pick an implementation approach for your problem. Based on this choice, would you use:

   - Coverage techniques
   - Interprocedural data flow analysis

   Provide rationale for your choices. If you select more than one technique, then prioritize them in order of importance to your testing approach.

2. Identify "parts" of the system that may impact reliability (HINT: you may have to define what reliability is). Define 1 or more test cases to test those "parts".

3. Document 1 or more expected sequences of actions for your problem.

Exercises ...

4. Is prototype evaluation appropriate for your problem? What about mutation testing? Provide rationale.

5. Exchange your work with another team. Study the problem. Ask yourself the following:

   - Does their implementation approach match the problem?
   - Are there any "holes" or inconsistencies in their descriptions?
   - Did they pick the right techniques for their implementation approach?

Static Testing

Static Testing Techniques

Anomaly Analysis

- Involves looking at sequences of events for certain types of "anomalies".
  - » Data flow anomalies such as "use-set" and "set-set-use"
  - » Physical units mismatch such as "length * volume"
- Examples:
  - » After a light change, the clock counter is referenced before it is reset
  - » There is an expression involving "light color multiplied by time" which doesn't make sense

Static Testing Techniques ...

Object-Oriented Analysis

- Object = set of data + associated operations.
- The set of data has certain "legal" values.
  - Each operator accepts data with only certain values.
  - Analysis involves checking that no combination of operators will result in a data item getting an illegal value or an operator being called with an illegal input.
  - Analysis will assure that the object can never be put in an "illegal" state.
- Objects can be mapped to classes of scenarios.

Static Testing Techniques ...

Object-Oriented Analysis ...

- Example:
  - » Time_counter is an object
  - » Time_counter should never be negative
  - » Reset and decrement are operators on time_counter
  - » Reset sets time to 120
  - » Decrement decreases time_counter by 1 if time_counter is greater than zero, otherwise it does-nothing to time_counter
  - » Time_counter can be shown to be guaranteed to always be non-negative

Static Testing Techniques ...

Compilation Testing

- For some languages, such as Ada, the compiler can detect some kinds of errors in the architecture of software

Defect Analysis

- Involves identifying kinds of *common errors* such as *divide by zero*

- Checking for instances of these *common errors*

Static Testing Techniques ...

Axiomatic Analysis

<Pre-Condition>
... code fragment ...
<Post-Condition>

Given the pre-condition is TRUE

- Is post-condition TRUE after execution of the code fragment

Given a combination of fragments

- Post-condition matches next pre-condition

Can also be general conditions that apply to the system as a whole

- E.g., "The traffic light can only be green in one direction (NS or EW)"
- Not tied directly to a specific code fragment

Static Testing Techniques ...

Stepwise Refinement

Separating a unit into equivalent descriptions at varying levels of detail

Analyze by comparing each level of detail to the preceeding one,

- Consistency/completeness checks

Symbolic Execution

Formal program proving technique

Traces program execution to prove program properties

- Can help do axiomatic analysis and/or stepwise refinement

Uses symbols act as placeholders for real values (similar to classes)

Static Testing Techniques ...

Symbolic Execution ...

Consider the following procedure that determines when to switch the light

```
Procedure Process_Signal
$T_l := T_C + 60$;
$T_S := T_C+15$;
While $T_C < T_S$ Loop
        --<*$T_C < T_S$ And $T_S <= T_l$ And $T_C < T_l$ *>
        If Approaching_Traffic Then
                If $T_C+15 > T_l$
                Then $T_S := T_l$;
                Else $T_S := T_C+15$;
                End If;
        End If;
        --<* $T_S <= T_l$ And $T_C < T_S$ *>
        $T_C := T_C + 1$
End Loop;
--<* $T_C = T_S$ And $T_S <= T_l$ *>
```

<u>Static Testing Techniques ...</u>

<u>Symbolic Execution ...</u>

<u>Step 1</u>: Define program properties to be proved

- Lines 4, 11, and 14

<u>Step 2</u>: Build a graph of program flow

- Helpful to build smaller "sub" graphs
  - » Easier with pre/post conditions
- Framework for trace

<u>Step 3</u>: Trace program execution, proving properties "as you go"

- See Handout #6
- <u>Exercise</u>: Fill in the missing parts of the proof in Handout #6

Static Testing Techniques ...

Hazard Analysis

- Hazard: very undesirable situation
  - » e.g.,light is green in all directions
- Determine how hazards occur
  - » e.g.,hardware failure
- Verify system prevents occurrence
  - » e.g.,check hardware status before switching the light

Fault Analysis

- Fault: a potential system error
  - » e.g.,failure of the clock
- Identify safety effects due to faults
  - » e.g.,lights never change

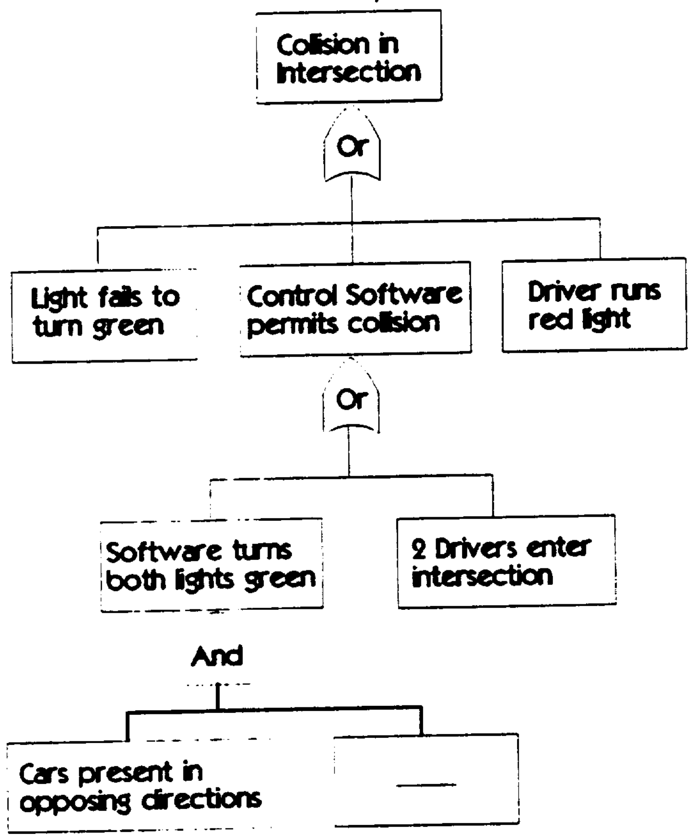
Static Testing Techniques ...

Software Fault Trees

Similar to Cause-Effect graphing

Maps faults to handlers

Maps failures to effects


Collision in Intersection
Or
Light fails to turn green
Control Software permits collision
Driver runs red light
Or
Software turns both lights green
2 Drivers enter intersection
And
Cars present in opposing directions

Static Testing Techniques ...

Software Fault Trees ...

Helpful in defining when to stop testing

- "... test until the consequences of failure no longer justify the testing cost."48

Hazard and Fault analysis identify external risks

Fault trees map those external risks to specific modules

Based on external risks, assess (for each module)

- Consequence of failure during operation
- Expected number of failures (MTTF)

Static Testing Techniques ...

Rule Consistency Checking

Attempts to find errors by checking for certain classes of "anomalies".

- Anomaly = a type of relationship between two or more rules that "seems wrong" , e.g.,

  A -> B and C

  B-> not C

- Anomalies generally indicate an error

Specific to rule-based implementation (forward or backward)

Can find all "anomalies" but a human must analyze anomaly to see if it is a problem.

Many research tools available, no significant commercial offerings.

Static Testing Techniques ...

Rule Consistency Checking ...

Reachability anomalies (non-modular version)

- Dead-end rules
  - » Del_old_changes does not affect any other rule
  - » Fact "signal_changes" should have been "signal_change"
- Unreachable rules
  - » Del_old_changes is also unreachable
  - » No rule asserts "signal_changes"
- Cycle Rules
  - » Update_time is in a cycle
  - » This "anomaly" does not indicate an error in this case
  - » Why?

Static Testing Techniques ...

Rule Consistency Checking ...

Redundant Rules (modular version)

- Set_long_timer:

```
if light_changed or
  signal.in_direction green
then
   set long_timer
   retract medium_timer
   retract short_timer
```

- Retract_medium_timer:

```
if light_changed
then
      retract medium_timer
      retract short_timer
```

- An attempt to retract medium timer twice if light_changed

Static Testing Techniques ...

Rule Consistency Checking ...

Conflicting rules (non-modular version)

- Set_long_timer:

```
if light_changed or
  signal.in_direction green
then
      set long_timer
      set medium_timer
      set short_timer
```

- Retract_medium_timer:

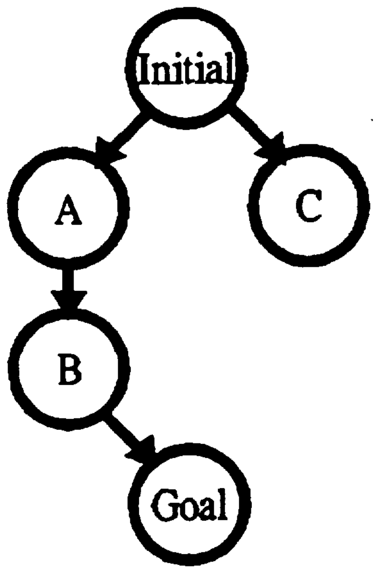
```
if light_changed
then
      retract medium_timer
      retract short_timer
```

- Two conflicting actions if light_changed (set and retract timer)

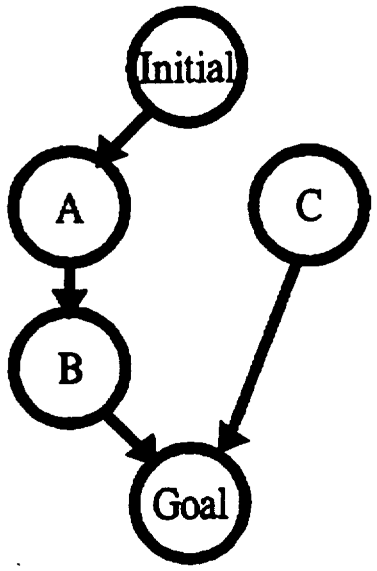
Static Testing Techniques ...

Rule Consistency Checking ...

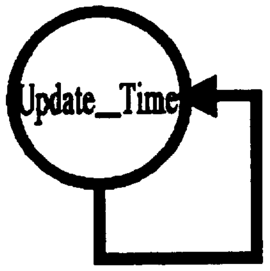
Dead-End Rule (Rule C)



Unreachable Rule (Rule C)



Cycle



<u>Static Testing Techniques ...</u>

<u>Rule Consistency Checking ...</u> (Graphing Techniques)

Petri-Nets

- Originally used to "trace" dynamic behavior of discrete event systems (e.g., rule firings)
- Similar to other diagramming techniques (e.g., state diagrams, cause-effect diagrams, etc.)
- Network of propositions (e.g., rule LHS and RHS)
- "Tokens" trace rule firings
  - » Completeness and consistency errors
- Tedious without automated help
  - » Modularity helps reduce complexity

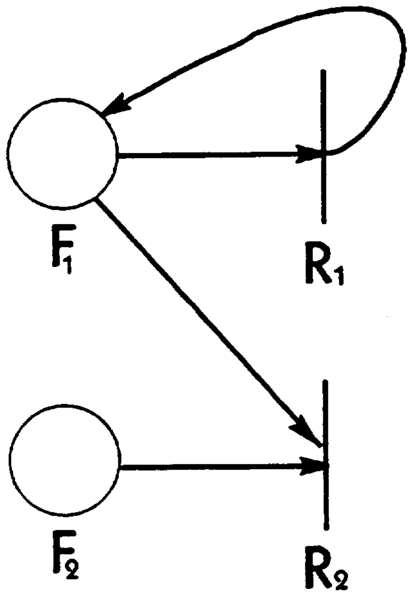
Static Testing Techniques ...

Rule Consistency Checking ...

Petri Nets ...

- Consider the TIME module of Handout #4

Facts		Rules	
F_1	(time (is ?t)	R_1	Count_Time
F_2	(stop-time ?t)	R_2	StopIt



Static Testing Techniques ...

Rule Consistency Checking ...

Directed Graphs (or Network Flows)

- Rules are converted into a collection of directed arcs (directed because of inference)
- First build a list of antecedent and consequent propositions
- Generate an edge to the graph for each antecedent/consequent pair
- Many algorithms exist for analyzing reachability issues

Static Testing Techniques ...

Rule Consistency Checking ...

Connectivity Graphs

- Different kinds of matrices:
  - » facts vs. rules, clauses vs. rules, clauses vs. facts, etc.
- Matrices can then be represented as undirected graphs connecting elements of the matrices
- Can Help to identify the major areas of correctness
  - » e.g., for Rulebases: completeness , consistency, redundancy, dead-end rules
- Can also assist in design (e.g., identifying modularity)
- Supported by simple matrix operations (see Handout #5)

Static Testing Techniques ...

Data Consistency Checking

Checking that data use is consistent with data definition

Checks data/facts

Can find mismatches between data definition and use

Is supported by some tools (e.g., CRSV)

- **E.g.,CRSV could detect "typos" such as the fact "signal_changes"**

Static Testing Techniques ...

Specification-Directed Analysis

Checking that implementation matches specification

- **Specification := assertion about a part of the implementation, like a "mini requirement"**

Useful for all aspects of a knowledge base

Useful for finding any type of implementation error

Not supported by any commercial tools but research prototypes exist

Static Testing Techniques ...

Specification-Directed Analysis ...

E.g., the "Timer" module

- Assertion: timer names are unique
- Analysis of timer_name-conflict rule, verifies assertion is true

Sometimes called "Formal Methods" (but can be informal)

Examples of useful types of assertions

- Data value constraints
  - » E.g., timer constraint
- Postconditions for rules
  - » E.g., timer_name-conflict satisfies postcondition "exactly one timer called ?name will exist"

Static Testing Techniques ...

Specification-Directed Analysis ...

Some useful types of assertions ...

- Abstract functions

  » E.g., light change action can be abstractly described as

  direction := { NS if direction = EW
                 EW if direction = NS }

- (precondition, postcondition) pairs

  » e.g., for change-light function

  pre: green-light' = NS or EW

  post: green-light = NS or EW
        and
        green-light /= green-light'

Static Testing Techniques ...

Partition Analysis

Coverage techniques use implementation

Using a specification can help find missing paths

- I.e., does the specification match the code being tested

A more formal specification is needed

Step 1: Define the inputs for each path along with the outputs

Step 2: Do the same for the associated specification

Step 3: Generate intersections of the results from steps 1 and 2

Step 4: For each non-empty intersection, verify that the spec matches the path

Exercises

1. Identify and define at least 1 "object" in your system (remember, objects consist of both data and operations on that data).

2. Write a pre-condition and a post-condition for each operation on the object.

3. Describe any general properties your "object" must satisfy. Discuss how you would analyze your "object"'s implementation to "prove" those properties are always satisfied.

4. Pick at least one operation and define some rules that implement its specification.

Exercises ...

5. Select one of the following techniques for analyzing these rules. Explain your answer.
   - Petri Nets
   - Directed Graphs
   - Connectivity Matrices
6. Identify 1 "hazard" in your system. Build a fault tree for for that "hazard".
7. Identify 1 "fault" in your system. Build a fault tree for that "fault".

Summary

Techniques

There are many more techniques than the ones discussed.

No technique by itself is sufficient for all levels of software and all types of faults.

Choosing the right set of techniques is important but can be difficult (the V&V puzzle).

Techniques can be selected based on three types of testing

1. Static Testing
2. Unit/Integration Testing
3. System Testing

Techniques ...

Each type of testing:

- Focuses on a different size of software
- Looks at different categories of errors/faults
- Uses certain techniques
  - » Can find errors more cheaply than a later type of testing
- Can reduce the cost of later types of testing by providing information (e.g., units, interfaces)
- Helps ensure a higher quality system (e.g., the system doesn't "crash" at the beginning of the first system test)

When to Stop Testing

Stop "when the money runs out" is a bad approach

Better approach: define a testing objective

- Coverage (e.g., branch coverage)
- Reliability (e.g., Mean Time To Failure)
- Number of errors found (e.g., 40% of what was found at code inspection)

Test until objective(s) reached.

May prioritize objectives

- Most important objectives first
- Most critical modules first
- Most critical error types first

Appendix A: References

References

1. Becker, S.A. and Medsker, L.. "The Application of Cleanroom Software Engineering to the Development of Expert Systems." *Heuristics The Journal of Knowledge Engineering. Quarterly Journal of the International Association of Knowledge Engineers (IAKE).* Volume 4 Number 3 pp. 31-40, Fall 1991.

2. Bellman, K.L., "The Modelling Issues Inherent in Testing and Evaluating Knowledge-Based Systems". *Expert Systems with Applications.* Vol 1., No. 3

3. Bezier, B.. *Software Testing Techniques.* Van Nostrand Reinhold Company, Publisher, 1983.

4. Boehm, B.. "Industrial Software Metrics Top 10 List". *IEEE Software.* Volume 4 Number 5, pp. 84-85, September 1987.

5. Boeing Aerospace Company. *Software Test Handbook: Software Test Guidebook.* Document No. RADC-TR-84-53 Volume 2 of 2. Rome Air Development Center, Griffis Air Force Base, NY 13441, March 1984.

6. Booch, G., *Software Engineering with Ada.*, Benjamin/Cummings, 1983

   Chapter 8 discusses type checking in Ada which is a kind of data consistency checking technique.

References ...

7. Fagan, M.E.. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* Volume 15 No. 3 pp. 182-211, 1976.

8. Fikes, R., Kehler, T., "The Role of Frame-Based Representation in Reasoning", *Communications of the ACM.*, Sept., 1985

> This is a general discussion of frames and their use in rule-based programming. It includes some discussion on necessary and sufficient conditions for classifying a frame instance as belonging to a certain class. This type of necessary and sufficient condition checking ensures a level of data consistency.

9. Franklin, W.R., Bansal, R., Gilbert, E., Shroff, G., "Debugging and Tracing Expert Systems". *Proceedings of the Twenty-first Annual Hawaii International Conference on System Sciences.* 1988

10. Goodenough, J.B. and Gerhart, S.L.. "Toward a Theory of Test Data Selection". *IEEE Transactions on Software Engineering.* pp. 156-173, June 1975.

11. Gries, D.. *The Science of Programming.* Springer-Verlag New York, Inc. 1981.

References ...

12. Hantler, S.L. and King, J.C.. "An Introduction to Proving the Correctness of Programs." ACM Computing Reviews. pp.331-353, September 1976.

13. Harrold, M.J. and Soffa L.S.. "Selecting and Using Data for Integration Testing." *IEEE Software* Volume 8 Number 2 pp. 58-65 March 1991.

14. Herod, J.M. and Bahill, T.. "Ameliorating the Pregnant Man Problem: A Verification Tool for Personal Computer Based Expert Systems". *International Journal of Man-Machine Studies*. pp. 789-805, 1991.

15. Hoare, C.A.R. "Introduction to Proving the Correctness of Programs." *ACM Computing Surveys* pp. 331-353, September 1976.

16. Howden, W.E.. "Reliability of the Path Analysis Testing Strategy." *IEEE Transactions on Software Engineering* pp. 208-215, September 1976.

17. Howden, W.E.. "Symbolic Testing and the DISSECT Symbolic Evaluation System." *IEEE Transactions on Software Engineering* pp. 266-278, July 1977.

References ...

18. Howden, W.E.. "Comments Analysis and Programming Errors." *IEEE Transactions on Software Engineering* Volume 16 Number 1 pp. 72-81, January 1990.

19. Howden, W.E.. "Weak Mutation Testing and Completeness of Test Sets." *IEEE Transactions on Software Engineering* Volume SE-8 No. 4, July 1982.

20. Jalote, P.. "Testing the Completeness of Specifications." *IEEE Transactions on Software Engineering* Volume 15 No. 5, May 1989.

21. Korson, T. and McGregor, J.D.. "Understanding Object-oriented: A Unifying Paradigm." *Communications of the ACM* Volume 33 No. 9 pp. 40-60 September 1990.

22. Landauer, C.A.. "Correctness Principles for Rule-Based Expert Systems." *Expert Systems with Applications.* Pergamon Press. Volume 1 Number 3 pp. 291-316, 1990.

23. Leite, J. and Freeman, P.. "Requirements Validation Through ViewPoint Resolution." *IEEE Transactions on Software Engineering* Volume 17 No. 2 pp. 1253-1269, December 1991.

References ...

24. Leveson, N.G.. "Safety." *Aerospace Software Engineering: A Collection of Concepts*. Ed. Christine Anderson and Merlin Dorfman. Volume 136 pp. 319-336, American Institute of Aeronautics and Astronautics, Publisher. 1991.

25. Leveson, N.G.. "Software Safety in Embedded Computer Systems." *Communications of the ACM* Volume 34 No. 2, February 1991.

26. Leveson, N.G., Cha, S.S., and Shimeall, T.J.. "Safety Verification of Ada Programs Using Software Fault Trees." *IEEE Software*. pp. 48-59, July 1991.

27. Linger, R.C., Mills H.D. and Witt, E.I.. *Structured Programming: Theory and Practice*. Addison-Wesley Publishing Company 1979.

28. Liskov, B. and Guttag, J.. *Abstraction and Specification in Program Development*. McGraw-Hill Book Company 1986.

29. Liu, N.K. and Dillon, T.. "An Approach Toward the Verification of Expert Systems Using Numerical Petri Nets." *International Journal of Intelligent Systems*. Volume 6, Number 3, pp. 255-276, June 1991.

References ...

30. Marcus, S., "SALT, A Knowledge Acquisition Tool That Checks and Helps Test a Knowledge Base". *AAAI Workshop Notes on Verification, Validation, and Testing of Knowledge-Based Systems*. 1988.

31. McGraw, K.L., Harbison-Briggs, K.. *Knowledge Acquisition Principles and Guidelines*. Prentice Hall, 1989

    pp. 312-323 includes a discussion of using experts to aid in review and testing of an expert system

32. Meyer, B.. *Object-oriented Software Construction*. Prentice Hall, Publisher 1988.

33. Miller, L.A., "Dynamic Testing of Knowledge Based Systems Using the Heuristic Testing Approach". *Expert Systems with Applications*. Vol. 1, No. 3, 1990

34. Mills, H.D., Linger, R.C. and Hevner, A.R.. "Box Structured Information Systems." *IBM Systems Journal* Volume 26 No. 4, 1987.

35. Mills, H.D., Linger, R.C. and Hevner, A.R.. *Principles of Information Systems Analysis and Design*. Academic Press, Inc. 1986.

References ...

36. Montalbano, *Decision Tables*. Science Research Associates, 1974

37. Myers, G.J.. *The Art of Software Testing*. John Wiley & Sons, Publishing 1979.

38. Myers, G.J.. *Software Reliability Principles and Practices*. John Wiley & Sons, Publishing 1976.

39. Myers, G.J.. *Reliable Software Through Composite Design*. Mason/Charter Publishers 1975.

40. Myers, G.J.. *Composite/Structured Design*. Litton Educational Publishing 1978.

41. NASA/JSC Software Technology Branch, *CLIPS Reference Manual.*, Voll III, Section 2

Section 2 is the description of the capabilities of CRSV

42. Nazereth, D.L.. *An Analysis of Techniques for Verification of Logical Correctness in Rule-Based Systems*. pp. 80-136. Catalog Number 8811167-05150. UMI Dissertation Service, Ann Arbor, MI 48106, 1988. (Phd. dissertation, Case Western Reserve University, 1988)

References ...

43. Nguyen, T.A., Perkins, W.A., Laffey, T.J., Pecora, D., "Knowledge Base Verification", *AI Magazine.*, Summer, 1987

44. Parnas, D.. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communcaionts of the ACM* Volume 15 No. 12, pp. 1053-1058, December 1972.

45. Richardson, D.J. and Clarke, L.A.. "A Partition Analysis Method to Increase Program Reliability." *Proceedings, Fifth International Conference on Software Engineering* pp. 244-253, 1981.

46. Rumbaugh, J.. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc. 1991.

47. Rushby, J., Crow, J., *Evaluation of an Expert system for Fault Detection, Isolation, and Recovery in the Manned Maneuvering Unit.* Final Report for NASA contract NAS1-182226 (NASA/LANGLEY)

References ...

48. Rushby, J.. *Quality Measures and Assurance for AI Software.* Prepared for NASA Langley Research Center. NASA Contracter Report #4187, 1988.

This is the last reference in the references section of this workshop. Pages 74-79 includes a discussion of minimum competency testing.

49. Science Applications International Corporation. "Task 1: Review of Conventional Methods." *Guidelines for Verification and Validation of Expert Systems.* Document No. SAIC-91/6660, 1991.

50. Sherer, S.. "A Cost-Effective Approach to Testing". *IEEE Software.* pp. 34-40, March 1991.

51. Voas, J., Morell, L., and Miller, K.. "Predicting Where Faults Can Hide from Testing". *IEEE Software*, pp. 41-48, March 1991.

52. Weld, D.S., de Kleer, J., eds. *Qualitative Reasoning about Physical Systems.*, Morgan Kaufmann, 1990

53. Yourdon, E. and Coad, P.. *Object-Oriented Analysis.* Prentice Hall, Inc. Englewood Cliffs, NJ 1990.

Appendix B: Techniques Vs. References

Techniques Vs. References

Techniques	References
Active Interface Testing	49
Anomaly Analysis	49, 5
Attribute-Based Test Case Selection	49
Axiomatic Analysis	11, 20
Boundary Testing	37-40
Branch Coverage	37-40
Cause-Effect Graphing	37-40
Competing Designs	23
Compilation Testing	49
Condition Coverage	37-40
Connectivity Matrices	22, 42
Data Consistency Checking	6, 8, 41
Defect Analysis	49
Disaster Testing	
Error Guessing	37-40

Techniques Vs. References ...

Techniques	References
Explicit Modelling	2
Expert Review	31
Fault Analysis	24-26
Flavor Analysis	18
Flow Graphs	42
Hazard Analysis	24-26
Inspections	7, 37
InterProcedural Dataflow Testing	12
Knowledge Acquisition Correctness	30, 14
Minimum Competency Testing	47, 48
Mutation Testing	19
Object Oriented Analysis	53, 37, 21, 44
Partition Analysis	45
Path Coverage	15, 37-40

Techniques Vs. References ...

Techniques	References
Performance Testing	49, 5
Petri Nets	1, 29, 42
Pre/Post Condition Testing	11, 15, 27, 28
Prototyping	
Random Testing	3, 49
Realistic Testing	3, 49
Regression Testing	3, 49
Reliability Testing	3, 49
Rule Consistency Checking	42, 43
Sensitivity Analysis	9
Software Fault Trees	26
Specification-Directed Analysis	Case Study #1, 47
State Diagrams	46
Stepwise Refinement	38-40, 34-35

Techniques Vs. References ...

Techniques	References
Structural Testing	33
Stress Testing	3, 37
Symbolic Execution	16, 17, 12
Testability Analysis	50, 51

Workshop on Verification and Validation of Expert Systems

Guidelines

Authors:

Scott W. French
FRENCHS@HOUVMSCC.VNET.IBM.COM

David Hamilton
HAMILTON@HOUVMSCC.VNET.IBM.COM

IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058

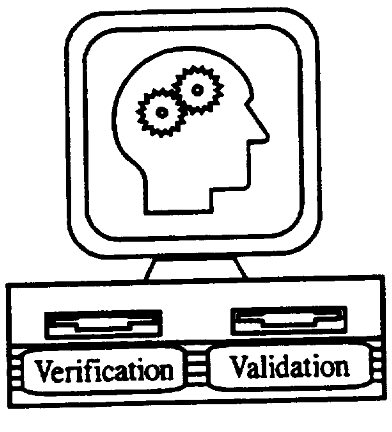


Table of contents

Introduction

Overview

Goals

1. To understand guidelines on the application of V&V techniques
2. To understand how to V&V a system which includes expert system(s)
3. To understand how to tailor V&V based on specific needs and characteristics

Approach

1. Make some inferences about what should be in a set of expert system V&V guidelines
2. Discuss a set of V&V guidelines
3. Discuss tailoring of guidelines

Implications for Guidelines

Overview

So far we have:

- Reviewed conventional and expert system V&V techniques

- Pointed out key V&V ideas (e.g., the V&V puzzle)

- Studied a sample problem (traffic light controller)

<u>Overview ...</u>

From this, we can make some inferences about what should be in a set of ES V&V standards and guidelines.

From these inferences, we can

- Develop a set of ES V&V guidelines
- Develop some tailoring criteria

<u>Note</u>: Many implications may seem trivial but they lead to important guidelines.

Conventional Validation Implications

Validation: "Am I building the right product?"

- **Must be able to know if a product is right or not**

- **There must be some known criteria that the right product will satisfy**

Conventional Validation Implications

...

Verification Puzzle: Different kinds of correctness

- Must know which kinds of correctness are important
  - » Utility Correctness at a minumum (satisfies user's needs)
    - * Must know user's needs
- Should check that the understanding of problem to be solved is both complete and consistent
- May tailor V&V based on size, complexity and criticality
- Must pick the V&V techniques to fit the puzzle

Conventional Validation Implications ...

Black Box View: Based on observable behavior

- Must be able to validate correctness based on observable response from known stimulus
  - » Can not validate system just by seeing that correct knowledge went into it

Operational Scenarios: Stimulus/response descriptions based on how the system is expected to be used

- User can describe how he expects to use the system and developer can obtain stimulus/response from the user's description(s)

Conventional Verification Implications

Prototyping: Early model of possible system

- Understanding of the desired system can be validated before system development begins

Verification Puzzle: Comprehensive validation of large complex systems is too difficult, but system can be "incrementally validated" by performing separate, static, unit/integration, and system testing

- Verification greatly reduces the difficulty of validation

Conventional Verification Implications ...

Verification: "Am I building the system right ?"

- **Must know/understand the system that is being built**
- **Must know how the system is to be built (i.e., need design)**

Modularity: Structured "divide and conquer" approach has many benefits

- **System should be modularized to reduce the verification effort**

Conventional Verification Implications ...

Different Techniques catch different types of problems and none are comprehensive

- **Mutliple V&V techniques must be used**

The earlier an error is found, the more cheaply it can be fixed.

- **Emphasize techniques which can be applied early**
- **Perform verification as early as practical**

Conventional Verification Implications ...

Techniques work at different levels (e.g., static analysis vs. statistical testing)

- **Verification should be planned so that techniques are applied when and where they are appropriate**

Static testing techniques work at many different levels and can be applied early

- **These techniques are important**

Abstraction, refinement, and proper documentation ease the application of static testing techniques

- **Design should use abstraction, refinement, and associated documentation (e.g., specifications)**

General Expert System V&V Implications

Expert systems are software

- Same basic conventional V&V implications hold for expert systems

Expert Systems may satisfy some, but not all, implementation and problem characteristics

- Verification approach must be tailored for the specific type of expert system being built

Expert System Validation Implications

May just mechanically apply expert's "rules of thumb" (as opposed to solving a problem)

- Validation must rely on comparison with the expert

May solve a very difficult problem (e.g., complex scheduling) where correct solutions are not known

- Validation may be able to only address "reasonableness" of solutions (e.g., feasible schedule)

May solve a problem with only fuzzy or subjectively correct answers

- Each test result must be checked by an expert

Expert System Verification Implications

Internal interactions may be unclear and/or complex

- Manual analysis may be very difficult (i.e., inspections)

Execution sequence may not be explicit

- Verification of problem solving method may be very difficult

Expert Systems often built iteratively (in small chunks)

- Testing should be iterative (to catch errors early)
- Regression testing will be done often

Other (Common Sense) Implications

There is no way to know if the system will meet the user's needs without doing something that would be called V&V.

- V&V must be done

V&V takes time (and money)

- Development schedule and cost should account for V&V

The best person to determine correctness is the expert

- The expert should be involved in V&V

A "fresh look" can often find errors better

- Independent (unbiased) V&V should be done if practical

Guidelines

Overview

The implications for V&V directly lead to some specific guidelines which will be discussed first.

Based on the guildelines, recommendations for how to develop a V&V approach will be discussed.

Finally, you will have the opportunity to practice developing a V&V approach on a case project.

Project Management Guidelines

Plan for V&V

- Include V&V in schedule (e.g., inspections)
- Include V&V cost in total development cost (typical V&V cost is 25% of total project cost, spread throughout the development cycle)
- Allocate resources for V&V (e.g., expert's time)

Plan to spend time developing a good design (so static testing won't be too hard)

Project Management Guidelines ...

Pick a Life-cycle that includes all 3 test phases (and follow it).

- Standardizing on a life-cycle aids in planning and management of V&V.

Tailor V&V approach based on:

1. Expected size and complexity

2. Type of expert system (based on characteristics)

3. Types of correctness that matter

Project Management Guidelines ...

Use Configuration Management

- Ensure system is correctly integrated
- Ensure testers know what they are testing (e.g., version control)
- Helps manage the effects of complex internal interactions

Reserve a significant portion of the expert's time for helping with V&V (25%).

Prototype for early validation but clearly separate prototyping from development

Plan to do V&V as the system is iteratively developed (not all at the end).

Problem Analysis Guidelines

Try to narrow the problem domain as much as possible

- **"Knowledge based systems have a greater likelihood of succeeding - and, in a sense, of being valid - when they address a narrowly defined problem."8**

- **"If an expert system starts with vague objectives, some may conclude that it doesn't matter what the eventual system does, because anything is better than nothing."7**

Problem Analysis Guidelines ...

Do not try to pre-determine whether the solution will be an "expert system" or not.

Expect .the System to work

- Survey results indicated a significant percentage did not expect the Expert System to be as accurate as the expert[5]
- "The difficulty with low expectations is that they become self-fulfilling"[3]

Requirements Guidelines

Write Requirements.

- **Something is needed to V&V the system against.**
  - » **"A good programmer understands what his program is supposed to do and why he expects his program to do it"[3]**

Document the following (at a minimum):

- **expected behavior**
- **operational scenarios (how the system is expected to be used)**

Requirements Guidelines ...

Consider each kind of correctness when writing requirements.

1. Functional

2. Safety

3. User-Interface

4. Resource Consumption

5. Utility

Design Guidelines

Design modular systems

- Modules can be V&V'ed separately
- V&V of many little systems is easier than V&V of one large system
- Reduces regression testing

Use abstraction and refinement

- Makes static testing easier
- Allows verification during design

Cross reference design to requirements and code

- Facilitates completeness checking

Design Guidelines ...

Some design hints

- Pick a design notation and stick with it across the application (needed to verify consistency).

- The Level of Formalism is NOT as important as the consistency of Formalism

  » "I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system ... reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas" - Fred Books[6]

General Guidelines

Consider an independent group for final V&V, or at least try to include some independent reviews

- A "fresh look" often finds additional errors
- Will help determine if system is adequately documented

Always try to find as many errors as early as possible

- Errors found early are much cheaper to correct

Use a mixture of V&V techniques

- There is no single comprehensive technique

V&V Technique Guidelines

During integration of large systems, test higher level control and user-interface functions first (stubbing out lower level details if necessary)

Perform regression testing at each iteration

- Emphasize modules that changed

- Perform "health test" of overall system

V&V Technique Guidelines ...

Emphasize static testing techniques for evaluation of detailed functional correctness

- Based on design notation/formalism, write design specifications and perform specification-directed analysis
- If rule-based implementation, perform rule consistency checking
- Use data-consistency checking, especially if implementation is frame-based.
- If developing a classification-type expert system, perform sensitivity analysis to evaluate sensitivity of classes to distiguishing criteria

V&V Technique Guidelines ...

Use realistic testing for evaluating utility and user-interface correctness

- Will the system satisfy the user needs based on how they plan to (would like to) use the system ?

Selectively choose test cases for testing functional correctness (do not attempt to be comprehensive, as in static testing)

- Emphasize critical and complex functions

- Randomly exercise other functions

V&V Technique Guidelines ...

Use stress/performance testing to evaluate resource consumption correctness

After selective testing, measure coverage and look for major "holes" in coverage (rules not covered, facts not used etc).

Recommended Approach

1. Analyze Problem (ongoing activity)

    - Identify areas of uncertainty and/or complexity that may require prototyping
    - Identify areas of high criticality
    - Identify available expertise
        - » Is problem to be solved by knowledge acquisition or analysis ?
    - Identify/document expected behavior and operational scenarios
    - Identify aspects of problem that match expert system criteria, but do not anticipate expert system implementation.

Recommended Approach ...

2. Do initial planning

- Do not attempt comprehensive up-front planning.

  » True expert systems are usually developed in a highly iterative manner

- Determine objectives for next iteration.

- Determine criticality of correctness.

- Estimate size and cost (include V&V).

  » If V&V is listed as separate cost, it is in danger of being "cut"

- Define milestones that follow a life-cycle.

Recommended Approach ...

2. Do initial planning ...
   - Reserve resources
     - » Expert's time
     - » Consider identifying IV&V group
     - » Look for available V&V tools (especially those that assist an expert[5])
   - Ensure:
     - » Problem is not too broadly defined
     - » Adequate requirements exist / will exist

Recommended Approach ...

3. Perform design and specification-driven analysis

   - As each module is refined/completed, verify functional correctness and completeness.

   - Always map back to higher level design, requirement, prototype, or problem description.

   - Hold periodic inspections and involve expert(s).

   - Based on implementation approach, use additional static testing techniques (e.g., rule consistency checking)

Recommended Approach ...

4. As each increment is completed

- Test overall execution (high level control) e.g.,
  - » Screens/windows look OK
  - » Files opened/closed correctly
  - » Functions respond to appropriate user inputs
  - » Output appears in the right place

Recommended Approach ...

4. As each increment is completed ...

- Perform realistic and/or statistical testing
- Perform stress testing
- Measure coverage and look for "holes"
- Regression test unchanged features
- Perform field testing with user's and experts

Exercise

1. Determine whether the recommended approach fits your problem. Identify additional issues that need to be considered.

2. Generate a detailed development plan for your problem. Try to include specific milestones and how they will be achieved.

3. Define specific development increments. Update your plan to reflect those increments.

4. Consider the test cases you have selected so far. Are there any other kinds of testing you need to do? When will you know to stop testing?

5. Build a high-level requirements outline for your system definition. How well does the original problem definition map to your outline?

Appendix A: References

References

1. Parnas, D.L., Clements, P.C., "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering*. Feb., 1986

Describes why one would wish to document a product as if it were designed according to an idealized development process/methodology, even if was developed in a very ad-hoc manner. Also includes suggestions on what the documentation of a product should contain.

2. Fox, M.S., "AI and Expert System Myths, Legends, and Facts", *IEEE Expert*. Feb., 1990

Contains personal observations by the author that help expain some causes of ineffective AI applications; many are due to a misunderstanding of AI technology.

References ...

3. Guttag, J.V., "Why Programming is Too Hard and What to Do About It", *Research Directions in Computer Science: An MIT Perspective,* MIT Press, 1991

Contains personal observations by the author on the difficulties in software programs. The author, a respected professor and researcher in software development techniques, offers some very candid opinions in this paper.

4. Schank, R.C., "Where's the AI ?", *AI Magazine*, Winter 1991

A very readable description of some personal observations by the author on some difficulties in developing truly intelligent systems. This article is highly recommended reading.

References ...

5 "KBS V&V - State of the Practice and Implications for V&V Standards"

This paper is included in the references section. It summarizes a survey that was performed of 60 expert system projects to determine what techniques were currently being used to V&V expert systems and what difficulties were being encountered.

6. Brooks, F., *The Mythical Man Month*, Addison-Wesley, 1975

The classic book on software engineering. It is a collection of personal observations on software development. Although the book is many years old, the observations are just as true today as they were 15 years ago. This book is very highly recommended reading.

References ...

7. Geissman, James R.. "Verification and Validation for Expert Systems: A Practical Methodology." Abacus Programming Corporation, Van Nuys, CA., SOAR Conference, 1990 (???)

8. Marcot, Bruce. "Testing Your Knowledge Base." *AI Expert*, July 1987

> This article offers some practical advice for testing knowledge bases by listing some very general guidelines. It also has a good detailed list of types of correctness.

9. Hall, A., "Seven Myths of Formal Methods", *IEEE Software*, September, 1990

References ...

10. Bundy, Alan. "How to Improve the Reliability of Expert Systems." *Proceedings of Expert Systems '87: Seventh Annual Technical Conference of the Pontish Computer Society Specialist Group on Expert Systems*. December 1989, pp. 3-17.

11. Culbert, Chris. "Knowledge-Based Systems Verification and Validation." *The Verification and Validation of Expert Systems Workshop*. Austin, TX, June 18, 1991.

12. Froscher, Judith N., Jacob, Robert J.K.. "A Software Engineering Methodology for Rule-Based Systems." *IEEE Transactions on Knowledge Engineering* Volume 2. No. 2, pp. 173-189, June 1990.

13. The Institute of Electrical and Electronics Engineers (IEEE). "IEEE Standard Glossary of Software Engineering Terminology." *ANSI/IEEE Std. 729-1983*. 345 E. 47th Street, New York, NY, February 18, 1983.

14. Waterman, Donald A.. *A Guide to Expert Systems*, Addison-Wesley Publishing Company, 1986, pg. 187.

Case Study #1: A Solution For The Traffic Controller Problem Using Terms, Operators and Productions

Introduction

Case Study number one will provide a detailed example of designing an Expert System solution to the Traffic Light Controller problem. The example is founded on work done by IBM's Houston Scientific Center. This effort (with assistance from Texas A&M University) combined the strengths of Production systems, Term Subsumption Languages and Object-Oriented programming to define a design language, called TOP (Terms, Operators and Productions), suitable for building verifiable Expert Systems. For a more thorough discussion of these different paradigms please refer to the ***References*** section of your class notebook. A complete design for the Traffic Light Problem written using the TOP design language is provided at the end of this study.

The design approach detailed in this case study represents an approach that focuses on continually refining the problem definition as understanding of the problem expands. Fortunately, as in conventional software design, this approach can be neatly broken into steps. Verification and Validation techniques, as appropriate, should be applied at each step. This discussion will address appropriate Verification and Validation approaches at each step of the development process.

Step 1: Knowledge-Base Architecture

To ease the verification effort, knowledge should be broken up into different parts (i.e., modules). This analysis should focus on identifying the primary ***ideas*** that describe the domain for a given system. In the case of the Traffic Light problem, this can be done very easily. Be aware that the results of this step are rarely final. As the problem becomes more clearly understood additional changes to the architecture of the design will probably be needed.

TOP supports partitioning a knowledge base by allowing the designer to build Ada-style packages. Each package defines the key ideas associated with a given unit of knowledge. For example, from the Traffic Light problem, one could easily identify several different units based on the key objects in the problem description. These would be *sensor*, *traffic_light* and *signal.* Shown below is the initial unit definition, using TOP syntax, for the *sensor* knowledge unit.

```
package SENSORS is
 . . .
end SENSORS:

package body SENSORS is
 . . .
end SENSORS:
```

Each unit will have a specification and a body. The specification will define the interface to other units in the design. Each unit of knowledge should be loosley coupled (i.e., it has few, if any, dependencies on other units) and strongly cohesive (i.e., a given specificiation fully implements the knowledge).

Knowledge in one unit may be required to define another knowledge unit. For example, the definition of the signal unit depends on the defintion of the sensor unit. This is true because the indicators that define a signal are received from an open sensor. To show these relationships in a TOP design, use the WITH (this syntax is also derived from Ada) clause. For example, the signal unit specification would appear as follows:

```
with SENSORS;
package body SIGNALS is

  . . .

end SIGNALS;
```

Verification/Validation Approaches:

Verification approaches at this level are very dependant on how well the problem is understood. This understanding must come from the expert in the field along with a detailed requirements document that specifies the required behavior of the expert system. Analysis using these two sources should focus on showing that the units defined cover the problem space (i.e., nothing was left out) and that the partitioning of the problem into units is consistent and maintainable. Visualization techniques such as structure charts, semantic nets, etc. can be helpful in analysis of the architecture.

Step 2: Define the Knowledge Terms

The next step in developing an Expert system using TOP would be to completely define each of the knowledge units. As mentioned, each knowledge unit in the design captures a unique part of the overall knowledge. In TOP, these unique parts are described using *Terms*. The technique for identifying these terms is called ***conceptualization of the domain.***

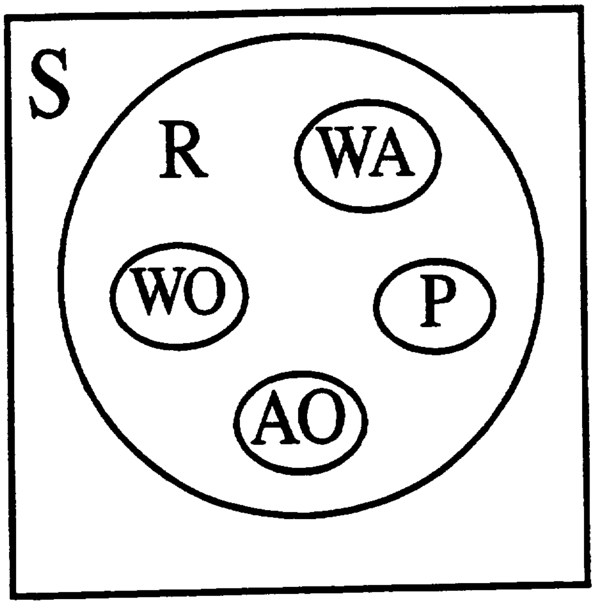
What are Terms? Terms capture declarative domain knowledge. In other words, terms are the words used to describe things in the problem domain. Terms can be either *concepts* (an idea) or *relations* (something that relates concepts). A simple method of identifying the highest levels of these terms is to look for nouns (i.e., concepts) and adjectives (i.e., relations). For example, from the Traffic Light Problem, one could define a concept for each of the units described previously such as *signal, sensor*, etc.. These particular concepts represent the highest level idea to be captured by their respective knowledge units. These are the easiest concepts to identify. Further understanding of the problem reveals refinements to these high level concepts, such as *Open_Sensor*, *Received_Signal*, etc.. Each of these refinements serve to clarify the primary idea captured by the knowledge unit and therefore belong in the same knowledge unit as the highest level concept. Relations are also identified based on an understanding of the problem. For example, from the Traffic Light problem, the relation *Has_Approaching* would serve to relate the concepts of a *Signal* and an *Indicator* (a special kind of number).

In TOP, refinement of high-level concepts and relations is captured by (1) the *specializes* keyword and (2) the ability to specify what makes one term a specialization of another. For example, the idea of an *Received_Signal* is the same as that of a *Signal* except that the *Has_Approaching* and *Has_Waiting* indicators are associated with a *Received_Signal* (the reverse is not true). There may be cases where no definition is possible or desired. These terms are considered *primitive*.

Verification/Validation Approach:

Conveniently, concepts and relations can be thought of as *sets* or *classes* of things. The members of these sets are called ***instances***. The definition associated with a given concept or relation describes when something can be classified as belonging to that given concept or relation. Clearly, if there are sets then there are subsets. The specializes keyword serves to identify those terms that are subsets. For example, instances of the concept *Received_Signal* are also instances of *Signal*, but not necessarily the other way around. Only when the instances satisfy the *Received_Signal* definition would they be classified as both a *Signal* and a *Received_Signal.*

The advantage of viewing concepts and relations as sets is that there are lots of good analysis techniques based on set theory. One simple technique to assist in analyzing the concepts in a given unit is the Venn Diagram. Each knowledge unit should capture one major set with all terms defined in that unit being subsets of that one primary set. For example, from the Traffic Light Problem, all terms in the unit, *Signals*, belong to one major set called *Signal.* If a term in the unit does not fit quite right into the main set then it should be partitioned into its own knowledge unit.



S = {Set of all signals}

R={Set of all *received*

AO = {Set of all received signals that indicate only appraoching traffic}

WO = {Set of all received signals that indicate only waiting traffic}

WA = {Set of all received signals that indicate both waiting and approaching traffic}

P = {Set of all received and processed signals}

The Venn Diagram should help in defining good concepts and relations and help in finding those things that do not make good sets, but rather define some global constraint that the system should operate under. As the Venn Diagram is defined, there will be some parts of the unit definition that are not conveniently described as sets. These parts describe more general constraints or conditions on the knowledge. Typically they involve more than one term. TOP designs include the definition of *Global Constraints* for the purpose of capturing these important parts of the knowledge. These parts are best left out of the Venn Diagram since they are constraints and not sets. However, the Venn Diagram can help in analyzing the conditions that define each *global constraint*. Some examples of these will be shown later as we expand the scope of the solution to the Traffic Light Problem.

Verifying the terms is the simplest part of verifying the ES because of their declarative nature. Just like the first step in this process, showing that the definitions are correct depends on the requirements and inputs from the expert. Many of the more difficult aspects of the ES design, such as sequencing, are not an issue at this early step. However, declarative definitions can become quite complex (i.e., they involve many conditions). To make the verification process easier, it is helpful to capture small groupings of conditions into a higher level condition (i.e., stepwise refinement/abstraction).

