EXPERT SYSTEM VERIFICATION AND VALIDATION STUDY

ES V&V Workshop

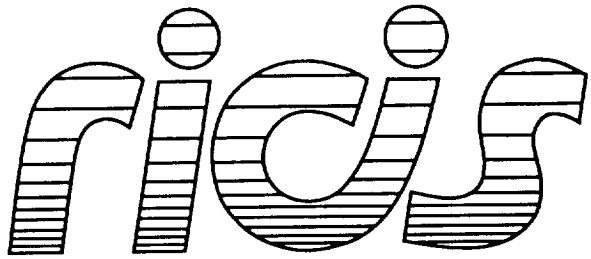
Scott French
David Hamilton

International Business Machines Corporation

March 1992

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

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



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

INTERIM REPORT

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

ES V&V Workshop

Preface

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

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 technical monitor for this activity was Christopher Culbert, Chief, Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of RICIS, NASA or the United States Government.

Expert Systems V&V Guidelines Workshop:

Part One: Verification and Validation of Software

By Scott W. French and David O. Hamilton

This section is part of the Expert Systems V&V Workshop. It summarizes conventional approaches to verification and validation of software. The material provided here will explore general approaches in applying verification and validation to conventional procedural software implemenations. For more specific details on topics and techniques please refer to the attached references.

Table of Contents

Introduction

Overview

The primary purpose of this document is to build a foundation for applying principles of verification and validation of Expert Systems. To achieve this end, some background discussion of verification and validation (V&V) as applied to conventionally implemented software is required. Part One will discuss the background of V&V from the perspective of (1) what is V&V of software and (2) V&V's role in developing software. Part One will also overview some common analysis techniques that are applied when performing V&V of software. All of this material will be presented based on the assumption that the reader has little or no background in V&V or in developing procedural software.

As the discussion of Part One unfolds, some additional aspects of software development will be mentioned, but not discussed in any detail. These aspects will be highlighted now.

V&V can be characterized as the application of a collection of techniques in a manner whose goal is to show that a piece of software has been built correctly and solves the right problem. Expanding on the goal of V&V as just outlined Part One will demonstrate that based on some insights into how software is developed, application of V&V tehcniques is most effective when applied in a logical sequence of tasks. Based on this conclusion, therefore, one can infer that a process can be applied to software development. This process can be referred to as a *life cycle*. There have been many life cycle models proposed and appendix E shows some examples (these examples were adapted from the cited references) of just a few. Part One will not attempt to dictate which of these models is the best to use (different projects may need to use different models). Rather, Part One will focus on the specifics of each V&V task. Application to the life cycle model your project chooses should be simple based on this discussion of V&V tasks.

Another aspect related to the discussion of V&V found in Part One relates to software design. Part One will demonstrate that implementing V&V tehcniques is much simpler when software has been designed correctly (or in other words, has been designed with V&V in mind). Those aspects of software design that are most impacted by V&V will be discussed in Part One However, this will not cover the more general topic of how to design a software system. Regardless of this omission, it should be clear that V&V and software design are very similar activities and that each directly impacts the other.

Goals

After reading Part One of this document it should be clear that (1) V&V should be done, (2) V&V works best when performed as the system is developed and (3) the system can be developed in a way that makes V&V easier.

First, Part One will show that V&V should be done. Unfortunately, V&V often succumbs to the myth of being an activity that is excess baggage to the

software development process. It is seen as an activity whose impact on developing timely, cost-effective software is too great. In reality, nothing could be farther from the truth. V&V will be seen here as an activity that actually reduces cost and improves productivity by focusing on finding errors earlier and satisfying customer desires. This results in software systems that are more reliable and user-friendly.

Next, Part One will show that V&V works best when it is performed as the system is built. V&V is NOT the final step in a software development project, nor is it the first step or somewhere in between. Rather, it is a collection of activities performed throughout the software development process. Part One will demonstrate this be relating specific V&V tasks to specific development tasks.

Last, Part One will show that the way a system is developed can make the job of applying V&V much easier. Special emphasis will be given to selecting and performing those V&V tasks that can be done early in the process where errors are simpler and cheaper to correct. Given that emphasis, one will be able to conclude that V&V, rather than adversely impacting cost and productivity during software development, actually lowers overall cost and improces overall productivity during software development.

The Verification Puzzle

What does it mean for software to be correct? On the surface, this seems like an easy question to answer. However, upon further examination of the nature of software it quickly becomes obvious that this is not easy to answer. There are many aspects of software that make showing correctness difficult. Customers are often either unable to adequately explain what functions the system must perform or unable to decide what functions are wanted. This leads to an environment where software development is in a constant state of flux.

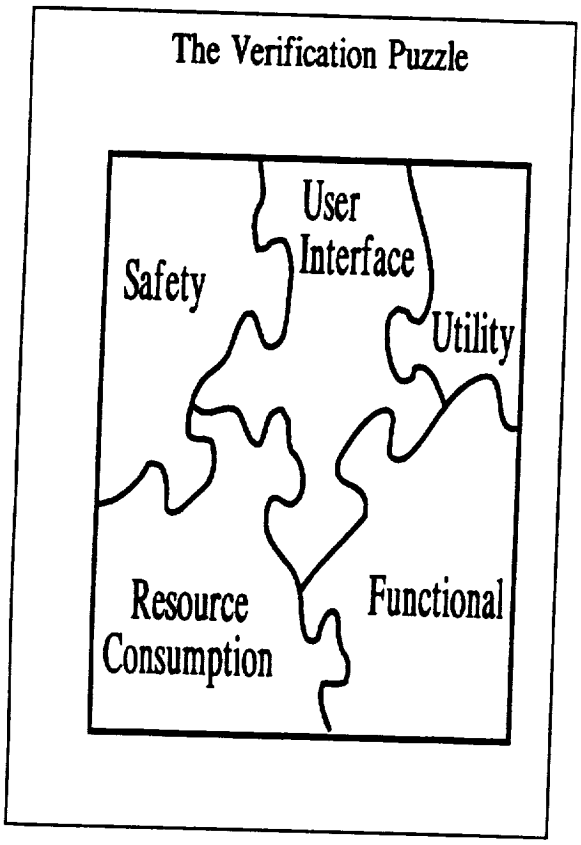
Unfortunately, even if this problem were to be resolved so that exact specifications were available and these specifications did not change during development, complete correctness would not be possible to demonstrate. During the early years of software development, software was demonstrated to be correct via dynamic testing (i.e., executing the software on a computer within its target environment). At first, this seems reasonable, but it reality it is not, because this only demonstrates that errors exist. It does not give the development team confidence that no additional errors (i.e., those not uncovered by testing) are in the software. Yet, this confidence was needed. So to achieve that level of confidence inordinate amounts of time were spent at the end of the development cycle doing testing.

It soon became apparent that this approach was not sufficient. Software systems were becoming too large and complex to wait until the last step of the process to try and demonstrate correctness. Better techniques were needed to help in finding errors earlier in the process when they are easier to correct.

As a result, much has been written and studied over recent years on better ways to demonstrate that software is correct. The ideas of verification and validation have grown from this work.

Verification and validation is a two part approach to demonstrating correctness. The first part focuses on shwoing that the software is being built correctly (i.e., *verification*). The second part focuses on showing that what is being built satisfies the customer (i.e., *validation*). The remainder of thise section will focus on both of these approaches and refer to them collectively as verification.

Figure 1—Pieces of the Verification Puzzle



How should a developer begin in verifying software? It turns out that the approach to verifying software is a lot like a puzzle composed of several pieces (see 1 on 6). Each of the pieces of the verification puzzle focus on a different part of correctness. Showing that a software system is correct, therefore, must include approaches that solve each part of this puzzle. The primary pieces of the puzzle are:

- Functional Correctness
- Safety Correctness[17,18]
- User-Interface Correctness
- Resource Consumption Correctness
- Utility Correctness

Functional Correctness: The first piece to this puzzle, *functional correctness*, is probably the most widely known and practiced. It also includes the largest number of techniques. functional correctness is concerned with documenting the expected behavior of a piece of software (i.e., the software should, given input X, generate output Y) and then showing that the software's implementation matches that specified behavior.

Safety Correctness: The next piece of the puzzle, *safety correctness*, involves showing that there are not conditions under which the software reaches an unsafe state of operation (e.g., a state where danger to man or machine is imminent). One reason this piece is so important is the inherent complexity involved with implementing the other pieces of verification. When software complexity grows, it may actually be easier to show that the software does not perform any unsafe operations than it is to show, from a functional correctness standpoint, that the software will always do the right thing.

User-Interface Correctness: The next piece, *user-interface correctness*, focuses on the human factors involved in using a piece of software. This aspect of vericiation (based on our earlier

definitions, this piece falls quite nicely into the validation category) was rarely considered in the early days of verifying software. The technology did not exist to provide sophisticated interfaces. Software was typically executed in a batch fashion with minimal user interface. However, over recent years, the advances in both software and hardware technology have created an environment that is rich in software that stretches the limits of user interface. These interfaces must be correct and user-friendly.

Resource Consumption Correctness: The next piece, *resource consumption correctness*, focuses on how well the software operates on a computer within its target environment. For example, can a real-time software system meet its scheduling constraints when it runs on a given processor. Other factors such as disk usage, transmission bandwidths, etc. must also be considered when performing this type of correctness testing.

Utility Correctness: The final piece, *utility correctness*, focuses on the overall software solution. This step is primarily a validation step. Does the software provide a solution that truly satisfies the user's needs? It is not unreasonable (nor unheard of) that a user would reject a software system that matches its specification. This is one reason why developing software can be so difficult and frustrating. It should also be clear that testing for this kind of correctness is not very *scientific* and for that reason is very difficult to do. It is a task that depends heavily on personal skills more than technical skills (e.g., how well do the development and customer teams work together to define system requirements).

Aspects

What does a developer look for when demonstrating that each of these pieces of verification is satisfied? There are three primary aspects to demonstrating correctness: *consistency*20, *completeness20* and *termination19*.

Consistency involves many things both internal and external to the software. Demonstrating consistency involves looking at each aspect of the software's definition with respect to all other aspects of that definition to determine if these aspects are consistent.

With respect to the external behavior of the software, this may involve, for example, showing that when a certain kind of input is received (e.g., when an operating system receives a hardware interrupt) the system responds in a consistent fashion (e.g., an interrupt handler is invoked). Showing external consistency also involves looking at the user interface to determine whether its *look and feel* is consisten (e.g., consistent use of colors, consistent use of function keys, etc.).

Demonstrating internal consistency involves showing that the pieces used to build the software system (refer to "Unit/Integration Testing" for a complete description of these pieces) are consistent. For example, one might verify that a variable that is declared to be integer is never assigned a non-integer data value. One might also check that this variable is used in a consistent manner (e.g., the variable is supposed to be used to index a specific

array and, therefore, it should only be used to index that array).

Completeness (sometimes completeness is referred to as *closure*) is more difficult to demonstrate than consistency. Consistency works with parts of the system that have already been defined. Completeness, however, seeks to determine if any parts of the system are missing. Looking for missing things is difficult because one is not always sure what to look for.

Some key criteria for focusing this search for the complete solution are:

- accepts all required inputs
- generates all required outputs
- performs all required actions
- maintains all required data

Even though demonstration of completeness is more difficult that demonstrating consistency, both of these approaches are undecidable problems (i.e., no algorithm exists for demonstrating either concept). Unfortunately, this means that developers can not definitively say that software is complete and consistent. The best that can be done is to demonstrate *sufficient* consistency or completeness. Fortunately, this is generally good enough to demonstrate correctness.

The final aspect of software correctness is *termination*. Mills states that "a program will be correct with respect to its specification, if and only if, for every initial value permissable by the specification, the program will produce a final value that corresponds to that initial value in the specification."[23] Consistency and completeness will show that generated outputs are correct. Yet, based on this definition, that is not enough. The developer must show that each input will, in fact, generate an output. That must mean that the program, in order to be correct, must always terminate. One part of demonstrating that a program terminates involves looking for all looping control structures and then showing that those loops terminate[19,7]. Demonstrating termination may also focus on showing that programs never terminate abnormally.

In summary, software is shown to be correct by using (1) consistency and completeness analysis to demonstrate a program generates all the right answers for all inputs and (2) termination analysis to show that the program always generates an answer.

Characteristics

The discussion so far has proceeded based on the assumption that all software systems are the same. Of course, this is not true. There is a large difference in the kinds of software developed today. There are large systems that must address a wide variety of interfaces (e.g., an operating system) as opposed to smaller self-contained systems that address a narrow problem domain. Software systems can also differ in criticality and complexity (i.e., bigger does not necessarily mean more complex). Software systems also differ in the ways they are represented (e.g., declarative versus procedural).

All of these differences indicate that verification must be approached based

on these differences. For example, software that is not man-rated (e.g., the space shuttle flight software) may not need the same level of verification as, say, a video game. In addition some of the pieces of the verification puzzle may also be eliminated. For example, a batch payroll system is probably going to have a minimal (if any) requirement for demonstrating user-interface correctness. In other cases, for example, development of sophisticated data entry software may focus more on user-interface correctness than on safety correctness or functional correctness (i.e., the user may be willing to live with less functionality if the user interface is really good; the reverse is probably not true).

In conclusion, the characterisitics of a software system impact how it should be verified. The trick is to identify the right tasks and techniques to use so that the maximum cost-benefit ratio is achieved.

Testing Phases

The verification puzzle shown in figure 1 dramatizes that *proving* software correctness involves many activities. As it turns out, the verification puzzle provides an abstract view of what is required to verify software. The pieces of the verification puyzzle can be neatly divided into some smaller puzzles. These puzzles, if you will, are:

- Dynamic Testing
- Static Testing

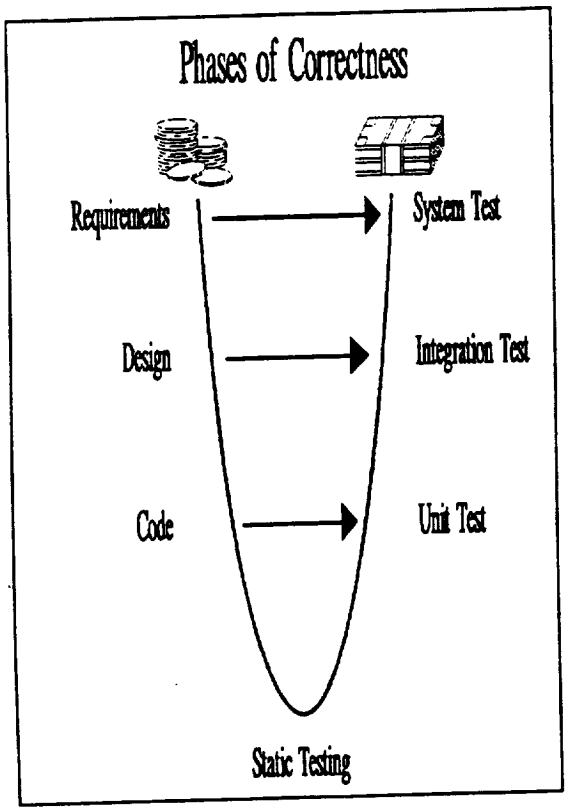
Dynamic testing involves executing software on a computer within its target environment. As will be discussed later, dynamic testing can be partitioned into two separate testing activities: *System* and *Unit/Integration*. System testing is the final activity applied to software. It focuses on demonstrating that the system meets all stated objectives by the use/customer. Errors found during this stage of development are more difficult and expensive to fix than in any other stage of software development (because its the last step).

The other part of dynamic testing is *Unit/Integration* testing. This approach to testing involves dynamic execution of small self-contained pieces of the software's internal structure in a stand-alone fashion. Techniques that apply to this kind of testing are often very time consuming. However, they are good at finding errors. This activity, because it works with the internal structure of the software, should precede system testing (which assumes the structure is complete). For this reason, erros found during this activity will be less difficult and costly to fix than during system test.

The *Static Testing* puzzle involves human analysis (e.g., desk checking, CASE, etc.) of software specifications at varying levels of abstraction. Techniques that address this puzzle are time consuming, but are by far the most cost-effective because they typically uncover the most errors and they uncover these errors much earlier in the software development process.

This section will focus on providing more detailed descriptions of each testing phase. Figure 2 maps each testing phase in relation to the other phases based on the cost of fixing errors found during that phase.

Figure 2----Testing Phases and Costs



The emphasis of figure 2 and the discussion that follows is that finding errors early in the development process is the most cost-effective approach to developing software. With that in mind, developers should focus on identifying and using techniques that meet that goal.

The sections that follow will provide an overview of each testing phase, a description of a simple example system and brief illustrations of applying the testing techniques in each phase to that example system. This overview will focus on describbing specific characteristics of a given testing phase, what inputs are required to adequately perform that testing phase and what implications there are based on both the input and output from that phase. These implications will support the notion that V&V testing phases should be applied in a logical sequence or order.

System Testing

System testing is a testing phase that does what its name implies: *test the system*. At this stage of software development, the system is considered complete (i.e., no need to worry about the internal structure of the system). There are several key characteristics of software and testing during this stage of development.

First of all, since the system is viewed as a single entity it is considered to be like a *black-box*[24,25]. Viewing a system like a black-box means that the user can not see inside the box. Related to software this means that the tester can not see the internal structure of the software.

Demonstrating the system is correct does not require knowledge of the internal system structure. The system tester is taking the view of the customer. The customer is not concerned with how the software is constructed, just that the software generates the correct responses for given inputs. It is very important that this black-box view be used at this level. Too much knowledge about the internals can lead to some false testing assumptions[24]. In fact, it is probably wise to use individuals that did not participate in developing the software implementation for system testing.

So what, then, does the system tester do? The system tester seeks to show that the system *exhibits required behavior*. What does it mean for the system to exhibit required behavior? Typically this means that the tester should specify[1] a complete list of stimuli and responses (i.e., inputs and outputs) and then test that the software generates the correct response for each stimulus. This is what it means for a system to be *correct*.

These descriptions would contain information that maps to each specific piece of the verification puzzle. For example, each documented stimulus/response pair should provide some indication of, say, the expected response time. It would be helpful if some indications were given regarding the criticality of the operation (i.e., this operation must always work or the shuttle will launch its payload before the payload bay doors have been opened). Descriptions of expected interfaces for entering the stimulus and displaying the response should also be specified.

For most systems, documenting each stimulus/response in this way is impossible. The number of pairs is infinite. This principle is true for almost all software systems. This principle also leads to a somewhat disconcerting conclusion: **exhaustive testing is not possible**. Since exhaustive testing is not possible and we know that showing a program is correct means we must show that a correct response is generated for each stimulus, it is impossible to show that a program is completely correct.