For example, from the Traffic Light Problem, an *Approaching_Only_Signal* is a *Received_But_Not_Processed_Signal* that indicates that a given signal indicates that approaching traffic was detected while no traffic was waiting. By capturing this detailed set of conditions as a concept, a name (or abstraction) can be associated with those conditions. This means that other portions of the design can check an instance's membership in the set Approaching_Only_Signal, rather the specific conditions.

Step 3. Defining Tasks for Knowledge Units

After steps one and two the declarative part of the domain knowledge is complete. Each knowledge unit captures a collection of terms that define a piece of domain knowledge. However, nothing has been defined to transition instances of a given term (or set) to instances of another set. Therefore, the next simplest step in our design process will be to identify tasks (e.g. object-oriented programming refers to these as operators) that perform these transitions. These tasks relate very nicely to the verbs in the problem description. For example, the unit, Traffic_Light, contains a task (or operator) called Switch that changes the light.

TOP uses the Method construct to allow designers to define the different tasks in a given knowledge unit. TOP does not declare a task (or operator) explicitly, but rather defines it as a collection of its methods. A given task may have many different methods based on different situations under which they might be used. For example, the method, Switch, from the Traffic_Light knowledge unit performs a different function based on whether the light is currently red or the light is currently green. These differing situations are specified using the *Used When* clause of the Method.

Methods also contain pre and post conditions. Pre-conditions are specified using the *Requires* clause and the post-conditions are specified using the *To Produce* clause. For example, the method *Open* from the unit, Sensors, requires that a given sensor is not already open. A post-condition specifies the conditions that must be true when the expressions contained in the *Involves* portion of the Method have finished execution. For example, when the method *Open* finishes execution, the given sensor should be now classified as an *Open_Sensor*. In fact, it is very straightforward to show that the post-condition for this method will always be satisfied, because the method asserts that the given sensor is now an *Open_Sensor.*

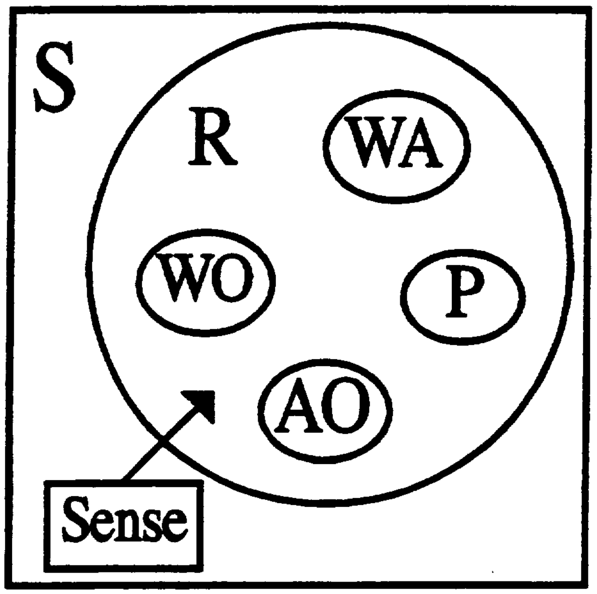
It is important to recognize the difference between the situation conditions and the pre-conditons. Pre-conditions express a collection of binding conditions that must be true for all methods of a given task. Situation conditions, however, specify a disjoint collection of conditions used to determine which particular method is selected for execution.

Verification and Validation Approach:

Verification and Validation at this step in the design focuses on showing that the correct tasks have been identified and that each method of a given task is correct. Verifying that the correct tasks have been identified is fairly straightforward. Once again, input from the requirements and an expert are important is showing the correct tasks have been identified. Another technique involves using the Venn Diagram approach outlined above. Since all concepts of the unit are being viewed as sets one can analyze the identifed tasks to see that these tasks perform all possible transitions (i.e., an instance of one kind of set can always be transitioned to another kind of set). For example, in the Venn Diagram that follows, the task *Sense* is shown to transition any instance of the set *Signal* to its subset, *Received_Signal.* This does give the complete coverage argument required. How does an instance of *Received_Signal* become an instance of *Approaching_Only_Signal?* This one can be answered directly from the definition of the concept, *Approaching_Only_Signal.* How can an instance of *Received_Signal* become a *Received_But_Not_Processed_Signal?* That happens as a direct result of the task, *Sense.* How does an instance of *Received_Signal* become an instance of *Received_And_Processed_Signal?* Apparently, given the definition of the *Signals* unit there is nothing defined to perform that mapping. Is this a problem? In some cases this might identify something that has been left out of the design. In this case, maybe not. The intention is to allow what ever unit that is processing the *Received_Signal* to indicate when it has finished processing that signal (hence the concept, *Received_And_Processed_Signal* is primitive). Therefore, no problem exists. The diagram shown does not indicate how the opposite transitions can be made (e.g., how does an instance of *Received_Signal* become an instance of just *Signal?*). Take a few moments and figure out how to modify the diagram, based on the TOP design, to reflect the missing parts.

Having shown that the correct tasks were identified, each task must be shown to be correct. This is a three part process: verifying the situations, verifying the pre-conditions and verifying the post-conditions. Verifying the situation expression involves showing that the combination of all situation expressions (i.e., each situation for each particular method of a task) *covers* all possible conditions under which the task operates. For example, coverage exists for the Switch task in the *Traffic_Light* unit, because a method is defined for each possible state of the light (i.e., red or green). The arguement is easily shown to be true because an instance of a light can only be a *red-light* or a *green-light.*

Verification of pre-conditions involves showing that the *Requires* condition is a necessary condition for all methods of a task. Verifying the post-condition involves showing that the result of executing the Involves portion of the method will produce the expected results. Showing that both the pre and post conditions are correct depends a lot on input from the requirements and experts.



S = {Set of all signals}

R={Set of all *received* signals}

AO = {Set of all received signals that indicate only appraoching traffic}

WO = {Set of all received signals that indicate only waiting traffic}

WA = {Set of all received signals that indicate both waiting and approaching traffic}

P = {Set of all received and processed signals}

Step 4. Specifying Problem Solving Behavior/Tasks

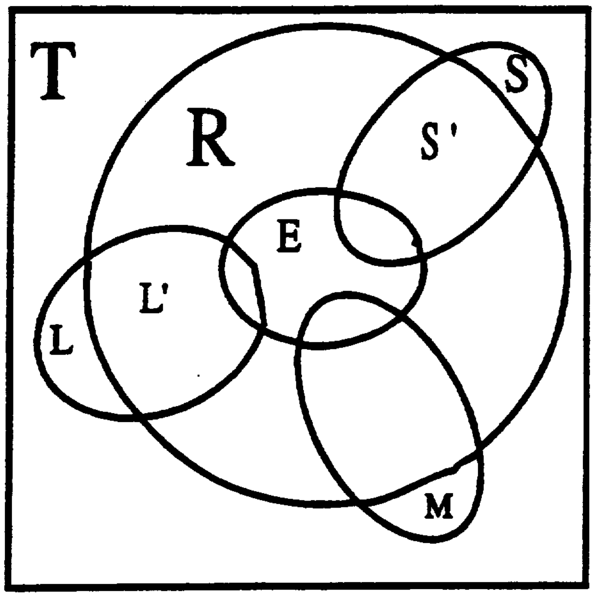
Now that steps one through three have been completed, the basic building blocks exist for defining the problem solving behavior of the Expert System. To define this behavior it is beneficial to try and identify the problem solving behavior by abstracting the specifics of what the system does to a general approach. For example, using the Traffic Light Problem definition, an abstracted problem solving approach might be as follows.

A goal exists that some activity should be performed (in this case, the light should change). In order for this activity to be performed, however, a specific event must take place (in this case, a period of time must expire). A subgoal, then, is to watch for this specific event to take place. This subgoal depends on other events (in this case, defining the desired interval of time to wait). Another subgoal, then, is to watch for completion of these events.

Let's refine this description to be more specific for the Traffic Light Problem. The desire is for the traffic light to change. What is required for this to happen? A period of time must expire in order for the light to change. How does a period of time expire? Clearly a period of timer expires when that exact number of time units has passed. But, what period of time should expire? There are many different circumstances under which a period of time is selected for expiration. These different circumstances map directly to the specific scenarios (i.e., stimulus histories) discussed at the black-box view of the problem.

At this point, something interesting happens that was alluded to in step one. At this point the Traffic Light Problem design has focused on three main units: *Sensors, Signals* and *Traffic_Light*. However, refinement of the problem has introduced a new unit that was not so apparent when the architecture was initially defined. This unit, *Timer_Unit*, focuses on defining the measurement of time periods to support the goal of periodically changing the traffic light. Should this happen during design (and it usually will), the appropriate step is to re-work steps one through three by adding in the new design unit. Venn Diagrams describing *Timer_Unit* are shown next.

Timer Unit Term Analysis:



T = {Set of all timers}

R = {Set of all running timers}

S = {Set of all short timers}

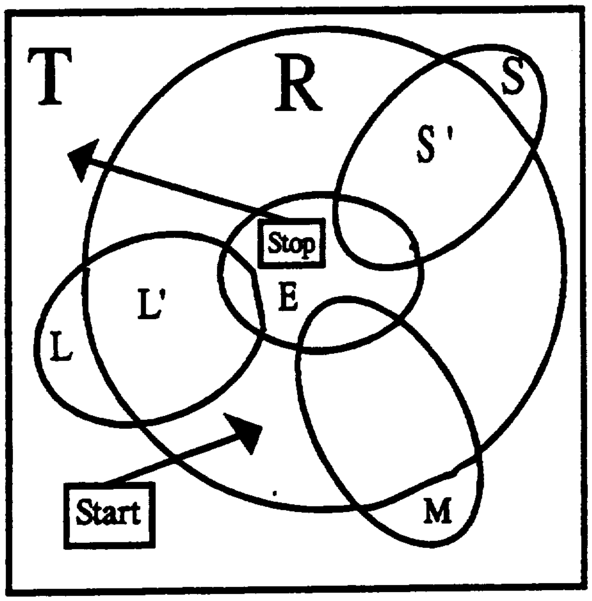
S' = {Set of all unexpired short timers}

M = {Set of all medium timers}

L = {Set of all long timers}

L' = {Set of all unexpired long timers}

Timer_Unit Task Analysis



T = {Set of all timers}

R = {Set of all running timers}

S = {Set of all short timers}

S' = {Set of all unexpired short timers}

M = {Set of all medium timers}

L = {Set of all long timers}

L' = {Set of all unexpired long timers}

Having modified the design to accomodate the Timer_Unit, the domain knowledge is complete and sufficient for capturing the problem solving behavior. TOPcaptures each part of the problem solving behavior as a *Production.* Each production has a name that describes the intended action this production will perform, a condition that must be satisfied in order for the desired action to be taken, a body that performs the action by invoking tasks and a post-condition that describes the expected result of performing the actions in the production body. Given this description let's examine who our description of the problem solving behavior for the Traffic Light Problem maps to the solution shown at the back of this study. The unit, *Traffic_System,* contains the highest level productions that exhibit the problem solving behavior described.

At the highest level of the behavior description is the goal to change the light. The production, *Change_The_Light*, performs this action. As specified in the *If* condition of the production, achieving this goal depends on the required period of time expiring; which, of course, matches the problem solving behavior defined above. Next, let's examine the subgoal of causing a period of time to expire. Well, the declarative knowledge explicitly states what causes a period of time to expire, but how is that state achieved? Clearly, this state is achieved by reducing the number of seconds until expiration to zero. The production, *Tick_The_Running_Timer*, performs this action.

Let's examine our next subgoal and that is selecting a period of time to expire. The global constraints shown in the unit, Traffic_System, capture the conditions that guide selection of the appropriate timer based on the requirements (note that these capture conditions involving more than one term). For example, the global constraint, *Timer_Should_Switch*, will flag when a 15 or 60 second interval should be used instead of the longer 120 second interval. Using these abstratct conditions, the productions, *ReStart_The_Running_Timer* and *Switch_Timer* perform the action of selecting the required interval of time to expire.

Now that the problem solving method has been defined, the specific actions each production will take must be defined. Typically, this will involve a stepwise refinement activity involving specification of more abstract tasks that invoke less abstract tasks. For example, the task, Switch_Light in unit Traffic_System invokes the task Switch from unit Traffic_Light to change the light and the tasks Start and Stop from the unit Timer_Unit to set a new expiration time for the next change of the light. The other tasks in Traffic_System also reflect this process of stepwise refinement.

Verification and Validation

Verifying this final step in the process is the most difficult part of the process. The first step is to show that all necessary productions have been defined to achieve the problem solving behavior. It is also necessary to show that the sequencing of these activities is correct. The discussion outlined above is an informal way to describe the problem so that sequencing can be verified. Another way is to use a *state-sequence expression*. A state-sequence expression explicitly dictates the expected order of invoking productions. A simple expression for the Traffic_System unit might be as follows:

```
{ [ Tick_The_Running_Timer |
    ReStart_The_Running_Timer |
    Switch_Timer] -> Tick_The_Running_Timer -> Change_The_Light}
```

This expression simply states that *Tick_The_Running_Timer*, *ReStart_The_Running_Timer* and *Switch_Timer* can be fired in a non-deterministic fashion, but *Tick_The_Running_Timer* must always precede firing the *Change_The_Light* production.

Next, all *pre* and *post* conditions must be verified as correct. This is a very detailed process of mapping conditions in the productions to the composition of conditions from the invoked tasks. For example, the If condition of the production, Change_The_Light, must match the Requires condition for the Switch_Light task. In addition, the result of executing Switch_Light must produce a result that is compatible with the post-condition, if any, of *Change_The_Light.* Fortunately, this is easy when post-conditions have been specified. For this case, simply match the To Produce clause of the *Switch_Light* task and the To Produce clause of the *Change_The_Light* production.

Next, any tasks invoked by higher level tasks need to have their *pre* and *post* conditions matched against the conditions in the invoking task. For example, in the task, *Switch_Light,* it follows that the task *Stop* can be invoked for the timer that just expired because an *Expired_Timer* is considered a *Running_Timer* and the passed timer must be a *Running_Timer* for Stop to be used. This process is repeated until all tasks are shown to produce the correct results with respect to the productions that invoked them.

Specifications

Package Sensors Is

```
--<*
-- State Data
--
-- Model
--
-- A sensor is an item that contains (or sends) signals. Other
-- objects "read" the sensor to access new signals. A sensor
-- can be "read" only after it has been "opened."
--
            Concept Sensor Is Primitive;
            Concept Open_Sensor Specializes Sensor And Is Primitive;

-- Constraints
            -- N/A

-- Initialization
            Traffic_Sensor Is_A Sensor;

-- End State Data
--*>

--<*
-- Transitions
--

-- Problem Solving Method
            -- Whenever a signal has not been received and sensor is
            -- "open" then the sensor should be "read" for new signal
            -- values
            --
            Production Open_Sensors Is
              If
                S Is_A Sensor And
                NOT S Is_A Open_Sensor
              Then
                Perform Open(S)
            End Production;

            --<*
            -- Method Open(S: In Out Sensor)
            --   will open a sensor for processing
            -- End Open;
            --
            Method Open(Sn: Sensor);
            --*>
```

```
  --  End Transitions
  --*>

End Sensors;

With Sensors;
Package Signals Is

--<*
--  State Data
--
--  Model
--
--  The signals package captures the notion of a signal.  A
--  signal (represented by a 0 or 1) is used to notify the
--  traffic controller that some external event has happened.
--  A signal is considered to be "received" when a new indicator
--  is received from the sensor.  A signal is considered to
--  be "triggered" when the sensed value is a 1 from a "received"
--  signal.
--
            Concept  Signal Is Primitive;
            Concept  Indicator Specializes Number And Is Primitive;
            Concept  On_Indicator Specializes Indicator And Is Defined By
               {
                 An indicator is ON when its value is 1
               }
               i Such That i Is_A Indicator And i = 1
            End Concept;
            Concept  Off_Indicator Specializes Indicator And Is Defined By
               {
                 An indicator is OFF when its value is 0
               }
               i Such That i Is_A Indicator And i = 0
            End Concept;
            Relation Has_Approaching(S: Signal; I: Indicator) Is Primitive;
            Relation Has_Waiting(S: Signal; I: Indicator) Is Primitive;
            Concept  Received_Signal Specializes Signal And Is Defined By
               {
                  A Received_But_Not_Processed_Signal is a Signal
                  that Has_Indicator I that has just been received from a
                  sensor.
               }
               r Such That r Is_A Signal And
                              r Has_Approaching i1 And
                              r Has_Waiting i2
            End Concept;
            Concept  Received_And_Processed_Signal Specializes
                   Received_Signal And Is Primitive;
```

```
Concept Received_But_Not_Processed_Signal Specializes
    Received_Signal And Is Defined By
    {
      If a received signal has not been processed then it is
      a "received_but_not_processed" signal
    }
    t Such That t Is_A Received_Signal And
              NOT t Is_A Received_And_Processed_Signal
End Concept;

Concept Waiting_Only_Signal Specializes
    Received_But_Not_Processed_Signal And Is Defined By
    {
      S Is_A Waiting_Only_Signal when only the
        Waiting_Signal is triggered
    }
    s Such That
      s Is_A Received_But_Not_Processed_Signal
      s Has_Approaching i1 And
       i1 Is_A Off_Indicator And
       s Has_Waiting i2 And
       i2 Is_A On_Indicator
End Concept;

Concept Waiting_And_Approaching_Signal Specializes
        Received_But_Not_Processed_Signal And Is Defined By
        {
          S Is_A Waiting_And_Approaching_Signal when
            both the Waiting_Signal and
                      Approaching_Signal is triggered
        }
        s Such That
            s Is_A Received_But_Not_Processed_Signal
              And
            s Has_Approaching i1 And
            i1 Is_A On_Indicator And
            s Has_Waiting i2 And
            i2 Is_A On_Indicator
End Concept;

Concept Approaching_Only_Signal Specializes
        Received_But_Not_Processed_Signal And Is Defined By
        {
          S Is_A Approaching_Only_Signal when only the
          Approaching_Signal is triggered
        }
        s Such That
            s Is_A Received_But_Not_Processed_Signal
              And
            s Has_Approaching i1 And
             i1 Is_A On_Indicator And
            s Has_Waiting i2 And
             i2 Is_A Off_Indicator
End Concept;
```

```
          Concept No_Waiting_Or_Approaching_Signal Specializes
                 Received_But_Not_Processed_Signal And Is Defined By
                 {
                   S Is_A Approaching_Only_Signal when only the
                   Approaching_Signal is triggered
                 }
                 s Such That
                     s Is_A Received_But_Not_Processed_Signal
                       And
                     s Has_Approaching i1 And
                      i1 Is_A Off_Indicator And
                     s Has_Waiting i2 And
                      i2 Is_A Off_Indicator
          End Concept;

    -  Constraints
       -- N/A

    -  Initialization
       Traffic_Signal : Signal;

    -  End State Data
    --*>

  --<*
  -  Transitions
  --
          --<*
          -- Whenever a signal has not been received and sensor is
          -- "open" then the sensor should be "read" for new signal
          --  values
          --
          Production Get_New_Signals Is
               If
                 Traffic_Sensor: Open_Sensor And
                 NOT Traffic_Signal: Received_Signal
               Then
                  Perform Sense(Traffic_Signal, Traffic_Sensor)
          End Production;
          --*>

          --<*
          -- Method Sense(s: in signal)
          --    will retrieve a new indicator from the sensor
          -- End Sense;
          --
          Method Sense(s: Signal; sn: Sensor);
          --*>

          --<*
          -- Method Reset(s: in received_signal)
          --    will indicate that the received_signal, s, has been
          --    processed and cannot be processed again until a
          --    new indicator has been received
```

```
            --  End Reset;
            --
            Method Reset(s: Signal);
            --*>
--  End Transitions
--*>

End Signals;


With Signals;
Package Timer_Unit Is

--<*
-- State Data
--
-- Model
--
-- A Timer is an item that serves to mark the elapse of a given
-- period of time.  A Timer is considered to by "set" when a
-- given period of time is associated with that timer.  A "set"
-- timer is "expired" when that given period of time expires
-- (i.e., is 0)
--
            Concept Timer Is Primitive;
            Concept Tick Specializes Number And Is Primitive;
            Relation Expires_In(T: Timer; CT: Tick) Is Primitive;
            Relation Has_Expiration_Value(T: Timer; CT: Tick) Is
                 Primitive;
            Relation Has_Secondary(P: Timer; S: Timer) Is Primitive;

            Relation Is_Secondary_To(S: Timer; P: Timer)
               Is Defined By
               {
                 P Is_Secondary_To S when S Has_Secondary P
               }
               (s, p) Such That p Is_A Timer And s Is_A Timer And
                                  p Has_Secondary s
            End Relation;

            Relation Switches_To(P: Timer; S: Timer) Is Primitive;
            Concept Running_Timer Specializes Timer And Is Primitive;

            Concept Long_Timer Specializes Timer And Is Defined By
               {
                 The Long_Timer expires in 120 seconds
               }
               t Such That t Is_A Timer And
                              t Has_expiration_value ev And ev = 120
            End Concept;

            Concept Medium_Timer Specializes Timer And Is Defined By
```

```
    {
     The Medium_Timer expires in 60 seconds
    }
    t Such That t Is_A Timer And
                   t Has_Expiration_Value ev And ev = 60
End Concept;

Concept Short_Timer Specializes Timer And Is Defined By
    {
     The Short_Timer expires in 15 seconds
    }
    t Such That t Is_A Timer And
                   t Has_Expiration_Value ev And ev = 15
End Concept;

Concept Expired_Timer Specializes Running_Timer And
    Is Defined By
    {
     Only an "running" timer can expire.  Expiration occurs
     when the seconds remaining before expiration is 0.
    }
    t Such That t Is_A Running_Timer And
                   t Expires_in w And w = 0
End Concept;

Concept UnExpired_Short_Running_Timer Specializes
    Running_Timer And Is Defined By
    {
     A short timer that is running but has not expired
    }
    t Such That t Is_A Running_Timer And
                   t Is_A Short_Timer And
                   NOT t Is_A Expired_Timer
End Concept;

Concept UnExpired_Long_Running_Timer Specializes
    Running_Timer And Is Defined By
    {
     A long timer that is running but has not expired
    }
    t Such That t Is_A Running_Timer And
                   t Is_A Long_Timer And
                   NOT t Is_A Expired_Timer
End Concept;

-- Constraints
    Global Constraint
        Timer_To_Use_When_None_Are_Running
        Specializes Timer And Is Defined By
        {
         Use the long timer when no other timers are running
        }
        t Such That t Is_A Long_Timer And
                       NOT t Is_A Running_Timer And
                       ( s Is_A Short_Timer And
                                NOT s Is_A Running_Timer) And
                               ( m Is_A Medium_Timer And
                                NOT m Is_A Running_Timer)
                End Global Constraint;

-- Initialization
                  M Is_A Timer
                    That Has_Expiration_Value 60;

                  S Is_A Timer
                    That Has_Expiration_Value 15 And
                         Has_Secondary M;

                  L Is_A Timer
                    That Has_Expiration_Value 120 And
                         Switches_To S;

-- End State Data
--*>

--<*
-- Transitions
--
                --<*
                -- Whenever all timers are not running, start the timer the
                --   primary timer (in this case, the long timer)
                --
                Production Initial_Timer_Start Is
                  If
                    t: Timer_To_Use_When_None_Are_Running
                  Then
                    Perform Start(t)
                End Production;
                --*>

                --<*
                -- Method Stop(t: Timer) Is
                --   Stop a running timer
                -- End Stop;
                --
                Method Stop(t: Timer);
                --*>

                --<*
                -- Method Start(t: Timer) Is
                --   Start a timer that is not running
                -- End Start;
                --
                Method Start(t: Timer);
                --*>
--
-- End Transitions
--*>
```

```
End Timer_Unit;

With Timer_Unit;
Package Traffic_Light Is

--<*
--  State Data
--
--  Model
--
--  A "light" is an item that controls the flow of traffic in
--  a given direction.  The control of traffic flow is achieved
--  through the use of colors (red and green).
--
              Concept Light Is Primitive;
              Concept Red_Light Specializes Light And Is Primitive;
              Concept Green_Light Specializes Light And Is Primitive;

--  Constraints
     -- N/A

--  Initialization
    NS_Light : Red_Light;

--  End State Data
--*>

--<*
--  Transitions
--
              -- Method Switch(l: light)
              --    will switch the color of the light in a given direction
              -- End Switch;
              --
              Method Switch(l: Light);
              --*>

--  End Transitions
--*>

End Traffic_Light;

With Traffic_Light;
With Timer_Unit;
Package Traffic_System Is

--<*
--  State Data
```

```
--
-- Model
--
-- Timers fall into certain "categories" based on the traffic
-- conditions. Timer_Should_Tick, Timer_Should_Switch and
-- Timer_Should_Be_ReStarted define the possible categories
-- for a timer based on traffic conditions.

-- Constraints
          Global Constraint Timer_Should_Tick(t: Timer; s: Signal)
            Is Defined By
            {
             A Timer_Should_Tick when the no approaching or waiting
             traffic is detected
            }
            t Such That t Is_A Running_Timer And
                         NOT t Is_A Expired_Timer And
                         s Is_A No_Waiting_Or_Approaching_Signal
          End Global Constraint;

          Global Constraint Timer_Should_Switch(t: Timer; s: Signal)
            Is Defined By
            {
             A Timer_Should_Switch when the long timer is running
              and a waiting signal is received.
            }
            t Such That t Is_A UnExpired_Long_Running_Timer And
                        ( s Is_A Waiting_Only_Signal Or
                          s Is_A Waiting_And_Approaching_Signal )
          End Global Constraint;

          Global Constraint Timer_Should_Be_Restarted(t: Timer;
                                                       s: Signal)
            Is Defined By
            {
             A Timer_Should_Be_ReStarted when the running timer
             has not expired and the current signal indicates
             approaching traffic. When the running timer is a long timer
             a waiting signal will take precedence over the approaching
             signal.
            }
            t Such That ( t Is_A UnExpired_Short_Running_Timer And
                          ( s Is_A Approaching_Only_Signal Or
                            s Is_A Waiting_And_Approaching_Signal))
                        Or
                        ( t Is_A UnExpired_Long_Running_Timer And
                          s Is_A Approaching_Only_Signal )
          End Global Constraint;

          Global Constraint Long_Timer_Expired_At(t: Timer; s: Signal)
            Is Defined By
            {
             A Long_Timer_Expired_At when the running timer is
                long and it has expired and a new signal has been
                received but not processed.
                }
                t Such That t Is_A Long_Timer And
                            t Is_A Expired_Timer And
                           s Is_A Received_But_Not_Processed_Signal
            End Global Constraint;

            Global Constraint Medium_Timer_Expired_At(t: Timer;
                                                      s: Signal)
              Is Defined By
              {
               A Medium_Timer_Expired_At when the running timer is
               medium and it has expired and a new signal has been
               received but not processed.
              }
              t Such That t Is_A Medium_Timer And
                          t Is_A Expired_Timer And
                         s Is_A Received_But_Not_Processed_Signal
            End Global Constraint;

            Global Constraint Short_Timer_Expired_At(t: Timer; s: Signal)
               Is Defined By
               {
                A Short_Timer_Expired_At when the running timer is
                short and it has expired and a new signal has been
                received but not processed.
                }
                t Such That t Is_A Short_Timer And
                         t Is_A Expired_Timer And
                         s Is_A Received_But_Not_Processed_Signal
            End Global Constraint;

-- Initialization
            NS_Light : Red_Light;

-- End State Data
--*>
--<*
-- Transitions
--
            --<*
            -- Whenever the long timer is running and waiting traffic is
            --    detected then switch to running the short and medium
            --    timers
            --
            Production Switch_Timer Is
              If
                Timer t Should_Switch Because of s And
                s Is_A Received_But_Not_Processed_Signal
              Then
                 Perform Switch_Timer(t)
                 Perform Reset(s)
            End Production;
            --*>

            --<*
```

```
-- Whenever no approaching or waiting traffic is detected
-- the currently running timer should be pulsed
--
Production Tick_The_Running_Timer Is
   If
     Timer t Should_Tick Because of s And
     s Is_A Received_But_Not_Processed_Signal
   Then
     Perform Do_Tick(t)
     Perform Reset(s)
End Production;
--*>

--<*
-- Whenever the long timer is running and approaching
-- traffic (only) is detected or the short/medium timers are
-- running and approaching traffic is detected (irregardless
-- of waiting traffic) the running timer should be restarted
--
Production ReStart_The_Running_Timer Is
   If
     Timer t Should_Be_Restarted Because of s And
     s Is_A Received_But_Not_Processed_Signal
   Then
     Perform Re_Start(t)
     Perform Reset(s)
End Production;
--*>

--<*
-- Whenever a running timer expires, the light should change
--    and all timers are stopped
--
Production Change_The_Light Is
   If
     t Is_A Expired_Timer
       Then
        Perform Switch_Light(NS_Light)
End Production;
--*>

--<*
-- Method Do_Tick(t: Timer) Is
--    Decrements the number of seconds until a timer
--    expires. In the case where a timer has a secondary
--    timer (i.e., one that runs at the same time), both timers
--    are decremented.
-- End Do_Tick;
--
Method Do_Tick(t: Timer);
--*>

--<*
-- Method Re_Start(t: Timer) Is
--    Stops and Starts the timer at its maximum expiration
```

```
            --    time.
            -- End Re_Start;
            --
            Method Re_Start(t: Timer);
            --*>

            --<*
            -- Method Switch_Timer(t: Timer) Is
            --    Stops the currently running timer and turns on the
            --     short/medium timers to measure when light should
            --     change
            -- End Swich_Timer;
            --
            Method Switch_Timer(t: Timer);
            --*>

            --<*
            -- Method Switch_Light(t: Timer) Is
            --    Changes the color of the light and stops running timer(s).
            -- End Switch_Light;
            --
            Method Switch_Light(l: Light);
            --*>

    -- End Transitions
    --*>

End Traffic_System;
```

Bodies

Package Body Sensors Is

```
--<*
-- Transitions
--
        --<*
        -- Method Open(S: In Out Sensor)
        --   will open a sensor for processing
        -- End Open;
        --
        Method Open(Sn: Sensor) Is
          Requires Sn Is_A Sensor And
                   NOT Sn Is_A Open_Sensor
          Involves Open physical file
                   Assert Sn Is_A Open_Sensor
          To Produce Sn Is_A Open_Sensor
        End Method;
        --*>

-- End Transitions
--*>
```

End Sensors;

Package Body Signals Is

```
--<*
-- Transitions
--
        --<*
        -- Method Sense(s: in signal)
        --   will retrieve a new indicator from the sensor
        -- End Sense;
        --
        Method Sense(s: Signal; sn: Sensor) Is
          Requires  s Is_A Signal And
                    sn Is_A Open_Sensor
                    NOT s Is_A Received_Signal
          Involves  i = indicator from Sensor
                    If sensor finished transmitting Then
                      halt
                    End If
                    Assert i Is_A Indicator
                    Assert s Has_Approaching i
                    i = next Indicator from Sensor
                    If Sensor finished transmitting Then
                      halt
                    End If
                    Assert i Is_A Indicator
                    Assert s Has_Waiting i
            To Produce s Is_A Received_Signal And
                        s Is_A Received_But_Not_Processed_Signal
        End Method;
        --*>

        --<*
        --  Method Reset(s: in received_signal)
        --    will indicate that the received_signal, s, has been
        --    processed and cannot be processed again until a
        --    new indicator has been received
        --  End Reset;
        --
        Method Reset(s: Signal) Is
            Requires  s Is_A Received_Signal And
                        s Is_A Received_And_Processed_Signal And
                       ( s Has_Approaching i1 And
                         i1 Is_A Indicator ) And
                       ( s Has_Waiting i2) And
                         i2: Indicator )
            Involves  Retract i1 Is_A Indicator
                      Retract s Has_Approaching i1
                      Retract i2 Is_A Indicator
                      Retract s Has_Waiting i2
                      Retract s Is_A Received_Signal
            To Produce s Is_A Signal And
                         NOT s Is_A Received_Signal
        End Method;


--  End Transitions
--*>
```

End Signals;

Package Body Timer_Unit Is

```
--<*
-- Transitions
--
            --<*
            --  Method Stop(t: Timer) Is
            --    Stop a running timer
            --  End Stop;
            --
            Method Stop(t: Timer) Is
                Requires   t Is_A Running_Timer And
                            t Expires_In e
                Involves   Retract t Is_A Running_Timer
                           Retract t Expires_In e
            To Produce t Is_A Timer
          End Method;
          --*>

          --<*
          -- Method Start(t: Timer) Is
          --    Start a timer that is not running
          -- End Start;
          --
          Method Start(t: Timer) Is
             Requires  t Is_A Timer And
                       NOT t Is_A Running_Timer And
                       t Has_Expiration_Value ev
             Involves  Assert t Is_A Running_Timer
                       Assert t Expires_In ev
            To Produce t Is_A Running_Timer
          End Method;
          --*>

--
-- End Transitions
--*>
```

End Timer_Unit;

Package Body Traffic_Light Is

```
--<*
-- Transitions
--
          --<*
          -- Method Switch(l: light)
          --    will switch the color of the light in a given direction
          --    (when red switch to green)
          --    (when green switch to red)
          -- End Switch;
          --
          Method Switch(l: Light) Is
             Used When  l Is_A Green_Light
             Requires  NOT l Is_A Red_Light
             Involves  Retract l Is_A Green_Light
                       Assert  l Is_A Red_Light
             To produce l Is_A Red_Light And
                        NOT l Is_A Green_Light
          End Method;

          Method Switch(l: Light) Is
             Used When  l Is_A Red_Light
             Requires  NOT l Is_A Green_Light
             Involves  Retract l Is_A Red_Light
                       Assert  l Is_A Green_Light
            To Produce l Is_A Green_Light And
```

```
                    NOT l Is_A Red_Light
            End Method;
            --*>
--  End Transitions
--*>

End Traffic_Light;


Package Body Traffic_System Is

  --<*
  --  Transitions
  --
            --<*
            -- Method Do_Tick(t: Timer) Is
            --    Decrements the number of seconds until a timer expires.
            --    In the case where a timer has a secondary timer (i.e.,
            --    one that runs at the same time), both timers are
            --    decremented.
            -- End Do_Tick;
            --
            Method Do_Tick(t: Timer) Is
               Used When  t Is_A Long_Timer Or t Is_A Medium_Timer
               Requires   Timer t Should_Tick Because of s And
                           t Expires_In w And
                           s Is_A Received_But_Not_Processed_Signal
               Involves   Retract t Expires_In w
                         Assert  t Expires_In (w-1)
                         Assert  s Is_A Received_And_Processed_Signal
              To Produce s Is_A Received_And_Processed_Signal And
                           t Expires_In (w-1)
            End Method;

            Method Do_Tick(t: Timer) Is
               Used When t Is_A Short_Timer
               Requires  Timer t Should_Tick Because of s And
                          t Has_Secondary m And
                          t Expires_In w And
                          s Is_A Received_But_Not_Processed_Signal
               Involves  Retract t Expires_In w
                         Assert  t Expires_In (w-1)
                         Perform Do_Tick(m)
               To Produce s Is_A Received_And_Processed_Signal And
                           t Expires_In (w-1)
                          m Expires_In 1 fewer seconds
            End Method;
            --*>

            --<*
            -- Method Re_Start(t: Timer) Is
            --    Stops and Starts the timer at its maximum expiration
            --    time.
-- End Re_Start;
--
Method Re_Start(t: Timer) Is
    Requires  Timer t Should_Be_ReStarted Because of s
                  And
              s Is_A Received_But_Not_Processed_Signal
    Involves  Perform Stop(?t)
              Perform Start(?t)
              Assert s Is_A Received_And_Processed_Signal
   To Produce s Is_A Received_And_Processed_Signal
              t Has_Expiration_Value w1 And
              t Expires_In w2 seconds And
              w1 = w2
End Method;
--*>

--<*
-- Method Switch_Timer(t: Timer) Is
--    Stops the currently running timer and starts the
--     short/medium timers for measuring light change
-- End Switch_Timer;
--
Method Switch_Timer(t: Timer) Is
    Requires  Timer t Should_Switch Because of s And
              t Switches_To pri And
              pri Has_Secondary sec And
              s Is_A Received_But_Not_Processed_Signal
    Involves  Perform Stop(t)
              Perform Start(pri)
              Perform Start(sec)
              Assert s Is_A Received_And_Processed_Signal
   To Produce NOT t Is_A Running_Timer And
              pri Is_A Running_Timer And
              sec Is_A Running_Timer And
              s Is_A Received_And_Processed_Signal
End Method;
--*>

--<*
-- Method Switch_Light(t: Timer) Is
--    Changes the color of the light and stops running
--     timer(s).
-- End Switch_Light;
--
Method Switch_Light(l: Light) Is
    Used When  Long_Timer t Expired_On s
    Requires  t Is_A Expired_Timer And
              s Is_A Received_But_Not_Processed_Signal
    Involves  Perform Switch(l)
              Perform Stop(t)
              Assert s Is_A Received_And_Processed_Signal
    To Produce
              NOT s Is_A Received_And_Processed_Signal
End Method;
```

```
        Method Switch_Light(l: Light) Is
           Used When  Short_Timer t Expired_On sig
           Requires   t Has_secondary s And
                      t Is_A Expired_Timer And
                      sig Is_A Received_But_Not_Processed_Signal
           Involves   Perform Switch(l)
                   Perform Stop(t)
                   Perform Stop(s)
                   Assert sig Is_A Received_And_Processed_Signal
           To Produce NOT t Is_A Running_Timer And
                         sig Is_A Received_And_Processed_Signal
        End Method;

        Method Switch_Light(l: Light) Is
           Used When  Medium_Timer t Expired_On sig
           Requires   t Is_Secondary_To s And
                       t Is_A Expired_Timer And
                      sig Is_A Received_But_Not_Processed_Signal
           Involves   Perform Switch(l)
                   Perform Stop(t)
                   Perform Stop(s)
                   Assert sig Is_A Received_And_Processed_Signal
          To Produce NOT t Is_A Running_Timer And
                        sig Is_A Received_And_Processed_Signal
        End Method;
--
-- End Transitions
--*>
```

End Traffic_System;

Case Study #2: A Cleanroom Approach to the Traffic Controller Problem[1]

Authors:

Fred Highland, Brent Kornman

IBM Corporation
100 Lake Forest Blvd
Gaithersburg, MD

[1]The following writeup has been edited slightly by Scott French and David Hamilton for inclusion in the classroom material.

Introduction

Technologies such as Cleanroom Software Engineering (Mills, et. al, 1987) promise to dramatically improve the quality of software products by allowing their correctness to be formally verified. In order to use these technologies, the design must be specified in a design language and verification techniques must be used to prove the design is correct. Numerous languages and techniques have been developed to specify and verify the designs for procedural software. However, very little has been done for Knowledge Based Systems (KBS). The methodologies for designing KBS are poorly understood and verification and test even less understood.

The purpose of this case study is to discuss a language for the design and verification of KBS application software. The basic intuitions and requirements for the design language are discussed first followed by an outline of the design language syntax and semantics. Next, the characteristics of the language are applied to defined a solution for the traffic controller problem.

Basic Concepts

The design language presented here is based on two important intuitions about KBS:

- they are a mixture of procedural and non-procedural programming techniques
- they are not just unorganized collections of rules and frames but are intended to operate in a specific manner by the developer

The idea that KBS are built from a mixture of procedural and non-procedural programming techniques derives from the fact that many solutions are not strictly procedural or non-procedural in nature. Rather, solution approaches are composed of a number of different subprocesses with different interactions. Some are dependent on the results of other processes and must be organized procedurally. Others may be performed independently or in parallel once the proper context is established. It is this latter type that KBS technologies, with their implicit control mechanisms, are best suited for. But it requires a mixture of the two forms to produce a complete solution.

The idea that KBS are not unorganized collections of rules and frames is more subtle. While some useful systems have been built this way, most applications are of such a complexity that some organization or process must be used to decompose the problem. This typically takes the form of a set of steps that must be performed or sequences of events that must occur in order to solve the problem. This may be represented with state or control variables which determine which rules are applicable at any point in time or it may be implicit in the changes and availability of the objects referenced by the rules. In the latter case, control is provided more by the inference engine than by the user. But often the implicit control is not exactly what is desired and meta-level controls or changes to the rules must be used to produce the desired result. In either case, there is implicit meta knowledge in the problem solving process which is usually present in the mind of the application builder but often hidden in the implementation.

These two intuitions suggest that KBS application design could be captured in a language that is based, in part, on existing procedural software design languages but with extensions that exploit the characteristics of KBS programming.

For practical reasons, the design language must also meet the following requirements:

- the design should be verifiable with a reasonable amount of effort and without a deep understanding of the underlying KBS tool
- the design should be easily translatable into the underlying KBS tool's knowledge representation language

These two requirements are conflicting, in that the language, to be easily verifiable, should be as procedural as possible since techniques for verifying procedural designs are understood. However, for the language to be translatable to a KBS tool's representation language, it must exhibit a non-procedural, declarative style, which is inherently difficult to verify.

Design Language Specification

The KBS Design Language (KDL) implements the requirements defined above for a design language. The following sections summarize KDL's definition in terms of syntax, semantics and correctness conditions.

Syntax

The syntax of the unique components of the KDL is summarized in figure . This design language is not meant to replace existing procedural design languages but rather to augment them to deal with the concepts embodied in KBS programming. The definitions of *global_data_definitions*, *local_data_definitions*, and *actions* in **WHEN** and **WHENEVER** statements are left unspecified in this definition so that structures from other design or implementation languages may be used to specify details. This allows the use of procedural control structures in the actions of **WHEN** and **WHENEVER** statements in order to express functions that may be better expressed using procedural means (e.g. **WHILE** loops, **IF** statements, etc.).

```
KB SEGMENT kb_segment_name (arguments)
  [segment_intended_function]

  GLOBAL DATA
  global_data_definitions

  LOCAL DATA
  local_data_definitions

  [when_intended_function]
  when_name WHEN
    [condition_expression]

  DO INTERRUPTIBLE
    [when_action_intended_function]

    actions

  END

  [whenever_intended_function]
  whenever_name WHENEVER
    [condition_expression]

  DO
    [whenever_action_intended_function]

    actions

  END

END KB SEGMENT kb_segment_name
```

Figure 1: KB Design Language Syntax

Semantics

The semantics of the design language are defined to accomplish the following goals:

- define the legal operation of the constructs
- restrict usage of the constructs to allow verification
- maximize the KBS tool independence of the language

The semantics of each of the basic components of the language, **KB SEGMENT**, **WHEN** statements, and **WHENEVER** statements, are discussed below.

__KB Segments:__ The **KB SEGMENT** provides the highest level of modularization and scoping for a knowledge base. It defines a logical unit of work that performs a single [segment_intended_function]. KBS applications may be composed of one or more **KB SEGMENTs** that may interact with other **KB SEGMENTs** or procedural functions.

:p.

A KB SEGMENT is composed of definitions for global and local data, one or more WHEN statements and zero or more WHENEVER statements. The WHEN statements completely implement the :pv.segment_intended_function:epv. of the **KB SEGMENT** in a non-deterministic manner. The **WHENEVER** statements support the **WHEN** statements by providing opportunistic and data driven functions that can be used to achieve the functions of a **WHEN** action. **WHENEVERs** are not active outside of the context of an active **WHEN** statement. However, their functionality can be shared by all **WHEN** statements.

__WHEN Statements:__ **WHEN** statements represent a condition under which one or more actions are to be performed. Their intent is to explicitly represent meta or control knowledge in the design of the system and the conditions under which that processing is appropriate.

The requirement of non-determinism of **WHEN** statements in accomplishing the [segment_intended_function] allows for the specification of multiple possible solution scenarios while forcing those scenarios to be independent of each other. This specifically disallows the execution of a sequence of **WHEN** statements to accomplish the [segment_intended_function] as such would represent an implicit intent of control which would be difficult to verify.

The **WHEN** statement is composed of a [when_intended_function], a [condition_expression], and a **WHEN** action part. The [when_intended_function] specifies the abstract condition under which this **WHEN** statement is appropriate, and the effect it will have. The [condition_expression] provides a more concrete specification of the appropriateness conditions. The **WHEN** action part specifies a sequence of functions that implement the [when_action_intended_function]. These functions are specified with procedural specifications that represent the sequence of processing. They may be implemented using a mixture of procedural design statements and **WHENEVER** statements. When **WHENEVER** statements are used, their intended function is specified in the **WHEN** actions so that the **WHEN** statement can be verified in a self-contained manner. The :pv.actions:epv. of a **WHEN** statement may also specify a **CALL KB SEGMENT** action whose intent it is to invoke another **KB SEGMENT**.

The actions. of a **WHEN** statement allow two fomns of execution to provide for different implementation approaches. The **DO** form specifies that all actions within the structure are executed sequentially without interruption. This is the nomal semantic of procedural programming languages and is appropriate if the implementation is to use either procedural programming or rule actions without demons.

The **DO INTERRUPTIBLE** fomn specifies that **WHENEVER** statements apply between each of the actions. This allows **WHENEVER** statements to be applied as soon as the appropriate condition exists. **DO INTERRUPTIBLE** blocks may contain **DO** blocks to specify that certain groups of actions are not interruptible. **WHENEVER** statements apply only between individual :pv.actions:epv. and **DO** blocks within a **DO INTERRUPTIBLE** block.

WHENEVER Statements: **WHENEVER** statements represent opportunistic or data driven rules or demons that may fire at any time, and as many times as necessary during the execution of a **DO INTERRUPTIBLE** block of a **WHEN** statement. If more than one **WHENEVER** is eligible to fire (i.e. its [condition_expression] evaluates to true) the order of firing of the **WHENEVER** statements can not produce different results. As with **WHEN** statements, such a required ordering represents an implicit control that should be explicitly stated in the design.

The components of a **WHENEVER** statement are similar to that of a **WHEN** providing a whenever_intended_function., a [condition_expression], and a **WHEN** action. Unlike the **WHEN** statement, however, the actions of a **WHENEVER** statement are performed sequentially and are not interruptible by other **WHENEVER** statements.

Correctness Conditions

A set of correctness conditions or proof rules for verifying that a design is correct have been defined. These allow verification of the design at various levels of abstraction, allowing either top-down or bottom-up verification techniques to be used.

Using a top down approach, the verification stages and associated primitives are as follows:

KB SEGMENT:	*[segment_intended_function]* is implemented by *[when_intended_function]*s
WHEN:	*[when_intended_function]* is implemented by **WHEN** statement
WHEN Action Part:	*[when_action_intended_function]* is implemented by **WHEN** actions

WHEN INTERRUPTIBLE Actions:	**WHEN** actions are implemented by their refinement and by applicable **WHENEVER** statements
WHEN (uninterruptible) Actions:	**WHEN** actions are implemented by their refinement
WHENEVER	*[whenever_intended_function]* is implemented by **WHENEVER** statement
WHENEVER Action Part:	*[whenever_action_intended_function]* is implemented by **WHENEVER** actions

Correctness conditions are defined for each construct or set of constructs at each level of abstraction as mentioned above. The general approach to the correctness conditions is to verify that the components of the construct implement the function of the construct and that the components are well behaved with respect to the restrictions imposed on them by the semantics of the design language. This involves verifying that improper interactions do not occur and that the results are deterministic.

The most significant part of the verification process with this design language is the verification of the **KB SEGMENT**. and the **WHEN INTERRUPTIBLE** actions. The verification of other parts of the language follows approaches similar to those used with procedural programming languages.

The **KB SEGMENT** is correct if:

1 For all arguments, does performing all **WHEN**s accomplish *[segment_intended_function]*?

2 Are all [when_intended_function]s independent of all other *[when_intended_function]*s? That is, could the result of one *[when_intended_function]* modify data used in another *[when_intended_function]*?

The first correctness condition is easily verified by comparison with the *[segment_intended_function]* and consideration of the data being processed. Each logical set of data must meet the condition of and be properly processed by the *[when_intended_function]*. The second correctness condition verifies that a **WHEN** applies only once to a logical set of data. If sequences of **WHEN**s are required to accomplish the intended function, then there is implicit control that has not been specified and has been left for the reviewer to discover. Hence, this restriction not only makes verification easier but forces control to be explicit.

A **WHEN INTERRUPTIBLE** Action is correct if, for all arguments:

1 Does performing the implementation of the **WHEN** action and applicable **WHENEVERs** accomplish the action

2 Does the execution of applicable **WHENEVERs** terminate?

3 Does the execution of applicable **WHENEVERs** produce the same results regardless of order (i.e. is the result of the execution deterministic)?