This leaves the tester with a dilemma. How do I know when enough testing (since I can't do it all) has been done to show the system is correct? Fortunately, there is an approach to doing this. This approach hinges on recognizing that *classes of stimuli*[26] exist.

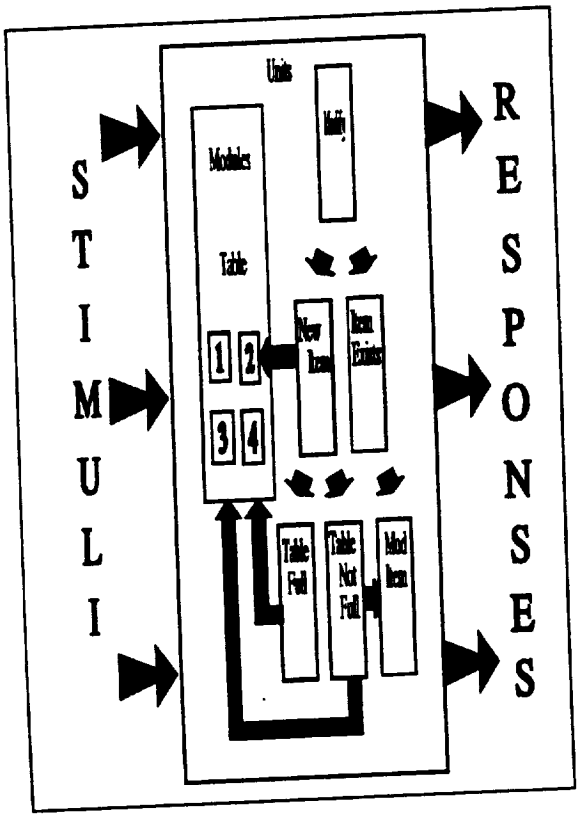
A class is a collection of items that exhibit some common properties. People take advantage of classes everyday. For example, human language uses words to denote classes. Humans can express the same idea in many ways using language (e.g., "yes", "you bet", "sure", etc.). This same notion applies to program stimuli. The advantage of recognizing this is that the tester need only test one item from the class since all items of the class should exhibit the same characteristics (i.e., responses). This significantly reduces the number of scenarios to be considered during system test.

Unfortunately, even the number of classes can be impractical to thoroughly test. The next section, "Unit/Integration Testing", will examine some ways to reduces or even eliminate some of this impracticality.

Implications

Before proceeding to the discussion of Unit/Integration Testing, there are some interesting implications to be drawn from the system testing phase. First, classes can be viewed as units. This is important because it gives the first hint as to what the internal software structure might be like (see figure 3).

Figure 3—Viewing the System as a Collection of Units



Each of these units also have stimulus/response pairs. This means that each unit can be testing independantly by applying the same system test approaches for each unit. This means that each unit will have classes of stimulus/response pairs just like the system as a whole. These classes can be viewed as units within units. The fact that there are units within units beginning at the outermost view of the system means that there must be some refinement taking place. This process of refinement is often referred to as stepwise refinement and should always be practiced so that the system can be viewed from differing layers of abstraction.

Another implication of the system test approach is that *operational scenarios*[23,24] can be identified. Operational scenarios represent a collection of stimulus/response pairs that uniquely identify operations that are the most likely to occur when the system becomes operational. Selection of these scenarios comes directly from the stimulus/response classes that have been identified. The purpose for identifying these is that these kinds of scenarios provide the basis for predict normal system performance (e.g., *Mean-Time-To-Failure*).

Unit/Integration Testing

Unit/Integration testing involves testing the pieces (i.e., units) of a system. For this reason, testing performed during this phase is often referred to as *white-box* or *clear-box*[24,25] testing.

There are some striking similarities between system testing and unit/integration testing. Unit/integration testing, like system testing, is

- dynamic (i.e., involves executing the software on a computer)
- focuses on identifying stimulus/response classes
- involves integration testing (testing how well the pieces, or units, work together)

Despite these similarities, there are some significant differences between

system and unit/integration testing. One difference is that integration testing at the system level only deals with one level of abstraction, the system level. Integration testing at the unit level may address many levels of integration depending upon the degree of stepwise refinement applied.

Another difference is that the developer must be concerned with showing the unit itself is correct. This gives an additional activity beyond testing a level of integration.

To address both of these differences the following key inputs are needed:

- the unit
- stimulus/response classes for each unit
- subsystems (i.e., different levels of integration) and their required behavior

Implications

The inputs and characteristics of this testing phase point to a very positive implication for system development. Software can be designed so that development, correction and modification is greatly simplified. How can this be? By recognizing that the software must be composed of smaller parts, the design technique, *Modularity*, can be applied.

Modularity has many meanings depending on who is spoken to. From a management perspective, modularity can be considered a *unit of work*[30]. From a programmer's perspective modularity can "refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own set of variable names."[26] Regardless of which perspective is chosen, modularity has a positive impact on performing verification and validation and should be applied to the maximum extent possible. To enhance the understanding of unit/integration testing, some of these positive aspects of modularity will be highlighted now.

Modularity and Its Benefits [19,23-25,26-29,30]

One benefit of modularity is that it can provide an effective basis for managing the development of software. Why? Because modularity allows a developer to develop small pieces of the system in a manner that keeps the separate from other pieces of the system. The word separate here implies the modules are separately compiled and have their own data space. Therefore, work assignments can easily be distributed among programmers based on modules.

Another benefit of modularity is that it can allow the system to be incrementally developed. Incremental development seeks to build larger and largers sytems by piecing together smaller systems. In other words, incrementally developed systems follow a *build as you go* approach. The cost savings associated with this approach should be clear. If only a small portion of the system is build and tested then any problems found therein require only a small portion of the system to change. Waiting until the later steps when the system is more completely defined then

there is much more that may possibly change when an error is found.

Additional savings are reaped during verification because, with modularity, verification can be done at the level of refinement. For example, if module A calls modules B, C and D to perform its function and that function changes only module A needs to be re-verified. Since, B, C and D did not change their verification status remains unchanged. Applying this principle to the scope of the system development and it can be seen that as the system becomes more and more defined the verification burden becomes smaller.

Another benefit of modularity that is related directly to the benefit just described is that modularity reduces the amount of re-verification. Since, by definition, a module isolates portions of the system so that changes to that portion do not affect other parts of the system. This can be done, in part, by defining a stable interface (i.e., a module's view of another module or as Parnas states, "... the assumptions those who write one module may make about the other modules"[30]). Changing a module without changing its interface means that all other modules that depend on that module do not need to change their *assumptions about* (or calls to) that module. This significantly reduces the modules that must change as a result of a single module changing.

Other Implications

Some more general implications can be drawn during this level of testing. First of all, unit/integration testing implies that there is a bridge that takes a much less specific description of the problem found in requirements to a very specific description found in, say, the target executable language. This bridge is the design and is necessary to provide an understanding of how the requirements, usually stated in English, were interpreted into the target executable language.

In fact, since the definition of units in the design has been drawn directly from the initial step of generating stimulus/response pairs (which are, themselves, part of the system requirements), mapping the requirements via the design to the implementation becomes a straightforward task of tracing through the levels of system refinement.

Unfortunately, system and unit/integration testing do not provide the exhaustive testing needed to demonstrate correctness. The next section, "Static Testing", will address other important testing issues not covered by either system or unit/integration testing.

Static Testing

Dynamic testing techniques focus on executing the software on a computer and then analyzing the results of that execution. Since exhaustive dynamic testing is not possible other techniques must be found to provide additional assurance that the software is correct. As it turns out, there are some kinds of analysis that humans do better than computers. These kinds of analysis do well in *filling the gaps* found in dynamic testing.

To begin to understand the value of these techniques let's look at how static testing differs from dynamic testing.

One difference is that each works from a different specification. Dynamic testing works from selection of stimulus/response pairs from stimulus/response classes. Static testing uses a variety of specifications to determine correctness (correctness means that the implementation correctly implements the specification). Some kinds of specifications prescribe specific properties of the system in a precise mathematical form (e.g., axioms)[10]. Others use a more informal, English-like style. Some specifications focus on prescribing conditions that must be true before execution of a given control structure (e.g., if-then-else, while-loop, etc.) and conditions that must be true when the control structure completes execution (*pre/post conditions*[7,8,20]).

Another difference is that static testing approaches the sytem from additional abstraction perspectives. Dynamic testing focuses on differing sizes of software (i.e., one level of refinement is bigger, or contains more units, than the next higher level of refinement). Static testing includes this view and the abstraction afforded by the software's representation. This representation may take the form of a requirements document, design, implementation, etc..

Abstraction and Refinement

Clearly, then, abstraction is one key analysis technique that differentiates static from dynamic testing. Fortunately, humans are pretty good at analyzing abstractions (unlike computers). To better understand why, let's examine some details regarding abstraction and its companion, refinement.

Abstraction is the process of simplifying a description of something by suppressing lower level details of the description. For example, the word *chair* is an abstraction. Using the word chair in a conversation allows the speaker to focus on the higher-level semantics of chairs rather than the specific details about chairs (four legs, seat, back, etc.) that everyone knows about. Using the abstraction is easier and still conveys the meaning. Abstraction also allows one to draw inferences based on the abstraction. Using the chair as an example again, if someone were to say that X is a chair and that Y is a chair, then one can infer similar meanings for X and Y.

An important corollary to abstraction is *refinement*. Refinement is a way to break apart an abstraction into its details. The process of applying refinement is often referred to as stepwise refinement. The opposite approach (building higher level descriptions from collections of lower level details) is called stepwise abstraction. Stepwise refinement is a good approach to designing a system because it closely models how a human analyzes a problem.

What does all this mean for demonstrating correctness? Look back at the discussion so far. Initially, one starts at the system view and attempts to show that is correct. However, before doing that one must show that the internal structure of the software is correct. This is stepwise refinement.

Now, coupe that fact with the fact that humans are good at analyzing abstraction and that humans can view

many more representations of the system than can the computer and there is a sound reason for applying static testing. This reasoning also has a financial aspect as well. Since humans can analyze software in any form then analysts can begin analyzing the system at its earliest point of definition well before the system is so rigid that it is difficult to change. These parts of the development are also the points where the system is least understood. For this reason, it is reasonable to expect static testing to find more errors than another phase of testing.

Why Do Dynamic Testing?

Now, one might ask why dynamic testing is needed at all. The primary reason for doing dynamic testing is to exercise code within its target environment. Without attempting to execute the software on a computer, testing would be relegated to humans exercising the software in their heads. This is definitely not sufficient to demonstrate correctness.

This is true because for various reasons. Humans can not execute large sequences of operations very quickly. Therefore, things such as response time can be projected, but not accurately measured. Humans can not handle large volumes of data (especially when coupled with trying to run this data through a sequence of steps). Humans can not emulate all the complexities of the target environment where other processes may affect how well the software performs. All of these factors contribute to the need for dynamic testing.

Implications

What does all of this mean? There must be a balance between static and dynamic testing based on the following principles:

- apply static testing during development as much as possible as early as possible (this should help avoid spending inordinate amounts of time in dynamic testing)
- use static testing primarily to prove the system satisfies all acceptance critieria

The Traffic Controller Problem

Now that each of the testing phases has been described, it is time to look at some techniques that can be used during these phases. To understand these techniques better, let's examine them within the context of the following simple 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 two 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 fifteen seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed fifteeen seconds with each new detection of continuing traffic up to a maximum of one minute.

Before discussing specific application of each test phase to this problem, begin by examining how testable this problem description is. Can this description be tested against to show that the implementation is correct?

To start answering this question one can analyze the system in terms of stimulus/response pairs. Remember from earlier discussions that the number of possible stimulus/response pairs is infinite (think about this problem and prove to yourself that this is true). Therefore, stimulus/response classes must be identified so that a workable number of scenarios can be addressed.

To help in the discussion of stimulus/response pairs for the traffic controller problem sequence expressions will be used. A sequence expression is an expression that describes a sequence of events (reading left to right). Each event is described as an ordered pair of stimulus and clock time. For example, the ordered pair, **(app-signal, t_0)**, would indicate that the approaching traffic was detected at time, t_0. Figure 4 shows some example scenarios for the traffic controller problem.

Figure 4—Sample Traffic Controller Scenarios

- { (switch-ligh, t_0),
  (, $t_1 = t_0$+120 seconds),
  (switch-light, t_0) }

- { (switch-light, t_0),
  (app-signal, t_1=t_0+1 second),
  (, $t_2 = t_1$+120 seconds)
  (switch-light, t_0) }

- { (switch-light, t_0)
  (app-signal, t_1=t_0+2 seconds),
  (, $t_2 = t_1$+120 seconds),
  (switch-light, t_0) }
  • • •

- { switch-light, t_0),
  (app-signal, t_1=t_0+n seconds),
  (, $t_2 = t_1$+120 seconds)
  (switch-light, t_0) }

- { (switch-light, t_0),
  (ped-waiting, t_1=t_0+1 second),
  (, $t_2 = t_1$+15 seconds).
  (switch-light, t_0) }
  • • •

- { (switch-light, t_0),
  (car-waiting, t_1=t_0+1 second),
  (, $t_2 = t_1$+15 seconds),
  (switch-light, t_0) }

The external stimuli found in the sequence expressions of figure 4 are defined as follows:

- *app-signal*: approaching traffic is detected
- *ped-waiting*: a pedestrian is waiting for the light to change
- *car-waiting*: a car is waiting for the light to change
- *switch-light*: the traffic light changed direction

Those ordered pairs that specify no external event represent an internal stimulus (e.g., a period of time expired). Now let's examine the scenarios in hopes of identifying stimulus/response classes.

The first scenario listed in the figured describes an sequence of events where no external stimulus occurred within two minutes, so the light changed. The next few scenarios describe a situation where approaching traffic is detected followed by a two minute period where no stimulus is received. The light then changes. The last scenarios describe situations where traffic is waiting and a period of fifteen seconds expires with no approaching traffic being detected. The light then changes.

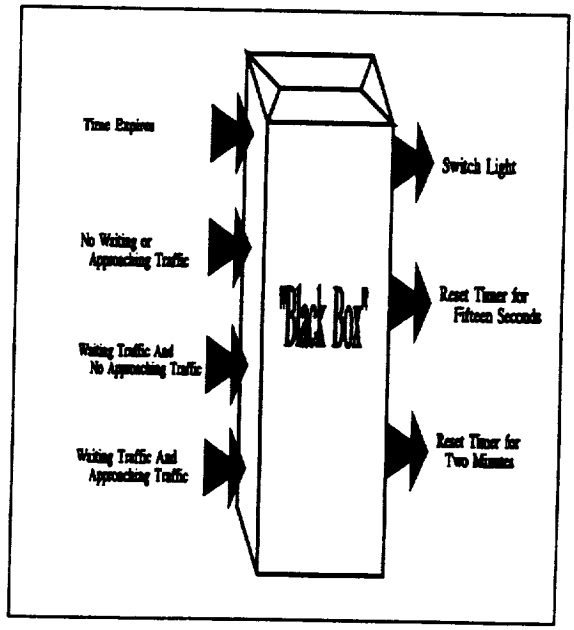
Where are the stimulus/response classes for these pairs? To answer this question examine the scenarios where approaching traffic is detected. The possible number of pairs is infinite (the interval, [0, 120], though bounded is infinite). Therefore, all combinations can not be tested. Yet, in reality, do all combinations need to be tried to demonstrate correctness? No. Regardless of whether the approaching traffic is detected one second from the last light change or 100 seconds is immaterial. The traffic controller should

exhibit the same behavior for each. Therefore a more general sequence expression defining the stimulus/response class can be written:

{ (switch-light, t_0),
(app-signal, t_1<t_0+120 seconds),
(, t_2=t_1+120 seconds),
(switch-light, t_0) }

Selecting a few cases from this class is sufficient to test the entire class of possible stimulus/response pairs. Other classes are easily evidenced from the description of figure 4. It is an exercise for the reader to determine these additional classes. Use figure 5 as your guide.

Figure 5---Initial Black-Box View of the Traffic Controller



The process followed in identifying these stimulus/response pairs may have seemed a little ad-hoc. It would be nice to use a more systematic approach. One good approach for identifying stimulus/response pairs to focus on the existance of *state*.

State refers to the notion of persistent data. Or, put another way, data that persists (has a value) over time. How does one identify state? A deeper analysis of the meaning of states provides some clues.

If state is data that will persist over time then one should be able to query the state (e.g., what is the value at time

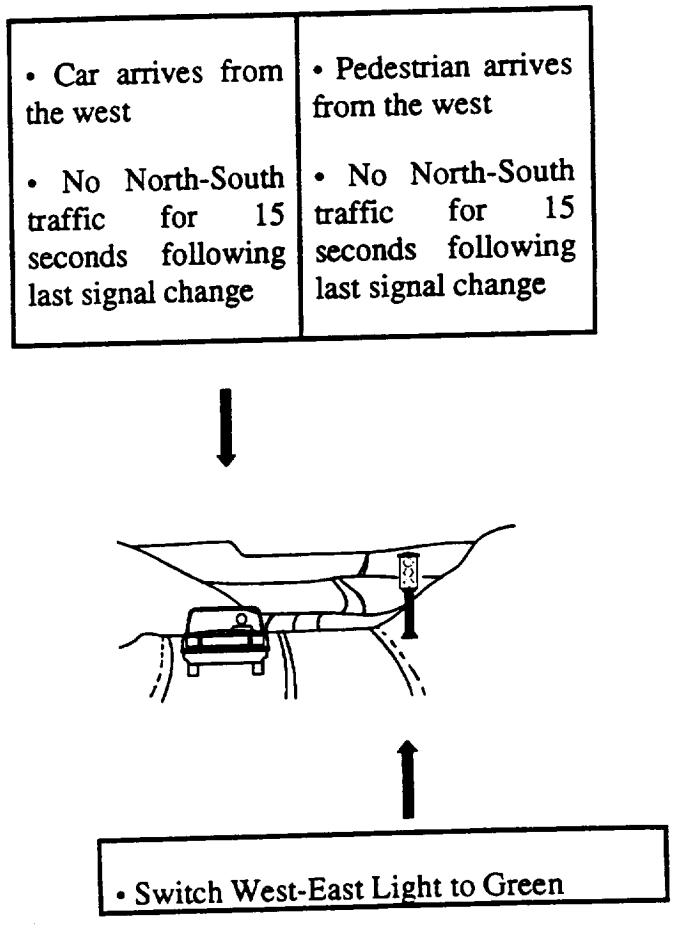
t_0). If state persists over time then there must be a method for changing or transitioning from one state to another. Likewise, there must be some way to create state (assign a value). Therefore, identifying state can be as simple as looking for classes of data whether these three categories of methods apply.

In the traffic controller problem state is easy to find based on this criteria. Each stimulus/response pair alludes to a changing light. Therefore, the light must have state since its value is queried and transitions from one state to another.

Another example of state is the clock or timer. Each stimulus/response pair ask how much time has elapsed since the last change of the traffic light?" This implies a query on some kind of timer or clock to access a value. In addition, each pulse of the clock changes the state of the clock. Given these examples of state, are there others?

Now that examples of state have been identified one can see how this helps isolate stimulus/response classes. For example, classes can be formed based on how the stimuli affect the state of the traffic light. Figure 6 shows partial results for this approach.

Figure 6---Isolating Classes Based on State



[1]These should typically be specified as a part of system requirements.

Testing Techniques

This section will focus on some common techniques for performing verification testing during the test phases described earlier. Each technique will be discussed in light of the specific phase where it applies and where it fits in the verification puzzle (see figure 1 on page **Error! Bookmark not defined.**).

General Techniques

Regression Testing[1,26]

The discussions with respect to testing have, so far, focused primarily on development of new software. However, statistics have shown that the development aspect of a software system is only a small portion of its life-span. Typically, as software is used over time the need for changing the software increases. New capabilitiies are requested, errors are found, processes become obselete. All of these point to changing the software.

What effect does changing the software have on testing? One of the trickiest tasks in testing is to test software that has changed. In part, because the unchanged parts of the software must be shown to still work the same. Performing this kind of testing is referred to *Regression testing*.

For example, assume that the traffic controller should be changed so that a pedestrian or car may have to wait up to one and half minutes for the light to change. Clearly, this change does not effect how the traffic controller responds when no traffic is waiting for the light to change. Regression testing, then, would work to show that this change in the traffic controller system would not change its response when no traffic is waiting for the light to change.

Regression testing works best when a test management tool is used to assist in capturing and retrieving test cases used to evaluate earlier versions of the software.

Prototyping

Another general technique for testing software is called prototyping. Prototyping is an approach that assists developers in gaining additional understanding of the problem to be solved. Prototyping focuses on building *scale models* of the real system. These scale models can then be evaluated by both the developer and user to test their understanding of how the system should work. This kind of approach works best when applied iteratively. What this means is that small prototypes are built early based on limited knowledge of the domain and then incrementally expanded as this knowledge increases.

Application of this approach to the traffic controller problem, for example, would probably indicate that immediately switching a red light to green is not good. The traffic needs some warning that the light is going to change so they can slow down. This observation is not in the original statement of the problem so based on the prototype the problem statement should be enhanced.

Competing Designs[16]

Another general technique capitalizes on the fact that different developers often have very different views of the system they are developing. To take advantage of this, the competing designs approach encourages different teams of developers to build their own views of the system. Once built, these views are then compared. This comparison may result in selection of one team's approach to building the system or may result in a merger of the differing system views into one coherent view. Regardless of the end result, the selection process should, at a minimum, force all developers to understand their different views to make sure the selected solution provides a complete system solution.

Independent V&V

Another organizational technique for doing V&V is called to define an independent organization that is responsible for performing V&V of the product. The term independent, as used here, refers to an organization outside the development organization. This independence removes any bias in analyzing the product that may be introduced from actaully helping build the product. This approach is typically performed at the requirements and system test level, but can be applied to any phase of the product development.

System Testing Techniques

Functional Correctness

Showing functional correctness during system testing involves demonstrating that the system generates correct responses for each input. For example, when the traffic controller does not detect any approaching or waiting traffic for two minutes after the most recent light change, then the correct response is for the traffic controller to change the light.

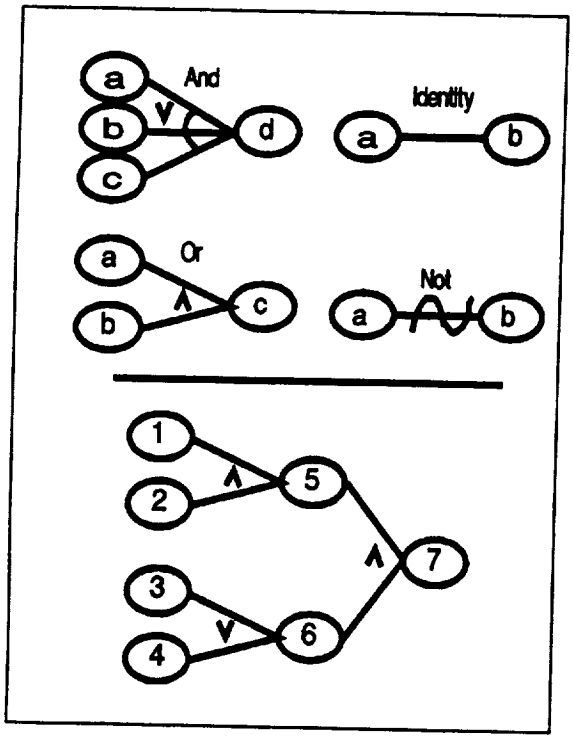
Unfortunately, as stated earlier, testing all possible combinations is not possible. Therefore, some approaches need to be used to help select from the sets of possible test cases those cases that provide the most benefit in identifying errors in the software. One of these techniques is called *Realistic testing*[32].

This technique focuses on selecting test cases based on their realism. Or, in other words, based whether the test case uses a scenario that would realistically be used during operation of the software. Additional effort should be expended in identifying and applying these kinds of cases because they are similar to how the software will actually be operationally used. For example, in the traffic controller system, one could reasonably expect that the majority of requests for a light change will be from cars and not pedestrians. Therefore, the majority of test cases that involve auto requests for light changes should be selected.

Another approach to selecting good test cases is called *Cause-Effect graphing*[26]. Cause-effect graphing examines the set of all possible stimuli and the set of all possible responses and attempts to identify *paths* from each of these stimuli to specific responses. This results in a Boolean logic network constructed from basic logic structures as shown in figure 7. The bottom of the figure shows a simple graph where the

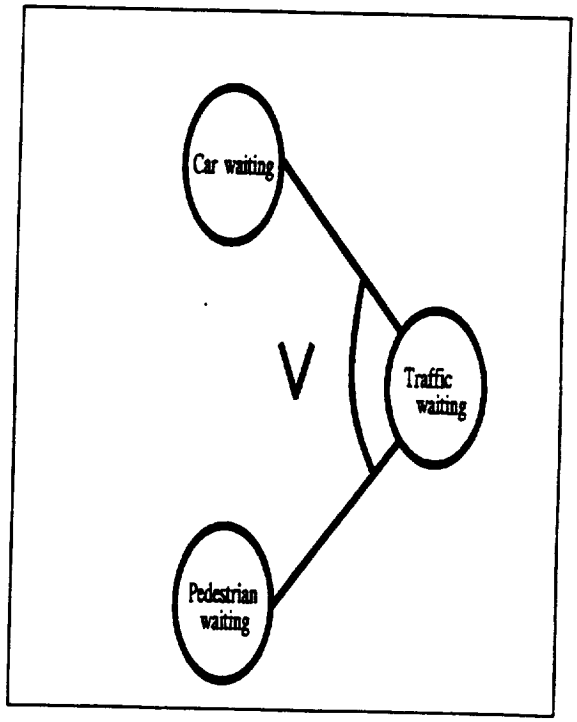
nodes on the far left are all the stimuli and the nodes on the far right are all the responses. Test cases are then selected based on these paths through the network.

Figure 7---Basics of Cause-Effect Graphing



A cause-effect graph also highlights levels of refinement and abstraction that will be useful at the unit/integration testing stage. This is because the graphing technique focuses on building many intermediate nodes when building paths from stimuli to responses. These intermediate nodes are the abstractions used in the system. For example, in the traffic controller problem either a car or a pedestrian can request that the light change. This can be viewed as an abstraction by using the *OR* structure as shown in figure 8 where the abstraction is the idea of traffic is waiting. As far as the controller itself is concerned, whether or not the waiting signal is received from a pedestrian or a car is immaterial. All that matters is that some kind of traffic requested that the light change. This abstraction then can help identify some stimulus/response classes as discussed earlier.

Figure 8---Cause Effect Graphing and Abstraction



Cause-effect graphing is not a perfect technique, because it provides little guidance on which paths through

the network might provide the most interesting results. One technique that addresses this is *Boundary testing*[26]. Boundary testing focuses on identifying test cases that exercise the boundary values between stimulus/response classes. Consider the following stimulus/response class:

{ (switch-light, t_0),
(app-signal, t_1<t_0+120 seconds),
(, t_2=t_1+120 seconds),
(switch-light, t_0) }

Boundary testing would focus on selecting specific test cases where approaching traffic would be detected at times right near t_0 and right near t_0+120 seconds.

Another approach to selecting good test cases is called *Attribute-based Test Case Selection*[32]. This technique uses attributes such as size, complexity, criticality, reliability, etc. to select test cases. For example, the traffic controller system could be considered critical since a failure in the system might cause an accident (e.g., it makes the light green in all directions). Criticality, then, is a more significant attribute than, say, size. Therefore, selection of test cases for the traffic controller will focus on picking those test cases that might cause the traffic controller itself to fail.

Other attributes of the system can be used to select test cases that may not directly relate to its functionality. For example, test cases could be selected based on some statistics regarding the intended use of the system or some other statistical record keeping technique.

A similar technique to the attribute selection method is called *Error guessing*[26]. This technique involves analyzing the system to identify functions that one might expect to have errors. This expectation can be based on many factors, both subjective and objective. Test cases are generated to address these expectations.

Sometimes, interesting results can be achieved by using no particular selection scheme at all. This approach is called *Random testing*[32] and does what its name implies. Random selection of cases from the many stimulus/response pairs and/or classes.

Safety Correctness

Techniques to demonstrate safety concerns focus on picking cases that when executed cause the software to reach an unsafe state. One technique that works rather well is *Stress Testing*[1,26,32,2]. Stress testing is primarily concerned with looking at off-nominal cases that might cause the system to reach an unsafe state (e.g., the traffic light is green in all directions). For example, what would happen if a pedestrian repeatedly hits the change light button? What happens if a power surge occurs at the same time the user pushes the button? Does the pedestrian get fried or does the light become green in all directions? These are the kinds of issues involved in stress testing.

User Interface Correctness

From the system testing perspective testing the user interface is most directly addressed by a technique called *Active*

Interface testing[32]. This technique focuses on showing that all interfaces with external entities work correctly. For example, in the traffic controller problem, external entities would include cars and people. Active interface testing is concerned , then, with showing that the interface for both cars and people work correctly. For example, can the system detect lighter weight cars? Will the system recognize the change light request if the button sticks when the pedestrian pushes that button? Questions such as these are asked and resolved as part of Active interface testing.

Resource Consumption Correctness

Resource consumption correctness is going to focus on showing that the system performs within a required level of efficiency within its operating environment. The most pertinent technique for demonstrating this type of correctness during system testing is *Performance testing*[26,32]. Performance testing focuses on choosing test that *push the envelope* of the system. Performance testing is somewhat similar to stress testing in that the tests selected should stress the system. However, performance testing will focus more on using nominal cases. An example of performance testing on the traffic controller system would be selecting a test case that will determine if delays in receiving a request to change the direction of traffic flow will adversely affect delays in changing the light. For example, what would happen at time t+14.9999 seconds when a pedestrian waiting signal was received at time t?

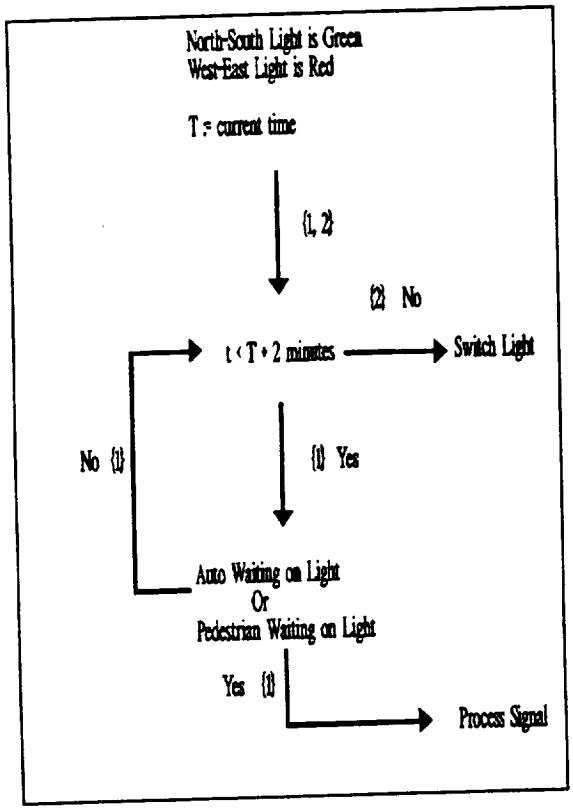
Unit/Integration Testing Techniques

Functional Correctness

Demonstrating functional correctness at the unit/integration testing phase involves demonstrating that each internal module or unit executes correctly. Associated with each of these modules is a structure or a sequence of execution steps that must be followed in order to achieve the desired result. Therefore, it seems reasonable to project that using test cases that exercise as many of these sequences as possible (i.e., coverage) would demonstrate functional correctness. In fact, research has shown that coverage techniques are the best and most comprehensive technqieus for showing functional correctness at this level of testing. There are other good techniques (and these may be better at finding certain kinds of errors) but at a minimum, coverage should be demonstrated during unit/integration testing[6].

There are three primary coverage techniques. One of these techniques is *Branch Coverage*[26]. To perform branch coverage the analyst must build a graph that shows the flow of data through the system. Figure 9 shows part of the graph for the traffic controller problem.

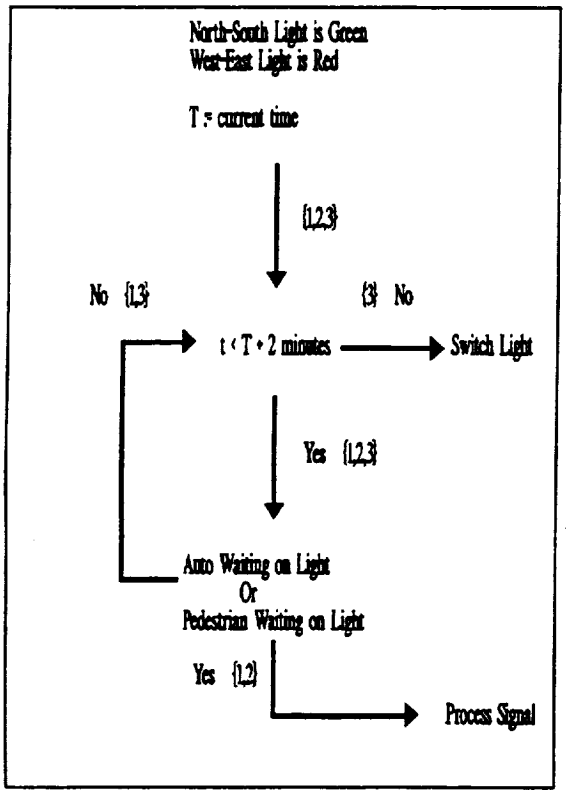
Figure 9---Branch Coverage



Branch coverage then examines the graph drawn to select test cases that follow all possible branches in the program (e.g., a branch is an arrow in figure 9). These cases will cause the program to execute each logical decision point (e.g., if-then-else). Using the graph of figure 9, two test cases are selected for covering all branches (follow the numbers beside the arrows). Describing what these two cases are is left as an exercise for the reader.

Another form of coverage is *Path coverage*[26]. Path coverage is more thorough than branch coverage because a single path may involve many combinations of branches or logical decision points. Path coverage, then , centers on showing that all combinations of branches works correctly. Figure 10 shows the same graph as that shown for branch coverage. However, selecting and executing test cases to perform path coverage results in three specific cases rather than two for branch coverage (once again follow the numbers by the arrows in the figure). Describing the specifics of these test cases is left as an exercise for the reader.

Figure 10—Path Coverage

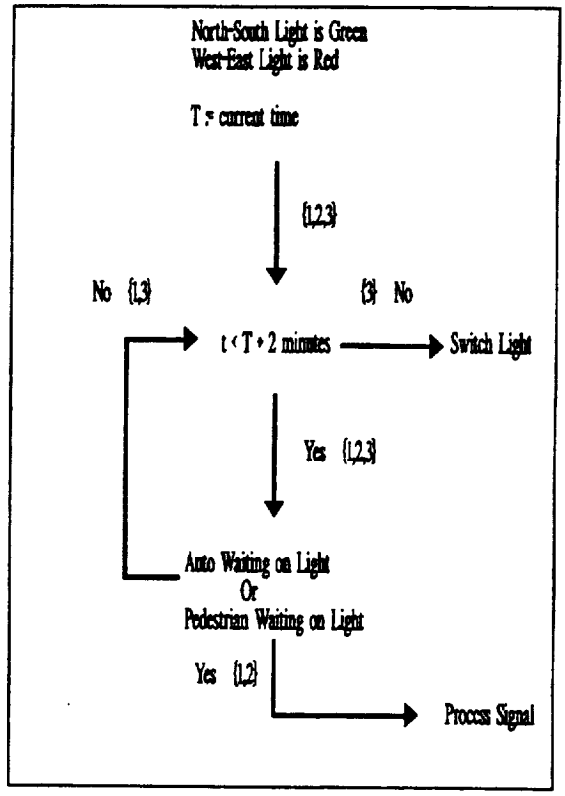


The last form of coverage testing is *Condition coverage*[26]. Condition coverage expands on path coverage to show that all combinations of logic or conditions at each decision point works correctly. For example, a logic decision point might be the statement:

If (A or B) then Perform C;

Condition coverage would attempt to show that C is performed when (1) only A is true, (2) only B is true and (3) both A and B are true. Figure 11 shows the same logic paths for the traffic controller problem. Condition coverage will generate more test cases than path coverage in this case because of the logic decision point where either a car or pedestrian could request the light to change (follow the numbers beside the arrows). Describing these test cases will, once again, be left as an exercise for the reader.

Figure 11—Condition Coverage



As always, these coverage techniques alone do not provide complete testing. In part, because they focus primarily on examining the implementation. The technique of

Partition analysis[31] attempts to merge analysis of specifications with that of implementation. This technique focuses on a kind of path analysis of the specification to identify partitions of test cases based on identify sub-specifications. These partitions are merged with those of the coverage techniques just described to form a more complete test suite.