These verification rules interact to verify that a set of **WHENEVERs** accomplish the intended function of a **WHEN** action. These rules allow latitude on the part of the designer in using **WHENEVERs**, but this must be balanced with verifiability. The first rule requires that all **WHENEVERs** in a **KB SEGMENT** be examined to determine if their applicability is appropriate. The second rule allows multiple **WHENEVERs** to be used to accomplish a function but requires that their termination must be verifiable. The third rule requires that the results of execution of multiple **WHENEVERs** be deterministic and that implicit control sequences are not present. Verification of WHEN **INTERRUPTIBLE** actions is potentially difficult because of the difficulty in predicting the sequence of **WHENEVER** application. However, the structure of the design language encourages isolation of function to small sets of **WHENEVERs** that are more easily verified.

Discussion

The KDL provides a structure that distinguishes control and opportunistic knowledge in the design of a KBS. The explicit representation of control knowledge is important because it provides a means to specify the abstract control flow the knowledge base was designed to use. As knowledge bases are typically data driven, this type of information is often encoded in rules along with other information using state variables, priorities, or the conflict resolution scheme of the underlying system. This makes the control strategies implicit and difficult to find, inhibiting understanding, debugging, and verification. By providing a mechanism to represent control, the intentions of the designer are made explicit and its correctness can be more easily verified. This does not restrict the implementation from using traditional techniques, such as state variables or priorities, but specifies the effect that must be acheived for the implementation to be correct.

While the explicit representation of control knowledge is important, the representation of data driven and opportunistic knowledge is a key feature of the KBS approach. This is also represented in the language in the form of **WHENEVER** statements. As these are pattern driven procedural statements, they can be used to represent any processing that should be performed under a given set of conditions. They can also be used to represent demons triggered by various actions that occur against data in the KBS making this representation useful for mixed KBS and Object Oriented paradigms.

The work done on TOP (Terms, Operators, and Productions described in the first solution to the Traffic Controller problem) embodies many similar concepts to the work presented here. TOP Operators have similar characteristics to **WHEN** statements and TOP Productions have similar characteristics to **WHENEVER** statements. TOP Terms provide a much more formal definition of knowledge base objects and their semantics than is specified in the KDL. In general, the TOP language is a precise KBS development language that can be used to specify designs and be automatically translated into a particular KBS tool langauge. The KDL is a much more flexible extension to existing design languages. Additionally, the verification arguments for TOP have only been informally defined and the language does not contain the semantic restrictions that simplify verification. The KDL provides restrictions on the use of language constructs, defines of the relationship between the constructs, and provides formal correctness conditions to allow verification to occur. However, the similarities of the two efforts should allow some of the verification characteristics of KDL to be applied to TOP.

A more general approach to knowledge base verification involving the use of relational verification techniques has been proposed. However, these techniques are difficult to use, making them currently impractical for use on real problems. The KDL attempts to avoid this problem by separating control and opportunistic knowledge and providing mechanisms for defining the function of groups of opportunistic rules to limit the need for relational verification to small, easily managed sets of rules.

The KDL is being used in the development of the Automated Problem Resolution (APR) prototype. The APR prototype is an aircraft flight replanning system being developed as part of a study for future upgrades the the U.S. Federal Aviation Administration's Air Traffic Control system. The system requires the generation of multiple aircraft maneuvers in a multiple problem environment and is a non-trivial problem in terms of representation, problem solving approaches, and performance.

Our experience with the design language to date has been very positive. It provides a vehicle to represent the designs that we are specifying for the APR project. It allows us to specify the types of processing we expected to do in with KBS tools (TIRS in this case) with a minimum of restrictions. It also provides a good mechanism to abstract the design at various levels allowing the use of top-down stepwise refinement techniques. Because of the issue of verifying the scope of applicability for **WHENEVER** processing, it sometimes forces the structuring of the design into multiple KB segments each with their own control and opportunistic sections. While this suggests the use of sub-KBs or similar restrictive scoping mechanisms, this is not required by the design as long as the semantics are the same. Hence, we expect that many of the KB Segments will be implemented as guarded sets of rules rather than sub-KBs. The verification rules for the design language are usable, allowing verification to occur quickly with minimal consideration of complex situations. The only problems occur with the use of **WHENEVERs**. The language allows **WHENEVERs** to be used in arbitrarily complex sequences. While this effectively allows the use of KBS programming techniques, it can be difficult to verify in complex cases. The need for verification of the design often encourages simplification of the design in these cases. Most importantly, the use of the design language allows us to verify the correctness of the designs and utilize Cleanroom Software Engineering effectively in the development of APR.

Summary and Conclusions

A design language for KBS has been described along with a brief description of the verification approach that is to be used with the language. The language is an extension of existing procedural design languages with structures for specifying control and opportunistic components of KBS designs. The language supports the development of KBS software using top down development and Cleanroom Software Engineering techniques in a practical manner.

The design language is being used in the development of the APR aircraft flight replanner prototype. Based on our experience to date, the language seems to provide sufficient representational power to specify the types of processing expected in a KBS while providing a practical mechanism for verifying the correctness of those designs.

While the language provides a good starting point for the use of design language and verification techniques with KBS, there are a number of areas still to be investigated. The language has only been used on a single project to date. While this project is relatively large (1500+ rules) and utilizes a number of different problem solving techniques, there is potential benefit from using this language in the development of other projects with different characteristics. It has also been suggested that this language would be useful for mixed KBS and object oriented paradigms, but this has not been investigated. Concepts such as formal descriptions of data and their semantics, such as that provided in TOP, are not currently part of the language and extension of the language to use data descriptions should be possible and beneficial. Finally, the use of the language to represent problems solved using backward chaining reasoning needs to be explored.

KDL Solution to the Traffic Controller Problem

A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every 2 minutes. With a car or pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Observations

The problem is inherently a realtime asynchronous processing problem. Such problems are not easily solved or understood. In that the intent is to provide a simple example, the problem will be formulated as a synchronous problem.

Assumptions

The following assumptions represent an interpretation of the requirements in areas that were potentially ambiguous:

1. Traffic flow in the direction of the signal has no impact on the changing of the signal when no traffic is waiting in the opposite direction. The wording of the requirements seems to indicate that the 15 second time extension applies only when traffic is waiting (It is possible to apply this 15 second extension to the 2 minute default when no traffic is waiting. Some traffic controllers do work this way as it minimizes impacts on traffic flow that are not necessary.)

2. The solution must allow for momentary action pedestrian crossing signals. While an auto sensor will generally be on once an auto is waiting to cross the signal, pedestrian crossing signals tend to be push-buttons that are only on momentarily. The solution will assume that once such a button is pushed. The pedestrian remains in the "waiting to cross" state until the signal changes. If this assumption were changed to use sample/hold circuitry in the sensors, the use of the traffic_waiting variable would not be required.

3. The pedestrian and auto waiting signals are "ored" together for a given direction of travel. This simplifies the processing of sensors as only one needs to be read for a direction.

4. The delay of traffic flow switch is interpreted to mean that a delay of 15 seconds from the time of detection is to be applied. Other interpretations, such as adding an additional 15 seconds to the current delay, are also possible. However, most traffic controllers seem to work in the manner assumed here.

Solution Approach

The solution utilizes a polling approach that polls the sensors and performs switching on a 1 second cycle. (Note that this is a simplification of the more general event driven approach with asynchronous timers that would probably be used to implement real traffic light controllers.)

On each cycle, the system will increment the internal timers, read the sensors and update the traffic light if necessary. This forms the basis for the control logic of the system that is represented in the WHEN statement.

Two timers are maintained. The "time" timer represents current time and is used in conjunction with the switch_time variable to determine when it is necessary to switch the traffic flow. The wait_time represents the number of seconds traffic or pedestrians have been waiting to pass. Only two timers are needed for this problem because there are only two directions of travel and the uses of the timer are mutually exclusive. If the problem were more complex, e.g. a three way intersection, more timers would be required.

The usage of the timers is as follows:

1. The time is incremented on every cycle of the system.

2. The wait_time timer is incremented whenever there is someone or something waiting.

3. Whenever a vehicle or pedestrian is first detected in the stopped direction, the switch_time is set to time + 15 seconds.

4. Whenever a vehicle or pedestrian is detected in the flowing direction and a vehicle or pedestrian is waiting in the stopped direction the switch_time is (re)set to time + 15 seconds.

5. Whenever the time = switch_time, the traffic lights are switched, the switch_time is set to time + 2 minutes and the wait_time set to 0.

6. Whenever the wait_time timer reaches 1 minute, the traffic lights are switched, the switch_time is set to time + 2 minutes and the wait_time set to 0.

Notational Conventions

1. We have adopted the notational convention that if there is only one When and the Segment intended function is the same as the When intended function then the intended function of the When can be omitted.

2. We have adopted the notational convention that TRUE -> I (the identity function in conditionals) is assumed if no alternative is given.

3. We have adopted the notational convention that frame instances or classes can be referred to in the design using their type/class name. This is used in the Crossing_traffic whenever.

Proof

1. <u>When Intended Function implements Segment Intended Function:</u>

   Since they are the same, this is obvious.

2. <u>When Statement implements When Intended Function:</u>

The When statement condition is always true. The When statement action consists of initializing variables to indicate that the light has just switched traffic flow to initial_flow_direction and and changing traffic flow for every second in time per the When Intended function. Hence, the two are equivalent.

3. <u>When Statement Initialize implements it's Intended Function:</u>

Using the correctness conditions for KDL, the statement verifies if its implementation and all applicable Whenever statements implement the intended function. In this case, the implementation implements the intended function, and it can be seen from inspection that no Whenever's are applicable since they all utilize a state variable that does not currently have a value.

4. <u>When For Statement implements it's Intended Function:</u>

By the correctness conditions for **For** statement verification, the statement verifies if the composition of its body intended function for each iteration implements the **For** statement intended function.

While the **For** appears to be infinite, making verification impossible, it is actually not. Since wait time is incremented if traffic is waiting, the wait time condition will eventually be reached. If traffic is not waiting, the third intended function will do nothing until the switch time is reached (which will eventually happen since time is incremented by the **For** loop). It is therefore sufficient to verify that the composition of the **For** body for all sequences up until the switch/wait time condition is met is correct in order to verify correctness of the **For**.

The verification of the **For** loop requires that the alternatives of the **For's** intended function be implemented. These are:

1. If no traffic is waiting to cross, change traffic flow in 120 seconds.
2. If traffic is waiting to cross and there is no traffic in the current direction of flow, change traffic flow in 15 seconds.
3. If traffic is waiting to cross and there is traffic in the current direction of flow, change traffic flow in 15 seconds, but not more than 60 seconds total wait.

__Verification of Condition 1__: If no traffic is waiting to cross, time will be incremented by the for loop until the switch time is reached. When the switch time is reached, traffic flow will be switched and the switch time reset. As time is set to 120 initially and is set to time+120 on each switching, traffic will be switched every 120 seconds if no traffic is waiting.

__Verification of Condition 2:__ If traffic is waiting and no traffic is detected in the direction of flow, the third intended function will set traffic switch time to time+15 seconds, and indicate that traffic is waiting. The traffic_waiting indicator will prevent the time from being reset if no other events occur. As time is incremented on each cycle, traffic will be switched in 15 seconds if no other events occur.

__Verification of Condition 3__: If traffic is already waiting and traffic is detected in the direction of flow, the second intended function will reset traffic switch time for time+15 seconds. If traffic is currently (sensor input) waiting, the switch time is reset to 15 seconds regardless of whether there is traffic in the current flow direction or not. In addition, the first intended function will increment wait time whenever traffic is already waiting. The "switch time" intended function will switch traffic flow whenever the switch time reaches 0 or the wait time reaches 60. Therefore, the condition is implemented by the composition of the intended functions.

5. __Sensor Input Intended Function implementation:__

By the correctness conditions for **DO INTERRUPTIBLE** intended functions, the function is correct if its immediate actions and applicable whenevers implement the intended function in a deterministic way.

The immediate actions consist only of read operation which is assumed to be correct. By inspection it can be seen that no whenevers are applicable as the value of state is not set.

6. __UPDATE WAIT TIME Intended Function implementation:__

The immediate actions consist only of an assignment to the state variable. The only whenever applicable as a result of this state variable assignment is Update_Wait_Time whose intended function is identical to the intended function of the statement here with the addition of the check for wait time update required.

While this is a trivial example, it indicates the use of state variables to isolate the function of whenevers and the use of whenevers to implement conditional logic.

7. Switch time/Wait time Intended Function:

The immediate action contains only an assignment to the state variable. By inspection of the whenevers, it can be seen that only the Switch_traffic and Crossing_traffic whenevers are applicable. From their intended functions, it can be seen that they each implement one alternative of the original intended function. Since they both indicate that traffic flow change is not required as part of their actions, they will be mutually exclusive.

8. Update Wait Time Whenever:

The condition and action of the whenever match the intended function of the whenever. By inspection, it can be seen that no other whenevers are effected.

9. Switch traffic Whenever:

The condition and action of the whenever match the intended function of the whenever. By inspection, it can be seen that no other whenevers are effected since they action of this whenever changes the state such that other whenevers are not applicable.

10. Crossing traffic Whenever:

The condition and action of the whenever match the intended function of the whenever. By inspection, it can be seen that no other whenevers are effected since the action of this whenever changes the state such that other whenevers are not applicable.

KDL Solution for the Traffic Controller Problem

```
KB SEGMENT traffic_light_controller (IN: sensor_stream, initial_flow_direction)
[ Given a traffic light just switched to initial_flow_direction,
  For every second in time:
   No traffic waiting to cross -->
     change traffic flow 120 seconds after last change
    / no traffic in current direction of flow -->
        change traffic flow 15 seconds after
          detecting traffic waiting to cross
        / change traffic flow 15 seconds after
          detecting traffic in current direction of flow
          but not more than 60 seconds after
          detecting traffic waiting to cross ]

LOCAL DATA
  Parameter Switch_time:
    Type: Integer
  end

  Parameter Flow_direction:
    Type:
(EASTWEST,NORTHSOUTH)
  end

  Parameter Time:
    Type: Integer
  end

  Frame Type Flow_sensor:
    Direction: Type:
(EASTWEST,NORTHSOUTH);
    Traffic_detected: Boolean;
  end

  Frame Northsouth_lane:
    Direction: NORTHSOUTH
  end

  Parameter Wait_time:
    Type: Integer
  end

  Parameter Traffic_waiting:
    Type: Boolean
  end

  Parameter State:
    Type: (UPDATE_WAIT_TIME,
SWITCH_TRAFFIC,NULL)
  end

  Frame Eastwest_lane:
    Direction: EASTWEST
  end
```

```
WHEN
  true
DO INTERRUPTIBLE
  [ Flow_direction,Switch_time,Wait_time,Traffic_waiting :=
    initial_flow_direction,120,0,FALSE ]

  Flow_direction := initial_flow_direction
  Switch_time := 120
  Wait_time := 0
  Traffic_waiting := FALSE
  State := NULL

  [ For every second in time:
     No traffic waiting to cross -->
       change traffic flow 120 seconds after last change
       | no traffic in current direction of flow -->
           change traffic flow 15 seconds after
             detecting traffic waiting to cross
           | change traffic flow 15 seconds after
             detecting traffic in current direction of flow
             but not more than 60 seconds after
             detecting traffic waiting to cross ]
  FOR time := 0 to forever
    [ Read traffic direction sensors ]
    Read(Sensor_stream,
        Eastwest_lane.traffic_detected,
        Northsouth_lane.traffic_detected)

    [ Traffic_waiting --> Wait_time := Wait_time + 1 ]
    state := UPDATE_WAIT_TIME

    [ time = switch_time | wait_time = 60 -->
      change traffic flow;
      switch_time,wait_time,traffic_waiting := time+120,0,FALSE
      | ((sensors detect traffic waiting & not traffic_waiting) |
        (traffic_waiting &
         sensors detect traffic in current direction of flow)) -->
          switch_time,traffic_waiting = time+15,TRUE ]
    state := SWITCH_TRAFFIC

  END WHILE

END
```

```
[ Wait time update required & Traffic_waiting -->
  Wait_time := Wait_time + 1 ]
Update_Wait_Time: WHENEVER

  state = UPDATE_WAIT_TIME and
  traffic_waiting

DO

  wait_time := wait_time + 1

END


[ traffic flow change required &
  (time = switch_time / wait_time = 60) -->
  change traffic flow;
  switch_time,wait_time,traffic_waiting := time+120,0,FALSE;
  indicate that traffic flow change is not required]
Switch_traffic: WHENEVER

  state = SWITCH_TRAFFIC and
  (time = switch_time or wait_time = 60)

DO

  [ Switch_time,Wait_time,Flow_direction,Traffic_waiting :=
      time+120,0,not Flow_direction,FALSE ]
  Switch_time := time+120
  Wait_time := 0
  Flow_direction := not Flow_direction
  Traffic_waiting := FALSE
  state := NULL

END
```

```
[ traffic flow change required &
  not (time = switch_time | wait_time = 60) &
  ((sensors detect traffic waiting & not traffic_waiting) |
  (traffic_waiting &
     sensors detect traffic in current direction of flow))-->
  switch_time,traffic_waiting = time+15,TRUE;
  indicate that traffic flow change is not required]
Crossing_traffic:  WHENEVER

  state = SWITCH_TRAFFIC and
  not (time = switch_time or wait_time = 60) and
  ((flow_sensor.traffic_detected = TRUE and
    flow_sensor.direction <> Flow_direction and
     traffic_waiting = FALSE) or
   (traffic_waiting = TRUE and
    flow_sensor.traffic_detected = TRUE and
    flow_sensor.direction = Flow_direction))

DO

  [ Traffic_waiting,Switch_time := TRUE,time+15 ]
  Traffic_waiting := TRUE
  Switch_time := time+15
  state := NULL

END
```

Launch Sequencing

Purpose and Background

The purpose of this system is to perform on-board functions required to prepare a space vehicle for liftoff, monitor for errors, and respond to errors. The space vehicle is a new type that has never been flown before. Because the pre-launch activities and checks must be performed so quickly just prior to launch, a human can not perform these functions; this means that there is no existing human expert that does this job.

Functions

The functions to be performed are:

1. Perform nominal launch sequence functions (NLSFs). Each NLSF has a command which will perform the function and a set of constraints about when must be met before the command can be issued. Each NLSF also has other constraints on when it can/should be performed depending on its relationship to other NLSFs. Finally, each NLSF is judged to have been successful depending on the truth of exit conditions: The NLSFs are documented in Table 1.
2. Monitor error conditions. Error conditions are context sensitive in that they are monitored under certain conditions. The monitoring conditions and when they should be monitored are documented in Table 2.
3. Respond to errors. An error condition occurs when a check (i.e., monitor) fails, a function fails to complete, or it has been determined that functions can not be issued at the right times to achieve main engine ignition at MET=0.0 seconds. The specific error recovery actions are documented in Table 3.

Table 1: Nominal Launch Sequence Functions

FUNCTION	CONSTRAINTS	EXIT CRITERIA
Main Engine Ignition	This is the main goal event and must occur at MET=0.0. It must also occur between 2 and 2.3 seconds after secondary engine thrust has built up.	command issued
Secondary Engines Ignition	Must occur within 2 seconds of propellant bleed valve closure.	engine thrust > 90% (usually takes about .5 sec)
Terminate direct ground link	main engine ignition	ground link termination confirmed
...	...	....

Table 2: Monitoring Conditions

MONITOR	CONDITION	CONTEXT

Engine Communication Failure	Engine Command Word Bit 1 not reset upon receipt	checked each .1 sec
Engine Failure	Thrust lower than expected (<90% 2 seconds after start)	checked each .5 sec after engine ignition
PIC ignition voltage	Must achieve count of 100 within 4 sec of arming and not drop below 90	checked each second after arming until ignition
...	...	...

Table 3: Error Recovery Actions

ERROR	CONDITION	RECOVERY ACTION
Engine Communication Failure	Bit 1 not reset on two consecutive commands.	If no engines are running, issue launch hold. If main engines not started, shutdown engines and issue launch hold. If main engines started, shutdown failed engine only if doing so will still maintain overall thrust within safety limits
...	...	...

Correctness Considerations

It is extremely critical that monitoring and error recovery be functionally correct and the correct recovery action is performed within .1 seconds after the error condition occurs.

It is critical that NLSFs be sequenced correctly.

Although the launch processing system has no direct user interaction, there is a need to display status of launch sequencing.

Hints

Note that the tables do not specify all the details and only include samples. Only develop a test approach, not a complete set of test cases. Note in your planning that safety is an important consideration (might influence cost). Also think about what things could go wrong and what the consequences might be.

File Management Interface

Background/Purpose

There is a simple file management system that accepts a command in a specific format and performs the indicated operation. For example, the user can type "COPY file1 file2" to copy file1 into file2. The purpose of this new program is to provide a natural language interface to the file management system (i.e., on top of the existing command line interface). The new program will accept a free form natural language command like "put file1 at the end of file2" and will figure out the correct file management command to issue like "copy file1 file2 /APPEND".

Functions

The commands accepted by the file management system are

COPY file1 file2 /APPEND /REPLACE /NOPROMPT

(noprompt option is used with the replace and move options; the user is not prompted if file2 already exists)

RENAME file1 file2 /NOPROMPT

(the noprompt option does not prompt the user if file2 already exists)

DELETE file1 /NOPROMPT

(the noprompt option does not prompt the user if file1 does not exist)

USE file1 file2 ... IN filen

(this command inputs files appearing before the word IN to the program specified in filen)

LIST pattern

(this command searches for files matching the pattern and lists them; the pattern allows an asterisk to appear as a wildcard for one or more characters)

The allowed natural language inputs should include the use of alternave verbs such as move, replace, put, erase, discard, throw away, execute, invoke, etc. The input sentences should also be allowed to occur in any natural order such as "replace file2 with file1".

Hints

This about safety, robustness, and how much the system should "guess" about what the user wants to do. Think about the possible test cases that might be needed for complete coverage and alternatives to complete coverage. Also, you can assume the existence of a dictionary online in machine readable (and searchable) format.

Car Won't Start Diagnosis

This is a standardly used simple car diagnosis problem. It requires little outside knowledge of how cars work. The purpose of this program is to query the user for information about symptoms and then determine the best guess of why the car will not start.

Functions

Objects

The relevant parts of the car are:

BATTERY
STARTER MOTOR
STARTER SOLENOID
SPARK PLUGS
DISTRIBUTOR
CARBURETOR
GAS TANK
FUEL PUMP

The function of this program is to determine which of the above objects is the most likely reason for the engine not starting. The following diagnosis information was obtained from an expert mechanic.

The easiest things to check are the gas tank and the battery. If the gas gauge reads empty and the engine turns over then the most likely cause is the gas tank is empty. If the headlights don't shine brightly and the engine does not turn over then the most likely cause is the battery.

If both the gas tank and battery are fine and the engine does not turn over then the most likely cause is either the starter or the starter solenoid. If you can hear a "clicking sound" when you try to start it, then it is probably the starter, else it is probably the solenoid.

If the engine does not turn over then the likely place is somewhere in the ignition system (spart plugs or distributor). Checking these is a little tricky for the novice but can be done. The first thing to do is to check spark getting to each plug. This can be done by removing the wire to each plug and holding it close to the plug (so the metal piece inside the wire is very close to the plug). If you can see a spark when trying to start the engine then the distributor is ok and the plugs are the likely problem, else the distributor is the likely problem. When performing this procedure, there is the possibility of a surprising, but not harmful, shock which can be avoided by wearing heavy rubber gloves (this should not be attempted by anyone with a heart condition).

Finally, if the engine turns over and runs for a little while (even if it is less than a second) then the likely cause is in the fuel system, either the carburetor or the fuel pump. The fuel pump can be checked by removing the line from the fuel pump to the carburetor and then very briefly trying to start the engine. If gas strongly squirts out from the line then the fuel pump is fine and the likely cause is in the carburetor. Note that this last procedure is very dangerous and should only be attempted by an experienced user (e.g., a mechanic) and only when the engine is cold.

Hints

Try organizing the diagnosis information in such a way that you can identify what the test cases should cover. Also think about what are critcal, not critical, safety, mandatory, and not mandatory requirements.

Wakup Call Processing

Purpose and Background

A group of hotels got together and decided to procure an automated wakeup call system for use by all of them. In the requirements discussion meetings, there was a lot of debate over the "peak load time" issue. At peak load time (around 7AM), there are many more wakup calls than the system can handle at once. so the calls must be prioritized. After much debate and consulting experienced operators, a prioritization scheme was agreed on. This system should implement the prioritization scheme.

Functions

A. Prioritization

Wakup call requests will be distinguished based on the cost of the room. All high class rooms (the most expensive) get first priority, then medium class ones and finally low class ones. Calls are further prioritized according to which calls were requested first and according to how late a wakeup call request is becoming. The lateness is given six times the weight as the earliness of the request. For example, if wakup call A was requested one hour before wakup call B but wakup call B is currently eleven minutes late then wakup call B has a higher priority (60 for A vs. 66 for B).

A call can be given a higher priority in two ways.

1. it is more than twenty minutes late
2. it is given "special priority" (this is not determined by this system but is predetermined)

If either priority-raising condition holds, then call will be given higher priority within a room class. If both conditions hold then the call with be given higher priority over all room classes.

B. Early Calling

During times of the day that are known to be peak load times, calls can be given in advance of the requested time (to try to avoid getting behind). For determining calls to be given early, the prioritization above works exactly in reverse with the following exceptions.

1. high class can be called up to 5 minutes early
2. medium class can be called up to 10 minutes early
3. low class can be called up to 20 minutes early
4. if the special priority flag is set then the call can be made up to 20 minutes early

There are two additional considerations. The first is that a late call always has priority over an early call. The second is that if two or more calls have the same priority then the choice is arbitrary.

Hints

Consider which parts of this system fit the expert system characteristics and which parts are more conventional. What implementation/design approach do you think would fit this problem best and how would this influence you test approach ? Are there any critical aspects that deserve more attention than others ? Could the system monitor itself to see if it were operating correctly ?

Description of Monkeys and Bananas Problem

This version of the problem description is due to Peter Ludemann (IBM).

Monkeys and Bananas

From the original NASA description. The presentation has been changed slightly.

Characteristics of objects and actions

The monkey has the following characteristics:

1. It has a location.
2. It is located on top of something (the floor or another object).
3. It may be holding an object.

An object has the following characteristics:

1. It has a location.
2. It is located on top of something (the floor or another object), or it is attached to the ceiling.
3. It has a weight (either light or heavy).

In addition, an object has the following characteristics if it is a chest:

1. It contains another object.[11]
2. It is unlocked by another object (a key).

The monkey may eat an object under the following conditions:

1. There exists a goal to eat the object.
2. The monkey is holding the object.

The monkey may hold an object under the following conditions:

1. There exists a goal to hold the object.
2. The monkey is at the same location as the object.
3. The object is attached to the ceiling and the monkey is on top of the ladder,[12] or both the monkey and the object are on top of the same place (either the floor or another object).
4. The monkey is holding nothing.
5. The weight of the object is light.

The monkey may move to a location under the following conditions:

1. There exists a goal to move to the location.
2. The monkey is on the floor.

[11] Editor's note: Presumably this should be "it may contain another object."

[12] Editor's note: and the ladder as at the same location.

The monkey may climb onto an object under the following conditions:

1. There exists a goal to climb onto the object.
2. The monkey is holding nothing.[13]
3. The monkey is at the same location as the object.
4. Both the monkey and the object are on top of the same place.

Initial Conditions

The goal is to eat the bananas.[14]

The initial conditions are:[15]

Table 1. Initial conditions. Empty entries indicate that the attribute does not apply to the object.

object	location	on top of	holding	weight	contains	unlocked by
monkey	t5-7	green couch	nothing			
green couch	t5-7	floor		heavy		
red couch	t2-2	floor		heavy		
big pillow	t2-2	red couch		light		
red chest	t2-2	big pillow		light	ladder	red key
blue couch	t8-8	floor		heavy		
blue chest	t7-7	ceiling		light	bananas	blue key
green chest	t8-8	ceiling		light	blue key	red key
red key	t1-3	floor		light		

Actions

The monkey may jump onto the floor under the following conditions:

1. There exists a goal to jump onto the floor.
2. The monkey is not on the floor (see jumping up and down).

The monkey may drop an object under the following conditions:

1. There exists a goal to drop the object.
2. The monkey is holding the object.

[13] Editor's note: From looking at the program, it appears that this restriction is not enforced — in fact, it is clear tha this restriction is incorrect because it would prevent the monkey from climbing the ladder with the key (to unlock chest containing the bananas).

[14] Editor's note: The goal is for the monkey to eat the bananas.

[15] The original description missed out the following:

- The red key is on top of the floor.

Note: the object may be dropped either onto the floor or the place the monkey is on.

The monkey may unlock a chest under the following conditions:

1. There exists a goal to unlock the chest.
2. The chest can be unlocked by another object (the key).
3. The monkey is holding the key.
4. The monkey is at the same location as the chest.
5. Both the monkey and the chest are on top of the same place.

Note: when a chest is unlocked, the object it contains is placed on top of the chest.

Commentary

Although the primary goal of this problem is for the monkey to eat the bananas, each of the goals must also be attainable separate from the primary goal. That is, it should be possible to change the goal from eating the bananas to walking to a certain location or unlocking a certain chest. The solution need not support multiple initial goals.

The word "goal" is used throughout this problem statement suggesting that this problem should be solved using goals. **Any appropriate methodologies may be used to solve the problem.**[16]
The problem, however, should be solved in a way that a knowledgeable user might be expected to solve the problem. Knowledge representation should not be sacrificed for speed when solving the problem.

The benchmark should be able to run under two modes. One mode should run the benchmark printing all the actions undertaken by the monkey, while the other mode should only print a message when the monkey has eaten the bananas. Two separate versions of the benchmark or a toggle switch in a single version of the benchmark are suitable to provide this capability.

[16] Editor's emphasis.

Handout #1: State Diagram for the Simple Traffic Controller

A state diagram is helpful in analyzing a procedural solution to the simple traffic light problem. This also reduces the implementation approach (see handout #2) to a simple process of checking the most recent state and the current state to determine a new state. The procedure in handout #2 implements the state diagram of figure 1. The specific states in that diagram are defined as follows:

S_1:	2M_Timer := Clock+2 Minutes NS-Light := Green
S_2:	2M_Timer Is Unchanged NS-Light := Green Clock Updated by 1 second
S_3:	1M_Timer := Clock+1 Minute 15S_Timer := Clock+15 Seconds NS-Light := Green
S_4:	NS-Light := Green 1M_Timer Unchanged 15S_Timer Unchanged Clock Updated by 1 second
S_5:	NS-Light := Green 1M_Timer Unchanged 15S_Timer := Clock+15 Seconds
S_6:	NS-Light := Red 2M_Timer := Clock+2 Minutes

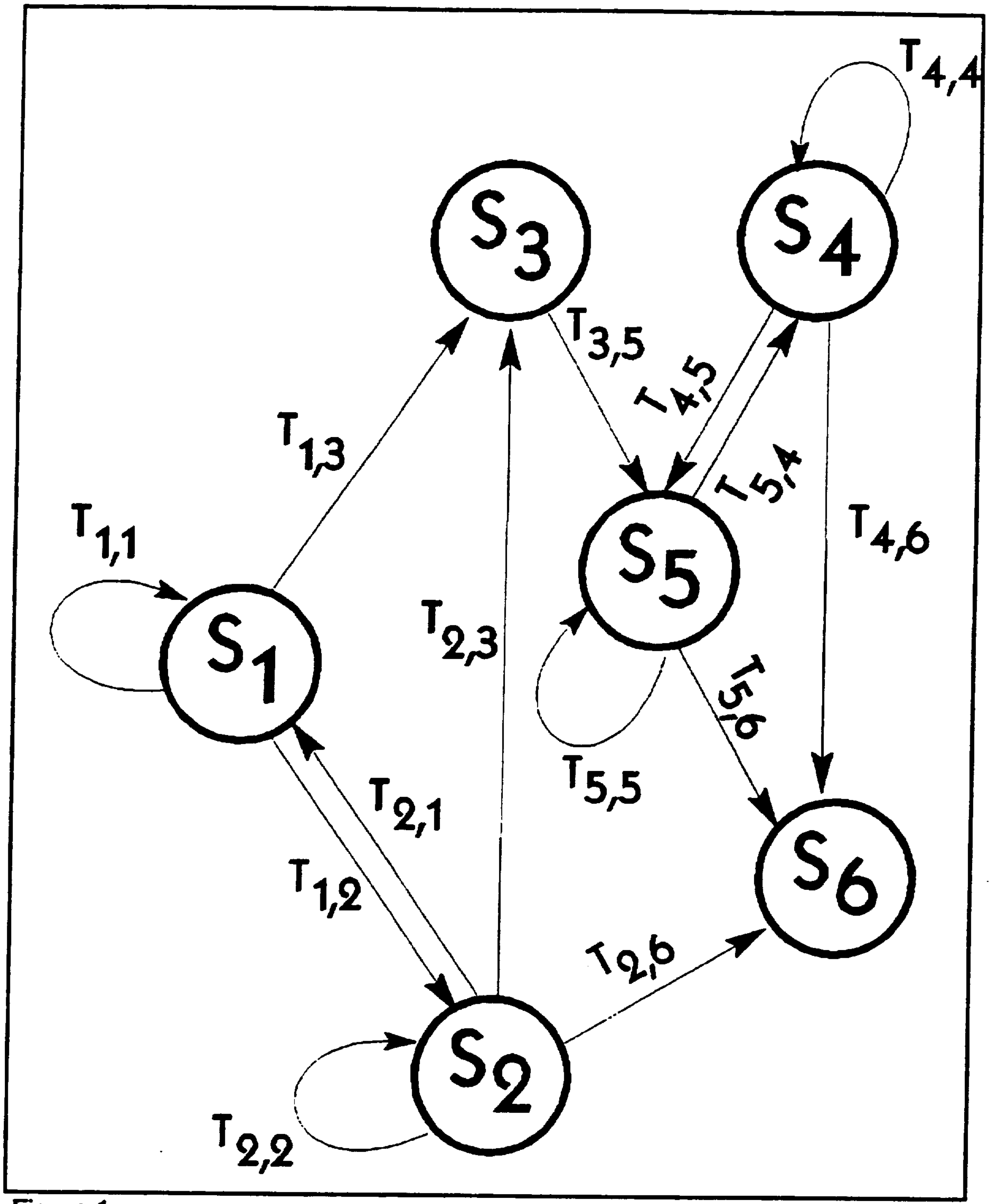


Figure 1

The matrix of figure 2 presents an alternate view of the state diagram in figure 1. Even though this state diagram eases the implementation of a procedural (or rule-base) solution, it can be complicated to use in defining test cases because of its level of detail. To address this complexity, define an abstract view of the state diagram in figure 1. Figure 3 shows an abstract state diagram that relates to the diagram of figure 1 (see figure 4 for the associated transition matrix). The abstraction considers the "essence" of what is taking place in the system. That essence is the process of deciding which the period of time to expire before changing the light (the * in the transition matrix of figure 4 indicates that the light will change as a "side-effect" of the state transition). It is important, however, to maintain a clear mapping between the abstraction and its refinement. In this example, the mapping is clear by examining the matrix of figure 2. Notice how the state transitions tend to fall into two distinct groups (the upper left and lower right corners of the matrix). These relate directly to the two abstract states shown in figure 3. Testing the system, then, is reduced to testing either view of the system. The premise for this approach is based on the correctness of the state diagram itself. If the state diagram is correct and the transitions between states have been implemented correctly then it would be reasonable to predict that the implementation is correct. Testing is now much simpler (especially when using the abstract state diagram) because the number of scenarios has been reduced. For example, using the state diagram of figure 3 there are only 4 transitions to consider (as opposed to a potentially infinite number of scenarios). Testing, then, will test the transitions. This is sometimes referred to as "conformance" testing (i.e., showing that the implementation conforms to the abstract view).

$T_{(i,j)}$	S_1	S_2	S_3	S_4	S_5	S_6
S_1	NS Approaching and no waiting traffic	No approaching and no waiting	EW Waiting			
S_2	NS approaching and no waiting and not 2M Timer expired	No approaching and no waiting and not 2M Timer expired	Waiting signal and not 2M Timer expired			2M Timer expired
S_3				No approaching	Approaching	
S_4				Not 15S Timer Expired And Not 1M Timer Expired And No Approaching	Approaching And Not 15S Timer Expired And Not 1M Timer Expired	(1M Timer Expired) Or (Not 1M Timer Expired And 15S Timer Expired)
S_5				No Approaching and Not 1M Timer Expired	Approaching And 1M Timer Expired	1M Timer Expired
S_6						

Figure 2

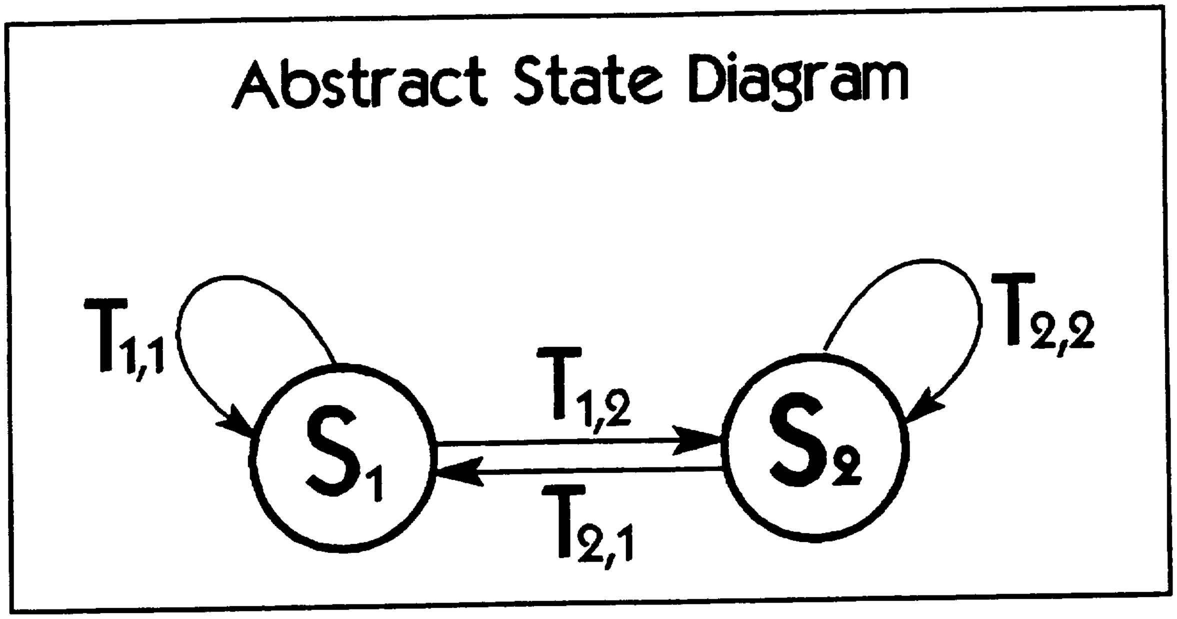


Figure 3

$T_{(i,j)}$	S_1	S_2
S_1	(No waiting) Or (2 Minutes expired and No approaching*)	Waiting
S_2	(1 Minute expired since switch from S_1*) Or (15 Seconds expired since last approaching*)	Approaching

Figure 4

Handout #2: Procedural Implementation

```
Procedure Traffic_Controller Is

--<*
-- The Traffic controller uses the notion of a Timer to determine
-- when to change the flow of traffic.  Each timer represents
-- a window in time beginning at the current clock time plus some
-- some delta.
--*>
        2M_Timer, 1M_Timer, 15S_Timer : Timer;

--<*
--  Returns TRUE when traffic is approaching in the current direction
--          of traffic flow at the current clock time
--  ELSE -> FALSE
--*>
        Function  Approaching_Traffic  Return (True, False);

--<*
--  Returns TRUE when traffic (auto or pedestrian) requests a
--          change in the light at the current clock time
--  ELSE -> FALSE
--*>
        Function  Wait_Signal_Received Return (True, False);

--<*
--  Returns the current time
--*>
        Function  Clock Return Time;

--<*
--  Returns TRUE when the current clock time exceeds the time
--          specified by the Timer
--  ELSE -> FALSE
--*>
        Function  Expired(T: In Timer) Return (True, False);

--<*
--  Switch from the current direction of traffic flow to the opposite
--*>
        Procedure Switch(L: In Out Light);
```

State : Current State of the Traffic Controller

Possible states the Traffic Controller can be in are:

S_1:	2M_Timer := Clock+2 Minutes NS-Light := Green
S_2:	2M_Timer Is Unchanged NS-Light := Green Clock Updated by 1 second
S_3:	1M_Timer := Clock+1 Minute 15S_Timer := Clock+15 Seconds NS-Light := Green
S_4:	NS-Light := Green 1M_Timer Unchanged 15S_Timer Unchanged Clock Updated by 1 second
S_5:	NS-Light := Green 1M_Timer Unchanged 15S_Timer := Clock+15 Seconds
S_6:	NS-Light := Red 2M_Timer := Clock+2 Minutes

```
State := S_1;
Loop
        Case State Is
                        When in S_1 => perform S_1 transitions
                        When in S_2 => perform  S_2 transitions
                        •  •  •
                        When in S_n => perform S_n transitions
        End Case;
        Update Clock;
End Loop;
End Traffic_Controller;
```

```
--<*
--  S1 Transitions
--
--  Decision Table:
--
```

Waiting	Approaching	Satisfied By:
T	T	1.3
T	F	1.3
F	T	1.1
F	F	1.2

```
--
--*>
<* 1.1 *> When S1 And (Approaching_Traffic And
                       NOT Waiting_Traffic)        =>
                       State := S1;
<* 1.2 *> When S1 And (NOT Approaching_Traffic And
                       NOT Waiting_Traffic And     =>
                       State := S2;
<* 1.3 *> When S1 And (Waiting_Traffic)            =>
                       State := S3;
--<*
--  End S1 Transitions
--*>
--<*
--  S2 Transitions
--
--  Assumptions : Once waiting traffic is detected detection of
--              oncoming traffic is irrelevant
--
--  Decision Table:
--
--          Waiting   | Approaching |  Expired  | Satisfied By
--        ------------+-------------+-----------+-------------
--             T      |      T      |     T     |     2.3
--             T      |      F      |     T     |     2.3
--             F      |      T      |     T     |     2.3
--             F      |      F      |     T     |     2.3
--             T      |      T      |     F     |     2.4
--             T      |      F      |     F     |     2.4
--             F      |      T      |     F     |     2.2
--             F      |      F      |     F     |     2.1
--*>
<* 2.1 *> When S2 And (NOT Approaching_Traffic And
                  NOT Waiting_Traffic And
                  NOT Expired(2M_Timer))  =>
                  State := S2;
<* 2.2 *> When S2 And (NOT Waiting_Traffic And
                  NOT Expired(2M_Timer) And
                  Approaching_Traffic))      =>
                  State := S2;
<* 2.3 *> When S2 And (Expired(2M_Timer))    =>
                  State := S6;
<* 2.4 *> When S2 And (Waiting_Traffic And
                  NOT Expired(2M_Timer))      =>
                  State := S3;
--<*
--  End S2 Transitions
--*>
```

```
--<*
--  S3 Transitions
--
--  Assumptions : Detecting additional waiting traffic does
--              not effect state transition
--
--  Decision Table:
--
--                              Approaching          Satisfied By
--                                   T                   3.1
--                                   F                   3.2
--*>
<* 3.1 *> When S3 And (Approaching_Traffic)  =>
                   State := S5;
<* 3.2 *> When S3 And (NOT Approaching_Traffic)    =>
                   State := S4;
--<*
--  End S3 Transitions
--*>
```

```
--<*
--  S4 Transitions
--
--  Assumptions : Once waiting traffic is detected detection of
--            oncoming traffic is irrelevant
--
--  Decision Table:
--
--        Approaching | 15S_Timer | 1M Timer Ex | Satisfied By
--                    |    Ex     |             |
--        ------------+-----------+-------------+-------------
--             T      |     T     |      T      |     4.4
--             T      |     F     |      T      |     4.4
--             F      |     T     |      T      |     4.4
--             F      |     F     |      T      |     4.4
--             T      |     T     |      F      |     4.2
--             T      |     F     |      F      |     4.3
--             F      |     T     |      F      |     4.2
--             F      |     F     |      F      |     4.1
--
--  What happens when the oncoming traffic is detected at the
--     exact same time that the timer expires?
--*>
<* 4.1 *> When S4 And (NOT Expired(15S_Timer) And
                  NOT Expired(1M_Timer) And
                  NOT Approaching_Traffic))        =>
                  State := S4;
<* 4.2 *> When S4 And (NOT Expired(1M_Timer) And
                  Expired(15S_Timer))        =>
                  State := S6;
<* 4.3 *> When S4 And (Approaching_Traffic And
                  NOT Expired(1M_Timer) And
                  NOT Expired(15S_Timer)) =>
                  State := S5;
<* 4.4 *> When S4 And (Expired(1M_Timer))   =>
                  State := S6;
--<*
--  End S4 Transitions
--*>
```

--<*
-- S_5 Transitions
--
-- Assumptions : Physically impossible for the 15S_Timer to
-- expire at the same time it is set
--
-- Decision Table:
--

Approaching	1M Timer Exp	Satisfied By
T	T	5.1
T	F	5.2
F	T	5.1
F	F	5.3

--*>
<* 5.1 *> When S_5 And (Expired(1M_Timer)) =>
State := S_6;
<* 5.2 *> When S_5 And (Approaching_Traffic And
NOT Expired(1M_Timer)) =>
State := S_5;
<* 5.3 *> When S_5 And (NOT Approaching_Traffic And
NOT Expired(1M_Timer)) =>
State := S_4;

--<*
-- End State_5 Transitions
--*>

Handout #3: First Rule Base Implementation

NOTE: To aid in understanding the syntax used for the rule-base that follows, consider the following. Each fact in the knowledge base is of the form **(x)** where **x** is a string of text. Variables are identified as names preceded by a "**?**". Variables are assigned during evaluation of a rule's LHS condition. This evaluation determines truth by pattern matching against facts. For example, pattern matching the expression, **(green ?direction)**, given the existance of the fact **(green NS)** would assign the value **NS** to the variable **?direction**.

```
Initial Facts is
  (green NS 0)
  (time 1)
  (signal NS car 370)
  (signal EW car 400)
  (signal NS car 420)
  (signal EW car 425)
  (signal EW car 450)
  (signal NS car 460)
  (signal NS car 470)
  (signal NS car 480)
  (signal NS car 490)
  (signal NS car 500)
  (end 600)
End Initial Facts;

Rule Update_Time With Priority -1 Is
      If  (time ?t)
     Then
         Retract (time ?t)
         Assert (time ?t + 1)
End Rule;
```

```
Rule Trigger_Signal_Change Is
      If (green ?direction ?) And
        (time ?t) And
        (signal ?other_direction ? ?t) And
        ?direction /= ?other_direction
      Then
          Assert (signal-change ?t))
End Rule;

Rule Del_Old_Changes Is
      If (signal-changes ?dt) And
         (time ?t) And
         (?t - ?dt) > 120
      Then
          Retract (signal-changes ?dt)
End Rule;

Rule Trigger_Signal_Delay Is
      If (green ?direction ?) And
        (time ?t) And
        (signal ?direction ? ?t)
     Then
         Assert (signal-delay ?t)
End Rule;

Rule Del_Old_Delays Is
      If (signal-delay ?dt) And
        (time ?t) And
        (?t - ?dt) > 15
      Then
          Retract (signal-delay ?dt)
End Rule;
```

```
Rule Change_No_Signal Is
       If (green ?direction ?last_changed)
         (time ?t) And
         ?t >= (?last_changed + 120 And
         not (signal-delay ?) And
         not (signal-change ?)
      Then
           Retract (green ?direction ?last_changed)
           If ?direction = NS
              Then ?other_direction = EW
              Else ?other_direction = NS
           End If;
           Assert (green ?other_direction ?t)
           Write "green " ?other_direction " (no signal) at " ?t crlf
End Rule;

Rule Change_No_Delay Is
       If (green ?direction ?last_changed) And
         (time ?t) And
         (signal-change ?sg) And
         not (signal-delay ?) And
        ?t >= ?sg + 15
       Then
           Retract (green ?direction ?last_changed)
           Retract (signal-change ?sg)
           If ?direction = NS
              Then ?other_direction = EW
              Else  ?other_direction = NS
           End If;
           Assert (green ?other_direction ?t)
           Write "green " ?other_direction " (no delay) at " ?t  crlf
End Rule;
Rule Change_Delay Is
       If (green ?direction ?last_changed) And
         (time ?t) And
         (signal-change ?sg) And
         (signal-delay ?sd) And
         ?t >= ?sg + 60
       Then
            Retract (green ?direction ?last_changed)
            Retract (signal-change ?sg)
            Retract (signal-delay ?sd)
            If ?direction = NS
               Then ?other_direction = EW
               Else ?other_direction = NS
            End If;
            Assert (green ?other_direction ?t))
            Write "green " ?other_direction " (delay) at " ?t crlf
End Rule;

Rule StopIt Is
       If (time ?t) And
         (end ?t2) And
         ?t >= ?t2
      Then
            Terminate ES execution
End Rule;
```