The dynamic techniques just described are most commonly applied to individual units. This leaves a gap in dynamically testing the integration of these units. One technique that focuses on this aspect of dynamic unit/integration testing is called *interprocedural dataflow testing*[9]. This approach focuses on analyzing the interconnections between units (e.g., parameter lists) and global data. This analysis results in a *definition-uses* table that maps each global data item and unit parameter to specific statements where these are defined and used. Once this table has been defined, test cases are selected that exercise the statements listed in the definition-uses table.

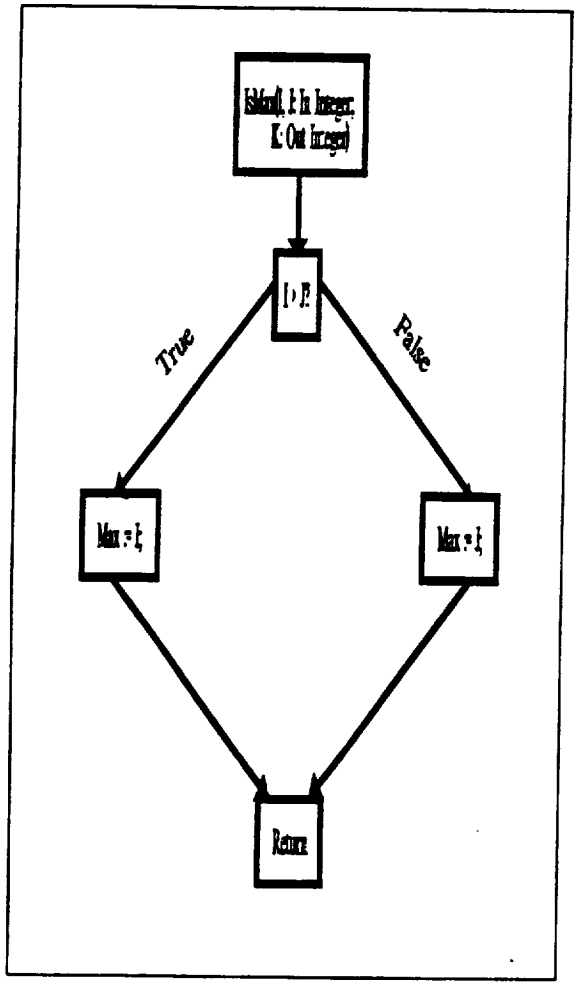
For example, consider the routine shown in figure 12 to compute the maximum of two integers. This technique will focus on analyzing the definition and use patterns of the variable Max since its value will affect the caller of IsMax.

Figure 12---Routine to Compute the Maximum of Two Numbers

```
1. Procedure IsMax(I, J: In Integer;
                   Max: Out Integer);

2.  If I > J
3.  Then Max := I;
4.  Else Max := J;
5.  End If;

6. End IsMax;
```

Having examined the routine one can build what is called a summary graph or an interprocedural flow graph that further isolates specific statements involving the parameter Max. Figure 13 shows how the flow graph might look for routine IsMax.

Figure 13—InterProcedural Flow Graph



By analyzing this flow graph one can build a definition/use table as shown in figure 14. Test cases will be selected, then, to exercise those statements where a given item in the table is used and defined.

Figure 14—Definition/Use Table for IsMax

Definition/Use Table for IsMax

Variable	Definition	Use
Max	3	6
	4	6

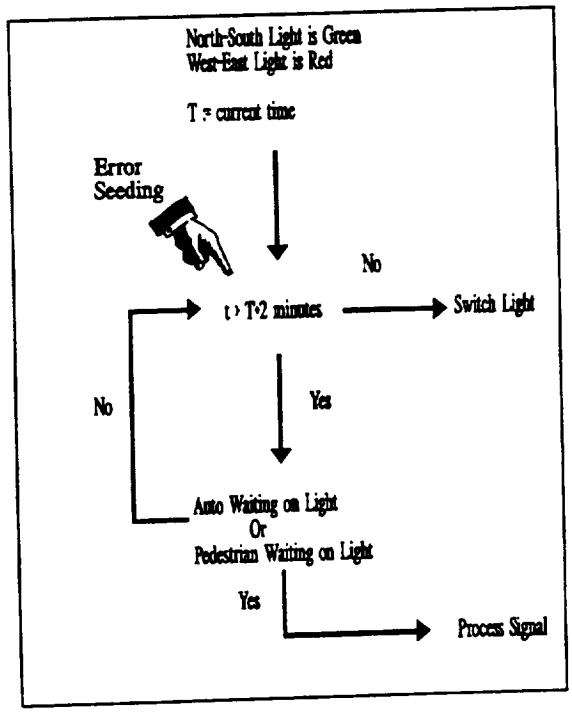
This approach is difficult to apply for large complex programs without some automated assistance. This is primarily due to the fact that parameters must be traced throughout the scope of their use. Often times this will include cases where the variable is referenced by a different name (e.g., for the call IsMax(A,B,C) the variable C is the same as the variable Max inside the routine IsMax).

Another approach for examining the integrated units (as opposed to each unit individually) is perform *flavor analysis*[13]. Flavor analysis is concerned with showing that the developers did not make any errors of omission. To do this, each developer must specify expectations about the software. Two ways to do this involve using data

comments or operator comments. Data comments use a combination of both properties and assertions about the data to model the actual construction of the program. When complete, the program can be compared against this for errors of omission. Operator comments describe a specific legal sequence of operator execution. These sequences are then matched against either human analysis or dynamic execution to determine if it matches the operator comments. Deviations in sequence highlight anamolies in the software. Further analysis may determine an actual error exists.

The approaches used so far have primarily focused on identifying good test cases. Yet, it would be helpful to know just how good these test cases are. One technique that helps answer that question is *Mutation testing*. Mutation testing is founded on the premise that a test case that is capable of demonstrating a program executes correctly ought to demonstrate that an incorrect program executes incorrectly. Therefore, this technique calls for generation of mutant programs (i.e., programs seeded with errors; see figure 15) that can be executed with the selected test cases. Those that show correct results for the mutants are not good test cases because they are not capable of distinguishing between a correct and incorrect program.

Figure 15---Mutation Testing



It should be noted, however, that mutation testing is a time consuming and risky task if performed without proper controls (e.g., configuration management). It is time consuming because the software is continually being seeded with errors and then re-instated to its original condition. It is risky because the developer always faces the risk of incorrectly removing errors from the program.

Safety Correctness

Reliability testing[32,26] is a good technique for demonstrating safety

correctness. This technique seeks to identify structures within the program that could, should they fail, adversely affect system reliability. These structures may not necessarily be error-prone. Looking at the traffic controller problem, one might wonder what would happen if the internal clock failed? Would the light simply flash red in all directions or would this failure cause the lights to be stuck in their current positions or would the lights become the same color in all directions? Any of these *MIGHT* happen. The key is to decide ahead of time which is the safe response and then test for it.

User-Interface Correctness

Testing the user interface will involve breaking the parts of the user interface into smaller pieces (this follows directly from our looking inside the black box view of the system) and testing those smaller pieces independantly. This will involve stubbing out different sections of the system to give the appearance of a complete system. Using this approach, user interface pieces can be simulated much earlier before the entire interface is complete. For example, the signal hardware could be simulated via software so that the traffic controller system could be prototyped and analyzed well before the entire system is constructed.

Static Testing Techniques

General Techniques

As trivial as this may sound, the most significant advancement in the practice of software verification is the *inspection*[5,26]. Inspections introduce an *active verification frame of mind* (e.g., sitting at a desk for the sole purpose of analyzing a work product for errors). Sound psychological evidence suggests (and, in fact, this has been shown true in practice) that this significantly reduces the introduction of errors and increases the teams confidence in the quality of the software. With this in mind, let's examine what inspections are all about.

Inspections fall into two categories: formal and informal. In either case, inspections are an approach to showing correctness by forcing a team of people involved with a given work product to inspect each work product for errors within the framework of some rules.

Kinds of Errors Caught by Inspection

Participants in an inspection will analyze a given work product to identify major errors, minor errors and suggestions. Both major and minor errors indicate the work product must be changed before the inspection is complete. Major errors address a work product that does not satisfy its intended function (or specification). Minor errors typically address non-function errors such as a violation of some pre-determined programming standard (e.g., variable naming conventions, etc.). Suggestions do not identify errors at all, but rather, indicate alternative solutions to the one being inspected. Often times suggestions speak to maintenance and efficiency aspects of the product.

Roles in an Inspection

Each person that participates in an inspection has a role. Typical roles are

those of moderator, developer and peer. The moderator is responsible for seeing that the inspection follows the rules. The developer is the person responsible for completion of the work product. A peer is someone else that has some involvement in the product (e.g., requirements writers, designers, project leads, etc.).

From the group of peers come two additional roles: reader and backup. The reader is responsible for guiding the inspection team through the work product during the inspection. The backup (who is usually also the reader) is someone who is not primarily responsible for completion of the work product, but is expected to have a level of knowledge about the work product that is at or just below that of the developer.

Inspection Rules

Each inspection must be governed by a set of rules. These rules serve to (1) allow measurement of the process (e.g., process error rate), (2) place accountability for the quality of the product at the team, not individual, level (i.e., egoless programming) and (3) prevent any issues from being unresolved prior to completion of the work product. Some example rules might be:

- work products must be distributed for review four days prior to inspection (i.e., give inspectors adequate time to inspect)

- a moderator must cancel the inspection if adequate prepartion has not been done by the inspectors

- the work product can not be released until its inspection is complete

One problem that always seems to arise when scheduling inspections is the size of the work product. One approach to handling the size of a work product to be inspected is *continuous inspections*. Continuous inspections focus on inspecting the work product as it is incrementally developed. For example, when a program function is being stepwise refined, there should be an inspection at each level of refinement. This approach obviously means more inspections, but there is less to review at each inspection and the later inspections are more error-free.

Continuous inspections involve primarily a developer and a peer. The peer is assigned a specific time each for inspecting the developer's work product.

Informal Inspections

Inspections can also be informal. Informal inspections (or *walkthroughs*) are informal because they use a less rigid format for inspection. Sometimes this means modifying some of the rules regarding inspections (e.g., only small changes can be reviewed informally) or possibly inviting fewer people to participate in the inspection. Usually, the less rigidity of informal inspections, means that there is no formal meeting of the inspection team *review* the work product.

Functional Correctness

Static testing techniques that demonstrate functional correctness fall into two categories: specification-based and structure-based.

Specification-based techniques focus on analyzing the specifications (or design) of the software to find errors. One technique that does this is called *Anomaly analysis*[32,2]. Anomaly analysis looks at sequences of events (as derived from the software's specification) to find anomalies. For example, a sequence of events might involve setting an entity and then using that entity. A sequence where that entity is set and then set again before it is used is an anomaly. Anomalies do not necessarily mean errors. They just indicate areas where there might be errors since these are not proceeded in the expected sequence.

A similar approach to anamoly analysis is *Defect Analysis*[32]. This techniques looks through the software to make sure that no general kinds of defects (e.g., divide by zero, index of the end of an array, etc.) exist.

Stepwise Refinement[27-28, 23-25] is a different kind of specification-based technique based on inspecting differing levels of abstraction. In this approach one checks to make sure that each level of refinement correctly and completely describes the previous level.

Another specification-based approach is *pre/post condition analysis*[7,20]. This analysis approach works with stated conditions within the design. The technique works to make sure the pre-condition adequately guards the associated implementation (e.g., if the implementation divides one variable by another, then the pre-condition ought to indicate that the variable acting as the divisor must not be zero) and that execution of the implemenation satisfies the post-condition.

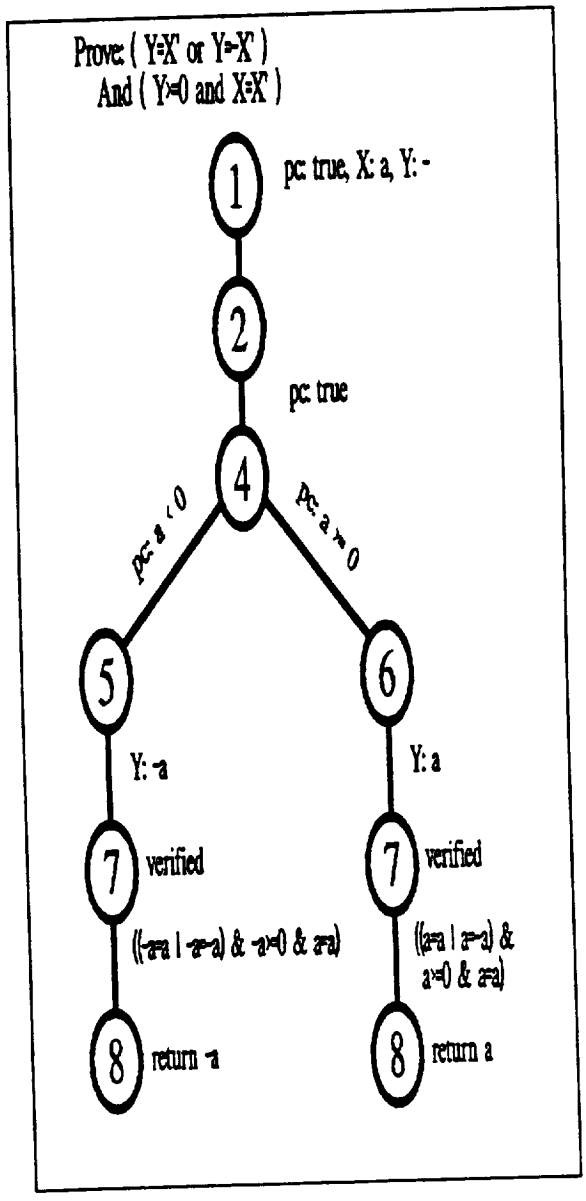
One approach that assists in analyzing pre/post conditions is *symbolic execution*[8,11]. This technique involves building a statement tree. Then, beginning with the pre-condition, mathematical symbols are used to trace through all possible paths in the statement tree. Mathematical symbols are used in place of trying to pass every possible value through the statement tree. The results of this trace are then matched agains the post-condition to determine whether the execution is correct. For example, consider the simple routine of figure 16.

Figure 16—Routine to Calculate Absolute Value

```
1. ABSOLUTE:
   Procedure (X);

2. Assume (true);

3. Declare X,Y: Integer;

4. If X < 0
5.    Then Y := -X;
6.    Else Y := X;

7. Prove (Y=X' Or Y = -X') And
          (Y >= 0 And X = X')

8. Return (Y);
9. End;
```

The pre-condition for routine Absolute is expressed in the *Assume* clause. Specifying the pre-condition as *true* is intended to indicate that any integer value can be processed. The post-condition appears as the *Prove* clause. Symbolic execution will show that this clause is satisfied by tracing mathematical symbols through the statement tree of figure 17.

Figure 17—Symbolic Execution for Routine Absolute



Another specification-based technique is *Axiomatic analysis*[7,14]. This technique seeks to match an

implementation against a specified collection of system properties or axioms.

Another type of functional correctness technique is structure-based. Structure-based techniques examine the architecture of the software for any errors. One structure-based technique is *Object-oriented Analysis*[36, 22,15]. This technique works on software that has been designed based on the object-oriented approach of building modules based on the data or state. Each object, then, can only process certain allowable or legal values. This serves as the foundation for analyzing the correctness of objects by showing that a given operator for the object can accept only legal inputs and produce only legal outputs. This can be as simple as analyzing one operator to analyzing many combinations of operators.

Applying Object-oriented design to the traffic controller problem might lead to the identification of a *timer* object. Associated with that timer object would be operations to *Reset* the timer and *Decrement* the timer. Given that the timer will only be used when it has a value greater than zero (i.e., the timer has been reset). Once the timer reaches zero is considered to be expired and is no longer used. Therefore, a general principle can be stated that the *timer* can only hold non-negative values.

Next, each of the operators, *Reset* and *Decrement*, should be analyzed to see if this principle is satisfied. *Reset* is responsible for setting the timer to a specific window of time that should elapse (e.g., 120 seconds). *Decrement* will be used to mark the passing of time (one clock pulse at a time) as long as the timer has a value greater than zero. Therefore, these definitions would indicate that, indeed, *Reset* and *Decrement* guarantee that timer will never have an illegal value.

Another structure-based approach for showing functional correctness applies when modular programming languages are used. This technique, called *Compilation testing*[32], is used with languages whose compilers can do certain checking across modules. For example, programs written in the language Ada using its package constructs can use the Ada compiler as a checking mechanism to show that the architecture of the packages is consistent.

Safety Correctness

Static testing that help demonstrate safety correctness focus on identifying potential problems and then analyzing how the system would respond should those problems occur. The first technique, *Hazard analysis*[17,18], involves identifying undesirable situations. For example, the most undesirable situation in the traffic controller system would be for all light to be green in all directions. Once identified, each hazard is analyzed to determine how it could happen. Once this scenario has been defined, the software is analyzed to determine whether this undesirable condition could ever occur. If it could occur, then the software should be modified to prevent that from happening (e.g., the software could perform a quick status check on switching hardware to make the lights can really be switched before trying to switch the light).

A similar technique is called *Fault analysis*[17,18]. A fault is considered to be any potential error in the system. This is a slightly different approach from Hazard analysis in that one is not looking at effects, but at possible stimuli for failures. Once these faults are identified then the software can be analyzed to determine what effect each fault would have. Any faults that would produce an undesirable effect need to be addressed in the software.

Summary

The purpose of Part One was to give insight into the current practice of verifying and validating software. To summarize the material presented, key points made during the overview will be reviewed along with key principles regarding the many testing techniques discussed.

Key Points

Part One presented several key points that are important when applying verification and validation approaches to software. First, each term refers to different parts of showing software is correct. Verification works to show the software was built correctly. Validation works to show the right product was built. Both of these are required to show the software is correct.

Second, testing software must be done both dynamically and statically. Static testing uses the skills of human analysis to find errors early in the software development process. Dynamic testing involves execution of the software on a computer. It has two distinct parts: unit/integration and system. Each of these addresses the many levels of structure and sub-structure associated with the software.

Third, verification and validation are analysis tasks that focus on showing software is consistent, complete and terminates.

Fourth, principled use of abstraction and refinement helps manage the scope of detail involved with any software system. Abstraction allows varying levels of detail to be expressed so that the analyst is not overwhelmed with lots of detail too soon. It also narrows the focus of any analysis to two basic levels, the abstract level and the refined level (i.e., $level_n$ and $level_{n+1}$).