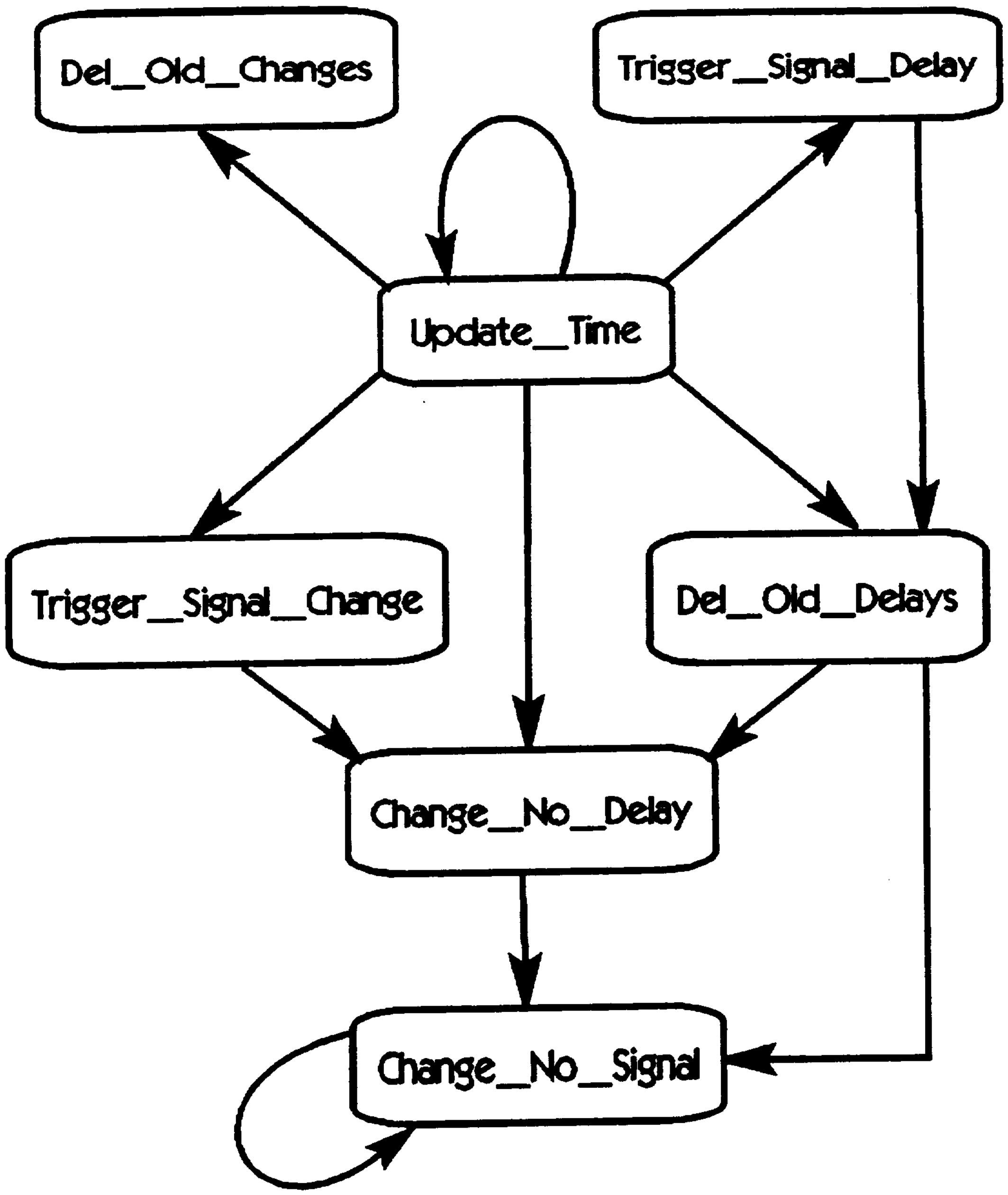


Figure 1: Diagram of Rule Relationships

Handout #4: Second Rule Base Implementation

/*

<u>Simulated Solution to Traffic Light Controller Problem</u>

<u>Problem Solving Method</u>

Time is simulated with a one second timer. This program cycles once each second. At the beginning of each cycle, certain definitions are set, then decisions are made about whether or not to change the traffic lights, and then at the end of each cycle, certain facts are reset (retracted).

```
Priorities: -2 : for updating the timer
            -1 : for things reset at the end of each cycle
             0 : figuring out if the lights need to be changed
```

<u>TIME</u> module

Update time count at end of each cycle

<u>State Data</u>

*/

```
Fact Time is (time (is ?t)) Where ?t must be a NUMBER;

Initial Facts Is
        (time (is 0))
        (stop-time 600)
End Initial Facts;
```

```
/*
        Transitions


        < update time at the end of each cycle >
        <* time := time + 1 *>
*/
        Rule Count_Time With Priority -2 Is
                If (time (is ?t))
                Then
                    Retract (time (is ?t)
                    Assert (time (is ?t+1))
        End Rule;


/*
        < halt when stop time reached >
*/
        Rule StopIt Is
                If (stop-time ?t) And
                  (time (is ?t))
                Then
                    Terminate ES execution
        End Rule;
```

```
/*
        TIMER module

        Allow timers to be asserted and figure out when they expire.

        Usage: Assert a time called some name and set for some time.
                When that time has elapsed, the timer will have the
                expires_at field set to true.

        State Data

        Model: Timer is a countdown timer that counts down with time

*/

        Fact timer Is (timer (called ?n)
                        (set_for ?t)
                        (has_expired ?f)
                        (expires_at ?e))
                          Where ?n must be a variable
                                ?t must be a NUMBER
                                ?f must be TRUE or FALSE with a default
                                  of FALSE
                                ?et must be a NUMBER with a default of
                                  99999
        End Fact;


        Rule Timer_Error Is
                If (timer (called ?name) (set_for ?sf)) And
                  ?sf <= 0
                Then
                   Write "TIMER_ERROR: " ?name crlf
        End Rule;
```

```
/*
        Constraint: only one timer of a given name.  This is resolved be
        deleting oldest timer.
*/
        Rule Timer_Name-Conflict Is
                If (timer (called ?name) (expires_at ?ea-1)) And
                  (timer (called ?name) (expires_at ?ea-2)) And
                  ?ea-1 < ?ea-2
                Then
                    Retract (timer (called ?name) (expires_at ?ea-1))
        End Rule;


/*
        Transitions

        Initial: expires_at := time + set_for
*/
        Rule Initialize_Expires_At Is
                If ( timer (expires_at 99999) (set_for ?sf) ) And
                  (time (is ?t))
                Then
                    Retract (timer (expires_at 99999))
                    Assert (timer (expires_at ?sf + ?t))
        End Rule;


/*
        < indicate timer has expired >
*/
        Rule Timer_Expired Is
                If ( timer (expires_at ?ea) (has_expired FALSE) ) And
                  (time (is ?t)) And
                  ?ea <= ?t
                Then
                    Retract (timer (has_expired TRUE))
                    Assert (timer (has_expired FALSE))
        End Rule;
```

```
/*
        Signal Controller Module

        Simulate car and pedestrian arrival sensors.


        State Data
*/

        Initial Facts Is
                (signal_data NS car 370)
                (signal_data EW car 400)
                (signal_data NS car 420)
                (signal_data EW car 425)
                (signal_data EW car 450)
                (signal_data NS car 460)
                (signal_data NS car 470)
                (signal_data NS car 480)
                (signal_data NS car 490)
                (signal_data NS car 500)
        End Initial Facts;
/*
        Model: Signal_data is a list of signal_names and times where the
                time indicates when the signal will be simulated
*/
        Fact signal Is (signal (in_direction ?d) (signalled_by ?sb))
                Where ?d is either NS or EW
                        ?sb is either car or pedestrian
        End Fact;
/*

        Constraint: none

        Initial: none

/*
```

```
/*

        Transitions


        < assert signal >
*/
        Rule Assert_Signal Is
                If (signal_data ?direction ?type ?time) And
                  (time (is ?time))
                Then
                    Assert (signal (in_direction ?direction)
                                    (signalled_by ?type))
        End Rule;


/*
        < retract signal at end of cycle >
*/
        Rule Retract_Signal With Priority -1 Is
                If ( signal (in_direction ?direction) (signalled_by ?type))
                Then
                    Retract (signal (in_direction ?direction)
                                      (signalled_by ?type))
        End Rule;
```

```
/*
        Traffic Light Module


        State Data

        Initial:
*/


        Global Variable ?green-light = NS;
        Global Variable ?red-light = EW;


/*
        Transitions
*/


        Procedure change-light () Is
                Assert (light-changed)
                If ?green-light = NS
                Then
                        ?green-light = EW
                        ?red-light = NS
                Else
                        ?green-light = NS
                        ?red-light = EW
                End If;
        End Procedure;


/*
    ;< reset light-changed fact at end of cycle >
*/
        Rule Retract-Light-Changed With Priority -1 Is
                If (light-changed)
                Then
                    Retract (light-changed)
        End Rule;
```

```
/*
```

Traffic Light Controller: Module

Problem Solving Method

OVERVIEW: Each cycle, figure out how long to wait to change lights, switching the light if it is time to do so.

A collection of timers are used to figure out when to change the lights. There is a long (2 min.) timer for "no signal" mode, a short timer (15 sec.) for "signal to change" mode, a medium timer (1 min.) for "signal to change but waiting on a car" mode.

The long timer is set when the light changes or there is a signal in the same direction.

The short and medium timers are set when there is a signal to change the light.

The short timer is reset each time approaching traffic is detected (and are waiting based on a signal to change the light).

The light is changed when any timer expires.

Constants

```
*/

Global Constant ?long-time = 120;
Global Constant ?medium-time = 60;
Global Constant ?short-time = 15;
```

```
/*
        Initial
*/

        Inital Facts Is
                (timer (called long) (set_for ?long-time))
        End Initial Facts;


/*
        Transitions


        < light-changed or approaching traffic -> set long timer >
*/

        Rule Set-Long-Timer Is
                If (light-changed) Or
                  ((signal (in_direction ?direction) And
                  ?direction = ?green-light)
                Then
                    Assert (timer (called long) (set_for ?long-time))
        End Rule;


/*
        < signal to change the light -> set medium and short timers >
*/

        Rule Set-Medium-Timer Is
                If (signal (in_direction ?direction)) And
                  ?direction = ?red-light
                Then
                    Assert (timer (called short)  (set_for ?short-time))
                    Assert (timer (called medium) (set_for ?medium-time))
        End Rule;
```

```
/*
        < approaching traffic detected and medium timer exists
              -> reset short timer >
*/
        Rule Reset-Short-Timer Is
              If (signal (in_direction ?direction)) And
                ?direction = ?green-light And
                (timer (called medium))
              Then
                  Assert (timer (called short)  (set_for ?short-time))
        End Rule;


/*
      ;< timer expires -> change light >
*/
        Rule Timer_Expires Is
              If (timer (has_expired TRUE)) And
                (time (is ?t))
              Then
                  Call (change-light)
                  Write "change light at " ?t  " " ?green-light crlf
        End Rule;


/*
      ;< light changed -> retract medium and short timers >
*/
Rule Retract-Medium-Timer Is
              If (light-changed) And
                (timer (called medium)) And
                (timer (called short))
              Then
                  Retract (timer (called medium))
                  Retract (timer (called short))
End Rule;
```

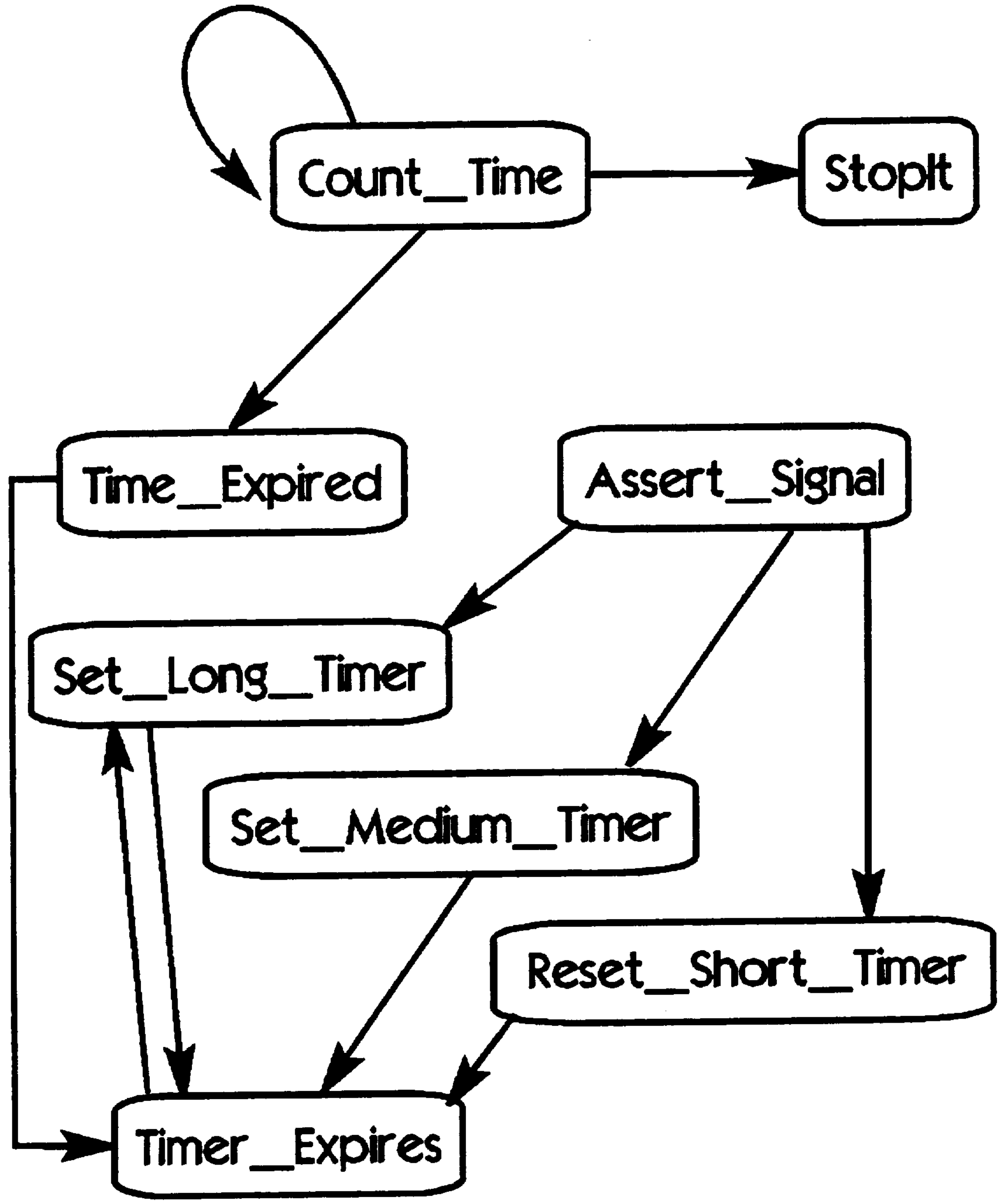


Figure 1: Diagram of Rule Relationships

Handout #5: Analyzing the Rule Base Implementations

Introduction

The purpose of this handout is to examine the benefits of applying *connectivity graph* analysis to the two CLIPS rule-bases generated for the traffic controller problem. Please refer to Landuaer (reference 21 in the "Techniques" section of the Presentation Material) for more complete descriptions of this approach. Nazareth (reference 41 in the "Techniques" section of the Presentation Material) also provides some of the more theoretical foundations for similar work in directed graphs (i.e., network flow). The first step in applying connectivity graphing techniques is to generate a complete list of rules and facts (this handout will only consider facts; other items such as clauses could be considered). Tables 1 and 2 show these lists from the first rule-base implementation of the traffic controller problem.

Tables 3 and 4 show the lists of rules and facts from the second rule-base implementation of the traffic controller problem. In general, whether building these connectivity graphs or not, generating a list of facts and rules can be very helpful in avoiding redundancies.

Identifier	Rule-Name
R_1	Update_Time
R_2	Trigger_Signal_Change
R_3	Del_Old_Changes
R_4	Trigger_Signal_Delay
R_5	Del_Old_Delays
R_6	Change_No_Signal
R_7	Change_No_Delay
R_8	Change_Delay
R_9	StopIt

Table 1: List of Rules from the Non-Modular rule-base implementation

Identifier	Facts
F_1	time ?t
F_2	green ?direction ?
F_3	signal ?other-direction ? ?t
F_4	signal_changes ?dt
F_5	signal_change ?t
F_6	signal_delay ?dt
F_7	end ?t

Table 2: Facts from the non-Modular rule-base implementation

Identifier	Rule Names
R_1	Count_Time
R_2	StopIt
R_3	Timer_Error
R_4	Timer_Name_Conflict
R_5	Initialize_Expires_At
R_6	Timer_Expired
R_7	Assert_Signal
R_8	Retract_Signal
R_9	Retract_Light_Changed
R_{10}	Set_Long_Timer
R_{11}	Set_Medium_Timer
R_{12}	Reset_Short_Timer
R_{13}	Timer_Expires
R_{14}	Retract_Medium_Timer

Table 3: List of Rules from Modular rule-base implementation

Identifier	Facts
F_1	time (is ?t)
F_2	stop_time ?t
F_3	timer (called ?) (set_for ?) (has_expired ?)
F_4	signal (in_direction ?) (signalled_by ?)
F_5	light_changed
F_6	signal_data ? ? ?

Table 4: List of Facts from Modular rule-base implementation

Generating Connectivity Graphs

Based on these tables, connectivity matrices can be generated. These matrices are good for examining a knowledge base to see how "interrelated" things are. Tables 5 and 6 show connectivity matrices derived from the fact and rule lists. These matrices are built by placing a 1 in each slot where a given fact is used on either the right or left hand side of the rule. A 0 in a given slot indicates that a particular rule does not reference the related fact. The equations of interest for tables 5 and 6 are:

- $(RF^{TR}) * (RF)$

- $(RF) * (RF^{TR})$

where **(RF)** is the initial Rule\Fact matrix and **(RF^{TR})** is the transpose of that matrix (i.e., creating a matrix by making the rows into columns and vice versa)

The first equation shown generates a matrix that shows, given an ordered pair of facts (f_i, f_j), whether a particular rule references both facts f_i and f_j (i.e., facts f_i and f_j have commonality). A graph can be generated based on this matrix where facts serve as the vertices of the graph and rules serve as the edges that connect these The second equation generates a similar matrix that shows, given any ordered pair of rules, (r_i, r_j), whether a particular fact is common to rules r_i and r_j. An undirected graph can also be generated from this matrix where the rules serve as vertices and the facts as edges.

Analyzing Connectivity Graphs

What can be learned about the two implementations of the traffic controller problem from these matrices? As it turns out, these matrices provide some important clues that can be used to assess the design of the two different implementations. To see these clues begin by considering the matrix generated from the non-modular rule-base implementation (see Table 7). As stated earlier, an undirected graph can be drawn based on the generated matrix where rules act as the vertices. Drawing a graph from the matrix in Table 7 generates, as expected, a very complex series of interactions. In fact, there is at least one edge between every rule and every other rule. This means that every rule has one or more facts in common with all other rules. Clearly, this would be a more difficult rule-base to analyze because of all these interactions.

What can be learned using the matrix generated from the modular rule-base implementation? The matrix should show that this implementation is easier to analyze. In fact, the matrix of Table 8 clearly shows a simpler connectivity structure as evidenced by the number of zeroes in the matrix (i.e., there are fewer edges in the graph). In addition, the matrix of Table 8 highlights the modules defined in the design (i.e., areas where higher numbers are clustered; e.g., the boxes in the inner portion of the matrix in Table 8). To prove this, compare the matrix of Table 8 to the modular rule-base design found in handout number five.

An interesting side-benefit to this is that, for the modular approach, one can assess, using the matrix of Table 8, the amount of coupling and cohesion that exists for each module. Every module should be strongly cohesive (i.e., the module is completely defined without any extraneous data or operations) and very loosely coupled (i.e., each module should have few, if any, dependencies on other modules) In the case of Table 8 one could make the arguement, for example, that the signal and timer modules should be combined to form one module due to the indications of coupling found in the middle box of Table 8. The loose coupling is evident by examining areas of the matrix in Table 8 that are not highlighted. The frequency of zeroes indicates that little or no coupling between modules exists.

Rules \ Facts	F_1	F_2	F_3	F_4	F_5	F_6
R_1	1	0	0	0	0	0
R_2	1	1	0	0	0	0
R_3	0	0	1	0	0	0
R_4	0	0	1	0	0	0
R_5	1	0	1	0	0	0
R_6	1	0	1	0	0	0
R_7	1	0	0	1	0	1
R_8	0	0	0	1	0	0
R_9	0	0	0	0	1	0
R_{10}	0	0	1	1	1	0
R_{11}	0	0	1	1	0	0
R_{12}	0	0	1	1	0	0
R_{13}	1	0	1	0	0	0
R_{14}	0	0	1	0	1	0

Table 5: Connectivity Matrix for the Modular rule-base implementation

Rules \ Facts	F_1	F_2	F_3	F_4	F_5	F_6	F_7
R_1	1	0	0	0	0	0	0
R_2	1	1	1	0	1	0	0
R_3	1	0	0	1	0	0	0
R_4	1	1	0	1	0	0	0
R_5	1	0	0	0	0	1	0
R_6	1	1	0	0	1	1	0
R_7	1	1	0	0	1	1	0
R_8	1	1	0	0	1	1	0
R_9	1	0	0	0	0	0	1

Table 6: Connectivity Matrix for non-modular rule-base implementation

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
R_1	1	1	1	1	1	1	1	1	1
R_2	1	4	1	2	1	3	3	3	1
R_3	1	1	2	2	1	1	1	1	1
R_4	1	2	2	3	1	2	2	2	1
R_5	1	1	1	1	2	2	2	2	1
R_6	1	3	1	2	2	4	4	4	1
R_7	1	3	1	2	2	4	4	4	1
R_8	1	3	1	2	2	4	4	4	1
R_9	1	1	1	1	1	1	1	1	2

Table 7: Connectivity Mapping between Rules (RF * RF^{TR}) for the non-modular rule-base implementation

Rules/Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}
R_1	1	1	0	0	1	1	1	0	0	0	0	0	1	0
R_2	1	2	0	0	1	1	1	0	0	0	0	0	1	0
R_3	0	0	1	1	1	1	0	0	0	1	1	1	1	1
R_4	0	0	1	1	1	1	0	0	0	1	1	1	1	1
R_5	1	1	1	1	2	2	1	0	0	1	1	1	2	1
R_6	1	1	1	1	2	2	1	0	0	1	1	1	2	1
R_7	1	1	0	0	1	1	3	1	0	1	1	1	1	0
R_8	0	0	0	0	0	0	1	1	0	1	1	1	0	0
R_9	0	0	0	0	0	0	0	0	1	1	0	0	0	1
R_{10}	0	0	1	1	1	1	1	1	1	3	2	2	1	2
R_{11}	0	0	1	1	1	1	1	1	0	2	2	2	1	1
R_{12}	0	0	1	1	1	1	1	1	0	2	2	2	1	1
R_{13}	1	1	1	1	2	2	1	0	0	1	1	1	2	1
R_{14}	0	0	1	1	1	1	0	0	1	2	1	1	1	2

Table 8: Connectivity Mapping between Rules (RF * RF^{TR}) for the Modular rule-base implementation

Generating Read/Write Matrices

Additional graph techniques exist for analyzing correctness criteria in a rule-base. One of these techniques works with matrices generated by examining the read/write relationships between facts and rules.This particular technique will be explored from the perspective of reachability (i.e., "can I get there from here?"). For example, Tables 9 and 10 show matrices that map rules to facts based on whether the fact appears on the right or left hand side of the rule for the non-modular rule-base implementation. Tables 11 and 12 show the analagous matrices for the modular rule-base implementation. Each of these matrices are built following a similar technique to the other connectivity matrices. A 1 is placed in each slot where a rule and fact are "connected." Zeroes indicate that there is no relationship between a given fact and rule.

Once these matrices have been built, two different equations can be used to analyze "reachability" issues within the knowledge base. The first equation below generates a matrix that matches facts against other facts (see Tables 13 and 19). The second equation matches rules against other rules (see Tables 14 and 20).

- $(\mathbf{Rd}^{\mathbf{TR}}) * (\mathbf{Wr})$

- $(\mathbf{Wr}) * (\mathbf{Rd}^{\mathbf{TR}})$

where **(Rd)** is the initial Rule\Fact read matrix and $(\mathbf{Rd}^{\mathbf{TR}})$ is the transpose of that matrix

Identifying Anamolies

Tables 13 and 19 show the fact to fact connectivity relationships for the non-modular and modular rule-base implementations respectively. What useful information does this matrix provide? These matrices indicate, for a given order pair of facts (f_i and f_j), whether a rule exists that reads f_i and writes f_j. Following this line of reasoning for the ad-hoc implementation, some anomalies in the rule-base are apparent. Anamolies, remember, do not necessarily indicate an error exists, but rather indicate that the possibility for an error exists. For example, consider the first column of the matrix. This column indicates that one rule reads f_1 and writes f_1, but no other rules write f_1. Is this a problem? Looking at the rule-base this can be explained. The rule Update_Time (this is the rule that both reads and writes fact f_1) is intended to update the time at the end of each cycle in order to simulate a clock. A salience value was added to the rule (i.e., this rule will not fire until a state is reached where no other rules at a higher salience can fire) to guarantee, among other things, that this rule is the only rule than can update the time (i.e., fact f_1). Therefore, this is not a problem.

Are there any other anamolies? Yes. Look at column three of the matrix in Table 13. The column contains all zeroes. This indicates that no rules write fact f_3 (this is also seen in the write matrix of Table 10). Yet, Table 9 indicates there are rules that read fact f_3. This is clearly an anomaly. Once again, though, this is not an error. As it turns out, all variations of fact f_3 have been defined within a deffacts structure (see page 1 of Handout #2). A similar line of reasoning can be used to explain the anomaly that the last column of the matrix (fact f_7) is also all zeroes.

What about the matrix for the modular implementation? Does this provide any useful information? There are two columns in this matrix that contain all zeroes. The column for fact f_2 can be explained using the line of reasoning from the previous paragraph. A *deffact* structure was used to do the write for fact f_2. The purpose of the rule that reads f_2 (which is rule R_2) is to terminate the rule-base. Therefore, should rule R_2 fire, the knowledge base terminates and no more "writes" are performed. The same arguements follow for fact f6 which also has all zeroes in its column. One process, then, for demonstrating correctness using these matrices is to look for anomalies and then provide arguements that these, in fact, are correct.

Anomalies also exist in the matrices of Tables 14 and 20. These matrices show rules that are related because they read and write the same facts. For example, the rules R_4 and R_5 are connected because they each read and write the fact f_6. One of the most curious anomalies in the matrix of Table 14 relates directly to the error discovered in Handout #3. Examine the row and column for rule R_3. Rule R_3 (Del_Old_Changes) is connected with itself, but is not connected via facts to any other rule. This indicates two things. First, R_3 is a dead-end rule. In other words, rule R_3 does not influence the firing of any other rules. Second, R_3 will, in fact, never fire because there are no other rules that write fact f_4. This is also evident in the inital read and write matrices, but is probably easier to analyze using one matrix than by trying to visually combine the results of two matrices.

Testing Reachability

Nazareth points out that for a connectivity matrix A, the equation A^n will generate a matrix showing whether a given rule, for example, can be reached from another rule across n edges (based on a graph that can be generated based on the connectivity matrix) of a directed graph. Using the matrices generated so far, the definition would look something like this:

$$A_{i,j} := \{ 1 \text{ iff rule}_i \text{ -> rule}_j \}$$

This equation states that the matrix A will contain a 1 whenever the result of firing rule_i influences the firing of rule_j to fire. The matrix generated from A^2, then, can be defined as follows:

$$A_{i,k} := \{1 \text{ iff rule}_i \text{ -> rule}_j \text{ -> rule}_k\}$$

This definition can be carried forward to show elements of reachability (i.e., can a given rule be influenced by another rule). In the framework of the matrices worked with in these examples, this connectivity is done, when working with rules, by facts. In other words, a given rule "writes" a fact and that influences the firing of other rules that also change facts that influence other rules and so on. Following Nazareth's approach generates a narrow result that allows one to focus on specific rules. For the examples here a more general reachability result was desired. To achieve this more general result, the following equation was used:

$$A + A^2 + A^3 + \ldots + A^n$$

This equation adds all of the A^n matrices (each value greater than 0 was converted to one since the concern was to show whether or not a rule was reachable from another rule not necessarily how many edges in a graph were required to achieve that reachability). Tables 15 through 18 show the results of applying this equation to the non-modular rule-base implementation. Tables 15, 16 and 17 show successive implementations while Table 19 shows the cumulative results of applying this equation to A^9. Tables 21 through 24 show the results as applied to the modular rule-base implementation. Tables 21, 22, and 23 show successive approximations while Table 24 shows the result up to A^5. The examples stopped at A^5 because the matrices generated following that up to A^{14} were all identical to A^5.

The primary result from applying this approach is that the anomalies mentioned earlier become more pronounced.These results become more pronounced because as the equation is carried out more slots become filled with one's until at some point the matrices begin to repeat. For the example, the row for R_3 never changes because as was already discovered this rule has essentially no bearing on the rest of the rule-base. The anomaly associated with rule R_1 also is still apparent because its column remained the same throughout.

The results of this equation when applied to the modular approach also provide interesting results. These results can be summarized by recognizing that there are fewer anomalies to consider for the modular case than for the ad-hoc case. This certainly supports the notion that designing modular knowledge bases results in easier analysis. While it is a positive thing that techniques such as these find anomalies, is it not better to design a system so that anomalies are avoided? Designing a system in this matter reduces the analysis of these matrices to confirmation that the system will perform as designed.

Landauer presents formulas for building other interesting matrices that can be used to analyze a rule-base. Nazareth also points to some interesting results that can be obtained by representing a rule-base as a directed graph and then applying elements of graph theory to do network flow analysis. These other techniques will not be considered here. However, the student is encouraged to examine these other techniques because of similar benefits they provide in analyzing a rule-base.

Rules \ Facts	F_1	F_2	F_3	F_4	F_5	F_6	F_7
R_1	1	0	0	0	0	0	0
R_2	1	1	1	0	1	0	0
R_3	1	0	0	1	0	0	0
R_4	1	1	0	1	0	0	0
R_5	1	0	0	0	0	1	0
R_6	1	1	0	0	1	1	0
R_7	1	1	0	0	1	1	0
R_8	1	1	0	0	1	1	0
R_9	1	0	0	0	0	0	1

Table 9: Read Matrix for non-modular rule-base implementation

Rules \ Facts	F_1	F_2	F_3	F_4	F_5	F_6	F_7
R_1	1	0	0	0	0	0	0
R_2	0	0	0	0	1	0	0
R_3	0	0	0	1	0	0	0
R_4	0	0	0	0	0	1	0
R_5	0	0	0	0	0	1	0
R_6	0	1	0	0	0	0	0
R_7	0	1	0	0	1	0	0
R_8	0	1	0	0	1	1	0
R_9	0	0	0	0	0	0	0

Table 10: Write Matrix for non-modular rule-base implementation

Rules \ Facts	F_1	F_2	F_3	F_4	F_5	F_6
R_1	1	0	0	0	0	0
R_2	1	1	0	0	0	0
R_3	0	0	1	0	0	0
R_4	0	0	1	0	0	0
R_5	1	0	1	0	0	0
R_6	1	0	1	0	0	0
R_7	1	0	0	0	0	1
R_8	0	0	0	1	0	0
R_9	0	0	0	0	1	0
R_{10}	0	0	0	1	1	0
R_{11}	0	0	0	1	0	0
R_{12}	0	0	1	1	0	0
R_{13}	1	0	1	0	0	0
R_{14}	0	0	1	0	1	0

Table 11: Read matrix for Modular rule-base implementation

Rules \ Facts	F_1	F_2	F_3	F_4	F_5	F_6
R_1	1	0	0	0	0	0
R_2	0	0	0	0	0	0
R_3	0	0	0	0	0	0
R_4	0	0	1	0	0	0
R_5	0	0	1	0	0	0
R_6	0	0	1	0	0	0
R_7	0	0	0	1	0	0
R_8	0	0	0	1	0	0
R_9	0	0	0	0	1	0
R_{10}	0	0	1	0	0	0
R_{11}	0	0	1	0	0	0
R_{12}	0	0	1	0	0	0
R_{13}	0	0	0	0	0	0
R_{14}	0	0	1	0	0	0

Table 12: Write Matrix for Modular rule-base implementation

Facts \ Facts	F_1	F_2	F_3	F_4	F_5	F_6	F_7
F_1	1	3	0	1	3	3	0
F_2	0	3	0	0	3	2	0
F_3	0	0	0	0	1	1	0
F_4	0	0	0	1	0	0	0
F_5	0	3	0	0	2	1	0
F_6	0	3	0	0	2	2	0
F_7	0	0	0	0	0	0	0

Table 13: Connectivity Mapping between Facts ($Rd^{TR} * Wr$) for the non-modular rule-base implementation

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
R_1	1	1	1	1	1	1	1	1	1
R_2	0	0	0	0	0	1	1	1	0
R_3	0	0	1	0	0	0	0	0	0
R_4	0	0	0	0	1	1	1	1	0
R_5	0	0	0	0	1	1	1	1	0
R_6	0	1	0	1	0	1	1	1	0
R_7	0	1	0	1	0	2	2	2	0
R_8	0	1	0	1	1	3	3	3	0
R_9	0	0	0	0	0	0	0	0	0

Table 14: Connectivity Mapping between Rules ($Wr * Rd^{TR}$) for the non-modular rule-base implementation

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
R_1	1	1	1	1	1	1	1	1	1
R_2	0	1	0	1	1	1	1	1	0
R_3	0	0	1	0	0	0	0	0	0
R_4	0	1	0	1	1	1	1	1	0
R_5	0	1	0	1	1	1	1	1	0
R_6	0	1	0	1	1	1	1	1	0
R_7	0	1	0	1	1	1	1	1	0
R_8	0	1	0	1	1	1	1	1	0
R_9	0	0	0	0	0	0	0	0	0

Table 15: Reachability Matrix (Rules\Rules) Step 2 ($A+A^2$)

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
R_1	1	1	1	1	1	1	1	1	1
R_2	0	1	0	1	1	1	1	1	0
R_3	0	0	1	0	0	0	0	0	0
R_4	0	1	0	1	1	1	1	1	0
R_5	0	1	0	1	1	1	1	1	0
R_6	0	1	0	1	1	1	1	1	0
R_7	0	1	0	1	1	1	1	1	0
R_8	0	1	0	1	1	1	1	1	0
R_9	0	0	0	0	0	0	0	0	0

Table 16: Reachability Matrix (Rules\Rules) Step 3 ($A+A^2+A^3$)

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
R_1	1	1	1	1	1	1	1	1	1
R_2	0	1	0	1	1	1	1	1	0
R_3	0	0	1	0	0	0	0	0	0
R_4	0	1	0	1	1	1	1	1	0
R_5	0	1	0	1	1	1	1	1	0
R_6	0	1	0	1	1	1	1	1	0
R_7	0	1	0	1	1	1	1	1	0
R_8	0	1	0	1	1	1	1	1	0
R_9	0	0	0	0	0	0	0	0	0

Table 17: Reachability Matrix (Rules\Rules) Step 4 ($A+A^2+A^3+A^4$)

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
R_1	1	1	1	1	1	1	1	1	1
R_2	0	1	0	1	1	1	1	1	0
R_3	0	0	1	0	0	0	0	0	0
R_4	0	1	0	1	1	1	1	1	0
R_5	0	1	0	1	1	1	1	1	0
R_6	0	1	0	1	1	1	1	1	0
R_7	0	1	0	1	1	1	1	1	0
R_8	0	1	0	1	1	1	1	1	0
R_9	0	0	0	0	0	0	0	0	0

Table 18: Reachability Matrix (Rules\Rules) Step 9 ($A+A^2+ \ldots +A^9$)

Facts \ Facts	F_1	F_2	F_3	F_4	F_5	F_6
F_1	2	0	2	1	1	0
F_2	0	0	0	0	0	0
F_3	1	0	5	0	1	0
F_4	0	0	3	1	0	0
F_5	0	0	2	0	1	0
F_6	0	0	0	1	0	0

Table 19: Connectivity Mapping between Facts ($Rd^{TR} * Wr$) for the Modular rule-base implementation

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}
R_1	1	1	0	0	1	1	1	0	0	0	0	0	1	0
R_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_4	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_5	1	1	1	1	2	2	1	0	0	0	0	1	2	1
R_6	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_7	0	0	0	0	0	0	0	1	0	1	1	1	0	0
R_8	0	0	0	0	0	0	0	1	0	1	1	1	0	0
R_9	0	0	0	0	0	0	0	0	1	1	0	0	0	1
R_{10}	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_{11}	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_{12}	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_{13}	0	0	0	0	0	0	0	0	1	1	0	0	0	1
R_{14}	0	0	1	1	1	1	0	0	0	0	0	1	1	1

Table 20: Connectivity Mapping between Rules ($Wr * Rd^{TR}$) for the Modular rule-base implementation

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}
R_1	1	1	0	0	1	1	1	0	0	0	0	0	1	0
R_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_4	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_5	1	1	1	1	2	2	1	0	0	0	0	1	2	1
R_6	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_7	0	0	0	0	0	0	0	1	0	1	1	1	0	0
R_8	0	0	0	0	0	0	0	1	0	1	1	1	0	0
R_9	0	0	0	0	0	0	0	0	1	1	0	0	0	1
R_{10}	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_{11}	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_{12}	0	0	1	1	1	1	0	0	0	0	0	1	1	1
R_{13}	0	0	0	0	0	0	0	0	1	1	0	0	0	1
R_{14}	0	0	1	1	1	1	0	0	0	0	0	1	1	1

<u>Table 21</u>: Reachability Matrix (Rules\Rules) Step 2 ($A+A^2$)

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}
R_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_4	0	1	1	1	1	1	1	0	1	1	0	1	1	1
R_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_6	0	1	1	1	1	1	1	0	1	1	0	1	1	1
R_7	0	0	1	1	1	1	0	1	0	1	1	1	1	1
R_8	0	0	1	1	1	1	0	1	0	1	1	1	1	1
R_9	0	0	1	1	1	1	0	0	1	1	0	1	1	1
R_{10}	0	1	1	1	1	1	1	0	1	1	0	1	1	1
R_{11}	0	1	1	1	1	1	1	0	1	1	0	1	1	1
R_{12}	0	1	1	1	1	1	1	0	1	1	0	1	1	1
R_{13}	0	0	1	1	1	1	0	0	1	1	0	1	1	1
R_{14}	0	1	1	1	1	1	1	0	1	1	0	1	1	1

Table 22: Reachability Matrix (Rules\Rules) Step 3 ($A+A^2+A^3$)

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}
R_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_4	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_6	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_7	0	1	1	1	1	1	1	1	1	1	1	1	1	1
R_8	0	1	1	1	1	1	1	1	1	1	1	1	1	1
R_9	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{10}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{11}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{12}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{13}	0	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{14}	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 23: Reachability Matrix (Rules\Rules) Step 4 ($A+A^2+A^3+A^4$)

Rules \ Rules	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}
R_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R_4	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_6	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_7	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_8	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_9	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{10}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{11}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{12}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{13}	1	1	1	1	1	1	1	1	1	1	1	1	1	1
R_{14}	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 24: Reachability Matrix (Rules\Rules) Step 5 ($A+A^2+ \ldots +A^5$)

Handout #6: Formal and Informal Proofs of Correctness

Introduction

It is important to argue the correctness of programs. Clearly, this is much easier when program specifications exist. One formal technique for analyzing a piece of software to determine if program specifications have been implemented correctly is called Symbolic Execution. There are three steps to doing symbolic execution. The first step, which is really a design step, defines specifications about program behavior. The next step graphs the program's flow of control (or execution). This graph is the framework for "tracing" a program's execution against its specifications. The last step traces execution of the program using symbols[1] to formally prove satisfaction of program specifications defined in the first step of this technique. This handout will demonstrate both a formal (based on symbolic execution) and informal proof of correctness style.

Formal "Proof of Correctness" Using Symbolic Execution

This section will outline the application of the three steps of the Symbolic Execution technique to the procedure, called Process_Signal, shown in figure 1. Process_Signal determines when the traffic light should change the flow of traffic after waiting traffic has been detected. Three variables are used to accomplish this task. These are:

- t_c : the current time
- t_s : the time to switch
- t_l : the maximum time traffic must wait before a light change

A function, called Approaching_Signal, is used to indicate whether traffic is detected in the current direction of traffic flow (i.e., returns the value TRUE when traffic is detected). For this demonstration one can assume that the function Approaching_Signal works correctly (i.e., there will be no additional proof of the correctness of this function).

[1]Symbols represent a "class" of values that a variable in the software may assume. This eases the trace of execution by eliminating the need to focus on specific data values.

```
Procedure Process_Signal Is
1   $t_l := t_c + 60$
2   $t_s := t_c + 15$
3   while $t_c < t_s$ loop
4           --<* $t_c < t_l$ and $t_c < t_s$ and $t_s <= t_l$ *>
5           if approaching_signal then
6               if $t_c + 15 < t_l$ then
                    $t_s := t_c + 15$
7               else
                    $t_s := t_l$
8               end if
9           end if
10          --<* $t_c < t_s$ and $t_s <= t_l$ *>
11          $t_c := t_c + 1$
12  end loop
13  --<* $t_c = t_s$ and $t_s <= t_l$ *>
End Process_Signal
```

Figure 1: The procedure Process_Signal

Step 1: Defining Specifications

As stated earlier, the objective of symbolic execution is to demonstrate that program specifications are correct. Line 4, 11, and 14 of figure 1 show the specifications that are to be proved correct for the procedure Process_Signal. The specifications shown are special kinds of specifications known as pre and post conditions. Pre-conditions appear define the required "state" of the program prior to executing a given program fragment. Post-conditions define the required "state" of the program when that specific program fragment completes execution. For example, Line 4 is a pre-condition for the If-then-else of line 5 and line 11 is the post-condition for that same if-then-else. Symbolic execution works best when using pre and post condition specifications.

Symbolic execution will prove that each of these pre and post conditions are satisfied by tracing the execution of the procedure Process_Signal. Based on the steps of the technique described earlier, step one is complete. These conditions are drawn directly from the statement of the problem. The desire is to, once waiting traffic has been identified, find the time when the light must change (when $t_c = t_s$). This time is determined by the amount of traffic in the direction of traffic flow. Regardless of this traffic flow, the light must change within at least one minute (t_l).

Step 2: Building a Graph

The second step is to generate graphs that allow for tracing procedure execution. There are two scenarios to consider when graphing the execution of a procedure. The first scenario involves procedures that have no loops (i.e., the graph shows be a single path of finite length). These are the simplest to trace and are done by examining each statement in sequence from beginning to end. The other scenario involves procedures that do have loops (i.e., the graph has an infinte number of paths with infinite lengths). For these scenarios, the infinite graph describing the loop can be analyzed as though it were finite by using mathematical induction[2].

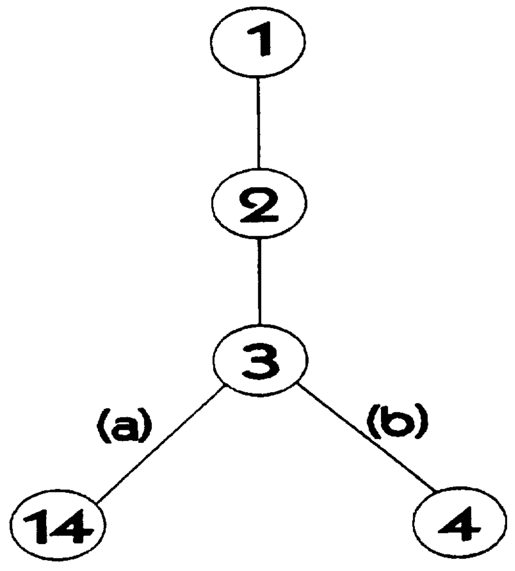
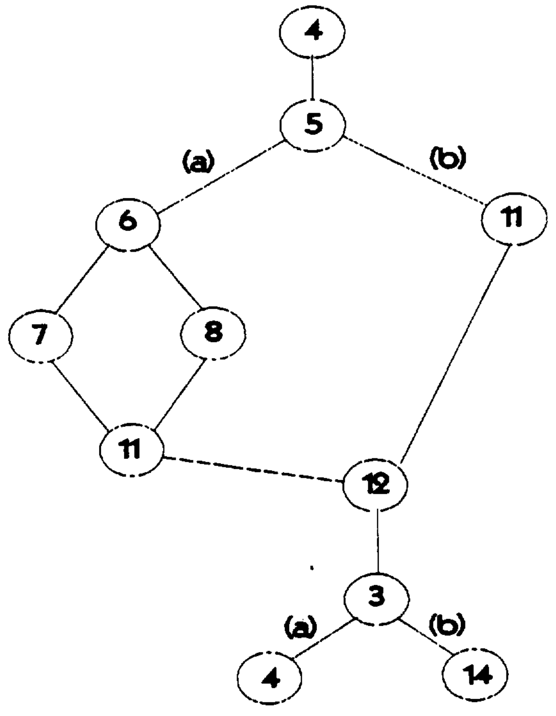
Graph assuming no loop iterations	Graph with loop iterations
	 4 5 (a) (b) 6 11 7 8 11 12 3 (a) (b) 4 14

Figure 2: Trace graphs for Process_Signal

Often times, whether there are loops or not, the graphs can become unwieldy. The best way to handle this is to break the graph into many smaller graphs (or "cuts"). The procedure can be shown to be correct by demonstrating each smaller graph is correct. This is a good technique for procedures with loops. For example, one might to prove the procedure works when (1) the loop never iterates and (2) the loop iterates one or more times. Figure 2 show this kind of separation for the procedure Process_Signal. It is worth noting that breaking a large graph into smaller ones can be difficult control structures are

[2]Induction for these cases involves demonstrating that if the loop executes correctly on the k^{th} iteration then it executes correctly on the $k+1^{th}$ iteration.

missing pre and post conditions. These conditions form natural boundaries that allow easier separation of a large graph into many small ones. For example, the pre-condition at line 4 of Figure 1 allows for a clean "cut" of a large graph (not pictured) into the two smaller graphs shown in figure 2.

Step 3: Tracing Program Execution

Now consider the final step of the symbolic execution technique. Each graph is traced using mathematical symbols to determine whether pre/post conditions are satisfied. It is worth noting that this technique is also very good at identifying things that are missing from a pre/post condition. In other words, the combination of a pre-condition and the trace may not provide enough information to prove that the post condition holds. Using the graphs from figure 2, begin with the simplest graph (i.e., the graph that assumes no loop iterations). Even though edge 3a depicts a condition where no iterations of the loop occur, by examination of lines 1 and 2 it is clear that under no circumstances will the loop at line 3 not iterate. For this reason this leg of the graph can be ignored (i.e., not traced). The edge 3b depicts the case where the loop will iterate any multiple of times. Proving this is simply the case of tracing lines 1,2, and 3 and then showing that the resulting values match the pre-condition shown at line 4. This step gets repeated as a part of tracing the more complicated graph, so specific trace results will be left to discussion of that graph.

The second. more complicated graph from figure 2 describes the true actions of the procedure Process_Signal. A simple way to trace the procedure is to build a matrix with a column for each value being traced. This eases the analysis burden by allowing the analyst to easily find which symbols map to which variables. The matrix of figure 3 shows the results of tracing the iteration graph of figure 2.

Initially, the symbol ß represents values for the variable t_c,. the symbol µ represents values for the variable t_s, and the symbol ø represents values for the variable t_l. These symbols, then, replace occurrences of these variables in the proof arguments that follow. The results of tracing lines 1 and 2 demonstrate this. For example, after executing line 1 the value of t_l, which initially is ø, is now ß+60 (replacing t_l with ø and t_c with ß in the equation $t_l := t_c + 60$). A similar result is attained after executing line 2 by following the same replacement strategy. The next line in the tracè is line 4. Line 4 is a pre-condition statement. Therefore, the pre-condition must be shown, using the symbols, to be satisfied. To do this, $t_c < t_s$ and $t_c < t_l$ and $t_s <= t_l$ must be shown to be true. By substituting the symbol values for the variables, these expressions become ß<ß+15 and ß<ß+60 and ß+15<=ß+60. These are all obviously true. A less formal arguement would contend that (1) the while-loop condition guarantees that $t_c < t_s$ at line 4, (2) since the value of t_l never changes and t_s is never assigned a value larger than t_l inside the loop then $t_s <= t_l$ and $t_c < t_l$.

Lines	t_c: ß	t_l: ø	t_s: µ	Arguements
1		ß+60		
2			ß+15	
4				<u>Prove</u>: $t_c < t_l$ and $t_s <= tl$ and $t_c < t_s$ TRUE : substitution of values for t_c and t_s yields the following: ß+15 <= ß+60 and ß < ß+60 and ß < ß+15
5b				Approaching_Signal is FALSE
11				<u>Prove</u>: $t_s <= t_l$ and $t_c < t_s$ TRUE : values for these variables have not changed since line 4 therefore, the same arguements apply
12	ß+1			
3b				$t_c >= t_s$
3a				$t_c < t_s$
15				<u>Prove</u>: $t_c = t_s$ and $t_s <= t_l$ TRUE : if, after substituting symbols, ß < µ (line 11) and ß+1 >= µ (line 3b) then ß=µ Also, since µ <= ø at line 11 and the values µ and ø have not changed, then $t_s <= t_l$ still holds
4				<u>Prove:</u> $t_c < t_s$ and $t_c < t_l$ and $t_s <= t_l$ TRUE : $t_c < t_s$ holds from line 3a $t_s <= t_l$ holds since values for t_s and t_l have not changed since line 11 Since $t_s <= t_l$ and $t_c < t_s$ are both true then $t_c < t_l$

Figure 3: Results of tracing the iteration graph from figure 2

Continuing with the trace, consider line 5. Line 5 is an If-then-else test on the condition that the function Approaching_Signal has detected traffic in the current direction of traffic flow. For simplicity, the trace of figure 3 follows the path that results from detecting no oncoming traffic (i.e., Approaching_Signal = FALSE). Tracing the opposite path (following edge 5b) will be left as an exercise for the reader. The next line to consider along the chosen path is line 11. Line 11 is the post-condition after execution of the If-then-else of line 5. Therefore, the properties $t_c < t_s$ and $t_s <= t_l$ must be shown to be true. By examining the matrix of figure 3 it is evident that the values for t_c, t_s, and t_l have not changed since line 4. The proof for line 4 showed that $t_c < t_s$ and $t_s <= t_l$. Therefore, these must still be true at line 11.

Line 12 is an assignment statement. Therefore, the original value, ß, for tc is now changed to ß+1. This is the last statement of the loop. From there the loop either iterates again (edge 3b) or exits the loop (edge 3a). Consider the case where the loop will iterate again. For this case, the properties of the line 4 must be shown to still hold true given new values calculated inside the loop (this is the induction step). The proof is straightforward. As in the first consideration of line 4, $t_c < t_s$ is true because of the condition on the while-loop. SInce the values for ts and tl have not changed since line 11 when $t_s <= t_l$ was shown to be true, $t_s <= t_l$ must still be true. Given the truth of these conditions, the expression $t_c < t_l$ must also be true. Therefore, line 4 holds for iteration.

Now consider the case when the loop does not iterate again. For this case the post-condition of line 14, $t_c = t_s$ and $t_s <- t_l$, must be true. Once again, this is fairly simple. Given the condition of the while-loop required to prevent iteration of the loop, the condition $t_c >= t_s$ must be true. The trace at line 11 indicates that (using the symbols) $ß < \mu$. Line 14 indicates, once again by substituting symbols for variables, that $ß+1 >= \mu$. For these conditions to both be true, $ß = \mu$ at line 14. The other condition, $t_s <= t_l$, must be true because the same condition was true at line 11 and the values for t_s and t_l have not changed.

Informal "Proof of Correctness"

This concludes a formal (partial) proof of the procedure Process_Signal. It is reasonable to argue that this formalism can be difficult to carry out when the program is very large. Yet, the essence of the formalism can be captured as an informal proof that is also correct. Informal proof of correctness seeks to informally argue the correctness of each program specification. For example, consider the informal proof of correctness for line 14.

Initially $t_c < t_s$ and $t_c < t_l$ and $t_s < t_l$ (by lines 1 and 2). Based on the internals of the loop, t_c is incremented with each iteration of the loop. Therefore, the proof for line 14 must demonstrate that there is an upper bound for t_c which is t_s. This also means there must be an upper bound for t_s since its value also increases inside the loop (i.e., if there is no upper bound for t_s, then t_c can not have an upper bound that is t_s). Lines 7 and 8 are the only places where the value of t_s is changed. Based on the condition of line 6 it is clear that t_s can never exceed t_l (which is constant throughout the loop). Therefore t_s has an upper bound and is guaranteed to eventually reach that upper bound. Therefore, t_c is guaranteed to reach an upper bound based on increments of 1 inside the loop. This means that when the upper bound of ts is reached, $t_s = t_c$ (since ts and tc are integers).

Figure 4: Informal arguement for the correctness of Process_Signal

The proof shown is as correct as the proof generated by Symbolic Execution. However, the informal proof of figure 4 is probably simpler to read and is certainly simpler to generate. The conclusion, then, is that the procedure Process_Signal was simple enough that doing an informal proof was probably the better, more efficient choice for proving correctness. However, following a formal approach is very good when dealing with complicated parts of programs that depend on the rigor of a formal method for ensuring important details are not overlooked.

Handout #7: Exercises on General Techniques

1. Define the "black box" view for your system.

2. Identify key terms from the problem description.

3. Which of the following techniques would you use? Explain your answer.

- Prototyping
- Competing Designs
- Independent V&V
- Inspections

4. Do a very high level specification for your system using one of the following techniques:

- Decision Table
- Cause-Effect Graph
- State Diagram

Handout #8: Exercises on System Test Techniques

1. Define 1 or more "realistic" test cases for your team exercise.

2. Define some attributes of your system. Define 1 or more test cases based on those attributes.

3. Define 1 or more test cases that do "boundary value" testing.

4. Define 1 or more test cases that "stress" test the system.

5. Define the external interfaces to your system. Define 1 or more test cases to test those interfaces.

6. Define 1 or more test cases to test the system's performance.

7. For each question, indicate how the results of each test case will be analyzed (i.e., how you will know the answer is correct).

8. Did the problem description provide enough detail to adequately perform the tests from questions 1-6?

9. Develop a "certification" test for your system.

10. Identify system "disasters" (i.e., things that should not happen). Explain how you will test your system for these "disasters".

11. Will your project need the aid of an expert (provide rationale)? If so, indicate the kind of expert required and the type of analysis to be performed.

12. Define 1 or more models to aid in your understanding of the system. Document each model.

Handout #9: Exercises on Unit/Integration Test Techniques

1. Pick an implementation approach for your problem. Based on this choice, would you use:

   - Coverage techniques
   - Interprocedural data-flow analysis

2. Identify "part" of the system that may impact reliability (HINT: you may have to define what reliability is). Define 1 or more test cases to test those "parts".

3. Document 1 or more expected sequences of actions for your system.

4. Is "prototype evaluation" appropriate for your problem? What about mutation testing? Provide rationale.

5. Exchange your work with another team. Study the problem. Ask yourself the following:

   - Does their implementation match the problem?
   - Are there any "holes" or inconsistencies in their descriptions?
   - Did they pick the right techniques for their implementation approach?

Handout #10: Exercises on Static Test Techniques

1. Identify and define at least 1 "object" in your system (remember, objects consist of both data and operations on that data).

2. Write a pre-condition and a post-condition for each operation on the object.

3. Describe any general properties your "object" must satisfy. Discuss how you would analyze your "object"'s implementation to "prove" those properties are always satisfied.

4. Pick at least one operation and defined some rules that implement its specificiation.

5. Select one of the following techniques for analyzing these rules. Explain your answer.

- Petri Nets
- Directed Graphs
- Connectivity Matrices

6. Identify 1 "hazard" in your system. Build a fault tree for that "hazard".

7. Idenfity 1 "fault" in your system. Build a fault tree for that "fault".

Handout #11: Exercises on Guidelines

1. Determine whether the recommended approach fits your problem. Identify additional issues that need to be considered.

2. Generate a detailed development plan for your problem. Try to include specific milestones and how they will be achieved.

3. Define specific development increments. Update your plan to reflect those increments.

4. Consider the test cases you have selected so far. Are there any other kinds of testing you need to do? When will you know when to stop testing?

5. Build a high-level requirements outline for your system. How well does the original problem definition map to your outline?

Handout #12: Alarm 1201!: A History Lesson in Some Important Aspects of Verification and Validation

Introduction

On July 20, 1969, Astronauts Neil Armstrong and Buzz Aldrin were preparing their spacecraft for the programmed descent toward the first landing on the moon. Armstrong gave a command to the guidance computer, instructing it to switch to descent mode. A few minutes later, he fired the descent engine. As they descended behind the moon, Aldrin gave another command which was due to a last minute change in the crew procedure; he instructed the guidance computer to begin autotrack mode for the rendezvous radar. The radar began interacting with the guidance computer to maintain lock on the command module in case of an abort back to the command module. This increased the workload on the computer. Just as Armstrong was getting to where he could see the landing location, Aldrin reported "Alarm ! 1201 1201 !" ([6]) The guidance computer was overloaded and beginning to shed less critical processes. Back in Houston, guidance officer Stephen Bales, who was unaware of the activation of the radar since it occurred behind the moon, quickly analyzed the computer overload situation. He saw that the descent profile was nominal and nothing appeared to be going wrong (Fortunately, the rendezvous radar, which was not needed yet anyway, was the only process low enough in priority not to be run.). However, the alarms continued to go off, causing "grave concern" ([1]) about the mission. As precious time was being spent dealing with the computer problem instead of looking out the window to pinpoint the landing site, Armstrong was unaware that he was headed toward a crater about the size of a football field that was full of boulders. Bales, fearful that if the computer overload continued then it would reject all its programs, instructed the crew to stop monitoring the landing radar data and leave that function to the ground. This reduced the load on the guidance computer, ceasing the disturbing alarms. Armstrong looked out the window and noticed the landing site was bad, quickly took over manual control and guided the vehicle to a safe landing with only seconds of fuel to spare.

The computer overload was later called the most serious problem of the Apollo 11 flight by mission analysts and was the center of the post-flight analysis. One's first reaction might be to blame the crew or mission planners for putting the rendezvous radar in autotrack. However, as they testified to afterwards, they were completely unaware that this action could lead to computer overload or any other problem. Although designer's of the guidance system had suspected such an action could cause a computer overload, they were unable to confirm this in simulations. In any case, it shouldn't be a problem because they had designed the system to handle overloads by shedding less critical processes. From their point of view, it could be said that there was no computer overload "problem"; the system functioned perfectly. Unfortunately, the guidance system was also designed to announce the problem by a vague alarm which did not convey the lack of seriousness of the situation. Ground controllers, though they quickly handled the situation and prevented a mission abort, could be blamed for not diagnosing the cause of the overload. However, they had no data on which to base such a diagnosis. They were not even able to see that the radar had been commanded to autotrack since this action occurred behind the moon.

What does this have to do with verification and validation (V&V) of expert systems? Certainly none of these groups can singly be blamed for this problem. It could all be classified as a pure accident due to several contributing factors. However, more complete verification and validation (V&V) could have prevented this problem from occurring. And, although no expert systems were involved in this situation, several intelligent agents that today could involve expert systems played a key role in the problem. We will use this Apollo 11 scenario to demonstrate several key points about V&V of expert systems by considering how V&V could have avoided this problem, even if one or more of the intelligent agents really involved an expert system.

Importance of Requirements and V&V of Knowledge

Obviously, the crew would not have put the rendezvous radar in autotrack if it had not been called for in the crew procedures. The mission planners would not have put this action in the crew procedures if they had been aware of its potential to cause a problem. We could say that the mission planners should have asked the design engineers if any procedure changes could potentially cause problems. If the crew procedures were designed by an expert system (ES), there would have to be a requirement for the ES to consult design engineers about potential problems caused by changes, especially last minute changes, in the procedures. So, more complete documentation of functional requirements, including the requirement to ask about potential problems of last minute changes, could have prevented the problem. However, this might not be realistic, especially for an ES. Instead, it might be more realistic to have documented possible problems associated with types of changes. For example, a simple rule stating that additionally activated processes (e.g., the one activated during autotrack) add additional load on the computer along with a rule that said that the computer was already expected to be close to its 90% processing design limit during descent would be enough information for the ES to be alerted to the problem of adding autotrack activation to the crew procedures. This is an example of the need to V&V knowledge that goes into an ES.

Importance of Static Analysis

Guidance design engineers suspected such a problem could arise; this potential problem had been reviewed and understood. Unfortunately, the engineers were unable to accurately simulate the wandering of the rendezvous radar and the processing load on the computer that would be required to keep the radar locked onto the command module. Yet they were easily able to analyze the ultimate cause of the computer overload problem and completely understood how the activation of autotrack mode could lead to the problem. According to one engineer, although the overload had been an acknowledge possibility, when the warning lights came on it "really brought us up out of our seats" ([1]). Though they could have predicted it, it was still a surprise when it actually happened. This illustrates that certain kinds of problems can be easy to find using static analysis (i.e., manual analysis) and can be difficult if not impossible to find (or confirm) using dynamic analysis (i.e., computer simulation). This holds true for ESs as well as conventional software.

Importance of Explicit Documentation of Design Constraints

Similarly, the guidance system designers had no requirement to tell the mission planners that there was a possibility of a computer overload problem if the rendezvous radar was in autotrack. This is an example of one intelligent agent (or part of the overall system) having knowledge about a constraint but this knowledge is not explicit so that other intelligent agents (i.e., mission planners) could see that they were violating the constraint. If both mission planning and guidance design were done by an ES, this would be an example of a implicit constraint. Consistency of the guidance design with mission planning could not be verified because the constraint(s) was implicit, i.e., known but hidden. For verification purposes, all constraints should be made explicit.

Importance of V&V'ing Even the Smallest Changes

The change to the crew procedures to put the rendezvous radar in autotrack during descent had been a last minute change and this may have contributed to the potential problem not being discussed more before the flight. Because ESs are changed frequently, many ES modifications are really last minute changes. As with the Apollo 11 planning, there may be a tendency not to V&V these last minute changes because they are small minor enhancements. But, as can be seen with this example, even small minor changes can have drastic consequences. All changes, no matter how minor, need to be V&V'ed.

Correctness of System Responses Can be Difficult to Judge

Although this example problem has often been used as an example of a computer software bug, it is really a very complex and subtle problem that can not accurately be classified as a software bug. In fact, it is very difficult to identify the specific fault or error that caused the problem. It is even debatable as to whether any problem occurred at all; the system did operate correctly. As with many complex problems, correctly classifying a system response as an error may require expert judgement; this is especially true of ESs.

Importance of Different Kinds of Correctness

:p.One reason this example problem is difficult to analyze is that, with any system, there are really many different kinds of correctness to be considered. For example, to the design engineers, the system was correct but to others, the system had a severe problem.

Functional Correctness

To the design engineers who were most concerned about functional correctness, the system was correct. That is because it correctly implemented all the functions that they thought it should. For all inputs, the system produced exactly the output they expected.

Performance Correctness

Performance correctness, whether the system can perform all mandatory requests with available resources, was a key issue in this example problem. And the system was correct from a performance point of view. It did handle all mandatory requests; only rendezvous radar processing, which was not mandatory, was not handled. However, a key point illustrated by this example problem is the difficulty in analyzing performance correctness. The design engineers were unable to predict when the computer might be overloaded; it would depend on a lot of variables that they could not analyze with certainty. This is especially true of ESs, which often must deal with varying situations and whose computation (or inference) times can vary depending on the combination of requests it is trying to satisfy.

User-Interface Correctness

To the crew; whose understanding of the system is primarily due to lights, alarms, and displays information produced by the system; the system was less correct than they would have liked. That is, the system had a user-interface problem. They had no way of knowing what the 1201 alarm really meant and how serious the problem was. Armstrong, when asked about the seriousness of the 1201 alarm during a press conference, said "... as soon as program computer alarms manifest themselves, you realize that you have a possible abort situation to contend with" ([4]). In other words, all he knew was that an alarm had sounded; this was a problem that had to either be resolved or the mission must be aborted. So although the system was functionally correct, from a user-interface point of view it was incorrect.

Safety Correctness

The most important type of correctness in most systems is safety - "Above all, do no harm." When one looks at the alarm issued by the guidance computer from a user-interface and safety point of view, one must conclude that it was definitely not correct. The issuance of the alarm actually created a safety problem where none had existed before. There was no problem until the crew and the ground began analyzing the 1201 alarm, wasting precious time while the vehicle headed toward an unsafe landing area. Yet, from a functional point of view, the same alarm was correct. It correctly indicated that the computer was indeed overloaded. So it really is important to look at things from different correctness points of view.

Expert Systems are Software, Only Different

When looking at many statements made about the guidance computer, one can get the impression that the system was truly intelligent. One explanation of what the computer was doing was "The computer in effect started to tell the crew that it was being asked to work beyond its capacity. It advised that interrogations from the rendezvous radar should cease because they were of lower priority" ([1]). This explanation really makes it sound like the computer "understood" what it was doing. Aldrin's description was a little more technically correct but still implied some more intelligence that might be due. He said the computer "... continually goes through a wait list of one item after another. This list was beginning to fill up and the program alarm came up" ([4]). In reality, this 20 year old machine language program neither explained its actions nor looked at its list of instructions and figured out that it was more than it could safely handle. Most users are not really aware of how "intelligent" the programs that they use are. For all they know, many of them could already be ESs. In other words, when looked at as a black box that does something, one doesn't really care whether the box has an ES in it or not. It is just software.

However, had the guidance computer actually contained an ES based on knowledge from an expert and implemented in an interpreted non-procedural language (e.g., a production system), V&V would have been done differently. It probably would have solved a more complex problem than simply calculating guidance data and passing it to different devices (e.g., the rendezvous radar). It probably would have made the decision of whether or not to put the rendezvous radar in auto-track. And instead of analyzing the correctness of a crew or mission planner decision, the correctness of the programmed computer decision would have been done. It also could have been a collection of rules, some of which processed things for the rendezvous radar as well as other things. If so, the analysis required to figure out that the radar being in autotrack was the cause of the problem would have been more difficult because of the complex intermingling of its processing with other processing. On the other hand, the handling of requests to the guidance computer could likely be handled by a far simpler and smaller rule-based program than the original machine language program. And smaller simpler programs are usually easier to analyze and test. Expert systems are software but they are a truly different kind of software.

Summary

Using the Apollo 11 computer overload scenario, the following key points have been illustrated.

1. Importance of complete V&V

2. Importance of requirements

3 Importance of knowledge V&V

4 Importance of explicit documentation of constraints

5 Importance of V&V'ing all changes, no matter how small

6 Subjective nature of correctness, in some cases

7 Importance of looking at all the many different kinds of correctness

8 Expert systems are software but a different kind of software

NASA/JSC's workshop on Verification and Validation explains all of these points in more detail and provides specific recommendations for how to handle all the different issues in V&V of ESs. By attending this workshop, you may be able to avoid one of your user's from experiencing an "Alarm 1201 !" situation.

References

[1] "Computer Overload Laid to Radar Mode", *Aviation Week and Space Technology*, Aug. 4, 1969

[2] "Armstrong's Piloting Reflexes Avert Rocky Landing for Eagle", *Aviation Week and Space Technology*, July 28, 1969

[3] Beyond the limits book (full reference needed)

[4] NASA press review document (full reference needed)

[5] NASA Apollo 11 anomaly summary document (full reference needed)

[6] Book about Apollo 11 (full reference needed)

Worksheet #1: State Diagrams

When To Use This Technique:

System Test, Unit\Integration Test, and Static Test

Who Uses This Technique:

Anyone interested in analyzing the system (e.g., users, developers, independent verifiers, etc.)

Why Is This Technique Used:

Generation of test cases, Design, Requirements definition, Correctness analysis

How To Use This Technique:

Key Terms

Automaton	A machine or control mechanism designed to follow automatically a predetermined sequence of operations or respond to encoded instructions. Sometimes called a "state machine".
Event	An external stimulus that either by itself or, in conjunction with other events, causes an object to change state
Event Class	An abstract name describing a collection of common events.
Function	A one-to-one mapping that has no states.
Module	A "piece" of the system that, in its most common form, captures a unique piece of data and operations on that data. Synonymous with an "object".
Output	Externally visible (to the module) results due to side effects of a state transitions.
Scenario	A sequence of events that occur during one particular execution of a state machine
State	A complete description of the state machine at a particular instant in time
State Diagram	A network of states and events where transition from the current state to the next state depends both on the current state and the occurrence of a specific event.

State Transition	When the current state changes (or transitions) to the next state. State transitions appear as the "arcs" between states in a state diagram. A state transition can occur automatically or as a result of a single event or event class. Arcs that represent transitions that happen automatically are not labeled.

Method

For each module:

1. Identify inputs and output for the state machine

   a. Identify stimuli and associated responses. If each event by itself leads to an output then a state machine is probably not appropriate. Use a function instead.

   b. Identify internal states and the events that cause transitions from one state to another

3. Identify initial and final "states"

4. Build a state transition matrix that maps states against states
   - Given n states, the matrix will be n x n

5. Draw a state transition diagram
   - States appear as circles
   - For each (S_i, S_j) in the transition matrix that equals "1", draw an arc from S_i to S_j
   - Label each arc with the specific events that cause the state transition

Helpful Hints

Unless the system in extremely simple, do not attempt to build a state diagram that describes the entire system.

Focus on building state diagrams centered on specific objects (or modules) and analyze those individually.

To enhance understandability, begin with high-level "abstract" state diagrams and then refine them into more detailed diagrams as the need for additional detail arises. A module that is in one state for the "abstract" state diagram must also be in one state for the refined state diagram.

The results of steps 4 and 5 are conceptually the same. They are simply represented differently. It is helpful to do both, but certainly not necessary.

Example

Consider, for example, the following description.

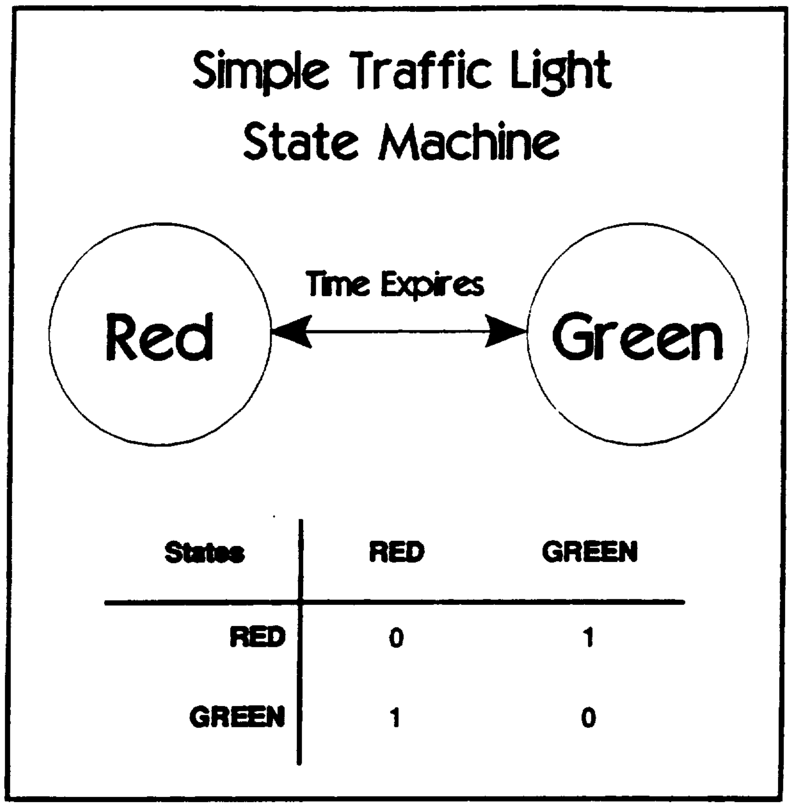
A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every two minutes. With a pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Some (there are others) possible scenarios that cause the traffic light to change are:

- No approaching traffic for a two minute period should change the light
- Waiting traffic is detected and no approaching traffic is detected for the next 15 seconds
- Waiting traffic is detected and approaching traffic is detected each second thereafter for at least one full minute.

From the highest level of abstraction it is clear that when a period of time (hereafter referred to as a timer) expires the traffic light changes. Figure 1 shows a state diagram modeling this abstraction. A simple state transition matrix that mirrors the state diagram is also included in figure 1. Even though, from a high level, this is an adequate description of the traffic controller, there are many details not represented. For example, the state diagram of figure 1 alludes to the use of a timer, yet details concerning the operation of timers is hidden. Also, from the scenarios, it is clear that traffic flow impacts when the light will change. Yet, this information is also hidden from the abstraction in figure 1.

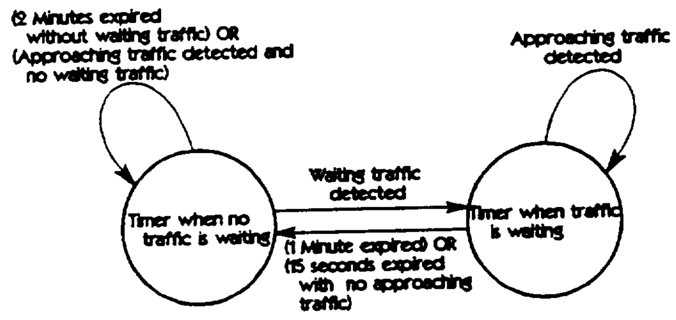
The state diagram of figure 2 shows a refinement of the state diagram shown in figure 1 that describes what a timer should do within the traffic controller system. Notice that details regarding the impact of traffic flow are now captured. This is because the traffic impacts the operation of the timers which in turn impact the operation of the light. Therefore, details regarding traffic flow are a needed part of the diagram in figure 2 in order to describe how the timers should work. Despite the additional detail, the mapping from one diagram to the other is relatively straightforward. Each transition that is caused by an expiration of time (as opposed to a transition caused by approaching or waiting traffic) maps to a change in the light.



States	RED	GREEN
RED	0	1
GREEN	1	0

Figure 1

Timer State Diagram


(2 Minutes expired without waiting traffic) OR (Approaching traffic detected and no waiting traffic)
Approaching traffic detected
Waiting traffic detected
Timer when no traffic is waiting
Timer when traffic is waiting
(1 Minute expired) OR (15 seconds expired with no approaching traffic)

States	Timer when no waiting traffic	Timer when traffic is waiting
Timer when no waiting traffic	1	1
Timer when traffic is waiting	1	1

Figure 2

Worksheet #2: Decision Tables

When To Use This Technique:

System Test, Unit\Integration Test, and Static Test

Who Uses This Technique:

Anyone interested in analyzing the system (e.g., users, developers, independent verifiers, etc.)

Why Is This Technique Used:

Generation of test cases, Design, Requirements definition, Correctness analysis

How To Use This Technique:

Key Terms

Action	Results caused by success of conditions (i.e., the right-hand side of a rule)	
Condition	Stimulus that, when satisfied, contributes toward one or more actions (e.g., part of the left-hand side of a rule)	
Decision Table	A matrix that has one column for each possible condition and one column for each possible action. Condition columns appear as far to the left in the matrix as possible. Action columns appear as far to the right as possible. Values appear in the columns to convey specific information. Two kinds of information are conveyed:	
	Binary (1 or 0)	e.g., Condition occurred or not, action performed or not
	Multi-value	e.g., column contains a value for a variable
Function	A one-to-one mapping that has no states.	
Module	A "piece" of the system that, in its most common form, captures a unique piece of data and operations on that data. Synonymous with an "object".	
Rule	Each row of a decision table is called a "rule". Rules are of the form:	
	If <left-hand side> Then perform <right-hand side>	
Scenario	A sequence of one or more rules in the decision table	

Method

For each module:

1. Identify all possible conditions and actions along with all possible values for each. Best to start by identifying and analyzing stimulus/response histories.

2. Build a matrix where rows will serve to map conditions against actions
   - For binary decision tables, given n conditions and m actions, the matrix will have 2^n rows and (n+m) columns. Matrix dimensions will vary for non-binary (e.g., multi-value) decision tables.
   - Condition columns appear to the left (C_1, C_2, .. C_n) and action columns appear to the right (A_1, A_2, ..., A_n).

C_1	C_2	. . .	C_n	A_1	A_2	. . .	A_n

3. Fill in the condition columns with all possible combination of 1's and 0's (1's indicate that the specific condition is true, 0's indicate the opposite)

C_1	C_2	. . .	C_n	A_1	A_2	. . .	A_n
1	1	. . .	1				
0	1		1				
1	0		1				

4. Examine each row and determine the actions that should occur as a result of the conditions. Place a 1 in the column for each action that should occur and a 0 in those that should not occur. For cases, where it is not clear whether a specific action results from a set of conditions, place a "?" (or any other special character of your choice) in the column for that action.

C_1	C_2	. . .	C_n	A_1	A_2	. . .	A_n
1	1	. . .	1	1	0	. . .	0
0	1		1	0	0		1
1	0		1	0	1		0
		. . .				. . .	

5. Work with users, experts, etc. to resolve specially marked columns.

Helpful Hints

The size of decision tables suffer from combinatoric explosion if they are used to describe an entire system (even a relatively simple one). Decision tables work much better at a unit or module level.

Decision tables are complementary to other techniques such as state diagrams and cause-effect graphing.

Decision tables translate fairly easily to rule-based languages.

Make sure all "slots" in the matrix are resolved before proceeding. Doing this guarantees that all possible conditions and their corresponding results have been considered.

Very good for analyzing correctness/completeness of a system, building test cases, and performing system design.

<u>Example</u>

Consider, for example, the following description.

> A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every two minutes. With a pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Some (there are others) possible scenarios that cause the traffic light to change are:

- No approaching traffic for a two minute period should change the light
- Waiting traffic is detected and no approaching traffic is detected for the next 15 seconds
- Waiting traffic is detected and approaching traffic is detected each second thereafter for at least one full minute.

At the highest level of abstraction, the traffic controller changes the traffic light when a specified period of time (hereafter referred to as a timer) expires. Therefore, there is only one condition to consider; whether a timer has expired or not. There is only one action that occurs based on this condition; the light changes. The decision table of figure 1 illustrates the simple decision table that results.

Timer Expires	Change Light
1	1
0	0

Figure 1

Obviously, from both the problem description and the scenarios, there are other conditions that indirectly impact when the light changes because they impact how the timer works. Therefore, the table of figure 1 can be "refined" to a more descriptive decision table that captures those hidden details. Figure 2 illustrates this refined decision table. The following conditions are considered in this table:

- Traffic is approaching in the current direction of traffic flow
- Traffic is waiting for the light to change
- 2 minute times has expired
- 15 seconds has expired
- Traffic has been waiting for 1 minute

The actions resulting from these conditions relate to selecting the appropriate timer to use. These actions are:

- Wait for 2 minutes
- Wait for 15 seconds
- Wait for 1 minute

A partially filled in table is shown in figure 2. The filled in values relate directly back to the scenarios identified at the introduction to this example. For example, the first filled in row of the table in figure 2 indicates that when traffic is detected in the current direction of traffic flow and no traffic is currently waiting for the light to change then the 2 minute timer is used (i.e., the controller will begin waiting for a 2 minute period. The second filled in row indicates that when waiting traffic is detected and the controller is waiting for 2 minutes to expire, the controller should begin looking for either 15 seconds to expire or 1 minute (depending on the presence of approaching traffic). There are many more rows (2^5 rows to be exact). Many of these rows will result in impossible scenarios or "don't care" scenarios. The important thing is that all scenarios have been considered. The table of figure 2 could be relatively easily related to the table of figure 1 by adding another action that indicates whether or not the controller needs to be signaled that, due to actions related to the timer, the light should change.

App	Wait	2M Exp	15S Exp	1M Exp	Wait 2M	Wait 15S	Wait 1M
. . .							
1	0	1	0	0	1	0	0
0	1	1	0	0	0	1	1
. . .							

Figure 2

Worksheet #3: Cause-Effect Graphing

When To Use This Technique:

System Test, Unit\Integration Test, and Static Test

Who Uses This Technique:

Anyone interested in analyzing the system (e.g., users, developers, independent verifiers, etc.)

Why Is This Technique Used:

Generation of test cases, Design, Requirements definition, Correctness analysis

How To Use This Technique:

Key Terms

Abstraction	A higher level, equivalent description that hides unnecessary implementation detail.
Cause	A stimulus that contributes toward one or more responses.
Cause-Effect Graph	A graph where all causes appear to the left and all effects appear to the right. Arcs are drawn from causes to effects. Arcs either go directly from one cause to an effect or, via boolean operators (see figure 1), combine with arcs from other causes to go to an effect.

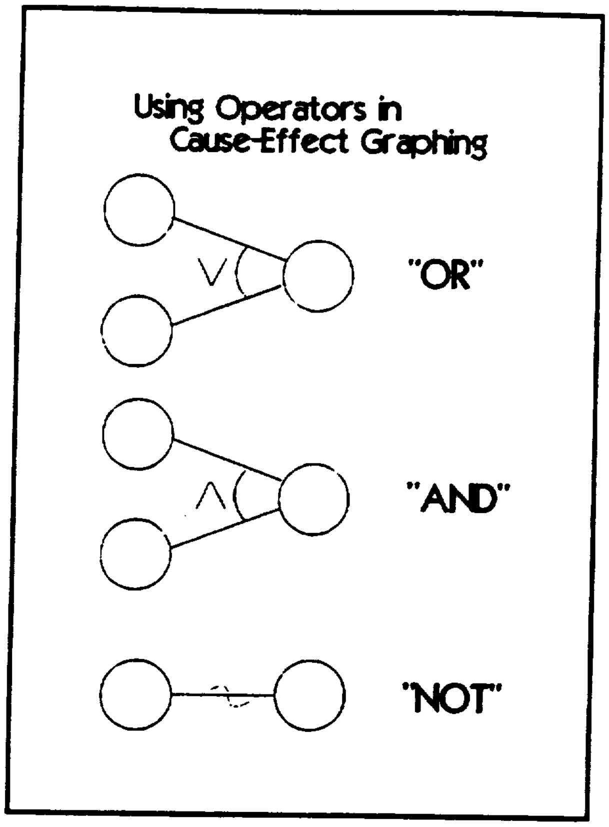


Figure 1

Effect	A response generated by combinations of stimuli.
Module	A "piece" of the system that, in its most common form, captures a unique piece of data and operations on that data. Synonymous with an "object".
Scenario	A path through the cause-effect graph.

Method

For each module:

1. Identify all possible causes and effects. This can be done at varying levels of abstraction. The best place to start is by identifying stimuli and responses.
2. Place all causes to the left side of the graph.
3. Place all effects to the right side of the graph.
4. Map causes to effects.
   a. Look at different combinations of causes to determine if those combinations are possible
   b. Try to generate interim nodes in the graph as a way to capture abstractions.

Helpful Hints

The size of cause-effect graphs suffer from combinatoric explosion if they are used to describe an entire system (even a relatively simple one). Cause-effect graphs work much better at a unit or module level.

Cause-effect graphs are complementary to other techniques such as state diagrams and decision tables.

Cause-effect graphs translate fairly easily to rule-based languages (e.g., each path is an if-then-else rule).

Very good for analyzing correctness/completeness of a system, building test cases, and performing system design.

Example

Consider, for example, the following description.

> A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every two minutes. With a pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Some (there are others) possible scenarios that cause the traffic light to change are:

- No approaching traffic for a two minute period should change the light
- Waiting traffic is detected and no approaching traffic is detected for the next 15 seconds
- Waiting traffic is detected and approaching traffic is detected each second thereafter for at least one full minute.

At the highest level of abstraction, the traffic controller changes the traffic light when a specified period of time (hereafter referred to as a timer) expires. Therefore, the expiration of a timer is considered a cause that contributes to the effect of changing the light. The current status is another cause that effects the final status of the light. The cause-effect graph of figure 2 illustrates these results.

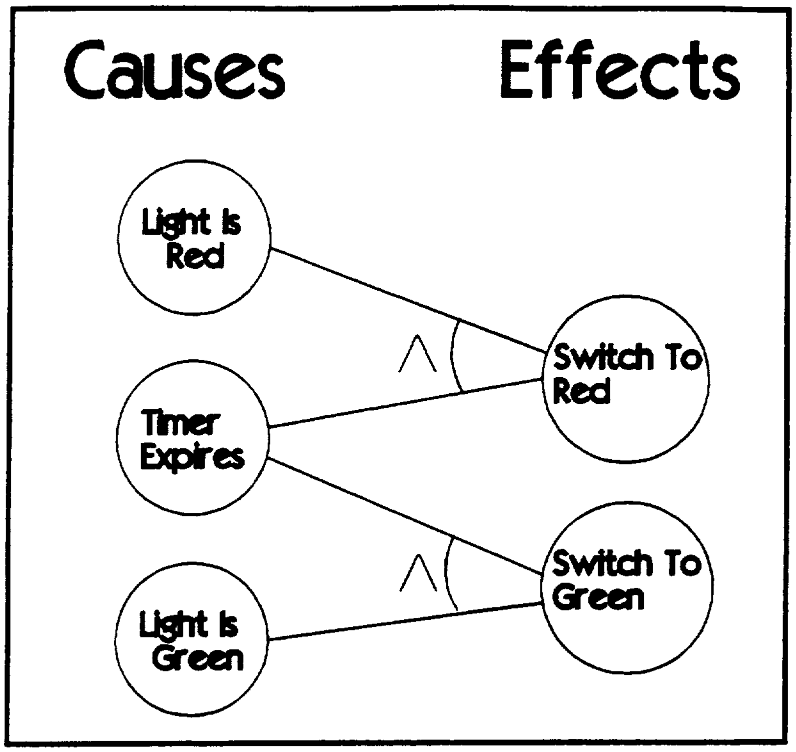


Figure 2

Obviously, from both the problem description and the scenarios, there are other conditions that indirectly impact when the light changes because they impact how the timer works. Therefore, the table of figure 2 can be "refined" to a more descriptive cause-effect graph that captures those hidden details. Figure 3 illustrates this refined cause-effect graph. The following causes are considered:

- Traffic is approaching in the current direction of traffic flow
- An auto is waiting for the light to change
- A pedestrian is waiting for the light to change
- 2 minute times has expired
- 15 seconds has expired
- Traffic has been waiting for 1 minute

The following timer effects result from combinations of the causes defined.

- Wait for 2 minutes
- Wait for 15 seconds
- Wait for 1 minute

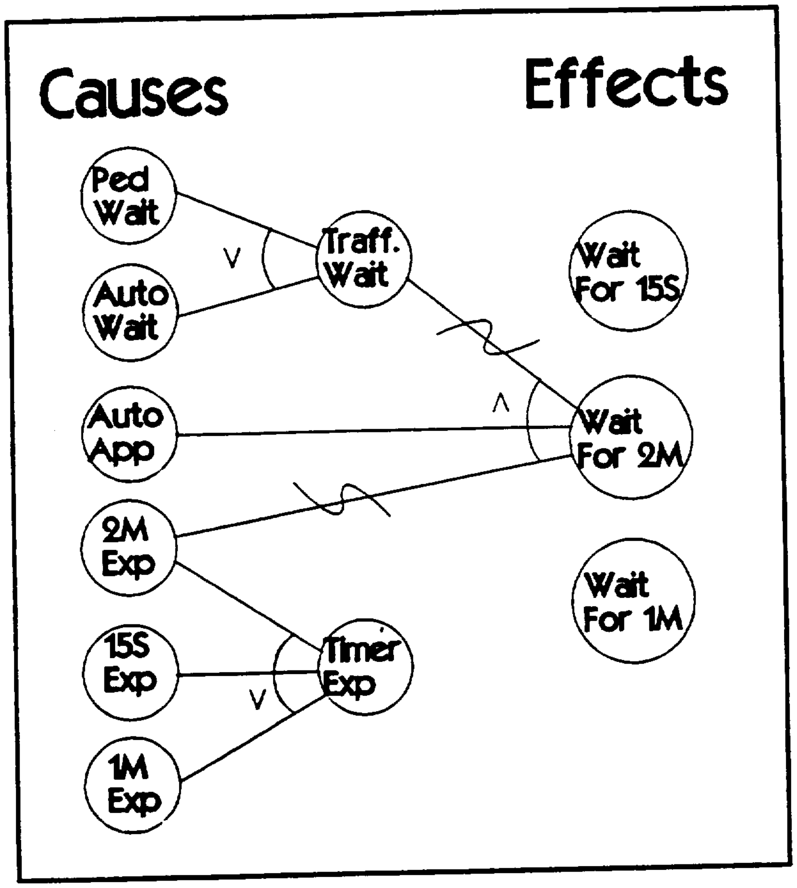


Figure 3

Note that the graph of figure 3 has interim nodes that are neither direct causes or direct effects. These are convenient abstractions that are helpful both in building and analyzing the generated graph. For example, the abstraction *traffic waiting* is used to determine a direct effect of the system (Wait for 2 Minutes). This abstraction captures that either a pedestrian or auto could have caused the system to detect waiting traffic. In terms of the direct result, however, this level of detail is not required. The only thing the system needs to know is that traffic is waiting, not the kind of traffic that is waiting.

The graph of figure 3 is not complete. Completion of the graph will be left as an exercise for the reader.

Worksheet #4: Program Proving (Axiomatic Analysis Using Symbolic Execution)

When To Use This Technique:

Unit/Integration and Static Test

Who Uses This Technique:

Anyone interested in analyzing detailed descriptions of the system (e.g., developers, independent verifiers, etc.). Users would probably not be interested in this technique.

Why Is This Technique Used:

Good for proving correctness of specifications. It readily highlights deficiencies in specifications. It also helpful in applying stepwise refinement.

How To Use This Technique:

Key Terms

Code Fragment	A "piece" of code. Can be as simple as one construct (e.g., If-Then-Else) or as complex as an entire module. Should be surrounded by a pre-condition specification and a post-condition specification.

```
< pre-condition >

... code fragment ...

< post-condition >
```

Mathematical Induction	A proof process that involves demonstrating that if something is true for i^{th} case, then it must be true for the $i+1^{th}$ case. In the case of loops, this means that by showing the first iteration works and that the pre-condition is satisfied at the start of another iteration, additional iterations will also work.
Pre-Condition	A specification that states the properties that must be true for the code fragment that follows it to be correct.
Post-Condition	A specification that states the properties that must be true after execution of the code fragment that precedes it.

Method

1. Define program properties to be proved. To get the most benefit from this technique, insert "pre" and "post" conditions around code fragments.

2. Build a graph of the program flow.

   a. Rather than build a large graph for the entire program, build several smaller ones. Use the "pre" and "post" conditions as boundaries for doing this separation.

   b. When loops are involved, two cases must be considered: no iterations and at least one iteration. Since one can not exhaustively prove all iterations, use the inductive process to demonstrate correctness of looping conditions of the latter case.

3. Assign symbols as values for each variable of interest (variables of interest are those that will "prove" program properties).

4. Trace program execution by substituting symbols for variables. Prove properties as they are encountered in the trace.

   a. Build a matrix with one column for each variable of interest and one row for each line of the program to be traced

Helpful Hints

Formal program proving works best on the most critical parts of the system. Informal proofs are more practical for the less critical parts of the code. Regardless of whether the proof is formal or informal, the goal is the same: prove the correctness of specifications.

When doing formal proving, focus on the "interesting" parts of system. Proofs can become long and unwieldy if everything (e.g., all program variables) is considered. Often, only a small subset of things are of interest. For example, if a program has ten variables and the loop you want to prove uses only three of them, then tailor the proof to analyze only those three.

Example

Consider, for example, the following description.

> A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every two minutes. With a pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Given this simple, consider the procedure shown in figure 1. This procedure when executed determines the appropriate time at which the light should change once waiting traffic has been detected.

```
Procedure Process_Signal
1                 $T_l := T_C + 60;$
2                 $T_S := T_C + 15;$
3                 While $T_C < T_S$ Loop
4                     $< T_C < T_S$ And $T_S <= T_l$ And $T_C < T_l >$
5                     If Approaching_Traffic Then
6                          If $T_C + 15 > T_l$ Then
7                               $T_S := T_l;$
8                          Else    $T_S := T_C + 15;$
9                          End If;
10                    End If;
11                    $< T_S <= T_l$ And $T_C < T_S >$
12                $T_C := T_C + 1;$
13                End Loop;
14                $< T_C = T_S$ And $T_S <= T_l >$
```

Figure 1

Based on the procedure shown in figure 1, the first step of the method is complete. Lines 4, 11, and 14 are the conditions that the symbolic execution will prove. Lines 4 and 14 are the "pre" and "post" conditions, respectively for the loop of line 3. Lines 4 and 11 are the "pre" and "post" conditions, respecitively, for the If-Then-Else at line 5.

Now that system properities have been defined, a graph is built to support tracing of the procedures execution. Rather than building one large graph that documents the entire procedure flow, two smaller graphs are built. Another reason for doing this is because there is a loop. Whenever there is a loop, behavior must be examined for two cases: the loop iterates at least once and the loop does not iterate. Figures 2 illustrates this reasoning by showing two graphs built by "cutting" the execution at line 4.

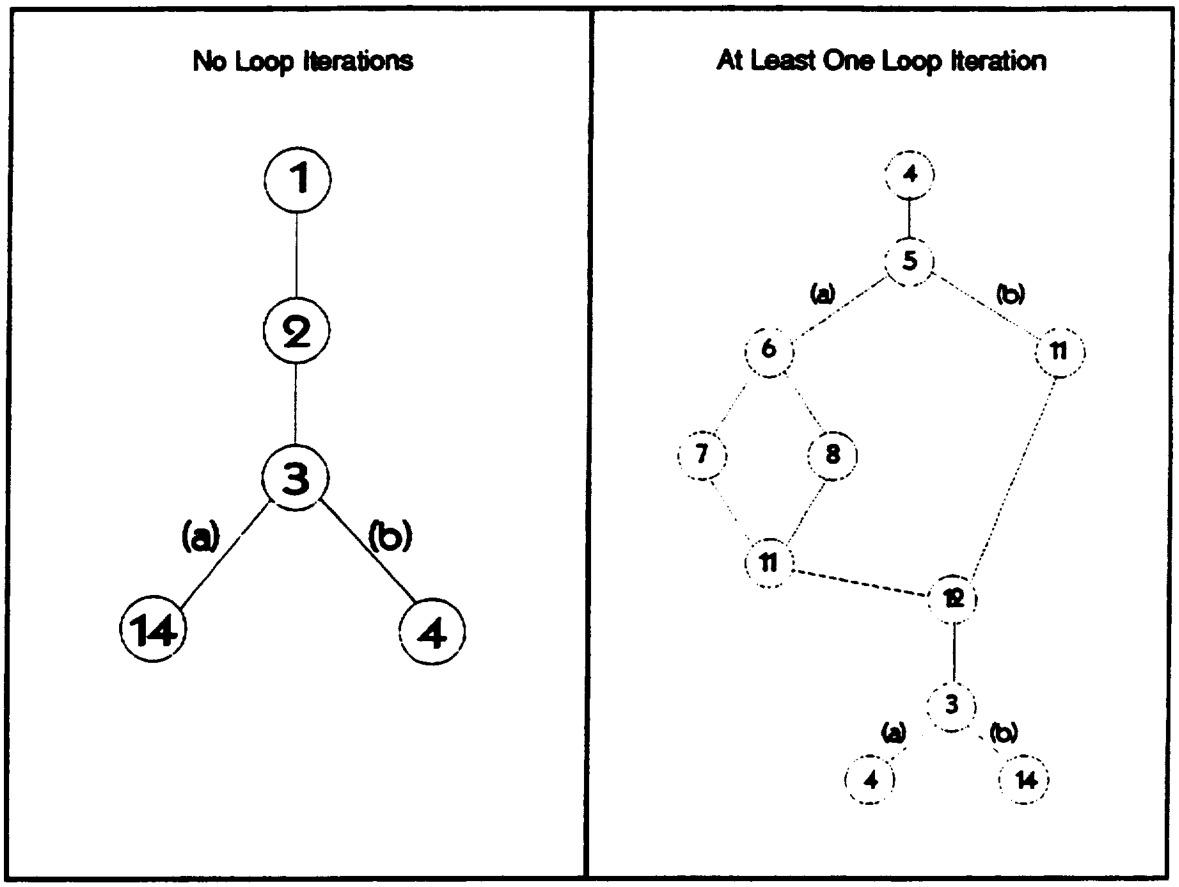


Figure 2

No proof is necessary for the graph showing no loop iterations (however, it would be worthwhile to demonstrate the path through branch (3b) is correct) because lines 1 and 2 show that the loop will always iterate. The more interesting graph covers the case where the loop iterates at least once. For this case a matrix is constructed to document the trace. Symbols are assigned to each variable of interest and traced in a unique column of the matrix. Every time the trace encounters a "pre" or "post" condition, arguements are provided, in terms of the symbols, as to how that property is satisfied. Figure 3 shows a partially complete proof for the graph with at least one loop iteration.