Last, modularity is an essential part of building systems that are easy to extend and modify (not to mention build). Modularity allows developers to cleanly separate a systems many parts using a *divide and conquer*-type strategy.

Comments on Testing Techniques

Many different techniques for testing were presented in this document. These are, by no means, the complete list of techniques that are available for use (refer to *34*, *2* and *32* for more complete lists of testing techniques). Hopefully, the discussion of these techniques has made it clear that (1) no testing technique is sufficient by itself to demonstrate correctness, (2) choosing which techniques to use is both difficult and important to do (never leave the *how* of testing to chance) and (3) a logical sequence or order exists for testing and should be followed (i.e., pick a life cycle and follow it).

Regardless of the techniques or testing phases being performed the same general principles apply:

- look for different categories of errors (e.g. look for the "weak links" in the system[6])
- select techniques that help find errors early (and devote most of the development effort to applying those techniques)
- "programs should be structured so logical testing of various abstractions of the program can reduce actual testing of the final program"[6]
- "specifications must be precise enough to be testable"[6]

Appendix A: References

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

2. 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.

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

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

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

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

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

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

9. 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.

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

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

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

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

Appendix A: References ...

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

15. 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.

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

17. 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.

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

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

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

21. 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.

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

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

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

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

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

Appendix A: References ...

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

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

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

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

31. 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.

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

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

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

35. 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.

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

Appendix B: Techniques Vs. Phases

Techniques	Phases			
	General	System	Unit	Static
Active Interface Testing		✓		
Anomaly Analysis				✓
Attribute-Based Test Case Selection		✓		
Axiomatic Analysis				✓
Boundary Testing		✓		
Branch Coverage			✓	
Cause-Effect Graphing		✓		
Competing Designs	✓			
Compilation Testing			✓	
Condition Coverage			✓	
Defect Analysis		✓		

Appendix B: Techniques Vs. Phases ...

Techniques	Phases			
	General	System	Unit	Static
Error Guessing		√		
Fault Analysis				√
Flavor Analysis			√	
Hazard Analysis				√
Independent V&V	√			
Inspections				√
Interprocedural Dataflow Testing			√	
Mutation Testing			√	
Object Oriented Analysis				√
Partition Testing			√	
Path Coverage			√	

Appendix B: Techniques Vs. Phases ...

Techniques	Phases			
	General	System	Unit	Static
Performance Testing		✓		
Pre/Post Condition Testing				✓
Prototyping	✓			
Random Testing		✓		
Realistic Testing		✓		
Regression Testing	✓			
Reliability Testing				✓
Stepwise Refinement				✓
Stress Testing		✓		
Symbolic Execution				✓

Appendix C: Techniques Vs. Correctness

Techniques	Kinds of Correctness					
	General	Functional	Safety	UI	RCC	Utility
Active Interface Testing		✓				
Anomaly Analysis		✓				
Attribute-Based Test Case Selection		✓				
Axiomatic Analysis		✓				
Boundary Testing		✓				
Branch Coverage		✓				
Cause-Effect Graphing		✓				
Competing Designs	✓					
Compilation Testing		✓				
Condition Coverage		✓				
Defect Analysis		✓				

Appendix C: Techniques Vs. Correctness ...

Techniques	Kinds of Correctness					
	General	Functional	Safety	UI	RCC	Utility
Error Guessing		•	✓			
Fault Analysis			✓			
Flavor Analysis	✓					
Hazard Analysis			✓			
Independent V&V	✓					
Inspections	✓					
Interprocedural Dataflow Testing			✓			
Mutation Testing			✓			
Object Oriented Analysis		✓				
Partition Testing		✓				
Path Coverage		✓				

Appendix C: Techniques Vs. Correctness ...

Techniques	Kinds of Correctness					
	General	Functional	Safety	UI	RCC	Utility
Performance Testing					✓	
Pre/Post Condition Testing		✓				
Prototyping	✓					
Random Testing		✓				
Realistic Testing		✓				
Regression Testing	✓					
Reliability Testing			✓			
Stepwise Refinement		✓				
Stress Testing		✓				
Symbolic Execution		✓				

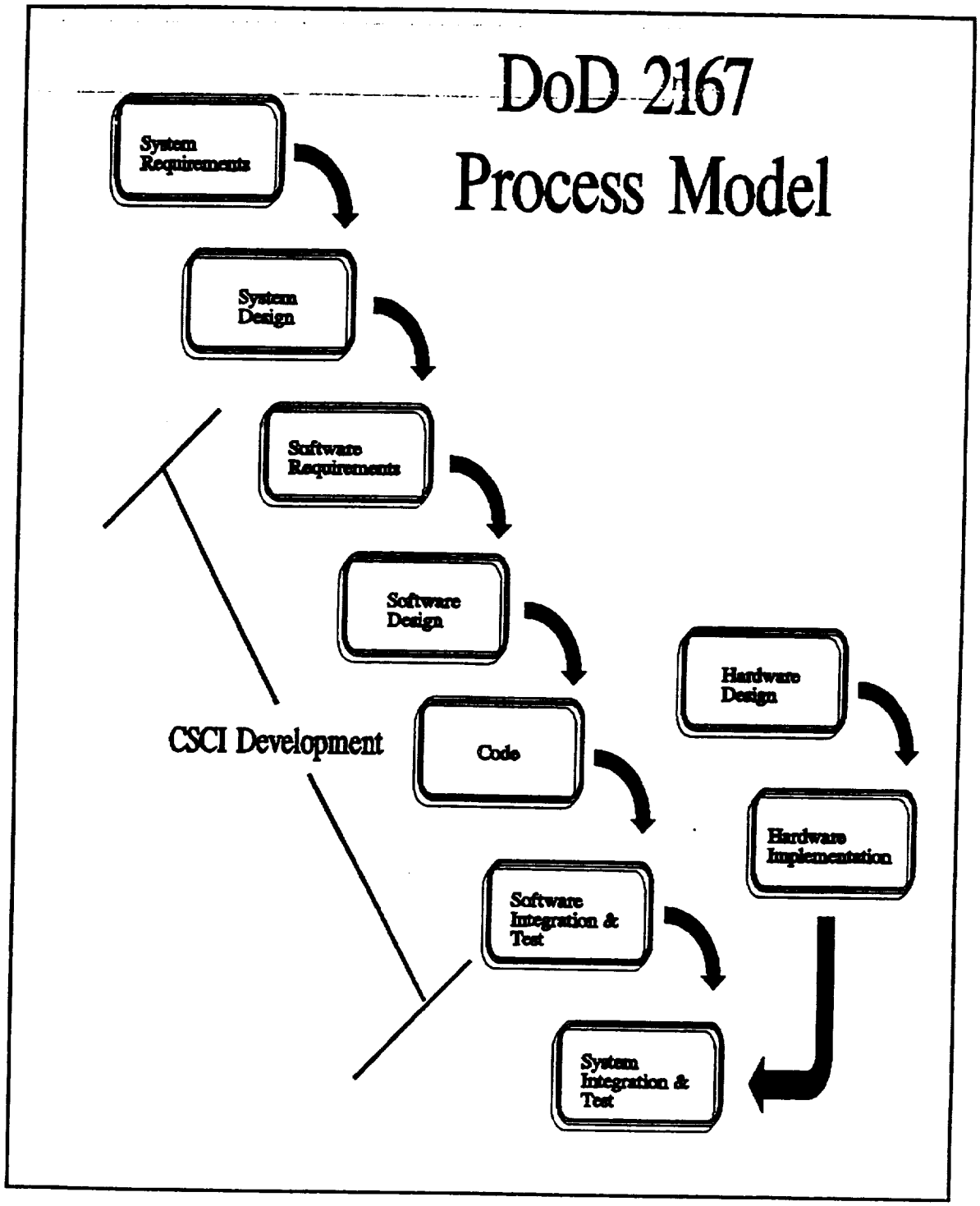
Appendix D: Techniques Vs. References

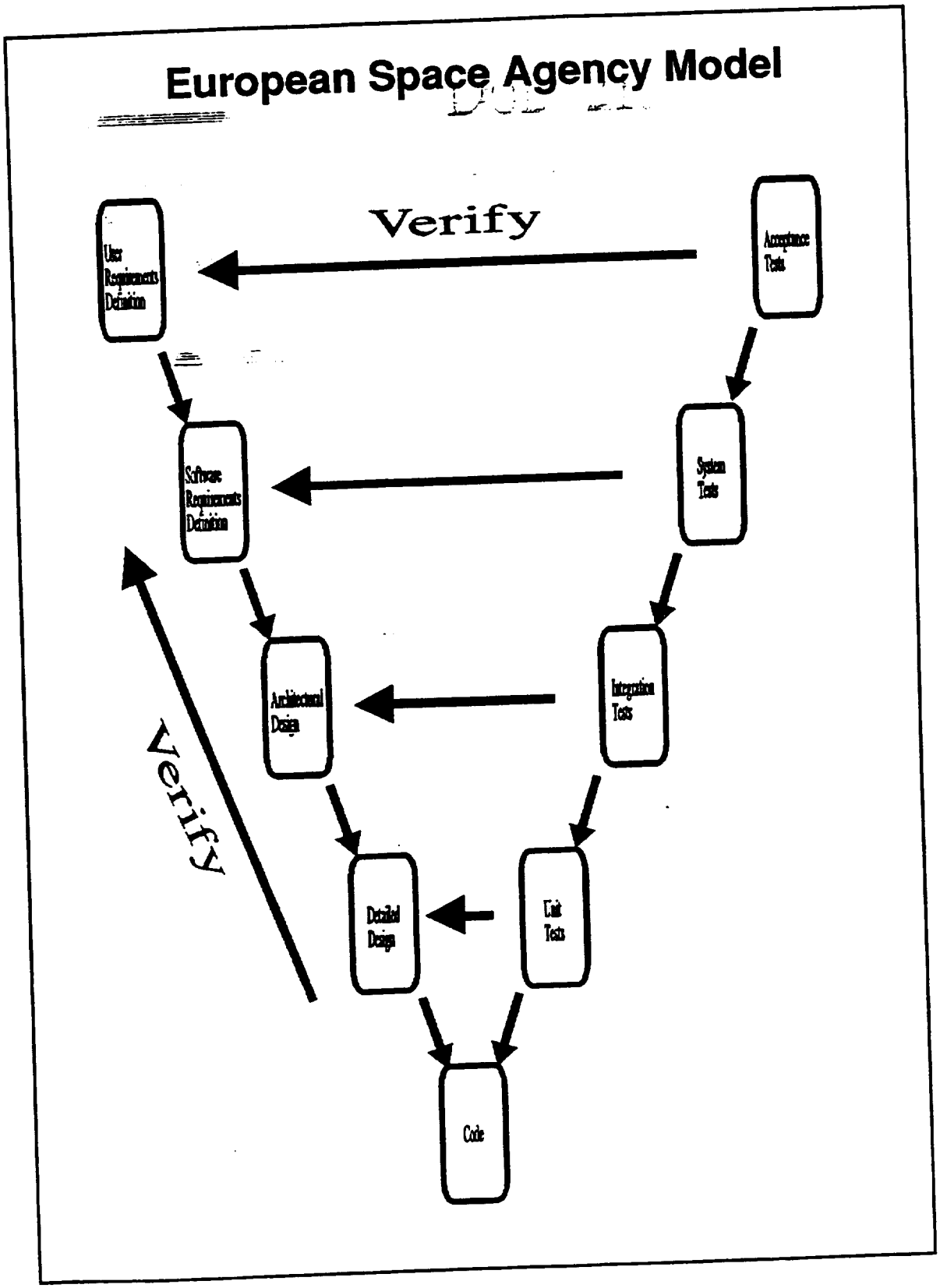
Techniques	References
Active Interface Testing	*32*
Anomaly Analysis	*32,2*
Attribute-Based Test Case Selection	*32*
Axiomatic Analysis	*7,14*
Boundary Testing	*26*
Branch Coverage	*26*
Cause-Effect Graphing	*26*
Competing Designs	*16*
Compilation Testing	*32*
Condition Coverage	*26*
Defect Analysis	*32*
Error Guessing	*26*
Fault Analysis	*17,18*
Flavor Analysis	*13*
Hazard Analysis	*17,18*
Inspections	*5,26*
InterProcedural Dataflow Testing	*9*

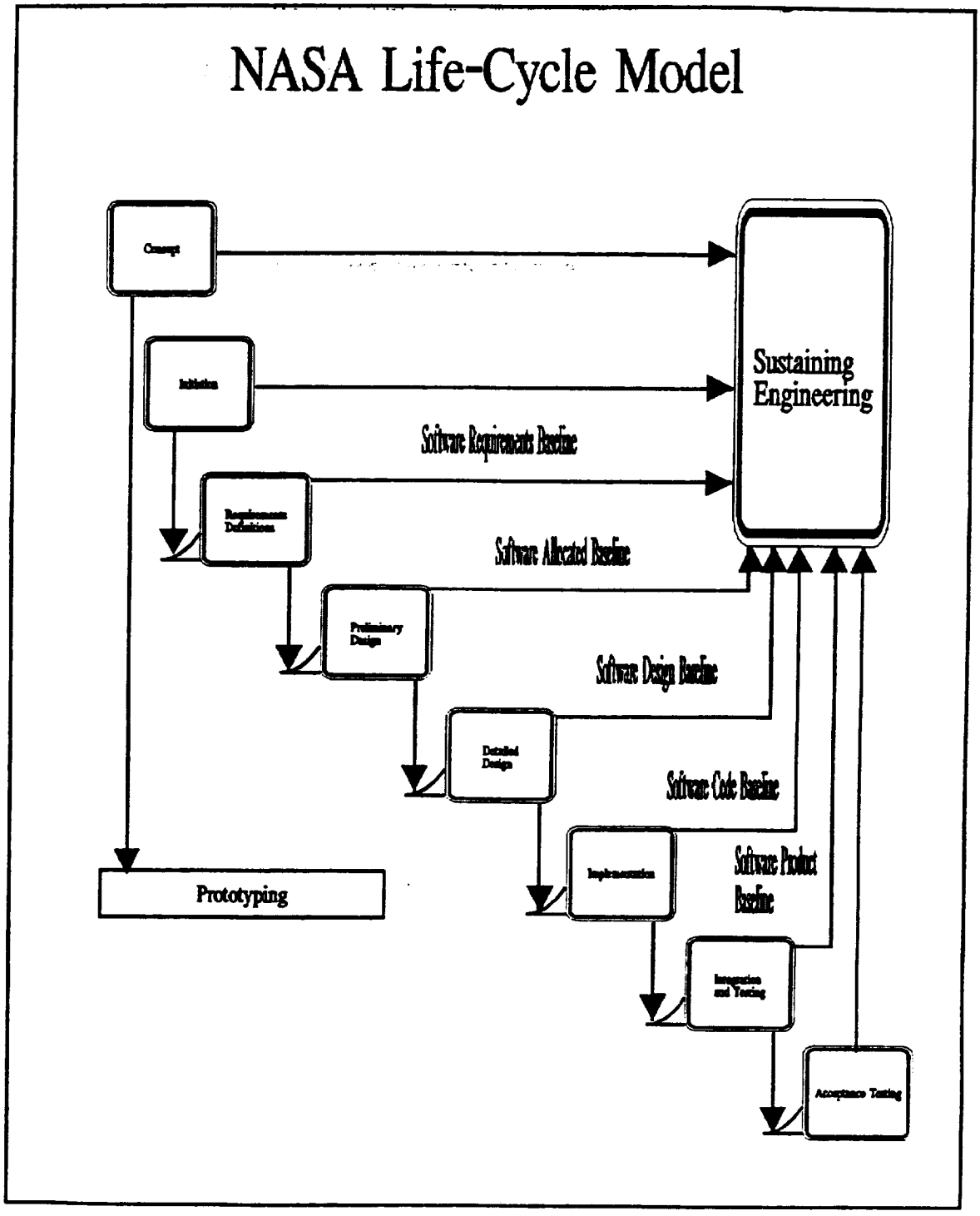
Appendix D: Techniques Vs. References ...

Techniques	References
Mutation Testing	*35,2114*
Object Oriented Analysis	*30*
Partition Analysis	*25*
Path Coverage	*31,31,2*
Performance Testing	*19*
Pre/Post Condition Testing	*31*
Prototyping	*31*
Random Testing	*31*
Realistic Testing	*31*
Regression Testing	*26-27,22-24*
Reliability Testing	*1,25*
Stepwise Refinement	*8, 11*
Stress Testing	
Symbolic Execution	

Appendix E: Examples of Life-Cycle Models




European Space Agency Model
Verify
User Requirements Definition
Acceptance Tests
Software Requirements Definition
System Tests
Architectural Design
Integration Tests
Verify
Detailed Design
Unit Tests
Code


NASA Life-Cycle Model
Concept
Initiation
Requirements Definition
Preliminary Design
Detailed Design
Implementation
Integration and Testing
Acceptance Testing
Prototyping
Sustaining Engineering
Software Requirements Baseline
Software Allocated Baseline
Software Design Baseline
Software Code Baseline
Software Product Baseline

02/07/92 09:17 AM

Expert Systems V&V Guidelines Workshop:

Part 2: Expert Systems

By Scott W. French and David O. Hamilton

This section is part of the Expert Systems V&V workshop. It summarizes the similarities and differences between expert systems and conventional software, how these similarities and differences impact V&V, and new techniques that address these impacts.

Table of Contents

Introduction

Goals

The primary purpose of this section is to explain the major techniques that have been developed for V&V of expert systems (ESs). These special techniques have been developed because there are significant differences between ESs and conventional software (CS); these differences make it so that existing CS V&V techniques are not adequate.

At the conclusion of this section, the student should:

1. understand the key similarities and key differences between CS and ESs
2. understand how the differences affect V&V of ESs (i.e., what new V&V problems or issues are created by these differences)
3. understand various V&V techniques that have been developed to overcome the problems and issues

Key Terms

There is no well established precise definition for ES or for Knowledge-Based Systems (KBS) but we will take the following definitions:

Expert System: computer programs that emulate the problem solving techniques of a human expert to solve complex problems

Knowledge-Based System: computer programs which use domain or heuristic knowledge to solve complex problems

Knowledge Engineer: a person involved in the development of an ES (includes knowledge acquisition as well as all the traditional activities of CS)

The key difference in these definitions is that a KBS may not use the problem solving techniques of a human expert. So an ES is a KBS but a KBS is not necessarily an ES. We will focus on ES characteristics with the understanding that some of the characteristics may not hold for just KBSs.

Overview

First, we will summarize both the similarities and the differences between ESs and CS. The basic similarity between ESs and CS is that they are both software. Students may accept this fact readily but others may strongly disagree, arguing that ESs are much more than just software. To eliminate debate on this issue, it is suggested that the instructor say that the V&V issues we can address rely on viewing ESs as software. All other V&V issues relate to the nature of knowledge itself and lie

VII-3 INTENTIONALLY BLANK

outside the realm of engineering and in the realm of philosophy and metaphysics (i.e., epistemology). If this is insufficient, then the Roger Schank quote from part 3 ("AI entails massive software engineering") and simly force the student to admit that the development of ESs at least involves a lot of software. That is, an ES may be more than software but it is mostly software.

ESs can differ from CS in two ways.

1. ESs are often implemented using non-procedural languages (we will call this group of differences the "implementation differences")

2. ESs problems have different characteristics (we will call this group of differences the "problem differences")

The first group of differences primarily affect lower level (white-box) testing while the second group of differences primarily affect higher level (black-box) testing. For this reason, we can focus on each group of differences independently.

We will illustrate both groups of differences using the Traffic Light Controller (TLC) problem. First, to illustrate the implementation differences, we will look at two solutions, one implemented as CS and another using a non-procedural language (CLIPS). Then, to illustrate the problem differences, we will introduce a new "enhanced" version of the TLC problem. This new problem will have more of the problem differences of ESs.

Finally, using the TLC problem, we will look at the major new ES V&V techniques. We will first look at techniques that have been developed specifically to address implementation differences. We will illustrate the techniques on the CLIPS solution to the TLC problem and provide some practice exercises. Then we will look at techniques which address problem differences, illustrate them on the enhanced TLC problem and provide some more practice exercises.

Expert System differences

Expert Systems are Software

First and foremost, ESs are software. That is, they are computer programs that are written in some type of programming language and are executed in a (digital) computer. It may be difficult, in general, to determine whether any given computer program is (or should be called) an ES. It may have only some of the characteristics of an ES or parts of the program may not have any ES characteristics at all.

For this reason, one should be careful when calling anything an ES or not an ES. It is better to simply think of a program as just that, a piece of software, but with certain characteristics. And when a piece of software has some ES characteristics, its V&V should account for those characteristics.

One should also be careful when analyzing a program to be solved with a computer program. It may be tempting to decide in the very beginning that the solution should be an ES. This approach can lead one into developing a solution that fits all the characteristics of an ES (because that is what was set out to be built) only to find out that an ES is not the best solution to the problem. Instead, one should just develop the best solution to the problem, noting the characteristics the solution begins to take and adjusting the development and V&V approach based on the emerging characteristics.

The problems that can be caused by pre-determining a solution to be an ES can easily be seen by noting the following. One might prototype a "pure" ES solution to a problem and find that it is not at all acceptable, say for performance reasons. This could possibly be fixed by re-coding small pieces of it in a different language. But if one has a fixed "all or nothing" vision of an ES solution, they might take much more drastic actions such as simply abandoning the problem or changing the problem so it is no longer an "ES problem".

Expert System Implementation Differences

The most recognizable ES characteristics are implementation characteristics such as the language used. This has led many people into using a so called "ES shell" if they think their problem qualifies as an ES. Because our focus is on V&V and not ES development, we will not discuss when an particular ES implementation approach should be taken but will instead discuss the V&V implications of various ES approaches.

Common "AI languages" used include:

- forward-chaining rules (also known as production systems)
- backward-chaining rules (e.g., Prolog)
- frame-based languages (e.g., KEE)
- LISP

With the exception of LISP, all the above languages share a common feature; they are non-procedural. Procedural programs are executed in a predictable fashion; it is straightforward to determine which statement will execute next. Each decision statement (which determines the statement that will execute next) is a test on a uniquely identified set of variables. By checking the values of those variables at any moment of execution, one can easily see which "direction" the branch will take.

Non-Procedural Languages

In non-procedural languages, it is not straightforward to determine which statement will execute next. There are no explicit decision statements that determine the next statement. Instead, there is a set of conditions, or tests, that are all looked at together to determine the next statement. Additionally, most often these tests do no operate on a unique set of variables. Instead they can potentially match or unify with a possiblly large number of different variable combinations. However, it is widely advocated that the developer not try to determine the sequence of statements in a non-procedural program; this is the "wrong" way to think about such a program. The "right" way to think is "declaratively".

Thinking about a program declaratively involves clearly separating control from data. Control in a non-procedural program is usually handled by a runtime "inference engine". The inference engine looks at the data (or more accurately, the entire state of the program) and decides which statement to execute next. The programmer, ideally, need only think about the correctness of data, leaving control to the inference engine. That is, the programmer should not be concerned about the order statements will be executed. In practice, programmers do need to be concerned about the order of execution. For example, a common error in production systems is to code a set of rules that work correctly only when executed in a particular order which happens to be the order they are initially executed. Then, when a seemingly innocuous change is made that affects the ordering of rule firings, the program no longer works. In this example, if both the expected and actual ordering were explicit, one could verify that they matched (i.e., the program was correct). As it was, both the expected and actual ordering was implict so it was very difficult to tell if

the program would executed as expected.

Iterative Development

Another ES implementation difference is that ESs are often developed in a highly iterative fashion. Almost all software is developed iteratively (e.g., by the Spiral life-cycle model) but ES are developed *more* iteratively. That is, many more iterations are typically used in the development of an ES and each iteration has only a small amount of added function.

The amount of iteration is especially high if the development of an ES heavily relies on knowledge aquisition. Experts typically can not express all of their expertise at once and this creates the need for many separate knowledge aquisition sessions. In between the sessions, it is generally good to add the newly aquired knowledge to the existing ES (to see how well the new knowledge works with the old).

Another reason for a high amount of iteration that, often with ESs, it is not clear whether the proposed solution is going to work well. In these risky situations, it is best not to go too far before evaluating/testing. This also leads to many small iterations of development.

The primary affect of a highly iterative development cycle on V&V is an increased amount of regression testing.

Lack of Explicit Algorithm

CS programs have been described as "algorithm + data structure". However, as we have previously discussed, ES generally have no explicit algorithm (i.e., no explict control over execution of statements). Instead, the ES progammer is encouraged to declaratively define the solution and leave the algorithm aspects to the inference engine. As we have seen, this can be a naive approach.

Experts do have problem solving approaches which contain procedural knowledge. The problem solving method may not be completely sequential and may be more heuristic than algorithmic. Whether or not the ES follows the same program solving method as a expert (in which case it would be just a KBS), it is important to make the problem solving method explicit and verify both that it is correct and that the ES follows it.

Expert System Problem Differences

In addition to how an program is implemented, it can be characterized by the type of problem it attempts to solve; this is especially true of ESs. ESs tend to solve problems that are either highly

complex or have been more easily solved by man than machine.

Man vs. Machine

Often, ESs are developed to solve problems that have been routinely solved by human experts. In these cases, a solution already exists (in the humans head) and may need only be translated into a form that can be executed by a computer. To a large extent, ES technology was developed to allow this to be done. Often this is done by translating expert heuristics (rules of thumb) into if-then rules that can be executed. These types of systems have come to be called "shallow" or "design by knowledge aquisition" systems because such systems are shallow in that they do not "understand" what they are doing and are developed directly from knowledge acquired from an expert.

Correctness of these types of systems is more like testing a human expert.

Complex Problems

Often, ESs are built in an attempt to solve a problem has been proven very difficult to solve (with a computer program). This is because the tools and languages usually associated with ESs make it much easier to solve certain types of problems. V&V of solutions to these complex problems can be difficult.

If it is difficult to determine the correctness of any specific answer (e.g., optimal scheduling of many hundreds of items) then it will be difficult to determine if the ES solution has come up with a correct answer. If the "correctness" of the answer can not be determined analytically and instead must be determined by "expert opinion", it will be difficult for anyone other than the expert to determine the correctness of the ES.

Finally, ESs are often used to assist or advise an human expert. That is, the ES does not make decisions but only suggests decisions to a human expert who will make the final decision. There is often a desire to decrease the amount of V&V of these types of systems because if the ES is wrong then it is felt that the expert will always catch it before any harm is done.

Two Traffic Light Controller Implementations

Overview

To illustrate both the implementation and problem differences between ESs and CS, we will look the Traffic Light Controller (TLC) problem. First we will look at testing black box solutions to the TLC problem and them we will compare an ES implementation to a conventional implementation for the the same TLC problem in order to illustrate the implmentation differences. Then we will look at a more "expert" type of TLC problem in order to illustrate the problem differences.

Scenario Testing

In Part 1 of this course, it was pointed out that black-box testing of a system involves identifying scenarios and the executing those scenarios (with differing stimulus) and looking to see if the correct response is received. Often, much of the work in black-box testing is identifying and describing all the scenarios. A good way to approach this is to first identify certain visible events that can occur in the system and describing scenarios in terms of combinations of these events.

For example, in the TLC problem, we can identify the following events.

1. *switch*: when the traffic light changes (e.g., from green to red)
2. *approaching*: when an approaching auto (or pedestrian) comes in the direction of the green light
3. *waiting*: when an auto (or pedestrian) is waiting for the light to change (i.e., they have approached from the direction of the red light)
4. *(t, e)*: when a certain time, t, has elapsed since the last event and at the same time, event e occurs

Given these events, we can begin to identify certain scenarios that could be tested. For example:

1. (2 minutes, switch) (2 minutes, switch) : this scenario occurs when there are no approaching or waiting traffic and the light switches every 2 minutes
2. (t: t < 2 mintues, approaching) (2 minutes, switch) : this scenario occurs when approaching traffic is detected before the light can change (i.e., less than 2 minutes since the last switch event); in this case the light should wait another 2 mintues before switching
3. (t: t < 2 minutes, waiting) (15 seconds, switch) : this scenario occurs when a car (or pedestrian) is waiting for the light to change; in

this case, the light should switch in 15 seconds if no other event occurs in that time

These are only some sample scenarios. We can not list all the scenarios because there are an infinite number of scenarios that can be constructed. There are an infinite number because there is no limit on the length of the chain of events in each scenario. For example, we the light can change every 2 minutes for ever; this simple event chain itself is, theoretically, of infinite length.

Testing State Changes

An alternative way of identifying tests is based on state changes in the system. The tests are used to verify that the system makes the correct state change in each situation. However, additional analysis is required in order to ensure that the states and state changes will result in correct behavior of the system. So this is not to be used as a validation technique but it is a good verification technique since it can comprehensively cover the complete range of possible scenarios.

This type of testing is sometimes called conformance testing. A state diagram (or equivalent representation) is created and analyzed to correctly solve the problem. Then the solution is shown to conform precisely to the state diagram through exhaustive testing.

Additionally, the state diagram can be constructed at differing levels of detail to correspond to varying levels of testing. For example, at a high level, we can identify the following states in an abstract solution to the TLC problem:

>> S_1: The system is currently in the state of "waiting for 2 minutes" before changing the light. That is, it is continuing to count off 2 minutes since the last event before changing the light.

>> S_2: The system is currently in the state of "waiting for 15 seconds" before changing light but has not waited 45 seconds yet. That is, some time in the past, the system began a 15 second wait, is still waiting for 15 seconds to expire, and may have restarted this wait a few times but not for more than 45 seconds.

>> S_3: The system is in the remainder of a 1 minute wait. That is, at some time in the past, the system begain a 15 second wait which it has restarted so many times (due to approacing cars or pedestrians) that it has been greater than 45 seconds since it first started the 15 second wait. And thus, the system will need to switch the light within one minute of starting the first 15 second wait.

The system could switch from one state to another based on the following possible events.

1. *approaching* (a car or pedestrian is detected in the direction of the green light)

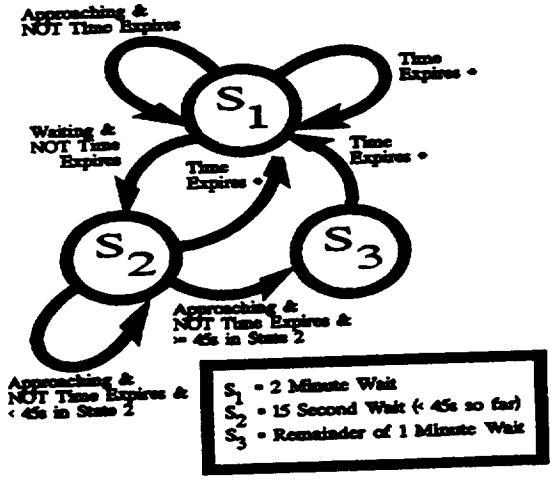
2. *waiting* (a car or pedestrian is detected in the direction of the red light)

3. *time expires* (the system is in the state of waiting for a particular amount of time, say t, and it has been t seconds since the system began waiting for t seconds)

4. *< 45 seconds in* S_2 (the system is in state S_2 and has been in S_2 for less than 45 seconds)

5. *>= 45 seconds in* S_2 (the system is in state S_2 and has been in S_2 for more than 45 seconds)

For each event, we can define a state transition (a change from one state to another) that will both ensure correct operation of the system. This is shown in the follow state diagram.



We should first analyze this state diagram and convince ourselves that it describes a system that correctly solves the problem. To do this, we need to look at completeness and consistency. Completeness includes looking to make sure that the system has all the right actions (i.e, a complete set of responses) and accepts all the right inputs (i.e., has a complete set of stimulus). Concistency involves making sure that the system has the correct response for each stimulus, without violating any constraints.

The system only has one response and that is to change the light. This response is covered when the system makes any transition into S_1 (i.e., a time expires). The system has only two stimuli, an approacing or a waiting car or pedestrian, which are both covered.

Now lets look at each of the seven state transitions and check that they are consistent with the requirements

1. (approaching and not time expires) If an approaching car or pedestrian is detected before 2 minutes has elapsed then there should be no response. This is consistent with the state diagram.

2. (waiting and not time expires) If a waiting car or pedestrian is detected before 2 minutes has elapsed then the system should change the light after 15 seconds (assuming no other stimulus. This is consistent with the transition to S_2.

3. (approaching and not time expires and < 45 seconds in state S_2.) If we are waiting for 15 seconds due to a waiting car and an approaching car or pedistrian is detected, then we should wait for another 15 seconds, assuming we have been waiting less than a total of 45 seconds. This is consistent with staying in state S_2.

4. (approaching and not time expires and >= 45 seconds in state S_2) If we are waiting for 15 seconds due to someone waiting and we detect someone is approaching but the waiting person has already been waiting for 45 seconds or more, then we basically need to ignore the approaching signal and change the light after a total of one minute. This is consistent with the transition to state S_3.

5, 6, 7. (if any timer expires) Regardless of what event occurs, if we have determined that it is time to change the light (i.e., someone has been waiting long enough) then we should change the light. This is consistent with the state diagram.

Thus far, we have shown that all the state transitions are consistent with what is prescribed by the requirements. We should also review the requirements to make sure that there are no situations that are not covered by any of the state transitions. Unfortunately, there is no mechanical technique for doing this (unless the requirements are already broken into small individual requirements which our requirements are not). One should just think through the requirements, looking for things that have not yet been considered.

Once we convince our state diagram describes (abstractly) a system that meetings the requirements, we now need only check that any proposed solution correctly matches the state diagram. Thus, by creating the state diagram, we have taken one large problem (verifying a proposed solution to the requirements) and decomposed it into two smaller problems (verifying the state diagram and verifying a proposed solution against the state diagram). This type of approach is sometimes called

"conformance testing" because a proposed solution is checked to see that it exactly conforms to a more abstract solution (e.g., a state diagram).

One difficulty is that if we are given a proposed solution and we can only observe the responses that the system makes (i.e., we can only see whether it changes the light or not), then we can not check that it conforms to the state diagram. This is because we can not directly see the internal state changes of the system. If we can not see inside the system (i.e., white box testing), then we are back to having to infer state changes from the different stimulus/response pairs.

One last point is that it would be much easier to check that a proposed solution conforms to the state diagram if its design is similar to the state diagram. For example, if there was an internal variable that could held either the value 1, 2, or 3 depending on the state the system was in (corresponding to the states in the state diagram), then it would be very easy to match the current state of the system to the state diagram.

Handouts and Exercise

There are 3 different handouts, each describing one proposed solution to the TLC problem. By dividing the class into 3 teams, each team can focus on one of the different solutions.

Each team should spend a few minutes analyzing the proposed solution given them and compare it to the state diagram. They should also develop a set of test cases that will determine if the proposed solution truly conforms to the state diagram.

Conventional Implementation

The conventional implementation is an Ada program. It is designed after the state diagram in that the internal state of the Ada program is easily recognizable at all times by looking at the value of the variable "State". There is a case statement that simply looks at the current event and decides which state, if any, to transition to next. This case statement is simply repeated endlessly.

Expert System Implementation

There are actually two different ES implementations. They are called ESs because they are implemented in the non-procedural language CLIPS. One may argue whether or not they are ESs because they are solving a rather conventional problem (the TLC problem). However, since they do illustrate the issues created by non-procedural languages, we will call them ESs for our purpose (and we will refer to them as either the "ES implementations" or the "CLIPS versions".

Later, we will discuss the differences between these two implementations but for now, we will compare the both of them to the Ada solution. We can do this because both CLIPS versions are similar in that they are both simply a set of rules that decide which state, if any, to transition to next.

Comparison and V&V Implications

If we compare the ES implementations to the conventional Ada implementation, we can notice a few features that make the ES solutions appear to be much easier to V&V.

First, the rules each look very much like a state transition statement. The general form of a rule is "if we are in state so and so and the current event is such and such then we should transition to state so and so (optionally performing an the external response of switching the light)". Thus, the ES implementations are relatively easy to compare the the state diagram.

Second, both CLIPS versions are much shorter than the Ada version. So there is much less ES code to verify. This is primarily due to the pattern matching feature of CLIPS enabling a single rule to match many different but similar situations (i.e., each rule can do more).

Finally, if one looks closely at how the Ada version was designed, one can notice that it resembles a crude inference engine. The loop around the case statement means that the Ada program will repeatedly look for a certain condition and do the associated action. This is very similar to a production system which repeatedly looks for the right rule to fire. In the Ada program, the inference engine, crude as it is, had to be coded and V&V'ed along with the rest of the program. But in the CLIPS versions, the built-in CLIPS inference engine was used.

Because of these differences, the ES approach seems much preferable to the procedural Ada approach. The ES approach leads to smaller programs which should reduce the V&V effort. But there are other characteristics of the ES implementations which should also be considered.