Lines	t_c: β	t_l: ø	t_s: μ	Arguements
1		β+60		
2			β+15	
4				<u>Prove</u>: $t_c < t_l$ and $t_s <= tl$ and $t_c < t_s$ TRUE : substitution of values for t_c and t_s yields the following: β+15 <= β+60 and β < β+60 and β < β+15
5b				Approaching_Signal is FALSE
11				<u>Prove</u>: $t_s <= t_l$ and $t_c < t_s$ TRUE : values for these variables have not changed since line 4 therefore, the same arguements apply
12	β+1			
3b				$t_c >= t_s$
3a				$t_c < t_s$
15				<u>Prove</u>: $t_c = t_s$ and $t_s <= t_l$ TRUE : if, after substituting symbols, $\beta < \mu$ (line 11) and $\beta+1 >= \mu$ (line 3b) then β=μ Also, since μ <= ø at line 11 and the values μ and ø have not changed, then $t_s <= t_l$ still holds
4				<u>Prove:</u> $t_c < t_s$ and $t_c < t_l$ and $t_s <= t_l$ TRUE : $t_c < t_s$ holds from line 3a $t_s <= t_l$ holds since values for t_s and t_l have not changed since line 11 Since $t_s <= t_l$ and $t_c < t_s$ are both true then $t_c < t_l$

Figure 3: Results of tracing the iteration graph from figure 2

Worksheet #5: Hazard and Fault Analysis Using Fault Trees

When To Use This Technique:

Static Test

Who Uses This Technique:

Anyone interested in analyzing the safety correctness of software (e.g., developers, independent verifiers, users). Users are included because they will need to help define the hazards that the software must account for along with what the software's response to those hazards should be.

Why Is This Technique Used:

From a testing perspective, identifying hazards and faults and their relationship to the software helps the process of building test cases for these conditions. From a design perspective, this technique can (1) drive the design of a solution that handles faults and hazards and (2) aid in demonstrating that the software never does anything "unsafe" due to the identified hazards and faults.

How To Use This Technique:

Key Terms

Fault	An error within that occurs within the software itself that could potentially cause a hazard.
Fault Tree	A graph (similar to a Cause-Effect graph) that, in the case of fault driven analysis, graphs from a given fault (or condition) to an end result or, in the case of hazard analysis, graphs from a given end result backwards to its cause (i.e., hazards are roots and faults are leaves in the tree).
Hazard	An undesirable external event that could potentially be caused by the software.

Method

1. Identify hazards and faults.

2. Build a tree.

a. For fault analysis, begin with a specific fault and work from the bottom of the tree to the top to determine what "results".

b. For hazard analysis, assume the result is a hazardous situation and then work down the tree to decide what conditions must happen to cause the hazard.

Helpful Hints

Very similar technique to cause-effect graphing. Since these techniques are similar you might be tempted to create a single cause-effect graph that captures both. However, it is probably better to analyze each kind of correctness separately. Fault trees for safety correctness and cause-effect graphing for functional correctness.

Best when done on a module basis. That could either be the module itself or when analyzing the where the module is used.

Example

Consider, for example, the following description.

> A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every two minutes. With a pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

The first step in analyzing this problem involves listing hazards and faults. Some examples of hazards are:

- There is a collision in the intersection (auto/auto, auto/pedestrian)
- Traffic is stopped in all directions

Some examples of faults are:

- The controller's internal clock fails
- The sensors detecting oncoming traffic fail

Once a complete set of hazards and faults have been defined a fault tree can be built. Figure 1 illustrates a fault tree for analyzing a hazard. Conditions that could contribute to this hazard appear inside boxes and are nodes in the tree. Since multiple conditions may contribute to any particular hazard (or multiple faults may contribute a single hazard), boolean operators (and, or, not) to "connect" these conditions.

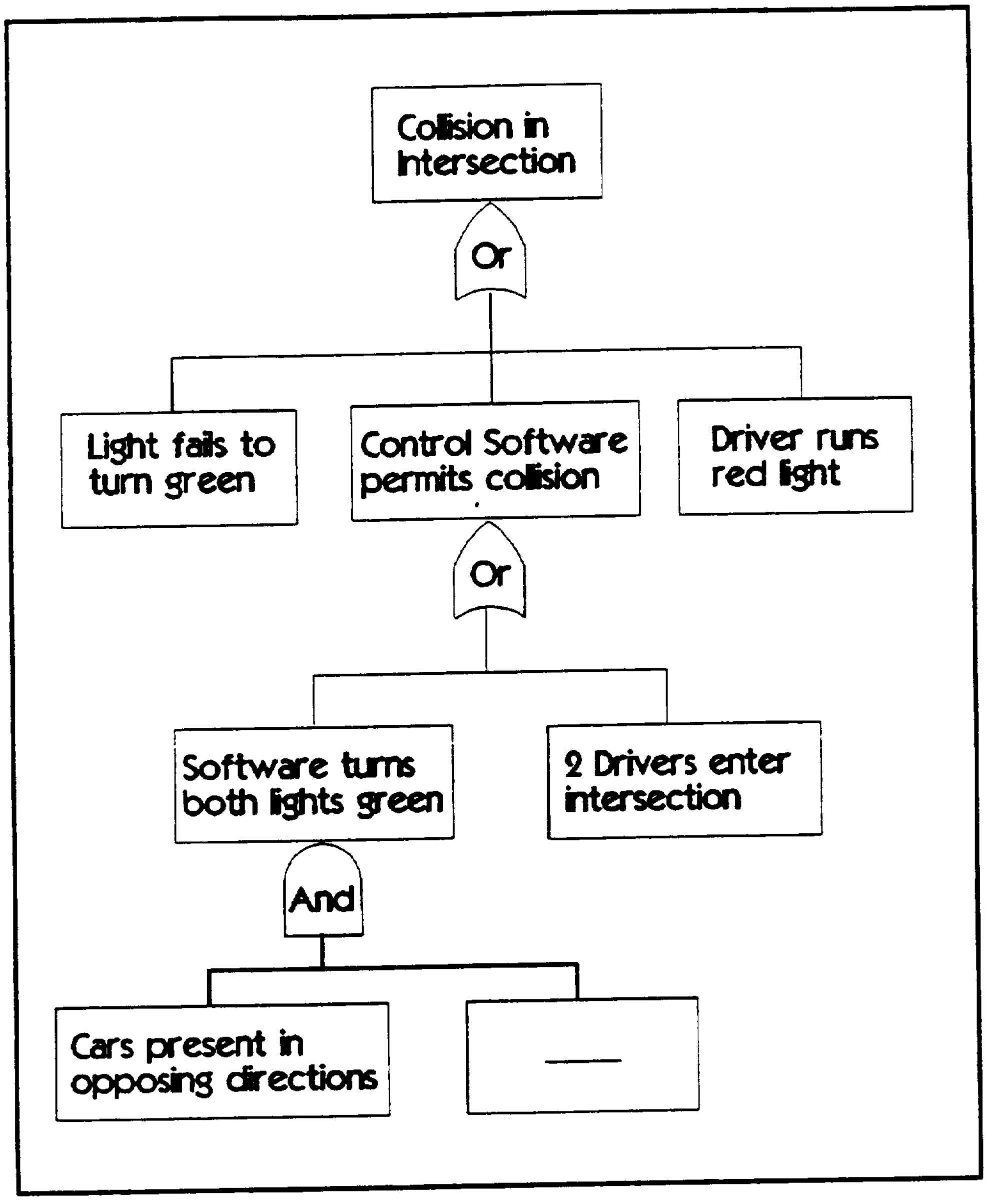

Collision in Intersection
Or
Light fails to turn green
Control Software permits collision
Driver runs red light
Or
Software turns both lights green
2 Drivers enter intersection
And
Cars present in opposing directions
____

Figure 1

Worksheet #6: Inspections

When To Use This Technique:

System Test, Unit/Integration Test, and Static Test

Who Uses This Technique:

The list of who participates in a particular inspection varies depending on where this technique is applied and to what work product it is applied. However, from an overall perspective anyone (user, developer, etc.) associated with the project will participate in an inspection at some point during the process.

Why Is This Technique Used:

Inspections are estimated to catch approximately 60% of all errors. It is THE most effective technique for analyzing a work product for errors.

How To Use This Technique:

Key Terms

Formal Inspection	A formal inspection requires a meeting where all required inspectors must participate and all rules must be followed
Informal Inspection	An inspection that does not necessarily require a meeting. All required inspectors may or may not participate and some rules may be relaxed pending agreement from the moderator
Walkthrough	Synonomous with a formal inspection

Method

1. Define the kinds of inspections and what they will inspect
   a. At a minimum, there should be one inspection per "phase"
2. Determine who should participate in each inspection
3. Assign specific roles to the participants
   a. Focus is on keeping the inspection orderly and efficient

4. Define the "rules" governing the inspection

   a. Focus is on preventing uninspected work products from being delivered to the user.

Helpful Hints

Do not review an entire system at one time. Rather, review incrementally. These increments naturally map to the modules of the work product.

Keep track of all inspection "results" that speak to the rationale behind the final form of the work product. This is helpful should the work product ever (1) need to be re-designed or (2) experience an error related to what was inspected.

The main focus of an inspection is identifying errors. Discussions regarding solutions to those errors should be done outside the inspection.

Example

1. What Should Be Inspected?

There are many kinds of work products that should be inspected. Some examples, are requirements documents, detailed design, code, knowledge documents, test cases, test results, etc.. In general, any product produced during the development process that will either be used outside the originating organization or will be used in another pahse of the development process should be inspected. The term "originating organization" refers the the organization that actually developed the product considered for inspection.

2, 3. Who Should Participate In The Inspection And What Should They Do?

At a minimum, a moderator, developer, backup, requirements analyst, and verifier should participate in an inspection. Hopefully, the organization structure is such that different people fill each of these roles. Each of these inspection pariticpants performs the following role in the inspection:

Moderator	The role of the moderator is to conduct the inspection process in a manner that assures the integrity of the process. To this end, the moderator ensures that the inspection team is prepared, has the right inspection material, and completes all identified actions. Ideally, the moderator should be someone outside both the development and test organizations.
Developer	The role of the developer is to be able to provide rationale for the implementation approach used on the work product.

Backup	The backup is a member of the development team, but is not directly involved in the implementation being inspected. However, the backup does have a general understanding of the work product. The role of the backup is to be another "set of eyes" from a similar perspective as the developer.
Requirements Analyst	The role of the requirements analyst (this could be the role for the expert) is to ensure that the inspected work product complies with stated requirements.
Independent Verifier	Assuming an independent verification team exists, they assume the role of examining the work product for testability.

4. What Are The Rules?

With these roles defined, the next step is to define some rules to govern the inspection process. For example,

- No inspection is complete until the moderator verifies that all issues related to the inspection have been satisfactorily closed.
- Each inspector must complete an error log for the work product being inspected. The inspector will classify issues as either a major error (the work product is incorrect as written), minor error (e.g., standards violation), or suggestion (an alternative approach). Suggestions are optional while major and minor errors must be corrected.
- No work product can be released until all inspections on that work product are complete.
- The work product inspection package must include the work product itself and an overview of what is to be inspected, and any other support material that would aid the inspector.
- Inspectors must be given sufficient lead time (e.g., 4 days) to prepare for the inspection
- The results of the inspection are to be "filed" by a project librarian for future reference.
- Work products may only be reviewed informally when they are small.
- An inspection shall not be longer than 2 hours in length.

Worksheet #7: Testability Analysis

<u>When To Use This Technique:</u>

System Test, Unit/Integration Test, and Static Test

<u>Who Uses This Technique:</u>

Primarily testers and developers.

<u>Why Is This Technique Used:</u>

A significant goal of any development project is to avoid implementations that are hard to test. This technique is helpful in assessing the testability of a particular implementation.

<u>How To Use This Technique:</u>

Key Terms

Execution Analysis	Probability that a given component is executed. A component here refers to a single implementation entity (e.g., a module, a line of code, etc.). Lower probabilities imply lower testability (i.e., harder to test).
Infection Analysis	Probability that a component is sensitive to errors. Lower probabilities imply lower testability (i.e., harder to test).
Mutation Testing	Intentionally seeding a "correct" program with errors. The goal is to identify test cases that can not distinguish between a correct program and one that is not correct.
Propagation Analysis	Probability that once a component is infected, it will affect the execution results (e.g., "what the user sees"). Lower probabilities imply lower testability (i.e., harder to test).

Method

1. Perform execution analysis
   a. Select random samples of test cases
   b. Run those test cases
   c. Generate a ratio between the number of times a component execution versus the number of opportunities.

2. Perform infection analysis
   a. Do mutation testing
   b. Trap the "state" of the output immediately after the mutated component executes
   c. Generate a ratio between the number of "states" that were infected (i.e., wrong) versus the number of opportunities
3. Perform propagation analysis
   a. Set breakpoints following the target component
   b. Intentionally modify the "state" of the at the breakpoint
   c. Generate a ratio between the number of modified "states" that affected execution output versus the number of opportunities

Helpful Hints

Without automated help (e.g., tracing when a component executes, random samples of test cases, setting breakpoints, modifying program variables at the breakpoint, etc.), this technique is very difficult to use. However, it does provide a nice breakdown of categories to consider when static analyzing a component to determine its testability.

Example

Consider the following simple rule base where A and B are initially TRUE and D is initially FALSE.

```
If A and B Then
        assert C

If C and D Then
        print "rule base complete"
        exit
```

The second rule shown is a dead-end rule (since nothing asserts D then the rule's LHS will never be true). Now, consider the testability of the second rule. Since it is a dead-end rule we would expect its testability to be low. This is obviously true since, for all possible test cases executions of this rule-base, the second rule will never fire. Therefore, its execution ratio is 0. Now, consider infection analysis. For this case, let's say a mutant is any unique condition involving C and D. Based on this definition, only one mutant will behave differently than the original program (If C or D Then ..., will fire and produce output). Therefore, infection analysis shows a number near zero for the second rule. Last, consider propagation analysis. This number will also be low for similar reasons to infection analysis. Since the rule never fires, there is no way to perturb the "state" after it fires (or, in other words, no output is ever produced). Therefore, the testability is low. Given these factors, we can conclude the obvious. The second rule is not testable.

Worksheet #8: Mutation Testing

When To Use This Technique:

Unit/Integration Test

Who Uses This Technique:

Testers

Why Is This Technique Used:

This technique aids in the analysis the effectiveness of a given test case. A test case is not very effective if it can not differentiate between a correct program and an incorrect mutant of that same program. In other words, if the output generated for a test case is the same when running both the correct and the mutant program, then that test case is not very effective at finding errors and should not be used.

How To Use This Technique:

Method

1. Generate a suite of test cases for the program being tested.
2. "Seed" the program with one or more errors
3. For each test case from the desired suite of test cases:
    a. Apply test case to the "seeded" program
    b. Apply test case to the un-"seeded" program
    c. Compare results generated
    d. If the results are identical, then the test case is should not be used (i.e., either modify it to make it effective or remove it from the test suite)
4. Steps 1 through 3 can be repeated for many different mutations of the program being tested.

Helpful Hints

Make sure the correct version of the program is kept separate from the mutant programs. This should help avoid accidental delivery of a mutant to the user.

To offset the considerable effort required to build and manage mutant programs, this technique could also be done statically (i.e., no mutant programs are built). In this case, test cases would be analyzed to predict their sensitivity to the correctness of the program.

Example:

Consider, for example, the following description.

> A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every two minutes. With a pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

Some (there are others) possible scenarios that cause the traffic light to change are:

- No approaching traffic for a two minute period should change the light
- Waiting traffic is detected and no approaching traffic is detected for the next 15 seconds
- Waiting traffic is detected and approaching traffic is detected each second thereafter for at least one full minute.

With this in mind, now consider the procedure of figure 1. This procedure performs the necessary actions to determine when the traffic light should change. Using this an example, then, examine the use of the mutation testing technique.

```
Procedure Process_Signal
1           $T_l := T_c + 60;$
2           $T_s := T_c + 15;$
3           While $T_c < T_s$ Loop
4                   $< T_c < T_s$ And $T_s <= T_l$ And $T_c < T_l >$
5                   If Approaching_Traffic Then
6                           If $T_c + 15 > T_l$ Then
7                                   $T_s := T_l;$
8                           Else    $T_s := T_c + 15;$
9                           End If;
10                  End If;
11                  $< T_s <= T_l$ And $T_c < T_s >$
12          $T_c := T_c + 1;$
13          End Loop;
14          $< T_c = T_s$ And $T_s <= T_l >$
```

Figure 1

The first step in the process is to build a suite of test cases. These test cases can be derived directly from the scenarios discussed earlier. The next step is to generate a mutant program. The procedure in figure 2 is an example of a mutant. Mutants are generated by modifying one or more lines in the a program (compare versions of line 6 in figures 1 and 2). The idea is to make subtle modifications so that the mutants are "close" to correct, but incorrect just the same.

```
Procedure Process_Signal
1           $T_l := T_c + 60;$
2           $T_s := T_c + 15;$
3           While $T_c < T_s$ Loop
4                   $< T_c < T_s$ And $T_s <= T_l$ And $T_c < T_l >$
5                   If Approaching_Traffic Then
6                           If $T_c + 15$ /= $T_l$ Then
7                                   $T_s := T_l;$
8                           Else    $T_s := T_c + 15;$
9                           End If;
10                  End If;
11                  $< T_s <= T_l$ And $T_c < T_s >$
12          $T_c := T_c + 1;$
13          End Loop;
14          $< T_c = T_s$ And $T_s <= T_l >$
```

Figure 2

The final step is to execute each test case against both the mutant and the non-mutant programs. If the results are the same, then the test case considered is not effective in finding errors in the mutated area of the code. For example, scenarios that involve no approaching traffic once a waiting signal has been received will not execute line 6 at all. Therefore, output for this test case will be the same for both versions of the procedure. However, a test case that involves repeated detection of oncoming traffic (e.g., oncoming traffic is detected every 2 seconds) once the waiting signal has been received will work differently for both versions. Therefore, that test case should isolate this kind of error. However, not all scenarios involving oncoming traffic after a waiting signal has been received will produce cause different results to be generated for the two versions of the procedure. Can you see why? Therefore, mutation testing can help isolate those that are most effective in exercising the mutated area of code.

Worksheet #9 : Planning for V&V

When To Use This Technique:

Planning starts at the earliest stages of development and continues throughout the development and maintenance of the system. No other development activities should begin until a workable plan is in place. A workable plan is not necessarily a complete plan. It is probably not possible to define a complete plan this early. A workable plan is one that contains enough of the right kind of information to properly guide development.

Who Uses This Technique:

Everyone associated with the development of a system has a part to plan in planning for V&V.

Why Is This Technique Used:

There are many reasons for doing good V&V planning. Failure to do planning often results in expensive and in-effective testing and poor use of resources. These can easily cause your project to fail. Good planning, on the other hand, increases the likelihood of success by focusing on two things: what you need to do the job (resources) and how you will do the job (implementation).

How To Use This Technique:

- What is the problem to be solved?
- Who are the "users"? Do they need to be involved? If so, when?
- Are there existing experts? How can they be used?
- What resources will be needed and when will they be needed?
- How much time/effort will be involved?
- Will prototyping be used? If so, what goal will the prototyping achieve?
- What increments
- What work products will be produced?
- What life-cycle will be followed?
- What implementation approaches should be considered and why?

<u>How To Use This Technique (cont'd):</u>

- What kinds of correctness apply to your system and why? Assign a priority to each kind of correctness
- Identify areas of potential risk
- Identify test techniques

VALUE AND COST EFFECTIVENESS OF V&V

Robert J. Boring
Ewel H. Hughes
Entergy Operations, Inc.
P.O. Box 756
Port Gibson, MS 39150

VALUE AND COST EFFECTIVENESS OF V&V

ABSTRACT

This paper describes a study of V&V costs for a small Software Engineering project at the Grand Gulf Nuclear Power Station by the plant staff. The development applied IEEE standards for software V&V and classical development methods that also complied with IEEE standards. The study examined value returned by the software V&V costs and describes the specific criteria that relate to value in the circumstances of a Nuclear Power Plant. V&V costs are summarized by phase of development, defect removal rate and cost per defect. Extrapolation from the data is made to evaluate alternatives for reduction of V&V methods.

VALUE AND COST EFFECTIVENESS OF V&V

INTRODUCTION

Verification and Validation is used to improve the overall reliability of software and control the integrity of the software delivered and to ensure that the product fulfills the purpose that was originally intended. It is desirable to control the expenses of software V&V and if possible utilize it to control delivery of value with the product. We define value as perceived return against costs of development and maintenance. There exist several constraints on value for systems intended to support Nuclear Power plant operation. First among these is the potential for abandonment of a system that may fill its purchase specifications but not be considered reliable enough for the important tasks that the staff wants the system to provide. Second is abandonment of the system from defects that frustrate the users when they attempt complex tasks using the system. Third, systems can be functional and in place but the much of the work is performed manually due to incomplete support of the actual tasks of the users. Manual work–arounds actually force maintenance of the system and simultaneous staffing to perform the work that the system was to provide. Finally the cost of system maintenance can exceed reasonable resource requirements and cost due to a corrective maintenance effort.

At GGNS we have completed a small project, less than 10,000 lines, and monitored the costs of development, testing and installation. The costs due to V&V are separable and will illustrate this unique situation. The overall behavior and proportion of these costs may be useful to similar situations. The small project represents the worst case for software V&V costs. Any fixed costs will show proportionally higher and variable costs will certainly have little economy of scale. In this project, we estimate the technical risk was high due to the following factors:

- Real–time requirements were present in the system
- 35% of the system was to be done using a real–time kernel for operating system services
- Integration with unfamiliar third party object code was required
- The system was distributed and used network communication as an underlying premise for development
- Work was done at system level or embedded system level exclusively – no applications level work was done

To meet these difficulties, two teams were established. The more experienced team was assigned to the Data Server Subsystem where the real–time requirements dominated. The other team built the Man–Machine Interface Subsystem.

VALUE AND COST EFFECTIVENESS OF V&V

I. Value is a function of what is returned.

The value of software is related to the benefits returned by the use of that software. More efficient work performance, error reduction, better planning or communication are all net positive returns that software can provide.

Constraints can limit the potential return of software. Software can limit the amount of work performed or increase the potential for errors. Maintenance costs for software can be high which reduces the net benefit. Possible consequences of too many defects include high maintenance costs, shortened system life, portions of the system abandoned or worked around, or system rejection at installation.

II. V&V Program

Many items were designed to be configurable items so that the software could be reused to support multiple functions so that the size of system could be as small as possible. These configuration items decrease V&V costs as well as future maintenance costs. Configuration items include expert system rule sets, database, screen displays, and reports.

Standards were adopted to insure conformance and decrease future maintenance costs. Standards used were taken from industry and developed in house. Standards used were:

- C Language Only
- Coding Standard
- Interface Standard
- Network Standard
- CASE Tool Enforced Design Documentation Standard
- Procedural Method for Development

The method used to develop the project was a modified waterfall development cycle. It is proceduralized and follows many IEEE Standards for V&V as well as development tasks. The method includes six phases of development:

- Software Requirements
- Functional Design
- Detailed Design
- Implementation
- Integration
- System Test and Installation

II. V&V Program (Cont'd)

The V&V program used throughout the project was based on IEEE 1012–1986, Standard for Verification and Validation Plans and IEEE 1028–1988 Standard for Software Reviews and Audits. A list of IEEE V&V tasks performed can be found in Appendix A. V&V tasks were combined and grouped to be more efficient. The V&V task groups were:

- Software Requirements Specification (SRS) Review
- Software Functional Design Specification (SDDS) Review
- Detailed Design Review
- Peer Code Review
- Unit Testing
- Integration Testing
- System Testing

III. Cost of V&V

The cost of the V&V program implemented was monitored by tracking the resources utilized in each V&V task. Cost is presented in terms of effort (man–days) and schedule (duration). Fifty–two percent (52%) of project effort was V&V tasks versus 48% non–V&V.

Fifty–seven percent (57%) project duration was V&V tasks versus 43% non–V&V tasks. Seventy–three percent (73%) of our V&V effort was spent in testing. Figure 3–1, Percentage of Project Effort on V&V Tasks, shows the relative effort of each V&V task performed.

Figure 3–2, Percentage of Project Duration for V&V tasks, shows the relative duration of each V&V task.

Figure 3–3, Percentage of Project Duration by phase, illustrates the relative duration of V&V tasks and non–V&V tasks in the project.

Figure 3–4. Percentage of Project Effort by Phase, illustrates the relative effort of V&V and non–V&V tasks in the project.

IV. Costs of Defects Removed

A record of defects removed by each V&V task was kept and is illustrated in Figure 4–1, Total Defects Removed by Phase. Defects that are removed in early phases are not propagated into the next phase where they are potentially more difficult to remove. Figure 4–2, Defects removed by Subsystem, illustrates the defect removal for each team.

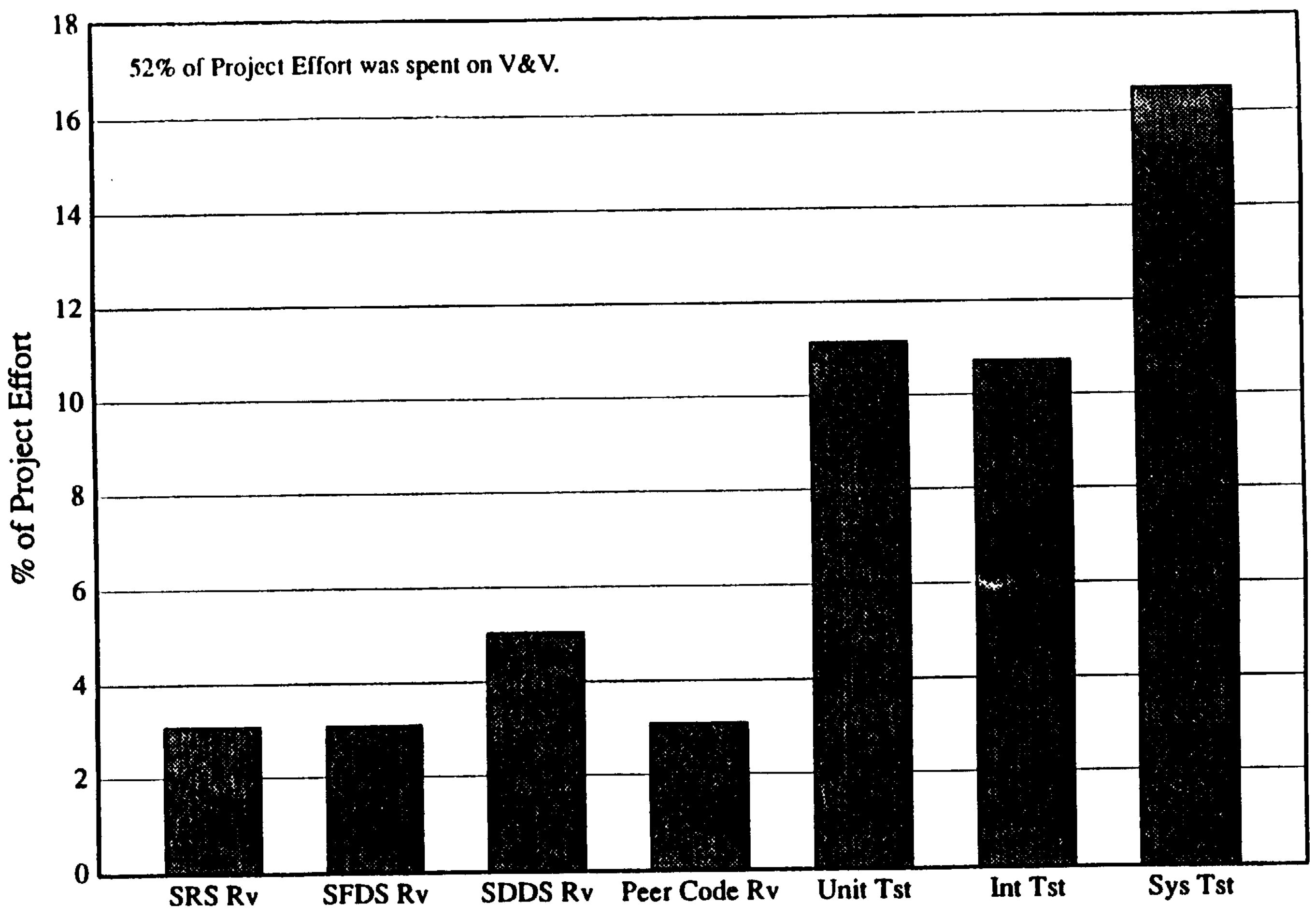


Figure 3–1. Percentage of Project Effort on V&V Tasks

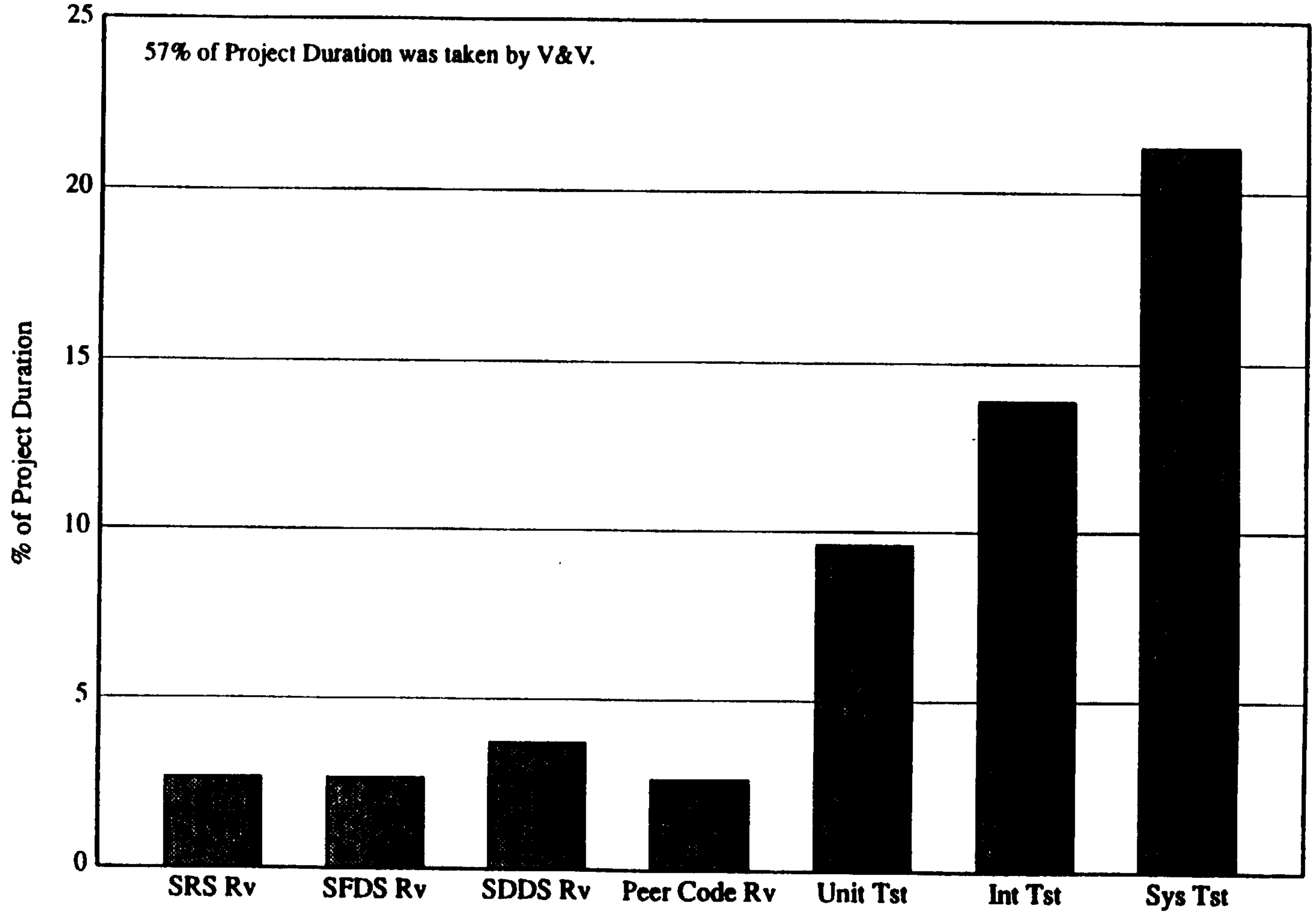


Figure 3–2. Percentage of Project Duration for V&V Tasks

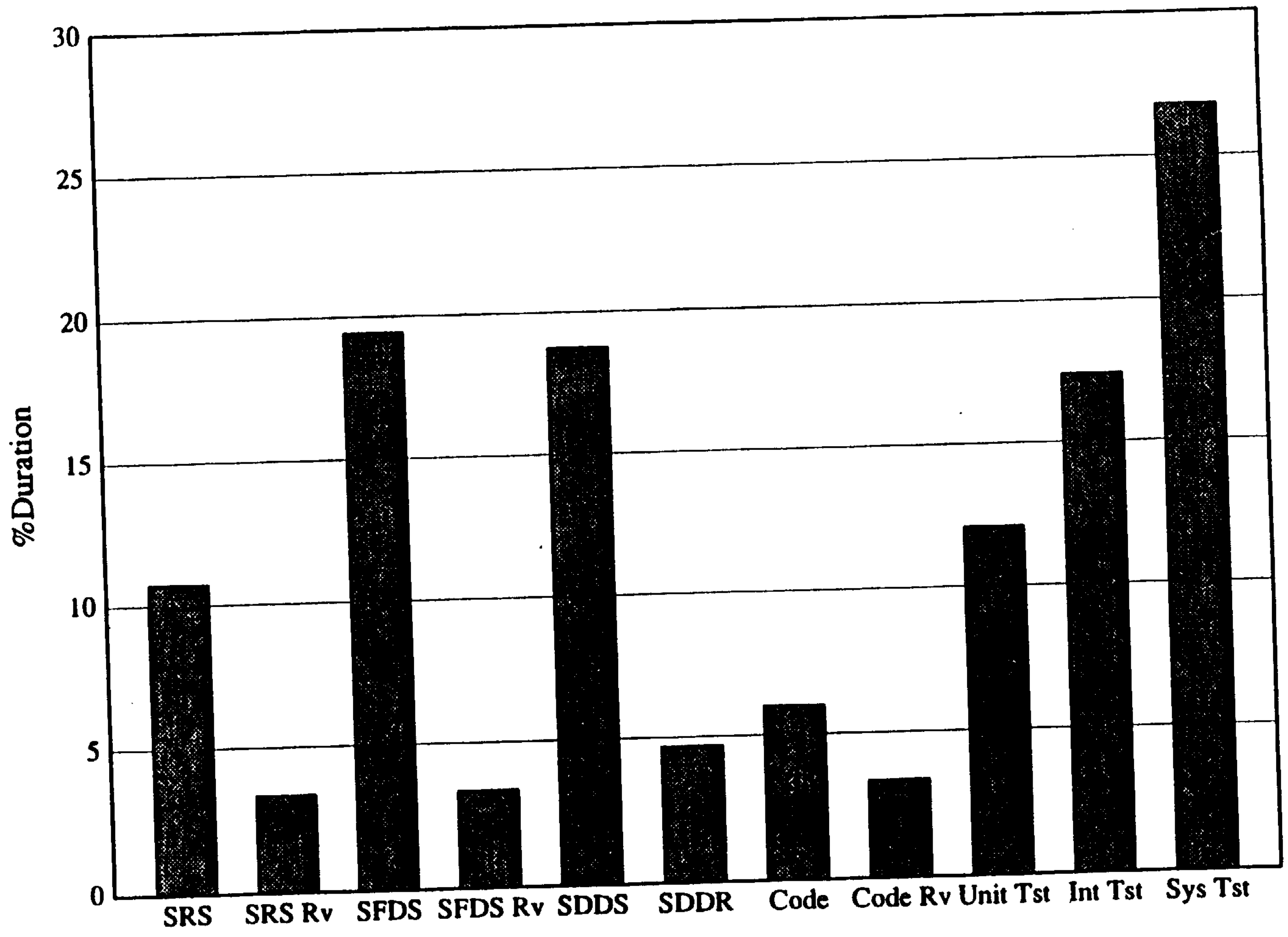


Figure 3–3. Percentage of Project Duration by Phase

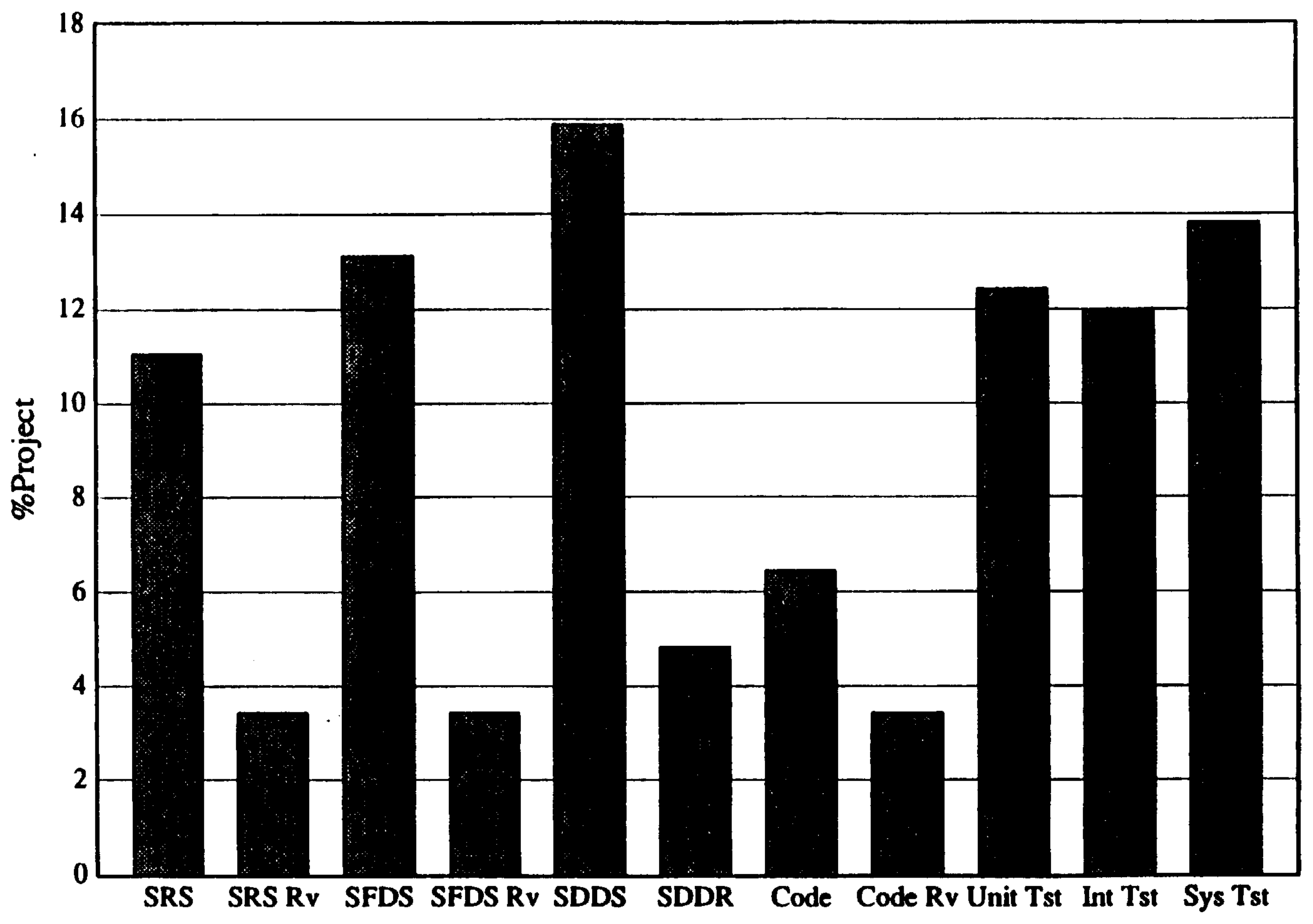


Figure 3–4. Percentage of Project Effort by Phase

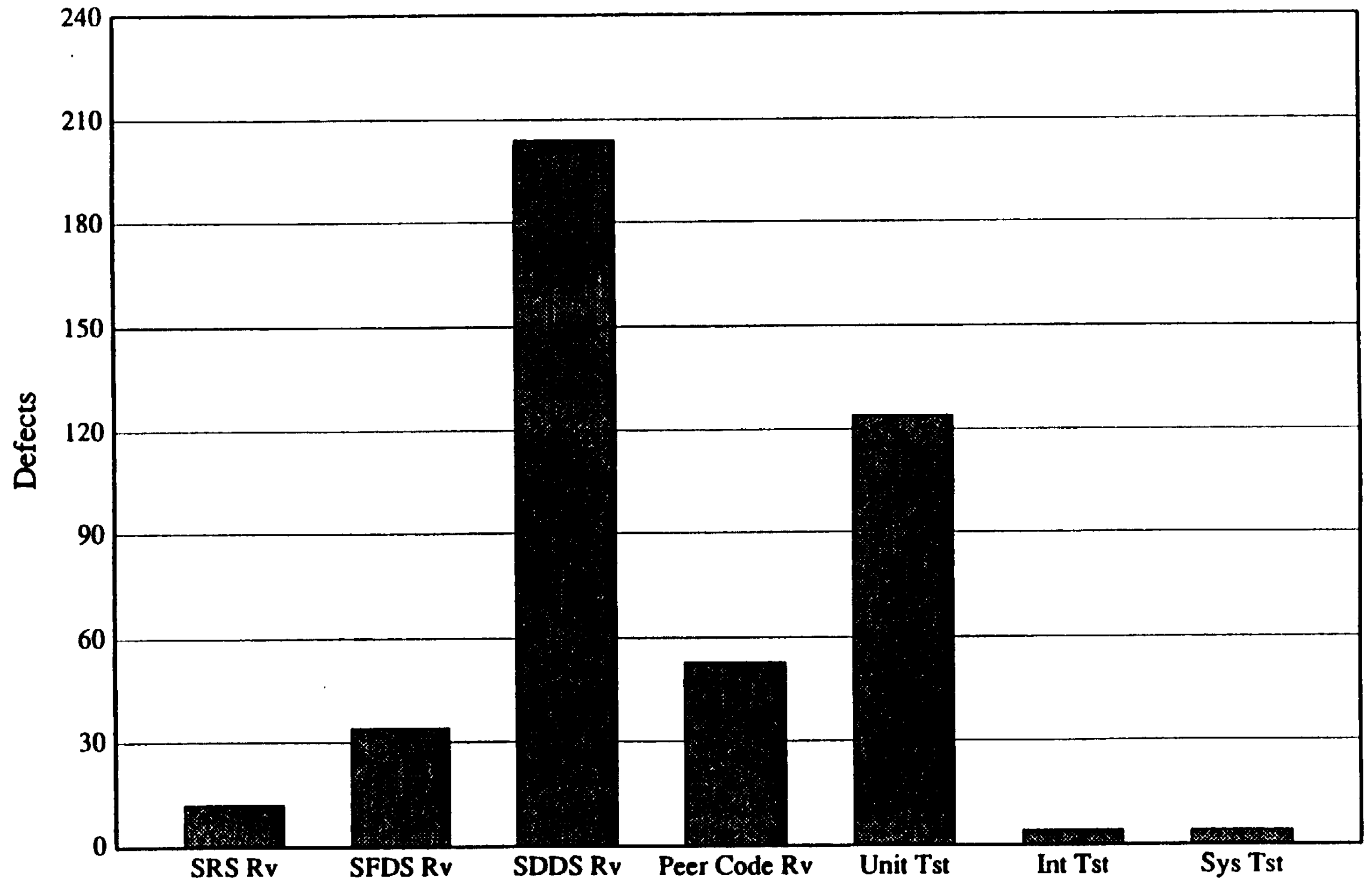


Figure 4–1. Total Defects Removed By Phase

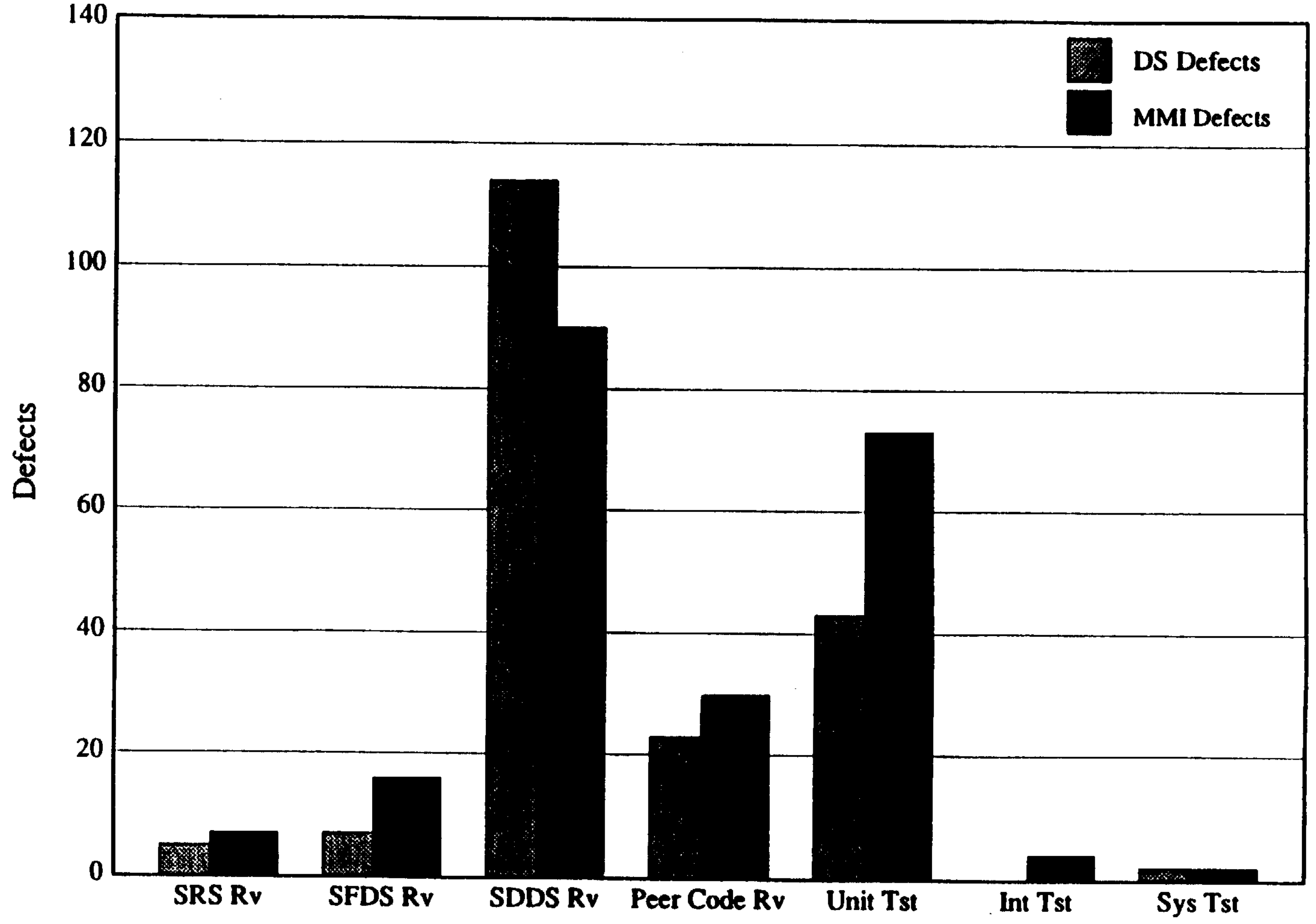


Figure 4–2. Defects Removed by Subsystem

VALUE AND COST EFFECTIVENESS OF V&V (CONT'D)

V. Cost of Defects Not Inserted

The data collected demonstrates that defects are more difficult to remove in latter phases. Figure 5–1, Cost in Man–days to Remove Defect by Phase, illustrates our cost per defect at each phase. The fact that testing is a more costly means of removing defects and the fact that defects not removed in the earlier phases would propagate to the testing phases emphasizes the cost effectiveness of early V&V implementation. This data suggests that delaying defect removal until testing has an opportunity cost of up to two orders of magnitude greater.