Although the ES implementations have less code, each rule can have compex interactions with other rules. In the procedural Ada version, it is realitively straightforward to determine the order of execution of each statement but in the CLIPS versions, it can be difficult to determine the order of rule firings. Rule firing order can depend on a combination of condition logic, priorities, and other critieria such as the number of conditions (specificity) on each rule. There are also other

implementation concerns for the ES implementations which are typically not used in procedural implementations such as the deletion of no longer needed dynamic variables (i.e., garbage collection). Thus, the Ada version, though larger, has more straightforward logic with less complicated internal interactions.

Much of the nature of procedural languages is due to efforts to simplify the internal interactions of programs in or to make them simpler and eaiser to understand (and thus to verify). Most of the structured programming concerns are directly related to issues of verification. Because most non-procedural languages are "unstructured", programs written in them can be harder to verify.

So it is not the case that ESs are harder or easier to verify but instead that they have different V&V issues. That is, in some ways they are harder but in some ways they are easier to verify. In summary, procedural programs have more decisions and control structures to test while ESs have more complex internal interactions such as side-effects, garbage collection, generality of pattern matching, and rule interaction.

The different concerns of ESs mean that existing V&V techniqhes, which were created for procedural progams, are not as appropriate for ESs. Existing procedural testing techniques are largely focused on ensuring that the implementation solves all aspects of the problem; techniques such as path coverage and requirements tracing are examples. ES testing also needs to address the potentially unexpected consequences of the complex internal interactions; ES V&V will have more of a focus on this issue than procedural V&V has.

Another way to view the differences between ES and conventional V&V is from the perspective of AI. ES languages are largely modeled after the way most psychologists believe humans think (at a high level, not at a neurological level) instead of on the design of a computer (like a procedural language is). Given this, we can look at the categories of mental errors humans make. These categories include things such as slips/lapses, exceptions to generalizations, and erroneous beliefs.

Slips/lapses are simple errors such as making a typographical error. These types of errors are most often attributed to interuption in trains of thought. For example, one may start to type the word "verification", think of its relation to "validation", and type "velidation". These types of mistakes can be likened to rules which are similar enough that they both fire but they have incompatible results.

Exceptions to generalizations occur when a generalization is made and then a new never-before-seen exception is

encountered. A similar type of error occurs when one simply forgets to treat the situation as a special case. This can be likened to either a rule that is too general (i.e., does not account for the new exception) or a frame system without a specialization for the exception. The similar type of human error (forgetting about special cases) can be likened to a rule or frame based system that contains the exception but, for some reason, does not recognize the situation as being a special case (a more concrete example of an production system analogue is when conflict resolution does not correctly identify the most specific rule and incorrectly fires a more general rule).

Erroneous beliefs are simply things that are thought to be true but are not. These can be likened to rules or frame definitions that are simply not right.

Error categories such as slips/lapses and generalizations directly relate to the effects of complex internal interactions typical of many non-procedural programs.

Handout and Exercise

Just as in CS programming, there are good and bad ways to design programs written in non-procedural ES languages, good and bad in the sense of being easier or harder to verify. Before presenting some ideas related to good and bad ES designs (and their effect on verification), some handouts (#2, 3, 4, 5) are given out that contain two CLIPS solutions to the TLC problem and some related information.

Students should spend a few minutes studying each proposed solution, trying to understand how they each work. Students should not try to understand each implementation in detail but should instead only try to get a high level understanding of each. Students should pay special attention to the interactions among the rules, trying to understand how the rules work together to solve the problem. To assist with this, the handouts include a rule interaction diagram that shows which rules are affected by each rule. Specifically, these diagrams are a directed graph where the edges are drawn from one rule, say A, to another rule, say B iff rule A modifies a variable in the LHS condition of rule B.

Testing Good and Bad Designs

The first difference to point out in the two different versions is that one of them is more modularized. Although this version has more rules overall, one need only understand a few rules at a time (i.e., one can focus on only one module at a time). Also, this more modular version can be more easily tested because each module can be tested separately. By comparing the two rule interaction diagrams, one can also see that the more modular version has less rule interaction; that is, the modules are

loosely coupled. Finally, the more modular version has simpler LHS conditions. For these reasons, the more modular version should be easier to understand; the students will be able to judge this for themselves.

Testing modular well-designed ES programs is generally easier and simpler than testing badly designed ones. It is possible to design ES prgorams so that they have less complicated internal interactions; that is, one can design "structured" ES programs.

The less modular version has a problem in it. This problem is not easily observable and most of the time, this version will work correctly. One rule, del-old-changes, which is used to delete old facts that are no longer needed, has a typo in it. "signal-changes" should have been "signal-change". This error can be seen by looking at the rule interaction diagram and noting that del_old_changes is a affects no other rules (i.e., it doesn't do anything). This problem could have easily been found by using the CLIPS CRSV tool and deftemplates. The more modular version uses deftemplates and thus many types of errors such as typos can be detected automatically.

A common objection raised against the development of highly modular programs is that they are less efficient. That is, they may be easier to understand but they execute slower. If this objection is raised by a student, the instructer can quickly eliminate it by simply pointing out that the more modular TLC solution executes significantly faster than the less modular one. The reason is that although the less modular TLC solution requires fewer rule firings, each rule firing takes longer because each rule is more complicated (i.e., the less modular CLIPS version requires a lot more pattern matching). Therefore *"simpler is better, simpler is faster"*.

"Expert" Traffic Light Controller Problem

For studying the V&V differences associated with ES problems, the TLC problem is not sufficient. Instead, we will use an expanded version of the TLC problems which is as follows.

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 and is also as fair as the policeman (i.e., he doesn't force any one direction to wait for a long time on the other direction).

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)

Using this information, the system will decide when to allow a street (north, south, east, west) to either go straight, turn left, or wait on another street.

Although this statement of the problem is not sufficiently detailed enough to develop a solution from, there is some analysis that can be done. One can check for inconsistent requirements and missing information. For example, if the requirements said that the system automatically decides when to switch to smart mode and also said that there was a switch to manually control the mode, then these requirements are potentially inconsistent. And if the requirements stated that traffic flow is a determining factor but gave no indication that traffic flow as an input then this would be missing information.

From looking at this new problem description, we can see that it has many features of an ES problem. It is automating a job that a human expert is currently performing. It is much more complicated than the conventional TLC problem and it would be difficult to

design an algorithm to solve the problem. It also contains some subjective requirements such as being "fair".

Knowledge Acquisition Results

The statement of the new TLC problem as it has been given so far does not have enough information to develop a solution from directly. As with most ES problems, the criteria for making the decisions mandated in the requirements must be obtained from an expert. For our purposes, we will assume that the following is from initial knowledge acquisition from the policeman.

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

With this new information, additional analysis can be done. In addition to looking for conflicting information, one should look to see if all decisions mandated by the requirements can be made from the criteria obtained during knowledge aquisition; that is "is the knowledge aquired sufficient to solve the problem ?" Ideally, one should aquire operational scenarios, both nominal and off-nominal (disaster) scenarios, to aid in defining tests.

Problem Features

At this point, the student can study the new TLC problem and think about scenarios as well as inconsistencies and missing information. A class discussion of how well this problem fits the notion

of an ES is also appropriate. Some questions that would be helpful in promoting class discussion are:

- Is it a shallow or deep reasoning solution ?
- Would this be difficult to solve with conventional software ?
- Does it rely on human judgement ?
- Will it replace or augment a human expert ?

Expert System Implementation V&V Techniques

Overview

The purpose of this section is to summarize several techniques for verifying programs written in non-procedural languages. These techniques address issues associated with how ESs are implemented (and thus will all be verification techniques as opposed to validation techniques). Although the student will not be a master of these techniques at the end of this section, they will be able to begin applying them. More importantly, they will be aware of techniques that do exist and when each one is applicable. References will be given that explain each technique in more detail.

For each technique, we will briefly describe the technique and then indicate when the technique is applicable (it may apply only for rule based systems and/or may only be able to detect certain types of errors). Most of these techniques are difficult to apply without some type of tool so we will discuss the availability of supporting tools for each technique.

An important part of the discussion of each technique will be an example of how it could be applied to the TLC problem. These examples serve to better describe the technique and how it can be used. The student should be encouraged to practice the techniques by further applying them to the TLC problem.

Rule Consistency Checking

The most difficult aspect of verifying rule-based programs is verifying rule interaction. This is because of the potentially large amount of interaction among rules. Most of the interaction is probable expected but there may also be some unexpected interaction. Verification should involve looking at all interaction to make sure that it is intended and is correct but, given the large amount of interaction, this can be difficult.

Rule consistency checking attempts to make this easier by identifying certain types of rule interaction that are usually unexpected and indicate some type of error. That is, they identify bad or anomalous types of rule interactions. There are several major categories of these anomalous types that we will illustrate.

First, it should be noted that these techniques are applicable only to rule based systems. They also do not find errors but only indications of possible errors (i.e., not all anomalous rule interactions are wrong). So a human will need to look at each anomalous rule interaction to determine if it indicates an error. Another important note is that it is very time consuming to identify all anomalous rule interactions by hand;

some type of tool should be used. Unfortunately, there are few tools available to do this and there are no known commercial products.

More information on rule consistency checking can be found in references [1] and [2].

Rule Consistency Anomalies

There are two types of reachability anomalies which are dead-end rules and unreachable rules. Dead-end rules are rules that do not affect any other rule. That is, the chain of inferencing reaches a dead-end at such a rule. An example of a dead-end rule is the rule "del_old_changes" in the short CLIPS implementation to the TLC problem. This rule only affects a fact called "signal_changes" which is not referenced by any other rule. So del_old_changes can not affect any other rule. This is an error; the fact that should have been changed by del_old_changes should have been "singal_change". It is worthwhile to point out that this error was made when the short CLIPS version was initially created and was actually found by doing rule consistency checking (this can be seen in the rule interaction diagram - handout #4).

Another type of reachability anomaly is unreachable rules. An unreachable rule is one that is not affected by any other rule. That is, the chain of inferencing can not reach this rule. For example, if there was a rule whose left hand side condition depended on a fact called "signal_changed" then that rule would be an unreachable rule. That is because there is no such fact created by any rule.

A cycle is a group of one or more rules that (can) repeat. The rule "update_time" is an example of a rule that repeats. In this case, the cycle is not an error; the rule is intended to fire repeatedly until a certain time is reached. But in general, cycles are potential problem areas because they could repeat endlessly (i.e., be the rule-based equivalent of an endless loop).

There are two types of overlapping rule anomalies. The first type is redundant rules. These are rules that do some of the same things. For example, the following two rules are redundant.

```
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
```

These rules are redundant because when light_changed is true, they both attempt to retract the medium (and short) timer. This could be an error, depending on how these rules are implemented. They could be implemented so that when light_changed, retract_medium_timer will fire and retract the medium and short timers, preventing the set_long_timer rule from firing and setting the long_timer. So if the long_timer must always be set when light_changed, the above rules are in error.

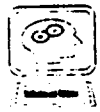
Another type of overlapping rule anomaly is conflicting rules. Conflicting rules are almost always in error. For example, the following two rules are conflicting.

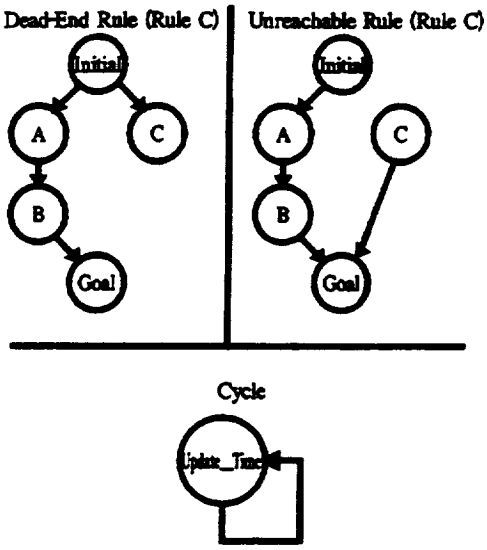
```
set_long_timer:
  if light_changed or
  signal.in_direction green
then
  set long_timer
  set medium_timer
  set short_timer

retract_medium_timer:
  if light_changed
then
  retract_medium_timer
  retract short_timer
```

These two rules are conflicting because when light_changed is true, the first rule will set the medium timer while the second rule will retract it. These two actions (setting and retracting) are contradictory.

All of the above errors could be found by inspecting the rules in question and seeing that they are wrong. For example, one could see that, in the previous example, set_long_timer should not also set the short and medium timers. Doing rule consistency checking is an aid in finding errors that are missed by simple inspection.

Finding all instances of rule anomalies in a large rule-base would take a long time to do by hand so it would be adventageous to use an automated tool. However, if the rule-base is modularized into small sets of rules, it is not unreasonable to check for several types of anomalies by drawing a rule interaction graph. That is, by drawing a directed graph, showing which rules are affected by other rules. Then, the following types of rule anomalies can be seen directly in the rule interaction graph.





Data Consistency Checking

Data consistency checking involves comparing the definition of data/facts with how the data/facts are used. This is very similar to type checking in conventional languages, such as Ada, where one must declare a data type and declare variables of that type before the variable can be used. This allows one to find errors involving a misuse of a variable.

Data consistency checking is often supported by tools that are provided with the programming language used. For example, CRSV, which is provided with CLIPS, can find many types of data/fact use errors such as the mispelling of the "signal_change" fact in the rule "del_old_changes". This error could have been found if "signal_change" were declared via a deftemplate.

More information on data consistency checking can be found in references [3], [4], and [5].

Sensitivity Analysis

Sensitivity analysis involves analyzing the sensitivity of one data item to other data items. Sensitivity analysis can be used as a debugging technique or can be used to increase the efficiency of programs. It can also be used to help construct test cases. Like other techniques, sensitivity analysis can be very tedious and time consuming if done by hand. Unfortunately, the only known sensitivity analysis tool is still a research prototype.

Sensitivity analysis is most applicable to classification types of problems instead of problems like the traffic controller problem. So suppose, instead of the TLC problem, we had the problem of classifying the current state of a soution to the TLC problem. That is, given the value of all the variables used in a TLC solution, classify the current state as being either S_1 (only the long_timer is running), S_2 (both the short and medium timers are running and the medium timer is not within 15 seconds of expiring), or S_3 (both the short and medium timers are running

and the medium timer is within 15 seconds of expiring). If we considered the sensitivity of each of the three output states to the input parameters (the variables of the program being analyzed), we can see that S_1 is the least sensitive because it depends only on long_timer running. Conversely, states S_2 and S_3 depend on both the short and medium timers existing as well as how long the medium timer has been running.

We could use this information about the sensitivity of S_1 to:

1. debug a problem involving no output when none of the timers are running
2. see that it would be more efficient to check for the program being in state S_1 first (because the fewest conditions need to be checked)
3 identify a set of tests to cover all possible ways state S_1 can be created

More information on sensitivity analysis can be found in reference [6].

Structural Testing

Structural testing involves identifying a set of test cases that will "cover" all parts of the program. "Covering" a rule means that the test has caused the rule to fire. "Covering" a frame means that the test has caused an instance of the frame to be dynamically created. "Covering" a frame demon means that the test has caused that demon to be invoked. By using these new definitions of "coverage", structural testing can be extended (from the realm of CS) to non-procedural languages. That, is by defining coverage for the type of knowledge base constructs used in the ES, structural testing can be adapted for any kind of ES language.

The purpose of structural testing is not to find errors directly but instead only to help ensure that testing is or has been done comprehensively. As with other techniques discussed in this section, structural testing can be tedious and time consuming. Tools can only partially help with this. Tools can be used to measure test coverage but it is generally still up to a human tester to create, execute, and analyzed the results of test cases.

Structural testing of ESs can be more difficult than for CS because ESs are developed in a more iterative way. Each time the knowledge base is changed, test coverage must be recalculated (to measure the coverage of the newly modified knowledge base) and tests must be rerun. However, when used as a part of regression testing, structural testing (i.e., measuring test coverage) can help one figure out which tests need

to be rerun and which may not need to be to be rerun. This can be done by recording the coverage of each previously run test case and comparing this to the knowledge base modifications. Each test case that covered a portion of the knowledge base that changed should be rerun. This may or may not be cost effective use of structural testing. The cost of rerunning all test cases should be compared to the cost of recalculating test coverage.

An important point about structural testing that should be pointed out is that the analysis involved in identifying test cases to cover all parts of a knowledge often does result directly in finding errors (i.e., errors are found before executing any tests). This is a serendipitous benefit of structural testing that should not be overlooked. This benefit seems to arise because it encourages programmers to:

1. look at their programs from a fresh point of view (that of test coverage)

2. study their programs in more detail

3. think about how their programs will execute under specific scenarios

More information on structural testing can be found in reference [7].

Specification-Directed Analysis

Specification-directed analysis, like structural testing, is an extension of a CS testing technique. It merely invovles extending the notion of specification to one that is compatiable with the constructs of non-procedural languges.

Recall that a specification is an assertion about a part of a program and can be thought of as a specific requirement that the program is expected to comply. Another way of thinking about a specification is that it is a requirement that the program designer places on the program. A key difference between specifications and typical requirements is that a specication is a precise and detailed assertion about how a program is expected to behave and there is usually a rigorous procedure that can be used to show that a program satisfies a specification.

Extending specification-directed analysis to programs written in non-procedural languages involves identifying for each new non-procedural construct, at least one format for recording specifications, and at least one technique for showing that a program satisfies a specification written in this format. Because there can be a variety of specification formats and "proof techniques", even for a single type of non-procedural construct like a forward chaining rule, specification-directed analyis is really a family of verification techniques.

There are no known commercial tools that support specification-directed

analysis for any non-procedural language. But, unlike the other techniques discussed in this section, specification-directed analyis can be done by hand (i.e., it is not as overwhelmingly tedious as the other techniques).

One example of specifying and proving an assertion for the TLC problem can be seen by looking at the longer CLIPS solution to the TLC problem. In the timer module, there is an assertion that the timer names are unique. Also in this module, there is a rule called "timer_name-conflict" that can be shown to fire at the end of each timer cycle to remove duplicately named timers. So, informally, this demonstrates that timer names are unique (at least at the end of each timer cycle).

The above example brings up an important point with respect to specification-directed analysis - informality vs. formality. Another name for specification-directed analysis is "formal methods". This is because specification-directed analysis is widely advocated as a being rigorously done using formal mathematical methods. And because many think that rigorous proofs about specifications using formal mathematical techniques are very very expensive to generate, specification-directed analysis is also often thought to be very expensive. However, as the above example illustrates, specification-directed analysis can be done very informally without an rigorous use of mathematics. So specification-directed analysis need not be very expensinve or difficult to apply. In fact, some believe that this family of techniques will eventually be used almost exclusively (i.e., will eliminate the need for the other techniques).