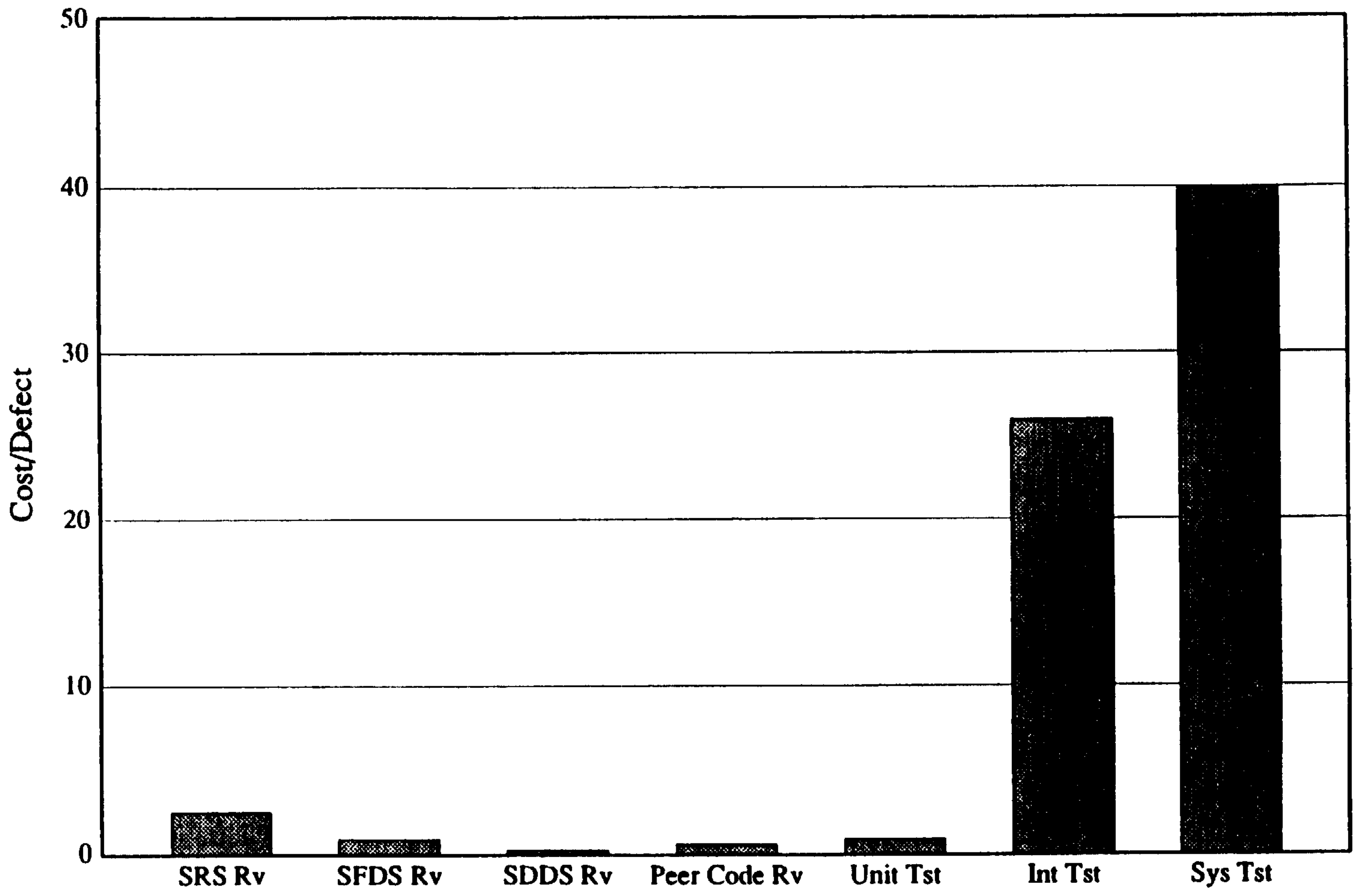


Figure 5–1.Cost in Man Days to Remove Defects by Phase

VALUE AND COST EFFECTIVENESS OF V&V (CONT'D)

VI. Value Returned

The system produced has been installed and in the hands of end users for seven months. Defects and comments have been recorded with favorable results. There have been two defects found, the first defect was isolated to an operating system error from the computer vendor, the other was a defect in third party display generation software detected with a new release of that software and did not reach the field.

No defects have been detected in the software produced in the project. Operator acceptance is high and requests have been made asking for other functions to be incorporated into this system.

We feel that these requests are a vote of confidence from the users and reflect their satisfaction with the new system.

VII. Conclusion

The V&V methods used in project and the metrics kept for the V&V tasks allow us to make a educated estimate that there are less than five undetected errors in the system. The fact that no defects have been detected in seven months of operation seems to indicate that the remaining defects are benign. User perception is that reliability and utility of the system is much higher than previous systems.

In comparing the V&V methods used in the project with this group's past performance, the defect rate prior to the project was greater than 5 defects per thousand lines of code and for the project was less than 0.5 defects per thousand lines of code. The rate of removal of defects prior to the project was estimated at 9 man–days per defect for corrective maintenance. In comparison, the system test defect removal rate with V&V is approximately 40 man–days per defect for system testing. It should be noted that there were approximately 200 known discrepancies in the previous system.

Commercial systems available for comparison exceed the three defects per thousand lines typical of the average software package. [1] Our estimate is that those systems in use at GGNS exceed five defects per thousand lines. At 1.1 million lines of supported software defects would total in excess of 5000. At a defect removal rate of 40 man–days per defect that is greater than 800 man–years of corrective maintenance.

All phases of the development process contribute to defect insertion. Figure 7–1, Insertion Points of Defects Removed in Testing, illustrates the number of defects removed in each phase of testing and where those defects originated, design or coding.

VII. Conclusion (Cont'd)

Testing can be costly way of removing defects. In terms of resources testing is inefficient because of the time required to go back through the steps leading up to a defect detection. In terms of schedule testing is difficult to work in any parallel fashion in the integration and system testing phases. This means that you cannot apply all resources to make things go faster. The result is that testing is less manageable in terms of schedule. Figure 7–2, Merit of V&V Tasks Normalized to System Testing, illustrates the relative efficiency of defect removal for each phase. This data suggests that V&V in the design phase is much more cost effective than testing in defect removal and manageable schedules.

Figure 7–3, Cost of Defects by Phase if not Removed, illustrates the cost in man–days to remove all defects for all previous phases. If all defects had to be removed at system testing, that phase alone would have taken more than eight man–years to remove all defects.

A V&V program reduces the large cost of corrective maintenance by delivering fewer defects to the production system. User acceptance is enhanced by getting their first impressions on a more correct, less defect ridden system. The potential of user rejection or lack of confidence is reduced by providing a more reliable system.

The potential for user rejection of a system can be described in terms of the number of defects and the importance of the system. Figure 7–4, User Rejection Potential, represents the characteristic behavior observed in our experience. The figure illustrates a lower threshold of defects for more important or critical software.

Figure 7–5, User Frustration Potential, illustrates a similar relationship when the function of software is to support a user task. The more complex the task that software supports, the less tolerant the user is of defects.

Recommendation for Cost Effective V&V

- Use V&V and tune the V&V tasks based on perceived complexity and perceived importance of the application.
- Manage using V&V by setting criteria to advance from a phase using review or testing.
- Emphasize early phase V&V tasks to remove persistent defects[2] and minimize exposure to uncontrollable and expensive defect removal by testing.
- Maintain records of results of evolutionary improvements in V&V and the development process. Refine V&V efforts as needs change and the organization matures.

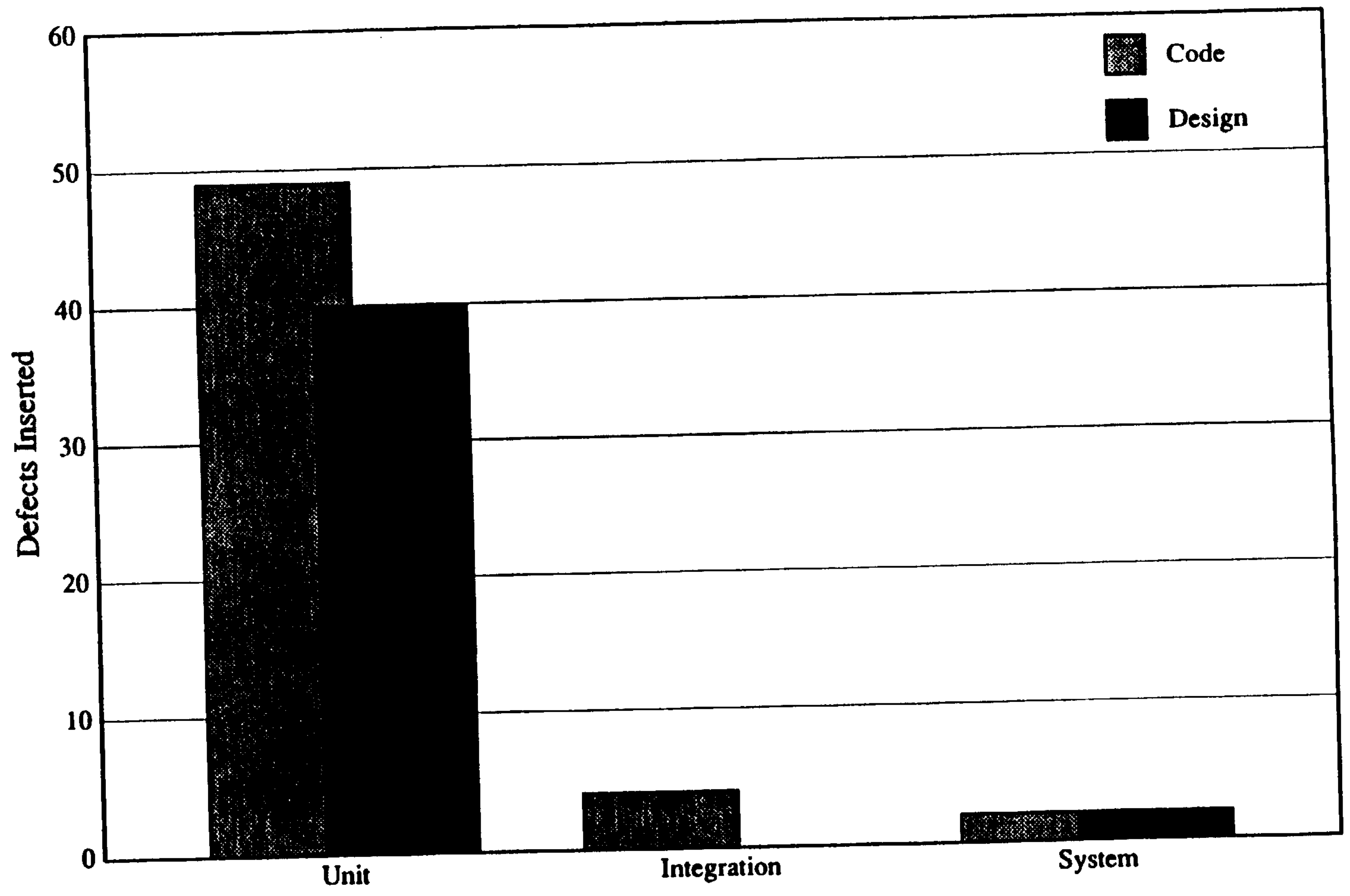


Figure 7–1. Insertion Points of Defects Removed in Testing

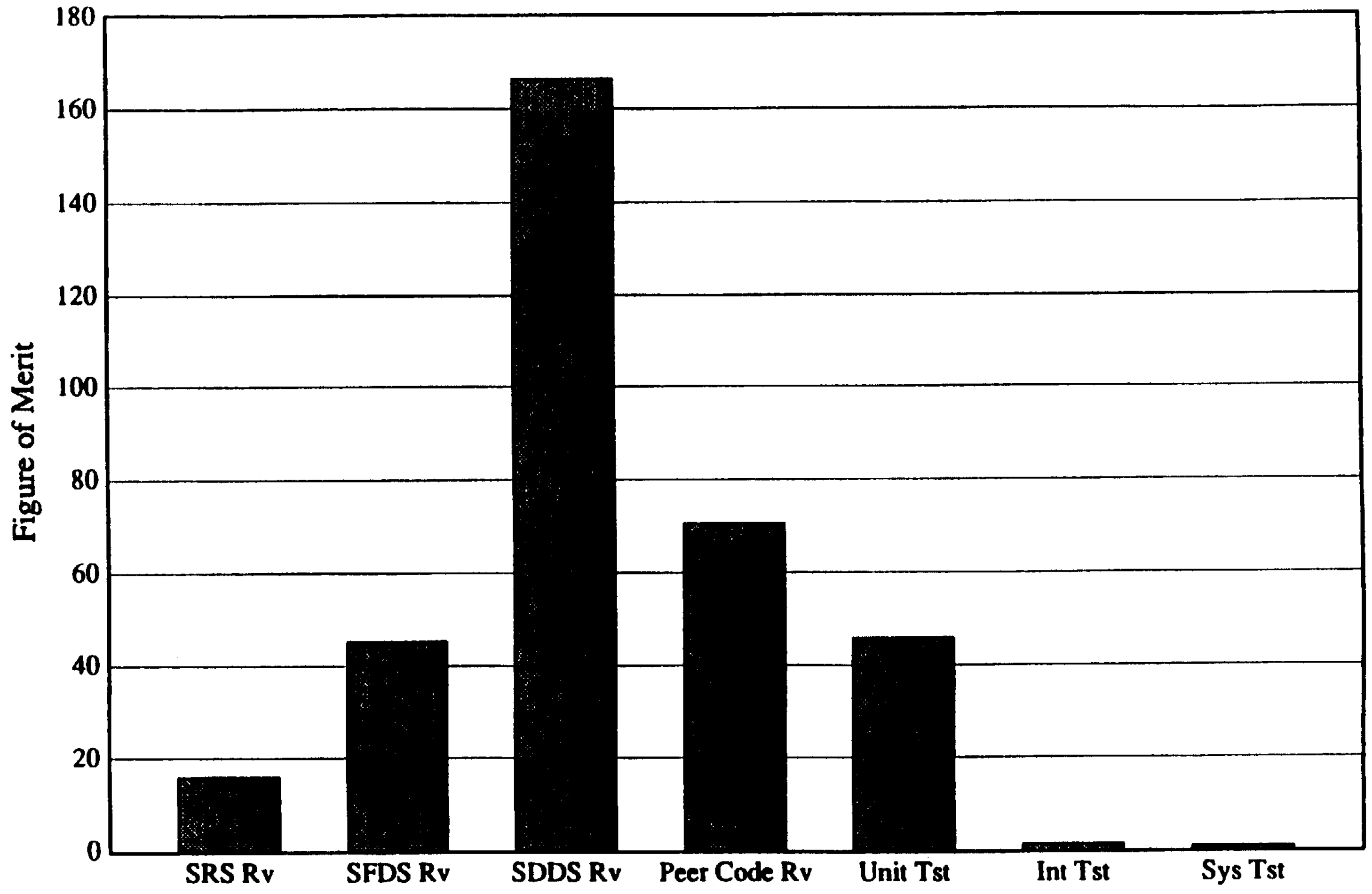


Figure 7–2. Merit of V&V Tasks Normalized to System Testing

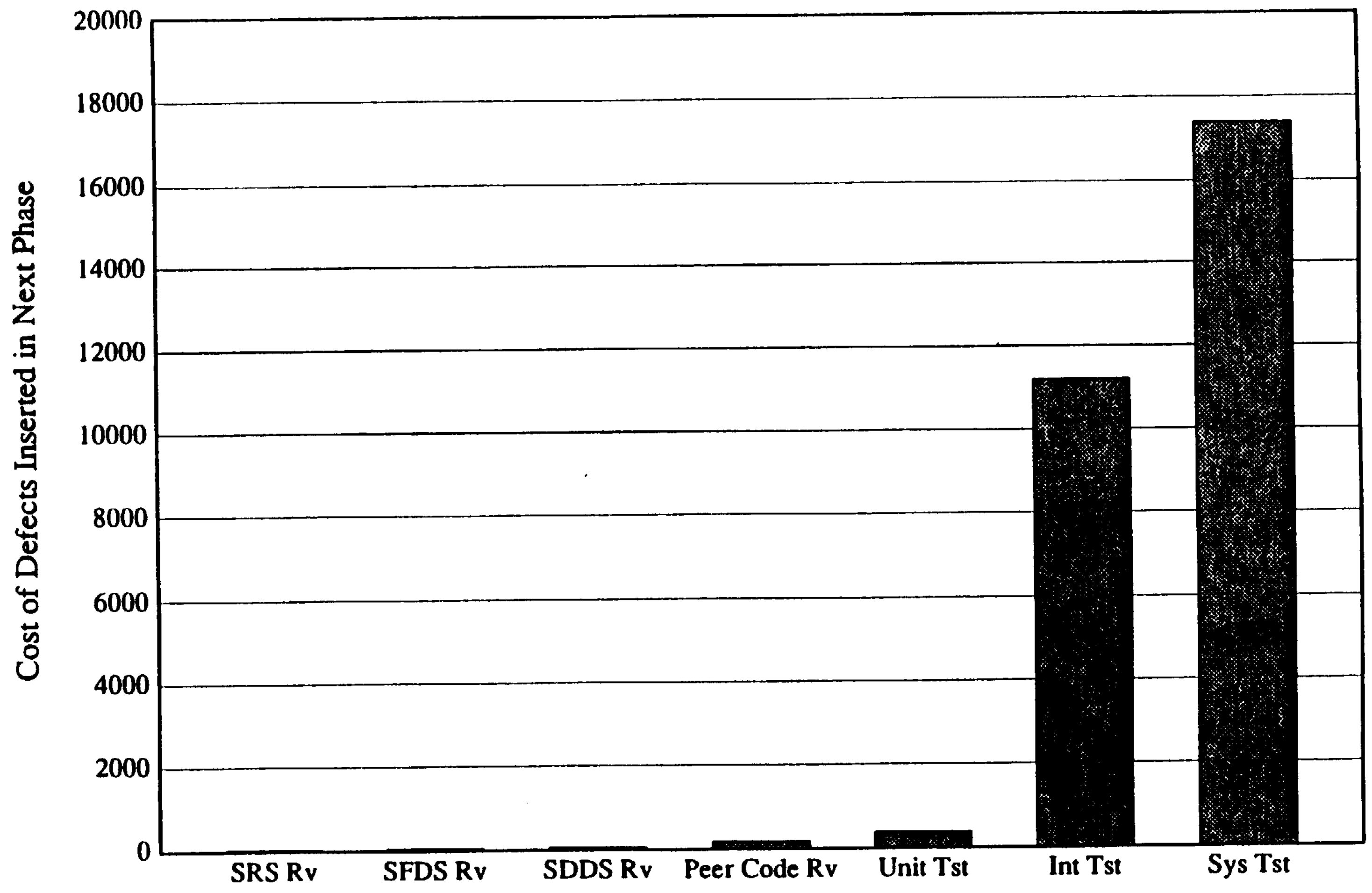


Figure 7–3. Cost of Defects by Phase if Not Removed

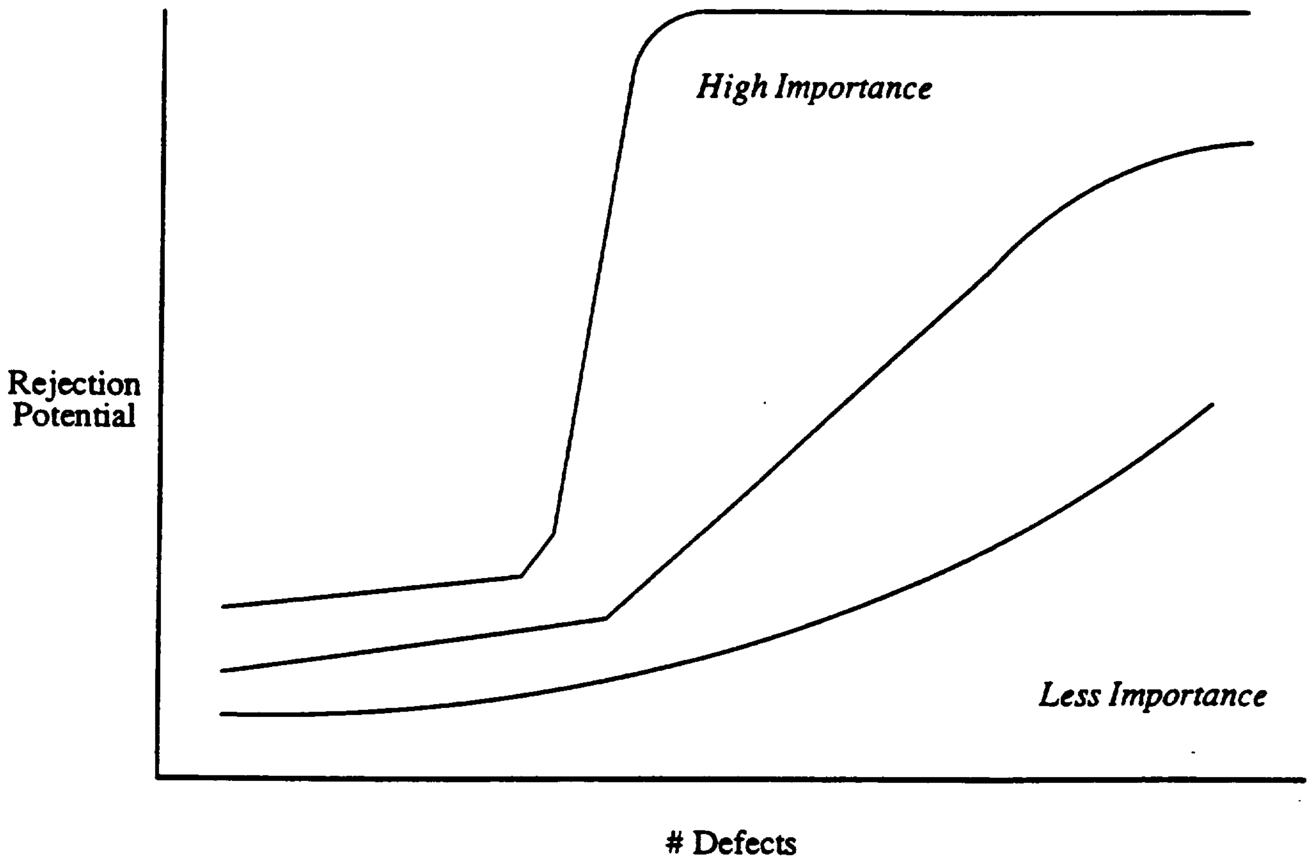


Figure 7–4.User Rejection Potential

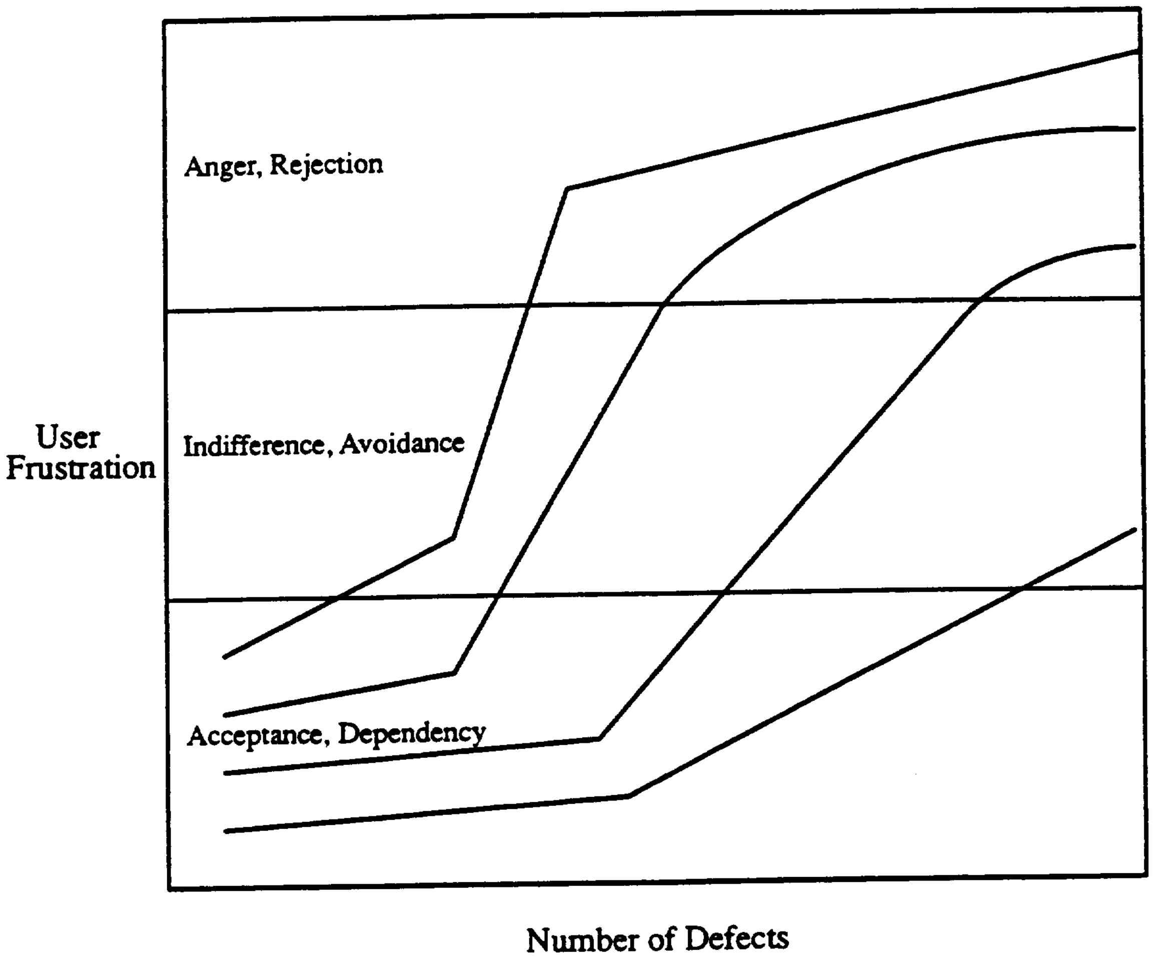


Figure 7–5. User Frustration Potential

VALUE AND COST EFFECTIVENESS OF V&V

REFERENCES

1. Quantitative Aspects of Software Validation, Raymond J. Rubey, Joseph A. Dana and Peter W. Ritche' (IEEE Transactions on Software Engineering, June 1975 pp150–155)
2. Persistent Software Errors, Robert L. Glass (March, 1981 IEEE, Transactions on Software Engineering)

VERIFICATION AND VALIDATION FOR EXPERT SYSTEMS: A PRACTICAL METHODOLOGY

by

James R. Geissman
Abacus Programming Corporation
Van Nuys, California

INTRODUCTION

Expert systems will only be used in critical applications if they are carefully verified and validated according to a product assurance methodology. The methodology should apply through the expert system's entire lifecycle, from conception through design, programming and testing, and should be based on objective verifiable standards. This paper advocates basing the methodology on prototyping for requirement development, design based on formally-defined knowledge processing paradigms, ceritifed inference engines, structured design-for-testing, knowledge base verification, and formal validation testing. A development organization or project can develop a product assurance plan to operationalize the methodology for any specific environment.

ELEMENTS OF V&V

First, some brief definitions. Verification means ensuring that an expert system has been developed in the correct manner and does not contain technical errors. Validation is ensuring that the expert system satisfies its users' needs, or that it solves the right problem.[1]

Verification and validation have clear meaning in the world of non-AI software engineering. Verification is a determination that software has been developed in a "formally correct" manner, in accordance with a specified software engineering methodology. In practice this means demonstrating that each stage in software development is a correct mapping of the requirements established in the previous higher-level stage. To do this, one examines both the process and the outcome. Was a sound methodology was followed, such as Structured Analysis or a design language with a pre-processor? Is the design reasonable and traceable? Are all the elements determined at stage n covered at stage n+1?

Validation means demonstrating that the completed program performs the functions in the requirements specification and is usable for the intended purposes. Just what this entails depends on what the requirements specify and how detailed they are. Very detailed requirements are usually possible for applications like a boiler controller; for an expert system, however, the limits of the possible, the potential, the essential, and an acceptable compromise may be fuzzy and continually evolving. The mere existence of an expert system V&V methodology may help to give order to expert system requirements by encouraging the early formalization of these concepts.

V&V FOR EXPERT SYSTEMS

Up to this time, V&V have not been commonly associated with expert systems for a number of reasons related to the way in which expert system seem to go beyond the assumptions of procedural software engineering. A number of specific problems seem to inhibit V&V for expert systems:

- o If an expert system project starts with with vague objectives, some may conclude that it doesn't matter what the eventual system does, because anything is better than nothing.
- o Green and Keyes[2] cite a "vicious circle," where nobody requires expert system V&V, so nobody does it. Since nobody knows how to do it, nobody requires it.

- Testable requirements are hard to find. Sometimes, attempts to write requirements for a procedural program may already have failed, leading to the expert system project in the first place. Even if there is a definition, it may be as vague as, "Build a machine that will do 'just what Charlie does'"[3] or, "Build a machine that will make most of the hard underwriting decisions as well as an experienced underwriter."

- Expert systems and AI started within the scientific research communicty rather than engineering production. From this came an emphasis on evaluation of systems with a view to determining the current state-of-the-art (which is assumed to be continually advancing). From the research perspective, systems' pluses and minuses are important for what we can learn from them to improve the next systems[7]. The engineering view, by contrast, is more interested in each system for itself, and V&V is performed to determine whether contractural requirements have been met and a system is safe to deploy.

- Common non-procedural architectures for expert systems do not result in code that bears any resemblance to the execution sequence. Hence, techniques for tracing execution flow from an examination of the code do not apply (although they may well apply to the inference engine itself). The declarative knowledge can be examined at face value, but how the system will work can only be predicted with knowledge of how the inference engine operates.

- A modularized, top-down hierarchically decomposed design may be hard to achieve in some expert system architectures. In a backward-chaining rule-based decision-tree system, for example, the progressive decomposition of goals eventually arriving at specific questions that can be resolved in terms of obtainable data is generally an achievable design strategy. In other more fluid architectures, however, side-effects may be frequent and a continual chain-reaction can result so that order is observed only at a micro-level (e.g., a frame-based system with procedural attachments where the procedures are user-written and do not follow the basic paradigm).

- Expert systems (especially those that operate under uncertainty or with incomplete data) may have so many possible states as to make exhaustive testing infeasible.

- Expert system environments or shells tend to have complex user interfaces where inputs are imprecise and hard to reproduce, such as pointing to a place on a picture with a mouse. This magnifies the potential solution space.

SOLUTION: A SIX-STEP APPROACH

The V&V methodology described here has aspects of classical software engineering, especially top-down decomposition. It is based on a four-stage development methodology going from problem definition to initial prototype to expanded prototype (iteratively enhanced) to delivery system. Each stage results in V&V artifacts.

Step One: Develop Initial Prototype Resulting in Testable Requirements

According to the definitions above, V&V of any computer program is not possible unless there are requirements with which the program can be compared. There is some controversy whether requirements can even be established for artificial intelligence programs, and some authors argue that there are fundamental problems with the applications most like human thinking such as natural language processing[4]. However, expert systems differ from some other AI applications in that they are more problem-oriented rather than process-oriented, and allow the specification of the attributes of an acceptable or correct solution. Even, "Do just what Charlie does" is the beginning of requirements; the problem is to refine it into a series of testable statements. This V&V methodology provides a means of doing just that.

PRECEDING PAGE BLANK NOT FILMED

Requirements can be developed from the iterative prototyping methodology, similar to the spiral model of software development. Initially, the problem definition stage results in some 'ely "Charlies." The next stage is to quickly build a prototype of a meaningful subset of the ,blem, followed by a stage of iteratively enhancing the prototype to deal with more and more of the problem. The initial and enhanced prototypes ar› followed by the delivery system, for which the requirements are developed. In this methodology, the prototypes serve as the basis for writing requirements. Even where the prototypes are incomplete or go slightly astray, developing them gives a clear insight into the problem from a knowledge engineering perspective. This insight enables clear and testable statements about the expert system in terms of what it will actually do (even at a detailed internal level).

Step Two: Design in Terms of Formal Paradigms

A crucial step in building a testable system is to ensure that the code-level artifacts are structured in order to be analyzed in a meaningful way. This is part of the design for testing principle.

To take an analogy from procedural software, consider checking a program by examining the code. If the code is single-entry, single-exit, structured and well-documented, a desk-check or walkthru can result in a level of confidence in the program's correctness. With self-modifying "spaghetti" code that shares global data with other processes running in parallel, even a minute code-level examination does not lead to confidence in the program, because there is no way to tell what state the code will be in when it runs.

In expert systems, "structured design" can mean the following:

- Formally define knowledge processing paradigms. These can include inheritance networks, backward- and forward-chaining production systems, and goal-driven logic (e.g., PROLOG), all of which are among the most straightforward of the commonly-used paradigms. Because these paradigms represent discrete event networks rather than the equivalent of "spaghetti," the future state of the system can be expressed as a function of the initial state and a number of transitions, and tools can be built to analyze the logical consistency of a knowledge base expressed in terms of one of the paradigms. Stachowitz has described EVA--the Expert Systems Validation Associate--an automated tool for checking certain paradigms that follows this logic[5].

- Where reasonable, design using the limited set of certified paradigms. This is Procrustean and might be limiting at first, but effort directed to specifying and checking out new paradigms will pay off over time.

- Independently perform V&V on any escapes to procedural code. These may introduce problems, especially if they result in modifications to the problem space (for example, changing working memory in a production system, or modifying the value of an item in an inheritance network).

Step Three: Certify Inference Engines

Certifying an inference engine means testing it to determine whether it in fact carries out one or more of the knowledge processing paradigms specified in step two. If this can be proved, then a certain amount of knowledge base verification can be achieved by code inspection of the knowledge base.

Without a verified inference engine, the only sort of V&V possible is black box testing, or, giving the expert system a set of problems and seeing what it does. Although black box testing is a necessary part of V&V, it is not feasible to test a non-trivial expert system exhaustively in this way.

Certification of inference engines might involve the following steps:

- Formal definition of an inferencing paradigm in a relatively abstract representation (an "ANSI standard").
- Specification of how a given inferencing paradigm is represented in terms of a given expert system tool.
- Development of test suites for the paradigm in an abstract representation.
- Translation of the test suites to the language of the particular tool.
- Performance of certification tests for particular paradigms.
- Checking that all debugging or explanatory information provided by the inference engine (e.g., current agenda, contents of working memory), is in fact correct and can be relied upon in later verification stages. Many expert system shells have very powerful graphic-based explanation facilities, and these can help the developer and tester immeasurably. It is necessary, however, to confirm that they do what they seem to do.

Inductive knowledge acquisition tools are adjuncts to inference engines, and may be verified similarly. Where a well-understood technique is followed, the tool can be checked against standard benchmark cases. For example, Quinlan's ID3 algorithm--or something similar--seems to be incorporated in several commercial products that derive decision trees from case history matrices of criterion data and resultant classifications.

A potential source of problems is where standard techniques are "improved" by new methods that remain proprietary and are not disclosed for commercial reasons. For example, one inductive knowledge acquisition tool seems to be following the ID3 algorithm, but the documentation suggests the tool is somehow better than standard techniques yet nowhere states how it works. The documentation associated with several expert system shells and inductive knowledge acquisition tools uses semi-mystical terms that leave a skeptical reader in some doubt, and is not augmented by a technical appendix that spells out details.

Step Four: Design for Verification

Initial design centers around a high-level statement of what knowledge will be used, where it will come from, and what the knowledge will look like (how it will be represented). Verification at this level consists of determining that the design covers each of the requirements. If the requirements and prototype are not sufficiently advanced to permit the development of a high-level design, this is a sign that they require more work.

The principal elements of an expert system high-level design are statements of the following:

- How each individual requirement will be dealt with.
- The knowledge processing paradigm(s) to be followed. These should be selected from those supported by certified inference engines.
- The principal factual knowledge that the processing relies on.
- The way in which the overall solution is broken into subproblems, the transitions from one subproblem to the next, communication between subproblems and how this structure is represented (e.g., the tree of goals in a backward-chaining system; blackboarding protocols).
- All interactions with the outside world, including how needed data will be obtained (e.g., asking the user, reading devices or querying a database system).

- o What basic assumptions underly the solution and what are the limits of validity or boundary conditions. This information corresponds to the assertions of invariants that help prove the correctness of procedural programs. Examples are the range of conditions or inputs under which the analytical assumptions are valid.

- o The way the current state of the problem or state of the world within the expert system will be represented, in general terms.

Step Five: Verify Knowledge Base

After the high-level design is completed, development (and verification) may go to a lower-level design with more detail, or it may go directly to the code. Because expert system languages and shells are more expressive than most procedural languages, it is often not necessary to do a detailed design step, and the next V&V activity is to verify the knowledge base. Verification should be done statically, with the code alone, as well as dynamically by observing the behavior of the system.

Operationally, one goes over the knowledge base to see if it matches the high-level design, and checks each individual rule/fact/object/procedural attachment/goal for correctness. This checking should be done by persons other than the developers. (See the article by Marcot for some suggestions.) The following are some specific things to check:

- o Confirm that the knowledge base conforms to one of the certified paradigms. (This is most easily accomplished by writing the knowledge base in the abstract paradigm definition form in the first place and mechanically translating it to the tool's input form.)

- o Verify subproblem structure and verify each subproblem independently. This means confirming that a solution to the subproblems is a solution to the larger problem and that the boundaries of the subproblems subsume the whole problem space.

- o Confirm that the knowledge is correct or at least reasonable in a static and individual sense. This means confirming the "facts", the relationships expressed in rules or other ways, the limits or boundary conditions and the overriding heuristics or meta-rules that guide the system's operation.

- o Identify the portions of the knowledge base that are not elements of the paradigm, such as escapes to C or calls to a database system. These can be individually verified in the normal software engineering manner.

- o Embed additional demons to signal failure of the boundary conditions (rules that fire when the system gets places it should never be).

The steps up to this point should be repeated whenever a new level of functionality is introduced, for example when a prototype is expanded to deal with a new problem or when the initial prototype is thrown away and a replaced with a new knowledge representation scheme, according to the spiral model.

Step Six: Perform Formal Validation

Even if each step along the way has been checked out, it is necessary to test the behavior of the integrated system, both to discover errors that only appear at this point, and to check against the user's "real needs."

The steps in validation, borrowing from procedural V&V, are as follows:

- o Determine validation criteria. This seems like an obvious first step, but it is frequently overlooked. Marcot proposes the following criteria: accuracy, adaptability, adequacy, appeal, availability, breadth, depth, face validity, generality, precision, realism, reliability, resolution, robustness, sensitivity, technical and operational validity, Turing test, usefulness, [plain] validity, and wholeness[6].

O'Keefe, Balci and Smith[1] suggest more formal statistical testing, which is especially appropriate for a classification system where cases with known properties can be provided and the outcome can be scored as right, wrong or somewhere between. For a system that performs a different kind of function, such as R1 that configures VAXes or EXCABL that cables Space Shuttle payloads, a number of possible solutions might be right or "good enough," which complicates scoring. Gaschnig, et. al. suggest a number of other evaluation standards[7].

One interesting criterion is that an expert system should "act like an expert" and demonstrate deep knowledge, rather than the shallow recipe knowledge generally associated with them. See [8] and [9] for a fuller discussion of what it means to be an "expert."

For any specific system, the criteria will be derived from the intersection of the criteria mentioned above with the system's particular requirements, as described in the requirements document and/or overall concept document.

- o Determine objective metrics for the selected criteria. This is a difficult step, and some appealing criteria may have to be dropped because of difficulty in coming up with a meaningful objective measure or surrogate.

- o Specify the realms or sets of input data that the expert system must correctly handle.

- o Develop a library of test cases and scenarios specific to the problem and perform regression testing when the knowledge base is modified. As with any software testing, testers should attempt to demonstrate the system's proper response to normal situations, and also attempt to induce system errors. (A good test is one that finds an error.) The test cases should include a mixture of obvious ones, more subtle yet still "average" cases, boundary conditions, meaningless combinations of valid and invalid data, load testing, and obvious error conditions outside the system's scope of validity. Specific cases are derived from the requirements, the design, and known quirks of the implementation environment. The results of these tests should be evaluated according to the criteria and metrics described above.

- o Develop test harnesses and drivers to administer the tests automatically. If possible, have the results, in terms of the metrics, automatically registered in a database that is part of the development system, where they will be associated with the specific software changes they correspond to. As part of the tests, have system performance re-validated by the domain expert(s) involved in the development.

- o Have system performance validated by an independent panel of experts not connected with the system development effort.

- o Use the expert system in parallel with existing systems and non-automated methods for a period of time and compare results.

- o Start the validation testing early, even when it is clear that only a subset of the functions have been implemented, and continue to perform regression testing as the system is elaborated.

- o Maintain detailed information on system performance as the knowledge base is elaborated, because interactions in working memory can cause substantial degradations in performance with seemingly insignificant increases in complexity. If performance is severely affected, a high-level redesign such as a revised subproblem structure to limit the focus at any one time may be called for. Where the requirements demand, perform load testing with realistic rates of inputs.

From Here to There: Work Yet To Be Done

The methodology argued here is not yet ready to use; some research and specification has to be done, including the following:

- o Formal definition of paradigms. This has already been done at least in terms of specifications for developing inference engines for such paradigms as forward- and backward-chaining, PROLOG, frames and inheritance networks. Other more versatile concepts like objects need to be defined in a way that links them to these. An community-wide body would be appropriate for this task.

- o Certification of inference engines. This is like certifying a compiler. Inference engines that follow the most straightforward paradigms (e.g., backward-chaining such as many or basic forward-chaining production systems such as OPS5, OPS83 or CLIPS) are the most likely candidates. Potentially, multi-paradigm tools like NEXPERT or ART could be incrementally certified for the different paradigms.

  A systematic specification of paradigms would resolve some issues influencing system performance that are currently rather muddy, especially among PC-based shells. For example, consider how undefined variables are treated in searching; Consider a backward-chaining shell that is programmed to perform a classification or interpretation problem (such as, evaluating household characteristics to decide whether to grant an insurance policy). Some commonly used shells that could do this include Insight 2+ (PRL) and Personal Consultant. Different shells are likely to perform differently in the face of unknown data: some systems ask the user for the value, whereas others may avoid branches including unknowns and search other regions of the decision space, if possible. Among those that query the user, there may be differences in the order in which facts are collected. These matters are certainly not spelled out in the documentation of most systems the author is familiar with, but can influence results.

CONCLUSION

Formal V&V is necessary for acceptance of expert systems into critical areas. V&V is a straightforward activity that parallels many of the steps undertaken in development and fits especially easily into an iterative prototyping development methodology. The recommended approach to expert system V&V centers on well-defined paradigms and certified inference engines, which permit both static and dynamic verification to be undertaken with confidence.

REFERENCES

[1] O'Keefe, R.M., O. Balci and E.P. Smith, "Validating Expert System Performance," IEEE Expert, Winter 1987.

[2] Green, C., and M. Keyes, "Verification and Validation of Expert Systems," Workshop on Knowledge Based System Verification, NASA/Ames, April, 1987.

[3] Culbert, C., G. Riley and R.T. Savely, "Approaches to the Verification of Rule-Based Expert Systems," SOAR Conference, NASA/JSC, August 1987.

[4] Partridge, D., and Y. Wilks, "Does AI Have A Methodology Different from Software Engineering?" Computing Research Lab, New Mexico State University, 1985.

[5] Stachowitz, R., et. al., "Building Validation Tools for Knowledge-Based Systems," SOAR Conference, NASA/JSC, August 1987.

[6] Marcot, B., "Testing Your Knowledge Base," AI Expert, July, 1987.

[7] Gaschnig, J., P. Klahr, H. Pople, E. Shortliffe, and A. Terry, "Evaluation of Expert Systems: Issues and Case Studies," in F. Hayes-Roth, D. Waterman and D. Lenat, eds., Building Expert Systems, Reading, Mass: Addison-Wesley, 1983.

[8] Berger, P., and T. Luckmann, The Social Construction of Reality, Garden City, NY: Doubleday, 1967.

[9] Swartout, W.R., and S.W. Smoliar, "On Making Expert Systems More Like Experts," Expert Systems, August, 1987.

VERIFICATION ISSUES FOR RULE-BASED EXPERT SYSTEMS

Chris Culbert, Gary Riley, Robert T. Savely
Artificial Intelligence Section - FM72
NASA/Johnson Space Center
Houston, TX 77058

ABSTRACT

Expert systems are a highly useful spinoff of the artificial intelligence research efforts. One major stumbling block to extended use of expert systems is the lack of well-defined verification and validation (V&V) methodologies. Since expert systems are computer programs, the definitions of "verification" and "validation" from conventional software are applicable. The primary difficulty with expert systems is the use of development methodologies which don't support effective V&V. If proper techniques are used to document requirements, V&V of rule-based expert systems is possible, and may be easier than with conventional code. For NASA applications, the flight technique panels used in previous programs should provide an excellent way of verifying the rules used in expert systems. There are, however, some inherent differences in expert systems that will affect V&V considerations.

INTRODUCTION

Expert systems represent one important by-product of Artificial Intelligence research efforts. They have been under development for many years and have reached commercial viability in the last three to four years. However, despite their apparent utility and the growing number of applications being developed, not all expert systems reach the point of operational use. One reason for this is the lack of well understood techniques for V&V of expert systems.

Developers of computer software for use in mission or safety critical applications have always relied upon extensive V&V to ensure that safety and/or mission goals were not compromised by software problems. Expert system applications are computer programs and the same definitions for V&V apply to expert systems. Consequently, expert systems require the same assurance of correctness as conventional software.

Despite the clear need for V&V, considerable confusion exists over how to accomplish V&V of an expert system. There are even those who question whether or not it can be done. This confusion must be resolved if expert systems are to succeed. As with conventional software, the key to effective V&V is through the proper use of a development methodology which both supports and encourages the development of verifiable software.

THE COMMON EXPERT SYSTEM DEVELOPMENT METHODOLOGY

Most existing expert systems are based upon relatively new software techniques which were developed to describe human heuristics and to provide a better model of complex systems. In expert system terminology, these techniques are called knowledge representation. Although numerous knowledge representation techniques are currently in use (rules, objects, frames, etc) they all share some common characteristics. One shared characteristic is the ability to provide a very high level of abstraction. Another is the explicit separation of the knowledge which describes how to solve problems from the data which describes the current state of the world.

Each of the available representations have strengths and weaknesses. With the current state-of-the-art, it is not always obvious which representation is most appropriate for solving a problem. Therefore, most expert system development is commonly done by rapid prototyping. The primary purpose of the initial prototype is to demonstrate the feasibility of a particular knowledge representation. It is not unusual for entire prototypes to be discarded if the representation doesn't provide the proper reasoning flexibility.

Another common characteristic of expert system development is that relatively few requirements are initially specified. Typically, a rather vague, very general requirement is suggested, e.g., "We want a program to do just what Charlie does". Development of the expert system starts with an interview during which the knowledge engineer tries to discover both what it is that Charlie does and how he does it. Often there are no requirements written down except the initial goal of "doing what Charlie does". All the remaining system requirements are formulated by the knowledge engineer during development. Sometimes, the eventual users of the system are neither consulted nor even specified until late in the development phase. As with conventional code, failure to consult the intended users early in the development phase results in significant additional costs later in the program.

So where does all this lead? The knowledge engineer is developing one or more prototypes which attempt to demonstrate the knowledge engineer's understanding of Charlie's expertise. However, solid requirements written down in a clear, understandable, *easy to test* manner generally don't exist. This is why most expert systems are difficult to verify and validate; not because they are implicitly different from other computer applications, but because they are commonly developed in a manner which makes them very difficult or impossible to test.

NEW APPROACHES TO DEVELOPMENT METHODOLOGIES

From the preceding section, it should be clear that the problem is the use of development methodologies which generally do not generate requirements which can be tested. Therefore, the obvious solution is to use a methodology which will produce written requirements which can be referred to throughout development to verify correctness of approach and which can be tested at the end of development to validate the final program.

Unfortunately, it's not that simple. Some expert systems can probably be developed by using conventional software engineering techniques to create software requirements and design specifications at the beginning of the design phase [1]. However, the type of knowledge used in other expert systems doesn't lend itself to this approach. It is best obtained through iterative refinement of a prototype which allows the expert to spot errors in the expert system reasoning before he can clearly specify the correct rules.

The goal of any software development methodology is to produce reliable code that is both maintainable and verifiable. A software development methodology for expert systems must serve a similar purpose as one for conventional software. However, there are some differences between expert systems and conventional software which will affect the development methodology. Development methodologies for expert systems are discussed in more detail in another paper by the authors [2]. Suffice to say here that some kind of development methodology must be chosen and applied to support effective V&V.

MAKING THE REQUIREMENTS WORK

Once we accept that requirements and specifications must be written and a methodology for how and when to write them has been adopted, the actual work of verifying and validating the program must be done. A very appropriate technique would be a direct derivative of the methods used to develop procedures, flight rules, and flight software for the Apollo and Shuttle programs. This technique consists of Flight Technique Panels which regularly review both the procedures for resolving a problem and the analysis techniques used to develop those procedures.

If expertise is not readily available from past experience, the analysis efforts typically use high fidelity simulations based on system models to derive and evaluate control parameters. If expertise is available, the knowledge is reviewed by the panel and placed in the appropriate context. The panels consist of system users, independent domain experts, system developers, and managers to ensure adequate coverage of all areas of concern. In previous programs, the typical output of such a panel was a set of flight rules describing the operational requirements for a system.

Sometimes these flight rules were translated into computer programs (typically as decision trees) and embedded in the onboard or ground computers. An additional verification step was needed to guarantee that the flight rules approved by the panel were properly coded. More often, computer limitations caused the flight rules to remain in document form used directly by flight controllers and mission crews.

For future programs, many of the flight rules which come from the Flight Technique Panels can be coded directly into expert systems. Expert systems developed in this manner will have undergone extensive verification through the panel review. They should also prove easier to verify in code form because the rule language will allow the program to closely resemble the original flight rule.

Programs of the complexity and size with which NASA regularly deals make this approach mandatory. Smaller programs generally will not require the resources or effort involved in verifying a system to this extent. The size of the panel and the length of the review process can be scaled down to something appropriate for the complexity and size of the application. For some applications, the panel approach could look very similar to independent code review techniques.

Exhaustive testing through simulation remains the most effective method available for final validation. However, for any system of reasonable complexity, exhaustive testing is both prohibitively expensive and time consuming. Space Shuttle applications typically used extensive testing with data sets representative of the anticipated problems or failure modes. This method is not guaranteed to eliminate all software bugs, but it can prevent the *anticipated* problems. If used properly, representative testing can eliminate enough problems to make the software acceptable for mission and safety critical applications.

The panel approach to verification discussed above is very effective at ensuring that the knowledge in the expert system is both correct and complete. Verification of conventional software also covers feasibility, maintainability, and testability. These verification efforts are generally done early in the design phase and should also be done for an expert system. The coded rules must also be examined to ensure that the consistency and completeness of the design is properly incorporated in the software.

Some of this work can be done automatically. Testing a rule language for completeness and consistency may actually be easier than testing conventional programs. The explicit separation of knowledge elements from control and data elements may allow relatively straightforward analysis of the rules by automated tools [3]. If automated methods are not used, other standard methods such as code reviews and manual examination of the rules may also be comparatively easy, again due to the independent nature of the knowledge elements. They can be done by the whole panel, or more likely, small teams of people drawn from the whole panel.

Feasibility of knowledge representation is usually fully tested in the early prototypes, but the feasibility of other elements of the expert system, such as performance, user interfaces, data interfaces, etc. must also be verified. The use of rapid prototyping can be extended from testing representation to testing some of these areas as well. Iterative development can go a long way to ensuring that the final system truly meets the user needs in these kind of areas.

Finally, the requirements must be examined to ensure that they are able to be tested. They should be specific, unambiguous and quantitative where possible. Objective requirements will aid in the development of rigorous test cases for final validation. A test plan should be written which discusses how the final expert system will be tested.

OTHER ISSUES FOR EXPERT SYSTEM V&V

There are other differences between between conventional software and expert systems, and those differences will affect V&V efforts. Some of the differences are discussed in reference [4] and summarized below.

Verifying the Correctness of Reasoning

Verifying that an expert system solves a problem for the right reasons is sometimes as important as getting the right answer. For a rule-based expert system, identifying all possible paths to a solution is very difficult. Therefore, it is important to ensure that the expert system has gotten the right answer for the right reasons.

Verifying the Inference Engine

The inference engine in a rule-based expert systems is a completely separate piece of code anc can be fully verified independently from the rest of the expert system.

Verifying the Expert

This question is automatically resolved as long as the expert system is validated. The panel approach discussed in this paper provides continual feedback on the correctness of the experts knowledge.

Real-Time Performance

Most conventional programs provide performance "guarantees" through extensive simulation of the expected performance environment. Expert systems can provide the same kind of performance "guarantees". Some kinds of conventional programs are analyzed at the machine instruction level to specifically determine the amount of time required to process a given data set. Achieving the same kind of capability in a rule-based expert system is more difficult, but can be done for a given data set entered in a specific sequence.

Complex Problems with Multiple Experts

The panel review method already discussed here is clearly the appropriate method for resolving a problem of this type. The review process used by the panel will allow inputs from any number of domain experts and will also establish the methods of validating system responses.

Traceability of Requirements

Tracing requirements after they have been coded in rules may be more difficult than for conventional code, particularly when hybrid representation techniques are used, i.e. when both rules and objects are used to satisfy the program's requirements. This is an area that needs further consideration.

Verifying the Boundaries of the Expert System Domain

V&V of an expert system must be carefully aimed at identifying the boundaries of a problem since the experts sometimes can not readily do so. V&V must also ensure that the expert system fails gracefully in these circumstances.

There are additional issues not discussed in reference [4]. These are discussed more fully below.

Reasoning under Uncertainty

Some expert system applications deal with incomplete, inconsistent, or uncertain information. Humans do a very good job of reasoning under uncertainty, but it can be very difficult to develop consistent models which exactly duplicate this process. Numerous methods have been developed to allow expert systems to deal with this type of information, such as fuzzy logic, probability methods like Bayes theorem, Dempster-Schafer theory, certainty factors, etc. The nature of how humans use this type of information makes it very difficult to verify in an expert system. Different people may give different answers when presented with the exact same information. V&V efforts must focus on two things; (1) verifying that the answers suggested in uncertain situations are 'acceptable' answers. The definition of 'acceptable' may be problem dependent, and (2) if uncertain information is combined, the method used to provide a certainty factor to the result must be consistent.

Maintaining a verifiable system

Long-term maintenance of an expert system is a poorly understood topic, primarily because there is little actual experience in this area. Soloway, et al. [5] discuss some of the difficulties in maintaining XCON, one of the largest and oldest expert systems in use today. They point out that XCON is a very dynamic system, with extensive changes occurring regularly. As with conventional software, most expert systems will change and V&V must be performed each time the modified system is released. The nature of almost all rule-based languages makes true modularization of code more difficult than with conventional software. Therefore, rule-based systems presently require complete retesting with every release, using a library of test cases. Good programming practices such as using explicit control features and simple rules are important aids, but may not be sufficient to prevent extensive retesting. This area will be better understood when more applications reach maintenance stages.

CONCLUSIONS

Verification and validation of expert systems is very important for the future success of this technology. Software will never be used in non-trivial applications unless the program developers can assure both users and managers that the software is reliable and generally free from error. Therefore, V&V of expert systems must be done. Although there are issues inherent to expert systems which introduce new complexities to the process, verification and validation can be done. The primary hindrance to effective V&V is the use of methodologies which do not produce testable requirements. Without requirements, V&V are meaningless concepts. An extension of the flight technique panels used in previous NASA programs should provide both documented requirements and very high levels of verification for expert systems.

REFERENCES

[1] Bochsler, D.C. and Goodwin, M.A., "Software Engineering Techniques Used to Develop an Expert System for Automated Space Vehicle Rendezvous", Proceeding of the Second Annual Workshop on Robotics and Expert Systems, Instrument Society of America, Research Triangle Park, NC., June 1986,

[2] Culbert, C.J., Riley, G., and Savely, R.T., "An Expert System Development Methodology Which Supports Verification and Validation", to be published.

[3] Stachowitz, R.A. and Combs, J.B., "Validation of Expert Systems", Proceedings Hawaii International Conference on Systems Sciences, Kona, Hawaii, January 6-9, 1987.

[4] Culbert, C.J., Riley, G., and Savely, R.T., "Approaches to the Verification of Rule-based Expert Systems", Proceedings of SOAR'87: Space Operations-Automation and Robotics Conference, Houston, TX., August 1987.

[5] Soloway, E., Bachant, J., and Jensen, K., "Assessing the Maintainability of XCON-in_RIME: Coping with the Problems of a VERY large Rule-Base", Proceedings of AAAI-87, Sixth National Conference on Artificial Intelligence, Seattle, WA., July 1987.

KBS V&V - State-of-the-Practice and Implications for V&V Standards[1]

David Hamilton, Keith Kelley & Scott French
IBM Federal Sector Division
3700 Bay Area Boulevard
Houston, Texas 77058

Chris Culbert
NASA/Johnson Space Center
Software Technology Branch/PT4
Houston, Texas 77058

Abstract

The majority of the work in knowledge-based system verification and validation (KBS V&V) has focused on developing techniques and concepts for performing V&V on expert systems. Little information is available on what V&V practices are currently in use by expert system developers and how current KBS practices compare to what is typically required on large systems. This paper summarizes the results of a survey whose purpose was to begin documenting some of the experiences and problems KBS developers have encountered. It also summarizes the results of analyzing the V&V requirements for a specific program (Space Station Freedom). The results of the survey suggest that current practices can be improved while the results of analyzing Space Station V&V requirements show that the conventional software state-of-the-practice is not completely applicable to KBS V&V. The results have implications for many large programs and for KBS V&V research.

Introduction

Knowledge-based systems (KBS)[2] are in general use in a wide variety of domains. As reliance on these types of systems grows, the need to assess their quality and validity reaches critical importance. As with any software, the reliability of a KBS can be directly attributed to the application of disciplined programming and testing practices throughout the life-cycle. However, there are essential differences between conventional software and knowledge-based systems, both in construction and use. The identification of how these differences affect the verification and validation (V&V) process and the development of techniques to handle them is the basis of work in this field.

Much of the work in KBS V&V has focused on developing conceptual approaches and postulating different techniques for performing some or all aspects of V&V on various types of KBS or expert systems (ES) [5]. Very little work in this field has demonstrated the usefulness of proposed techniques on operational KBS. Even more importantly, since effective V&V must be applied throughout the life-cycle, there has been almost no case study work in applying disciplined software V&V princi-

1 Elements of this paper have already been published in [9].

2 Or expert systems. Although there is a growing acceptance of different definitions for knowledge-based systems and expert systems, we will use the terms interchangeably in this paper. The differences between KBS and expert systems do not significantly affect the V&V process.

ples throughout the development of an operational KBS. The long term goal of our work is to develop guidelines, standards, tools, and techniques for V&V of all KBS applications which many be used in the Space Station Freedom Program (SSFP). As a precursor to determining the applicability or usefulness of many of the proposed KBS V&V techniques, it is important to develop an understanding of what V&V practices are commonly in use today and how proposed techniques can improve upon those practices.

It has been widely claimed that few expert systems are subjected to the same level of V&V that conventional software routinely undergoes [4]. However, this practice has not been well documented. More important for our purposes, little documentation[3] exists which describe the problems associated with KBS V&V from the developer or user's point of view. The specific purpose of our survey was to begin documenting the experiences and problems KBS developers have encountered in performing V&V on their systems and relate those problems to the kinds of issues KBS V&V researchers consider important. The overall strategy for determining the state-of-the-practice was to determine how well each of the potential expert system V&V issues are being addressed and to what extent they have impacted the development of expert systems. Our approach was to develop a set of survey questions for both KBS developers and users and then to follow that survey with selected interviews.

Because our ultimate goal is to develop guidelines, etc. for SSFP, we compared the results of our survey to the existing SSFP V&V requirements. We also analyzed all the SSFP V&V requirements to determine their general applicability to KBS V&V.

In this paper, we first summarize the results of this survey[4] and then we summarize the results of analyzing SSFP V&V requirements.

Survey Results

A total of 70 people, 93% of which were developers, responded to the survey concerning a variety of knowledge-based systems. Seventy percent of these systems were operational and the remainder were considered prototypes (although some of these "prototypes" had users). These systems covered a range of criticalities and sizes, requiring as little as one person-month of development effort to as much as two hundred person-months of development. Most (75%) of the systems were concerned with diagnosis, primarily in the aerospace field (73%).

Questionnaire Results

Much of the results can be derived by simply calculating the fraction of respondents that answered a question in a certain way. The following is a short summary of each type of information gathered[5].

Performance Criteria:

Thirty-nine percent estimated that the expert system performed with an actual accuracy of less than 90% and 54% estimated an accuracy of less than 95%. Most (50%) estimated the problem space coverage between 60% and 95%. In comparing the accuracy of the expert and the expert system, most (79%) expected the expert system to at least as accurate as the expert. Yet, the actual systems were often (75%) estimated to be less accurate than expected and also (62%) less accurate than the expert. Users, more often than developers, estimated the expert system as being less accurate than expected and less accurate than the expert.

Requirements Definition:

Seventy-five percent indicated that expert consultation was a basis for determining the behavior of the system. More revealing is that for 52% of the systems surveyed, there were no documented requirements. Forty-three percent indicated that prototypes or similar tools were used for requirements. Forty percent had medium difficulty in generating requirements, 35% said the requirements were hard to develop, 25% said the requirements were easy to develop. Fifty-eight percent of developers had a high level of contact with experts during development.

Development Information:

The most frequent (40%) life-cycle model used is the Cyclic Model (repetition of Requirements, Design, Rule Generation, and Prototyping until done). However, 22% of the respondents stated that no model was followed. Most development was done with an expert

3 An exception is documented in [8].

4 A more complete discussion of the survey results appears in [9].

5 Unless otherwise noted, the percentages shown are the percentage for all the responses, both developer and user combined.

system shell (CLIPS and others), and the predominant Interface Code was C and LISP. Applications were reasonably large, requiring an average of 23 person-months to develop. Developed systems were not reported to be particularly sensitive to change (77% said changes only occasionally caused an unexpected behavior).

V&V Activities Performed:

Most V&V activities relied on comparison with expected results and checking by the expert. Sixty-six percent used functional testing and 44% used structural testing. Fifty-nine percent had the domain expert check the knowledge base. On average, 24% of the development was spent on V&V. While all (100%) of the users rated V&V of expert systems as hard, the response from developers varied. Thirty-four percent of the developers said the V&V effort was of medium difficulty while 27% said it was hard and 33% said it was easy, 5% said it was impossible. Significantly, each V&V technique was used as the sole V&V technique in at least one project. Also, in general, there were wide ranging uses of V&V techniques; each technique was used by many projects.

V&V Issues Encountered:

The known issues most often cited as problems were: test coverage determination (63%), knowledge validation (60%), real-time performance analysis (33%), and problem complexity (40%). Other problems cited were: modularity (27%), configuration management (20%), certification (11%), and understandability (10%). The least cited problem was analysis of certainty factors (only seven respondents indicated that certainty factors were used). Every known issue was cited by at least one respondent. The expected system use varied widely (3-2000), while actual system use was relatively good. However, less than half of the respondents provided information, suggesting that actual use was much lower than reported. Of those who responded with an opinion, 96% felt that their expert system was at least as reliable as a typical conventional software system, and 51% felt it was more reliable.

Interview Results

In addition to acquiring written responses to the survey questions, interviews were performed to gather additional data and to clarify questions concerning the written responses. Additional information from these interviews are summarized in this section.

Structural Testing:

Based on the survey results, a commonly used evaluation approach was the use of structural testing. This was surprising because the common perception among KBS researchers is that many common forms of structural testing are relatively difficult to apply to expert systems. From the interviews, we learned that although some projects did attempt to measure the actual test coverage (i.e., percentage of rules executed during testing) many others did not actually measure the coverage. Instead, they attempted to develop test cases that would cover all of the knowledge base (or at least the important parts) but made no attempt to measure how well the knowledge base was actually covered. Also, there appeared to be no attempt to cover interactions between knowledge base elements (e.g., rule interactions). Generally, each element was tested as if it were an independent piece of the knowledge base. Some knowledge base developers felt that more formal structural testing would be too much effort and would hinder the development process too much. The interview results suggest that although structural testing was used, it was a very weak form of structural testing (at least compared to, say, branch coverage in procedural software testing).

Experts Developing Expert Systems:

It appeared that the expert was heavily relied upon to aid in evaluation of the knowledge base; this subject was probed more deeply during the interviews. The developers felt that a close interaction between the expert and the knowledge base developer was mandatory to successfully develop an expert system. This is not a surprising result and it has been discussed at length in the literature [1]. Many KBS developers feel this interaction is so important that they think the best approach is simply to have the expert develop the system. Though it is important for a knowledge engineer to understand the problem domain and to thoroughly represent that domain [6], it is generally accepted that the domain expert should not be the sole developer of an expert system[6]. There are many problems associated with the development of an expert system by a domain expert. Experts often use knowledge that is so highly compiled and implicit that they have difficulty defining that knowledge explicitly (so a machine can use it). Furthermore, collection of domain knowledge from "introspection" is generally held in doubt by psychologists [3]; that is, experts often don't solve a problem the way that they think they do. Finally, building expert systems often involves building highly complex software systems, systems that require skills and training that domain experts seldom have. Some of these issues were recognized by at least one interviewee who felt that when his group begins to tackle more sophisticated problems, they would need developers with better-developed software and knowledge engineering skills.

Requirements Writing and the Conventional Software Life-Cycle:

We anticipated that expert systems were being developed using a much more iterative and less structured life-cycle than the conventional waterfall model. Although the subject of life-cycle models was not intentionally addressed during the interviews, it often came up when discussing requirements. It seems that several respondents associated "requirements" with the conventional waterfall model. They felt very strongly that the conventional approaches to software development, such as the waterfall model, were much too formal and structured for expert systems development. Some even suggested it would be disastrous to apply them to expert systems. For many, this feeling extended to documenting requirements, others simply used a different approach to requirements. For example, in some cases, requirements were not written because it was felt that a requirements document was a formally written paper document that needed to be "approved" before development could proceed. In other cases, an iterative prototyping development effort took place and was followed by documenting system requirements. These requirements were then used to test the system to ensure that it worked as everyone thought it should.

Prototypes vs. Operational Systems:

Although we asked respondents to state that their system was either "a prototype" or "operational," we received indications that this distinction was often difficult to make. For example, responses included "it is both a prototype and operational," or "it is an operational prototype," or "it is just a prototype but we have many users." It seems that some systems are originally intended to be a prototype but are used operationally. Some intentionally approach the development of an operational system by first developing a "prototype" and once the prototype is "certified," it is considered "operational." Others acknowledge there is a danger that a prototype will be used as if it were operational. They have taken steps to ensure that a prototype system that is not accidentally relied upon in an operational setting.

Real-Time Performance Analysis:

In our survey, we intended "real-time performance analysis" to refer to the ability to predict the response time for an expert system. That is, the ability to analyze the time performance of the system. However, from the interviews we learned that many interpreted "real-time performance analysis" to mean the ability to get the system to run as fast as desired/necessary. While this is important, it is unclear from the survey and the interviews just how many (if any) of the respondents had quantifiable, rigid needs for expert systems which could generate a response in a guaranteed time frame. Certainly few of the system developers had formally analyzed or documented any "hard" real-time constraints.

Issues Independent of A System Being an Expert System

An important, but difficult, aspect of analyzing expert system development methodology is distinguishing properties of expert systems that are significantly different from properties of conventional software [2]. This is also an important aspect of the analysis of this survey of V&V issues. Several comments appeared to be due more to factors other than the fact that the system being developed was an "expert system." The interviews helped clarify this issue, and the important ones are discussed in this section.

Extensive Use of Prototyping and Rapid Development:

The conventional waterfall life-cycle model has proven to be ineffective for conventional software development. Therefore, it is no surprise that developers do not want to use it for expert system development. A more iterative model (e.g., the spiral model) that includes the use of rapid prototyping is being perceived as a better alternative to the waterfall model. "Conventional" software development projects often include the use of prototyping for activities like developing better user interfaces and having developers better understand the problem domain. These kind of issues are not unique to expert system development, but did come up often in the survey, particularly during the interviews.

Small/Simple vs. Large/Complex Systems:

Although some of the systems surveyed are fairly large (e.g., 200 person-months), they are generally much smaller than dedicated software development projects

[6] This is described in more detail in [7], p.154 as the Knowledge Engineering Paradox: "The more competent domain experts become, the less able they are to describe the knowledge the use to solve problems."

(e.g., Shuttle mission control center (MCC), Shuttle flight software, etc.). The systems surveyed seem to be isolated efforts to develop off-line applications for niches for which expert system technology was felt to be very suitable. They were generally systems that were not part of a larger software system, though they are often used in conjunction with a large data processing system (e.g., they receive real-time data from a large data processing system). This allowed the expert system developers to work without many of the constraints imposed on larger systems (e.g., tightly controlled configuration management).

Addressing a Knowledge Engineer Instead of a Programmer:

Although we did not intend to gather information on the experience and background of individual expert system developers, we did learn that several respondents involved in developing expert systems are experts in a problem domain without significant programming experience. This fact was important when formulating the detailed recommendations[7].

Issue Summary:

It may be the case that the above issues are indeed typical of expert system development projects and that they should be addressed when addressing V&V of expert system problems. However, it should be recognized that they are somewhat different than the other issues that are true of all expert systems regardless of their size and who is developing them. This may point to a need to tailor suggestions for V&V of expert systems to considerations such as the size of the expert system, the experience of the developer, whether the system is embedded in a much larger software system, etc..

Recommendations Based on the Survey

The major goal of this survey was to discover and document the current state of the practice in V&V of expert systems. Based on the survey results, it appears that much can be done to improve the practice. As a starting point, recommendations for improving KBS V&V were drawn from the survey and interview results. These recommendations are separated into two categories: direct recommendations which are directly supported by the survey results and inferred recommendations which can be inferred from the survey results by analyzing relationships among the responses.

[7] The detailed recommendations are discussed in [9].

Direct recommendations include:

- Develop requirements for expert system verification and validation
- Address most often encountered issues
- Recommend a life-cycle for expert systems development

Inferred recommendations include:

- Address readability and modularity issues
- Address configuration management issue
- Develop criteria to classify expert systems by intended use
- Investigate applicability of analysis tools

Survey Conclusions

The original goal of our survey was to gather data and document the current state-of-the-practice in KBS V&V. The survey and follow-up interviews have given us considerable insight into the kinds of problems that developers have really encountered in developing and verifying expert systems. Many of these problems will require additional work before solutions will be readily available. The analysis of the survey and interviews and the subsequent recommendations can serve as valuable reference for directing future KBS V&V research into those areas which are of the most value to KBS developers and users. In addition, managers of KBS development projects can learn from these results to structure life-cycle approaches for KBS development which are more likely to lead to high quality application software.

Space Station Freedom Program V&V Requirements Analysis

Overview

There are several software V&V requirements for the Space Station Freedom Program (SSFP) that are contained in SSFP documents. KBS V&V issues were not considered when these requirements were defined so it was felt that they might not be appropriate for the V&V of KBSs. To understand the scope of this problem and how it might be resolved, we defined a task[8] to:

- Identify all SSFP V&V requirements
- Analyze the applicability of the requirements to KBSs
- Make recommendations so that all V&V requirements would apply to KBSs. A recommendation

could be to change an existing V&V requirement or to develop a KBS V&V technique that could be used to satisfy a requirement.

Analysis

From several SSFP documents, we initially identified 93 SSFP V&V requirements which were specific to the technical work of software V&V. That is, we did not consider hardware requirements, general documentation requirements, or logistical requirements such as reporting procedures. Grouping similar requirements together and eliminating some minor duplication resulted in 50 distinct requirements.

We analyzed each of the 50 requirements to answer the following questions:

- What is the intent of this requirement ?
- Does this requirement make sense for a KBS ?
- Is this requirement currently satisfied in the current state-of-the-practice ?
- If it is not in the current state-of-the-practice, is there any inherent reason it could not be satisfied ?
- If there is no inherent reason it can not be satisfied, what is it about KBS development that makes this requirement difficult to satisfy ?

Results

Twenty-seven of the requirements are defined either at a level of generality or at a point in the life-cycle where specific software attributes are indistinguishable and can be applied equally to both KB and conventional software systems. Seven of these requirements can be applied to KBSs using existing processes. Thus, 16 requirements remained that were uniquely difficult or impossible to satisfy for KBSs.

We learned that many requirements that would be difficult to satisfy for KBSs were due to two major factors: "life-cycle model" (four requirements) and "system requirements" (five requirements). The "life-cycle model" factor existed because a general waterfall-type of life-cycle model was assumed to be used for system development. For example, the SSFP configuration management requirements would be difficult to apply to an highly iterative life-cycle by having a high overhead to document and release changes to the system. The "system requirements" issue existed because many of the requirements relied on the existence of a detailed set of requirements that identified many considerations; the general state-of-the-practice definitely does not include the generation of such detailed requirements. For example, there is an SSFP requirement to verify quality requirements yet there is no well-understood way of measuring the quality of a KBS.

The remaining V&V requirements that would be a problem for KBSs are:

- Identification of modules (There is no clear way of identifying "chunks" of knowledge as a module, e.g., a rule grouping.)
- Verifying maintainability (It is not clear what makes an expert system maintainable.)
- Requirements to code mapping (Can not be mapped to modules unless modules can be identified; mapping to individual rules/frames is too difficult.)
- Performance analysis (It is difficult to analyze the response time of non-procedural programs.)
- Path coverage (Paths in the conventional sense do not apply to non-procedural programs, paths in a broader sense are much more difficult to identify in non-procedural programs.)
- IV&V (Because of the heavy reliance on experts to aid in verification, independent verification [without the expert or using a different expert] may not be feasible.)
- Verifying off-the-shelf-components (There are not standards in KBS languages as there is in the standard procedural language, Ada.)

Implication to Other Programs

Most existing programs have V&V standards and guidelines that are similar to the SSFP V&V requirements and were generated with conventional procedural software in mind. An analysis similar to the one summarized here would be necessary to adapt the existing program standards and guidelines so they could be applied to KBSs. This approach would be preferable to generating a separate set of standards and guidelines for KBSs. As with SSFP, it is likely that the majority of standards and guidelines could be applied to KBSs without any difficulty so there would not be much duplication. Also, in practice, it may not be clear where in the system a KBS ends and conventional software begins. It may even be the case that a system that starts out being a KBS might end up being implemented as conventional software or visa versa. So having separate KBS and conventional software V&V standards and guidelines would create many difficulties.

[8] A more detailed discussion of this work is discussed in [10].

Summary

From the survey that we have performed, we have determined that there are some issues with respect to the state-of-the-practice in V&V of KBSs. We have also learned about common practice as well as problems. From the analysis of SSFP V&V requirements, we have learned that conventional V&V standards and guidelines are not completely applicable to V&V of KBSs. We have also learned that the state-of-the practice in conventional software V&V (as represented by standards and guidelines) is significantly different than the state-of-the-practice in KBS V&V.

References

[1] Bell, M.Z. (1985). Why Expert Systems Fail. *Journal of Operations Research Society*. 36 (7).

[2] Culbert, C., Riley, G., Savely, R.T. (1987). An Expert System Development Methodology Which Supports Verification and Validation. In *Proceedings of ISA 88*. Houston, TX: Instrument Society of America.

[3] Ericson, K.A., Simon,H.A. (1984). *Protocol Analysis*. MIT Press.

[4] O'Keefe, R.M., Lee, S. (1990). An Integrative Model of Expert System Verification and Validation. *Expert Systems with Applications* 1 (3).

[5] Rushby, J. (1988). *Quality Measures and Assurance for AI Software.* NASA Contractor Report No. 4187.

[6] Slagle, J.R., Gardiner, D.A. (1990). Knowledge Specification of an Expert System. *IEEE Expert*. 5 (5).

[7] Waterman,D.A., (1986) *A Guide to Expert Systems*. Addison-Wesley.

[8] Constantine,M.M, Ylvila. W.W. (1990). Testing Knowledge-Based Systems: The State of the Practice and suggestions for Improvement. *Expert Systems with Applications* 1 (3).

[9] Hamilton,D., Kelley,K., Culbert,C. *State-of-the-Practice in Knowledge-Based System Verification and Validation.* To appear in *Expert Systems with Applications.*

[10] Expert System Verification and Validation Study, RICIS Contract #069, Phase 2 - Requirements Identification, Delivery 2 - Current Requirements Applicability, University of Houston / Clear Lake.

Proposed Requirements Content

Robert C. Angier

IBM Federal Systems Division

ES V&V Workshop

Contents

Software Requirements Specification

The following is an annotated outline for a proposed detailed Software Requirements Specification (SRS) for software support tools.

Goals: The objectives of this document are to:

- provide a standard for detailed software requirements
- address the information needs of both users and developers
- allow for orderly expansion from user to developer requirements
- provide requirements content "checklist" that is complete enough to avoid surprises later
- include requirements related to software reusability
- simplify requirements maintenance by partitioning its content into independent sections (to the degree possible)
- provide a requirements model that supports a range of product sizes, from a simple, stand-alone tool to a large system of tools.

Sources: Sources used in developing this document include:

- DOD-STD 2167 Software Development Standards, 04 Jun 85
- Draft FSD Software Requirements Specification (SRS) Practice and Bullletins, SEB Spec. Development Working Group (SDWG), Nov. 85 and Jan. 86.
- Proposed Standard for SPF Transaction Requirements, Oct 83.

Use of Standards: Much of the information requested in each section could be defined once, as a standard, and referenced by individual specifications. This method is preferred, since it results in greater compatibility between products. It would also reduce the amount of work required to write a detailed specification, with no loss of content.

Document Variations: Since this document has been designed to support large-scale software systems, it is acknowledged at the outset that some of its provisions will not be needed for simple tools. Specific variation by section are summarized in "Application by Product Type" on page 17.

Terminology: The terms "item" and "software element" have been used interchangeably in the remainder of this document. Both refer to the software product which is specified by the requirements document.

Document Content

Section 1 - Introduction

This section provides general orientation material related to the software element and its specification.

1.1 - Document Description

A standard section that summarizes:

- the purpose of this document
- the contents of this document (by major section)

1.2 - Item Identification

Identifies the software element specified by this document, including its ID (if any) and full name.

1.3 - Item Purpose

Brief description of purpose of the software element (i.e., its intended use).

If the item is part of a larger system, then the following subsections should be used:

1.3.1 - System Membership: The identity of the system or systems that the software element is a part of.

1.3.2 - System Purpose: Real-world purpose of the system that the software element is a part of.

1.3.3 - Item Role: role of the specified software element within the system (i.e., what it is responsible for within the system)

1.4 - Item Scope

Summary of the software element's scope:

1.4.1 - Major Functions: Summary of major functions (actions) performed by the software element (i.e, scope of its functional responsibilities).

1.4.2 - Application: Identifies the software element's expected range of application (scope of its target domain).

1.5 - Item Classification

Identification of the type of software element specified (e.g., stand-alone tool, system component, reusable software component)

If this item is reusable, the following subsections apply:

1.5.1 - Item Type: Identifies the category of this item within a designated reusable software taxonomy.

1.5.2 - Item Characteristics: Identifies key distinguishing features of this item which aid in its selection.

Section 2 - Applicable Documents

This section contains a summary list of other documents that **form a part of** this specification, consolidated from other parts of this document. It identifies the exact versions of documents that apply.

2.1 - Specifications

Related specifications that affect the software product, including:

- Higher-level specifications (e.g., of the system that this element is a part of.
- Interface specifications of related software and hardware elements.

2.2 - Standards

Standards that apply to the software product, or the process by which it is produced.

2.3 - Other Publications

- Drawings
- Manuals
- Regulations
- Handbooks
- Bulletins
- etc.

Section 3 - Environmental Specification

This section is intended to make assumptions or requirements about the software element's surroundings explicit and visible. This is to avoid "surprises" later, when the product is delivered.

There are three parts to this section:

- the operational environment in which the software element is used,
- the target execution environment in which it runs, and
- the implementation environment in which it is produced and supported.

This organization recognizes that software IS a transformation of the target "machine" to an operationally useful one. Software is also transformed form its implementation form to the target machine form. These relationships are shown in Figure 1.

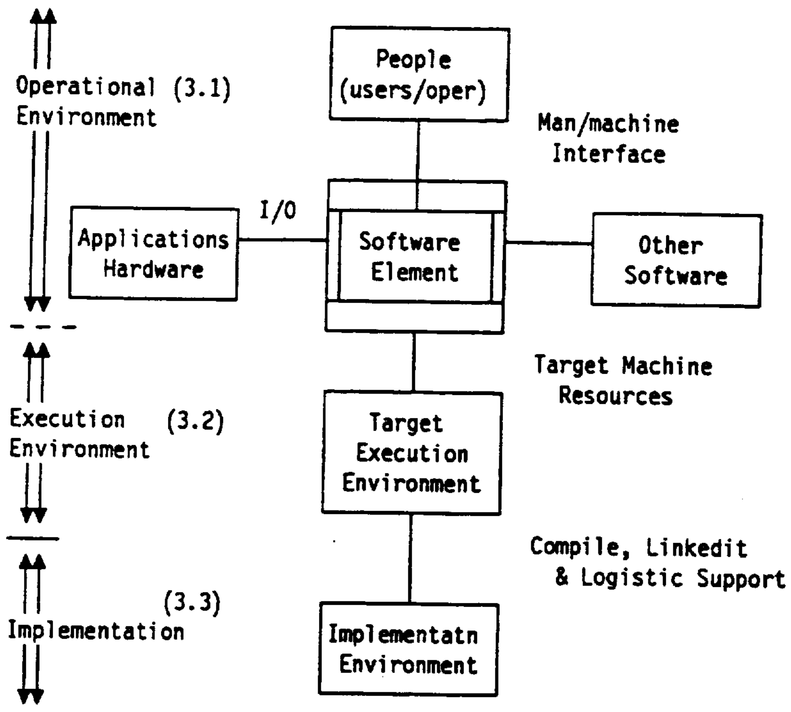


Figure 1. Relationships of a Software Element to Its Environment

3.1 - Operational Environment

This section is a general characterization of (1) how the specified software element is intended to be used, and (2) the people that will use it.

3.1.1 - Operational Objectives: Describe what the system's end users are trying to achieve, in terms of useful results. Also describe generally how the software element contributes to the user's real-world objectives.

3.1.2 - Operational Constraints: Identify real-world constraints that limit how the system can be used. (For example, sharing of terminals by several users, or limited time to get the work done).

3.1.3 - Item Users: Description of the software element's users. Where the software element supports more than one kind of user (e.g., an "author" and "reviewer"), these groups should be identified here, and characterized separately.

3.1.3.x - (Name X) User Group: For each distinct user role, identify:

3.1.3.x.1 - Tasks

- the user's job objective or responsibility
- tasks that make use of this software element

3.1.3.x.2 - User Characteristics: Describe this group of users, in terms of assumed (or required) knowledge, skill levels, and training.

3.2 - Target Execution Environment(s)

This section defines the hardware/software environment (or environments) in which the software element is to be operated.

3.2.1 - Target Machine Environment: Identifies the target programmable hardware in which this software element will operate. Hardware devices that it directly interfaces with, including terminals and workstations, should be identified. Where hardware configurations are restricted, those constraints should also be noted here.

3.2.2 - Target Software Environment: Identifies the operating system and other common software packages that make up the target operating environment (e.g., MVS, IMS, and ADF II). Only include those items on which the specified software element must depend. Include minimum release numbers if applicable.

3.3 - Implementation Environment

This section describes features of the software development environment that are significant to implementation of this software element. Its subsections describe the implementation hardware, software, languages, and process.

3.3.1 - Implementation Hardware Environment: The machine environment in which the software element is to be developed and supported should be described. If it is the same as the target execution machine, then simply reference that section.

3.3.2 - Implementation Software Environment: The software implementation environment includes identification of the implementation operating system, compiler(s), assemblers, linkage editors, and other tools that affect development of the software product.

3.3.3 - Programming Languages: Specifies the allowable programming language(s) in which the software element is to be implemented. Identify standard language variants if used (e.g., the project-specific part of the Ada language).

3.3.4 - Implementation Process Standards: Identify standard methodologies that are to be used in implementation of the software product, in order to control its content and quality.

3.3.5 - Software Product Standards: Identify standards to be met by deliverable products of the software implementation process, including design documentation, source code, and test procedures.

Section 4 - Interface Requirements

4.1 - Summary of Interfaces

This section defines the interface requirements that affect the software element interactions with other system elements:

4.1.1 - Interface Relationships: A summary of functional and physical interfaces between the software element and hardware or other software elements. This is usually satisfied by a block diagram with labelled arrows.

4.1.2 - Interface Identification and Documentation: Proper identification of each interfacing hardware or software element, and identification of associated documents containing interface requirements. For unique interfaces, the appropriate section of this document should be cited.

This section can be satisfied by an Interface Identification Table, such as the one shown in Figure 2.

Interface Name	Interfacing Element	Doc. Num.	Document Name
term if	Model XYZ Terminal	xyz-0014	Program. I/F
...	... ...	...	...
graf if	Graphic Display Subsys	This Doc	Sect. 4.2.5
mous if	WHITE-2300 Mouse	This Doc	Sect. 4.3.1

Figure 2. Sample Interface Identification Table

4.3 - Unique Software Interfaces

This subsection describes detailed interface requirements for software interfaces described above, which are not defined in separate specifications.

If any software interfaces are uniquely defined for this item, they should be documented here in separate subsections:

> **4.3.x - (Name X) Interface** This subsection specifies the "X" software interface by name, discusses its purpose, its partitioning of functional responsibilities, and provides a summary of information communicated via the interface. The summary may be provided by an Interface Summary Table, as in Figure 3 on page 7.

Interface Name	Information Description	Initiation Condition	Expected Response
term if A	Data Ready	Buffer Full	Clear
HW ==> SW	Sync	Receive Sync	none
term if B	Mode Select	Mode Change	Status
SW ==> HW	Sync	Startup	Sync
	Status Request	Cyclic 1.0 Hz	Status

Figure 3. Sample Interface Summary Table

Interface specifications should include:

- Identification of which element transmits data, and which receives it.
- The conditions for initiating each data transfer. If cyclic, specify the rate.
- The transfer protocol used for the interface (e.g., blocking, message switching, handshaking).
- The priority level of the interface and each signal, if applicable.
- Format and content of the data being transferred. Include units of measure, scaling, and representation conventions, where applicable.
- The expected response to each data transfer, including the maximum time allowed for the receiving element to acquire the data, and respond, if applicable. Also include the effects of not responding, if any.
- Identify whether the interfacing element executes concurrently or sequentially with the software element being specified. If concurrent, the method of inter-task synchronization should also be specified.

4.4 - Unique Hardware Interfaces

This subsection describes detailed interface requirements for hardware interfaces described above which are not defined in separate specifications.

If any hardware interfaces are uniquely defined for this item, they should be documented here in separate subsections:

4.4.y - (Name Y) Interface This subsection specifies the "Y" hardware interface, in the same manner as for "4.3 - Unique Software Interfaces" on page 6.

Section 5 - Detailed Functional Requirements

This section specifies what the software element must do (not how it must do it). The software element's externally visible behavior should be described as though it was a "black box". This approach provides the developer with enough flexibility to choose the best design, while giving the requirements engineers the ability to define what "best" means for this item.

5.1 - Functions

The functions (actions) performed by the software element are defined here. A simple tool can be completely described in one section. For complex systems, the behavior is often described by a model, consisting of simple functions that are linked together by data flow.

The subsections which follow describe a functional model for a large system.

5.1.1 - Functional Overview: A summary diagram, such as a Functional Black Diagram, or Level 1 Data Flow Diagram, should be used when more than one function is described. This provides a frame of reference, or "big picture", in which the individual functions can be understood. (This section may be omitted for a simple tool).

If there are any requirements that apply in common for all functions, then subsections can be used, as in "5.1.x - (Name X) Function."

5.1.x - (Name X) Function: The X Function is identified and briefly described. For interactive tools, each transaction should be described as a separate function. The subsections below provide a detailed functional specification:

5.1.x.1 - Activation: Describes how you get here. Specific considerations to be addressed are:

Activation Conditions: Describes when to perform this function, how it is invoked (either manually and/or by other software), and what must be done prior to to its activation.

Termination Conditions: Describes when this function may be exited, how, and what must be done prior to its termination.

Restart Conditions: Describes the conditions under which operation of this function is re-started (if any).

Checkpoint/Recovery: Defines requirements for checkpointing current state, and recovering that state (if any).

5.1.x.2 - Displays: This subsection describes the human interface to the function, as perceived by the user. It can include the following, in order of increasingly detailed specification:

Information Content: Identifies the information items to be displayed. Static items should be differentiated from those which may change.

Presentation Form: The form in which the information is to be displayed is defined:

data format	describes how each data item is to be represented
organization	describes the arrangement of the display as a whole, either in general terms, or in detail (e.g., a direct image of the intended display). Rules by which this organization can be adapted to different devices (e.g., with different display sizes) should also be given.

color use	rules for the use of colors, or direct assignment of colors to display elements.
highlighting	use of special features such as overbright, reverse, and blinking.
symbol use	rules for the use of special symbols on displays

Initialization: defines how the display should be pre-set when activated.

Other Human Inputs: Use of bells, alarms, or other devices that are intended for the user.

5.1.x.3 - Controls: This subsection describes the human interface by which this function is controlled. Typical means of control are by menu selection, data field entry, PF Keys, and command entry. It is desirable to separate the mechanism used from the action produced, since these assignments are frequently device-dependent, or are subject to customization. A Functional Control Summary Table, as in Figure 4, may be used to provide an overview of the controls available. Additional Activation columns may be needed for specific devices (e.g., an IBM 327x terminal has different PF Keys than a 3270-PC).

Type of Control	Control Name	Control Description	Means of Activation
Exit from Display	back	Return to previous level	PF Key-3
	home	Return to "home" display (main menu)	PF Key-2 / CMD=home
Invoke Another Display	help	Call "help" facility to explain this display	PF Key-1
Function Control	add	Add or Update a record	itemcmd=A
	del	Delete a record	itemcmd=D
	crank	Calculate current results	CMD=run
Display Content Control	recrd	Specify a record for display	CMD=Rec_ID
	frwd	scroll forward to next recd	PF Key-8
	hex	tranlate to hexidecimal	enter

Figure 4. Sample Functional Control Summary Table

Specification of each control may include the following:

Control Activation: describes the means of activating this control (if a table is not used).

Control Inputs: Identify the information items that define the control's action, and their effects.

Contraints: Defines any restrictions on the use of this control, and the system response when these are not satisfied.

Actions: Identifies the action(s) performed when this control is used. (These are further defined in "5.1.x.5 - Actions" on page 10).

5.1.x.4 - Inputs: This subsection describes the data input requirements of the function. For interactive systems, this corresponds to manual data entry of dynamic fields. For other applications, files or records may be specified.

Typical input specifications include the following:

Input Source: Identifies the source or sources of required data.

Input Organization: Define the arrangement of input data (e.g., record layout, sort order) if it isnot pre-defined elsewhere (e.g., by interface specifications or display definition).

Input Constraints: Defines any restrictions on the value of an input, and the system response when these are not satisfied.

Conversions: Defines the transformations to be made to the input to put it in in a form which is uasble by this function.

5.1.x.5 - Actions: This subsection defines what the function does. Actions that may be performed include data transformation, generation or detection of events, commanding devices, or performing mode transitions.

The principal objective of this part of the specification is to be as clear and concise as possible in describing the required action. A variety of means of expression are possible, including:

- mathematical formulation
- structured English description
- decision tables or trees
- data flow diagrams
- etc.

Choose methods which are appropriate to the function being defined (For example, decision tables are useful for expressing complex logical conditions).

5.1.x.6 - Outputs: This subsection defines the outputs that are produced by this function, including their destination functions, or external software or hardware elements.

5.2 - Modes or States

This subsection defines the major changes in function that result in characteristically different software element behavior. For example, if a tool provides both a "browse" and an "edit" capability on the same file, the way it operates is different in each case. Controls may be different, or have different effects; displays may vary; allowable operations in one case may be illegal or non-sensical in the other. The concept of a mode is generally more extensive in large systems, where modes or states are often directly related to the operational task.

5.2.1 - Operating Modes: The possible operating modes of the software element are defined individually. If there are several types of modes involved, it may be useful to arrange them into related groups. For each mode, its name, description, and main characteristics should be stated.

5.2.2 - Events: System events that can affect its operating mode (e.g., the failure of a hardware device) are described here. A name and definition should be given for each event.

5.2.3 - Mode Transitions: The ways in which system operation changes from one mode to another are defined:

Initial Modes

Identifies the system modes that are present at initiation of this software element.

Legal Mode Transistions

Defines the set of allowable changes in modes. This infomation can be provided in a mode-to-mode transition matrix, or by a state transition diagram. If several types of modes exist, they should be grouped into connected sets.

Mode Transistion Rules

Defines the conditions or events that determine when each leagal mode transition can occur.

5.2.4 - Relationship to Functions: Defines the effect of system modes on the software element's functions. This can generally be shown by a table that indicates which functions are valid in each operating mode.

5.2.5 - Relationship to Objects: Defines the effect of system modes on the information objects on which the functions operate. A table showing which objects are valid in each state can be used.

5.3 - Information Requirements

This subsection defines the major information objects that are used or produced by the software element.

5.3.1 - External Objects: The data which describe real-world objects to the system are defined.

An information processing system relies on information models of real world objects. For example, a person may be represented by a name, SS#, and department. Similarly, a hardware device may be represented to the system by its device type, model#, path, and I/O rate. Definitions should be grouped by the object to which they refer.

5.3.1 - Internal Objects: Where necessary, the representation of internal information objects is defined (e.g., I/O blocks, records, files, or data bases). This may be needed where pre-existing interfaces must be satisfied, as in hardware device interfaces, or interfaces to existing software systems. Where ever possible, reference source requirements for these definitions, rather that repeat them.

Section 6 - Performance Requirements

This section defines how well the software element must satisfy its functional requirements. Performance considerations are often the most significant determining factor in software design, and in user acceptance of the software product. It is therfore essential that these factors be explicitly defined at the outset of development.

It is particularly important that performance requirements be testable; in each case, the means of determining that the requirement has been satisfied should be stated.

6.1 - Availability Requirements

Describe the required probability that the software element is in readiness to perform its function. This figure should define real availability to the end user, which takes into account terminals, lines, controllers, intermediate processors, host machines, and necessary software elements (e.g., a specific OS and DBMS). Availabilty figures should be derived form operational user need; it may be desirable to relate them to system capabilities.

6.2 - Timing Requirements

Timing constraints on the software element's operation are defined. Timing factors that may be specified include:

response time
maximum allowable time from occurrence of a system stimulus to the system's response. Response time is usually expressed in terms of its probability distribution, rather than a single value (e.g., 95% of responses occur in less than 1 second). For batch processes, response times refer to the allowable time from initiation to completion of a task.

frequency
cyclic rate of occurrence, usually expressed in cycles/second (Hz).

jitter
allowable variation in cyclic rate (eg. +/- 200 msec).

currency
age of the information used or produced (a.k.a. data staleness)

time homogeneity
requirement that specific data samples be coincident in time.

The conditions under which these are to be measured should also be identified.

6.3 - Accuracy Requirements

Specify the required accuracy and precision of the software element's outputs. This usually applies to numerical software products, but may be needed in other areas as well.

6.4 - Capacity Requirements

6.4.1 - System Capacities: Defines the capacity of capabilities **provided by** the software element. It specifies required sizes of objects that are supported by the software element (e.g., table sizes, number of devices, concurrent users, etc.).

6.4.2 - System Resources: Defines the capacity of system resources **utilized by** the software element. It specifies required CPU power, disk storage, I/O Bandwidth, or other resources that are necessary for the software element to meet its performance requirements.

Section 7 - Adaptation Requirements

This section defines how the software product can be adapted to the variety of its actual use. It includes possible configurations, customization, tuning, and future growth of the software element.

7.1 - Configuration Requirements

This section defines the variety of configurations that the software product must take:

7.1.1 - Target System Configurations

7.1.2 - Required Subsets

7.2 - Customization Requirements

7.2.1 - Application Tailoring

7.2.2 - Installation Tailoring

7.2.3 - User Tailoring

7.3 - System Tuning

7.3.1 - Instrumentation Requirements

7.3.2 - System Parameters

7.4 - Provisions for Future Growth

Section 8 - Qualification Requirements

This section specifies the qualification methods to be used to ensure that each of the requirements has been satisfied. Major methods of qualification include:

Inspection	visual examination of the product
Demonstration	relies on observable functional operation
Test	relies on instrumented operation and analysis of the results
Analysis	engineering assessment, involving interpretation or extrapolation of accumulated data

Other qualification methods may also be defined for unique purposes.

Section 9 - Support Requirements

9.1 - Product Delivery and Installation

9.1.1 - Preparation for Delivery: This section specifies the form and medium of delivery, labeling, packaging, and handling.

9.1.2 - Product Distribution

9.1.3 - Installation Requirements

9.2 - Logistic Support Requirements

9.2.1 - Product Update

9.2.2 - Data Maintenance

9.3 - Training Requirements

Section 10 - Requirements Traceability

This section demonstrates that all allocated higher-level system requirements that have been allocated to this software element are satisfied.

Section 11 - Notes

General information that aids in understanding of this specification (e.g., background information, glossary, formula derivations). This section does not contain requirements.

Appendixes

Supplemental information which is referenced in the body of the document, but is separate for ease of document maintenance.

Application by Product Type