There are some commonly used types of specifications that can be adapted for use in non-procedural programs. One such type of specification is data value constraints. This is similar to data consistency analysis except that in addition to just defining the type of the variable, a constraint on its value is also given. The constraint may be just a simple list of possible values or a complex condition that depends on the values of other variables. We have already seen one example of this type of constraint and that was the unique name constraint on timer names in the Timer module of the longer CLIPS solution to the TLC problem.

Another type of specification is the use of preconditions and postconditions. For a given action, a precondition specifies a condition that must be true in order for the action to be performed while the postcondition specifies the condition that should be true after the action is performed. This type of specification can be adapted for use in forward chaining rules that perform some action. The condition part of a

such a rule already serves the purpose of the precondition in that the condition must be true before the rule fires. What is missing is the postcondition which states what should be true after the rule fires. By adding such a postcondition to rules, one can not only check that individual rules are correct (by verifying that the rule firing will result in the postcondition being true) but can also be used to check the interaction between rules. It helps with interaction analysis because the postcondition of one rule can be compared to the left hand side condition of another rule to more clearly see if one rule firing will lead to the firing of another rule.

Preconditions and postconditions can also be used in verifying the use of demons and other types of triggered functions. By associating a precondition with a demon, one can more easily compare the assumptions that a demon makes with the rest of the system. For example, if we had a frame-based solution to the TLC problem which had a demon that suppled the signal data (i.e., when a waiting or approaching signal was detected), we could define a precondition of "no signals unprocessed" and a postcondition of "exactly one unprocessed signal". Together, these would indicate that one signal should be processed before the next one is read.

Most knowledge bases include some functions. That is, they are not all just doing logical inference with an occasional external action. For example, the longer TLC CLIPS solution has a function to change the light. This function can be specified with a precondition and postcondition pair as follows:

precondition: green-light = NS or EW
postcondition: green-light = NS or EW
and green-light /= green-light'

Where "green-light'" denotes the green-light variables value before the change-light function was invoked. Together, these indicate that the change-light switches the light from NS to EW and from EW to NS. This function could be more easily specified with an abstract functional specification. Abstract functional specifications are basically very high level functional programs so they are generally more natural for programmers to use. An abstract functional specification for the change-signal function is:

direction:= NS if direction=EW
EW if direction=NS

Two of the case studies (#1 and #2) are more complete examples of specification-directed approaches to the development and verification of ESs.

More information on specification-directed analysis can be found in references [8], and [9].

Expert System Problem V&V Techniques

Overview

In the previous section, we discussed verification techniques that addressed implementation issues of ESs. Because those techniques were implementation oriented, they were all clear box techniques. In this section, we will discuss techniques that address issues associated with ES types of problems. Because these techniques are problem oriented, they will be black box techniques.

The use of these techniques will not depend on how the solution is "coded", in what language it is implemented, how it is designed, or with what reasoning strategies are used. In discussing these techniques, we will only be concerned with whether or not the solution adequately solves the problem, satisfying all correctness objectives (stated in some form of requirements). We will also be concerned with whether or not the system is based on a correct set of knowledge. One could argue that as long as the system adequately solves the problem, it does not matter whether or not the system is based on correct knowledge. For some purposes this may be true but for most purposes it will be a dangerous approach. A system based on incorrect knowledge could cause problems in several ways.

1. The system may be a source of knowledge in the future.

2. In the future, a maintainer may notice a difference between the knowledge used in the system (which is incorrect) and the behavior of the system (which is correct) and change the bahavior of the system to be compatible with the knowledge (so that it no longer works correctly).

3. Future maintainers will have difficulty understanding the system; they may be confused by the incorrect knowledge.

So we will assume that it is important whether or not the system is based on correct knowledge.

Knowledge Acquisition Correctness Checking

As just mentioned, it is important to check that the system is based on correct knowledge. It is also important to check for correct knowledge as early as possible during development. This is consistent with the goal of finding errors as early as possible. The earliest point in which incorrect knowledge can be identified is when it is being acquired from an expert.

Knowledge acquistion correctness checking involves checking consistency

and completeness of the knowledge being supplied by an expert. It does not necessarily involve checking the accuracy of the expert's knowledge; this would be difficult for the ES developer who is often not an expert in the subject domain or for an expert who is developing a system based on is own knowledge. This is similar to requirements analysis where the goal is not to judge whether the user really needs the capabilities stated in the requirments but instead involves looking for inconsistencies and incompleteness which would lead to difficulty in developing a system that satisfies the requirements.

Inconsistencies to be looked for include the following types of problems.

1. Contradictory statements such as terms being defined multiple times and in different ways
2. Redundant statements such as terms being defined multiple times in similar ways
3. Conflicting goals or assumptions

Types of incompleteness to be looked for include the following types of problems.

1. Terms referenced but not defined
2. Goals with no tasks defined for accomplishing them
3. Tasks defined which do not attempt to accomplish any defined goal
4. Situations which may arise but for which no goal or task is defined

It is important to note that incorrect knowledge identified during knowledge acquisition does not mean that the experts knowledge is wrong and it would be very prudent for the knowledge engineer to make this very clear to the expert. Instead, incorrect knowledge identified during knowledge acquisition is most often an indication of an error of communication, some type of misunderstanding between the knowledge engineer and the domain expert.

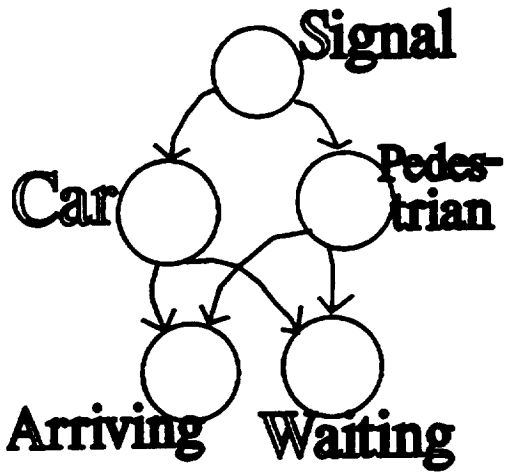
More information on knowledge acquisition correctness checking can be found in reference [10].

Incorrect knowledge can be most easily identified (and prevented) by representing the knowledge in a form that is easily understandable to both the expert and the knowledge engineer. Although a complete survey of knowledge representations is beyond the scope of this course, the following are some common ones that can be used.

- Decision tables: these are useful for identifying incompletness among goals, tasks, and situations. A decision table may list goals vs. tasks or situations vs. tasks. For example,

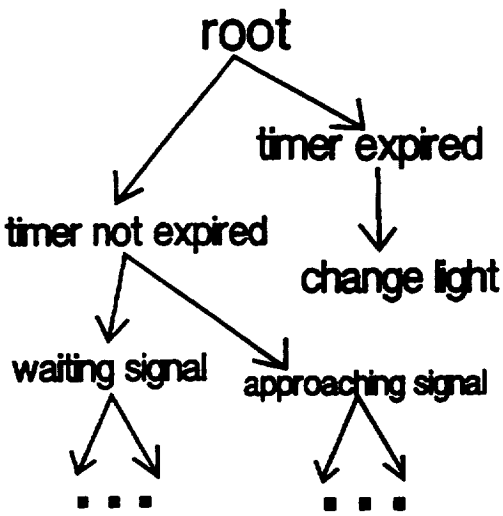
the state diagram for the TLC problem could be represented as a decision table by having the columns in the table being states and the rows being different situations. The elements placed in each row/column position would be the state to be transitioned into, given the system is currently in the state represented by the column and the situation represented by the row has occurred. References [14] and [15] give additional information on the use of decision tables.

- Concept trees or maps: these are useful for identifying inconsistencies in the definition of concept terms. Terms can be defined in a classification hierarchy (a tree) or in a type of Venn diagram where similar terms are drawn closer together than dissimilar terms (a concept map). For example, the terms associated with signals in the TLC problem could be represented in a classification hierarcy as shown below.



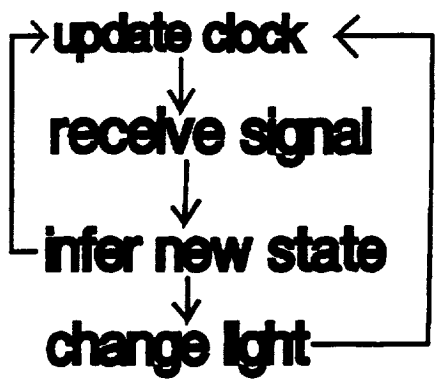
Reference [12] contains additional information on concept trees and maps.

- K-Trees: these are useful for identifying incompleteness and inconsistencies in heuristics acquired. K-Trees are similar to decision tables and classification hierarchies. K-Trees are n-ary trees where each non-leaf node represents a condition and the leaf nodes represent the consequences (some final decision). Tracing a path from the root node to any leaf identifies all the conditions that must be true in order for the consequence to be true. K-Trees are usually better than simple rules when analyzing heuristics. For example, conditions and actions for the TLC problem could be represented as shown below.



Reference [13] contains additional information on K-Trees.

- Task timelines: These are useful for analyzing expected sequencing and timing of tasks to be performed. These are most often used for representing high level problem solving (i.e., the flow of major tasks involved in the problem solving approach). For example, the overall TLC problem solving behavior could be represented as shown below. Reference [12] contains additional information on task timelines and task analysis.



Minimum Competency Testing

Many types of human experts are certified by giving them minimum competency tests. Examples included medical doctors, accountants, real estate brokers, professional engineers, and Space Shuttle flight controllers. So it is natural to consider developing similar types of competency tests for ESs, especially when an ES is to serve in the same role as an expert who had to pass a minimum competency test (e.g., requiring an ES for Space Shuttle flight control pass the same tests that a human flight controller had to pass).

In two ways, minimum competency testing is similar to statistical testing. The first way is that most human expert certification tests do not require perfect 100% accuracy. That is, human experts can fail to correctly answer some of the test questions and still pass the test. But if they answer a certain percentage correctly, then they are "good enough"; that is, they are not perfect but they have

the required minimal level of competency. The second way that minimum competency testing is similar to statistical testing is that minimal competency tests are often representative of the types of problems that the expert will encounter "on the job"; that is, they are typical operational scenarios.

A good exercise for class discussion at this point is the following. Assume that the TLC problem is to be solved by a human traffic controller (a traffic cop). What are some of the questions that would be on the traffic controllers certification exam (i.e., his minimal competency test) ?

More information on minimum competency testing can be found in reference [11].

Disaster Testing

As when validating most systems, validating ESs often involves spending more time testing situations that should never occur than is spent testing situations that occur routinely. That is because correctness of the system is sometimes most important when it is dealing with disaster or potential disaster situations which should rarely if ever occur. And because errors, especially incompleteness errors, are more often found in parts of the system which deal with disasters (this seems to be due to the complexity of disaster situations).

Disaster information (information associated with recognizing disasters and potential disasters, and with preventing them or recovering from them) is important for V&V of ESs. In addition to the importance of testing that ESs correctly react to diasaster and near disaster situations, disaster information is needed to check for completeness of the system (i.e., to make sure all situations are properly covered). It is also useful for exploring how well the system understands what situations it does and does not know how to handle. This is because disaster sitations can get complicated and require much more knowledge and reasoning than the ES can provide. Thus, disaster situations are useful for testing whether the ES can give up and declare that it can not (safely) handle a situation.

In addition to using disaster information for generating test cases, it can also be carried through the design. That is, the system can be verified to comply with disaster requirements during its development. This is most conveniently done using specification-directed analysis because disaster information is typically most conveniently represented as a constraint specification.

Human experts are often very good at quickly recognizing potentially disastrous situations and preventing disasters from occurring. Although some

experts enjoy recounting near diasasters they have encountered in the past (so called "war stories"), experts often sometimes neglect to discuss disaster situations; sometimes situations that should never occur are taken as "common sense". Because of the importance of disaster information for V&V of ESs, the knowledge engineer should pursue acquiring disaster information from the expert during knowledge acquisition.

An obvious example of a disaster situation for the TLC problem is that the light should never be green in both directions. This would allow for traffic in both directions to collide.

Expert Review

Generally, the best people to check correctness are the experts themselves. Because experts already understand the problem to be solved, the knowledge needed to solve the problem and may already know how to solve the problem manually, the only thing they may be missing is an understanding of the ES solution. So the key to expert review is representing the review material so that an expert can (easily) understand it.

Usually, the expert can easily understand test case scenarios and test results. So they almost always can and should be checked by the expert. Experts should also be involved in checking the correctness of acquired knowledge. Some form of the knowledge acquired from them or another expert should be given back to them for review. Many of the knowledge representations such as concept trees and K-Trees are easily understandable by an expert once they are given a little training in the representation.

And with a little extra work to train the expert and/or re-represent the solution in a more natural form (e.g., narrative English), the expert can check the system design. They can check the overall problem solving approach of the design (e.g., the major tasks and the flow between tasks). Sometimes they can even check the details of the implementation to ensure that the acquired knowledge was correctly interpreted during implementation of the system.

Reference [12] contains addition informations on and suggestions for the use of expert reviews.

Summary

This section has discussed the differences between ESs and CS, pointing out that the differences can be grouped into two categories.

1. Differences due to how ESs are usually implemented. These differences are due to the use of non-procedural languages.

2. Differences due to the types of problems that ESs are usually built to solve.

To illustrate the first category of differences, two approaches to solving the TLC problem were presented, one approach was conventional (written in Ada) and the other approach used a non-procedural language (CLIPS).

To illustrate the second categoory of differences, a new "more ES-ish" version of the TLC problem was presented and was compared to the more conventional TLC problem.

Finally, a set of ES V&V techniques were summarized. Some of the techniques addressed the ES implementation differences while other techniques addresssed the ES problem diffferences.

References

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

2. Nazareth, D.L., "An Analysis of Techniques for Verification of Logical Correctness in Rule-Based Systems", Phd. dissertation, Case Western Reserve University, 1988

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

   Section 2 is the description of the capabilities of CRSV

4. 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.

5. 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.

6. 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

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

8. "Designing a Solution for the Traffic Light Problem Using Terms, Operators, and Productions" - This is the first case study in the Case Studies section.

9. 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)

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

11. "Quality Measures and Assurance for AI Software", This is the last reference in the references section of this workshop

pp.74-79 includes a discussion of minimum competency testing

12. 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

pp. 135-151 discusses concept maps (concept distionaries) and other concept organization techniques

pp. 173-178 discusses task timelines and other task analysis topics

13. Kernam, G., Koltun, A., Schwartz, E., "Programming ESs at the K-Tree Level", Sigart Newsletter, July, 1989, ACM Press

14. Montalbano, M., "Decision Tables", Science Research Associates, 1974

15. Weiss, S.M., Kulikowski, C.A., "A Practical Guide to Designing Expert Systems", Rowman & Allanheld, 1984

pp. 118-123 discusses decision tables in the context of expert systems

Expert Systems V&V Guidelines Workshop

Part 3: Guidelines

By Scott W. French and David O. Hamilton

This section is part of the Expert Systems V&V workshop. It presents a set of guidelines for V&V of expert systems. These guidelines are based on the materials presented in Parts 1 and 2 of this workshop.

Table of Contents

C-2

Introduction

Goals

The goal of the workshop is to equip the students with the knowledge necessary to develop and implement an overall V&V approach for the ES they are currently working on or the very next ES that they develop. That is, the student should be able to leave this course and begin to apply all the material that has been presented. In order for the students to be able to do this, they will need to know not only about ES V&V techniques but also when to apply the various techniques and how to combine different techniques into an overall V&V approach.

The goal of this final section is to provide a set of guidelines on when and how to apply the various techniques. To be most useful, these guidelines are formatted into a step-by-step procedure.

Specifically, the goals for the student are to:

1. understand the guidelines, including the rationale for the guidelines

2. understand how to combine the ES V&V guidelines with conventional software (CS) V&V guidelines (That is, the student should be able to develop a combined V&V approach for systems that include CS as well as ESs)

3. understand how to tailor the guildelines for different types of ESs (That is, the student should be able to identify specific characteristics of their particiar ES and tailor the guidelines to fit these characteristics.)

Overview

Before beginning the discussion of guidelines, some general misconceptions are reviewed. These include misconceptions about software in general in addition to expert systems. The purpose of covering these misconceptions is to create the proper mind-set for discussing the guidelines and begin motivating the students so they will be more receptive to the guidelines.

The approach used to presenting the guidelines is motivational. Building on the mind-set created by discussing the misconceptions, a set of conclusions are made from the material in parts 1 and 2 of this workshop, conclusions on what can and should be done in the V&V of ESs. The conclusions of these inferences lead directly into the general guidelines.

The guidelines are simply listed and described. Because they have been

motivated by previous discussion, they should not need a lot of explanation.

The final discussion of tailoring the guidelines is, in many ways, the most important discussion of the workshop. It is where all the material presented is brought together into a format that the students should try to follow when they return to their jobs following this workshop.

Common Software Misconceptions

Software in General

A naive view of software development is that once the only end product is the executable software. But to understand how to use the software, some type of user's, reference, and/or training material must also be supplied. Also, software rarely remains unchanged after its first release but instead is continually updated and revised many times. In order for maintainers to safely and effectively change the software, some type of maintenance information must be provided. Finally, because each change to the software requires reverification and revalidation of the entire system (not just the changes made), V&V work products such as test cases must be preserved to for reverification and revalidation.

It has also been suggested[1] that the documentation be organized so that it appears that the system was developed in an ideally systematic way. That is, by looking at the documentation, it appears that the system was developed by perfectly following a particular development methodology (a "rational design process").

Rapid prototyping is becoming a frequently used development approach, especially for expert systems. To take full advantage of the prototype, people often simply continue developing the prototype, adding capabilities and performing V&V, and make the prototype into the product/operational system. However, this approach is usually a misguided one. Prototypes are almost always built as quickly and cheaply as possible, taking shortcuts whenever possible and using such a ad-hoc system as the foundation for a long term operational system can lead to many problems. Another problem regarding evolving from a prototype is more subtle but somewhat more important. This problem is the assumption that a prototype of a piece of the problem will scale up to a solution to the entire problem[2,3]. This is a particular problem for ESs because one can sometimes solve a portion of the problem very easily using some of the powerful non-procedural languages but run into extreme difficulty solving all aspects of the problem. The leap into a full development effort based on the success of a small prototype has been the cause of more than one ES project failure.

As discussed in part 2, formal methods (specification-directed analysis) are widely considered to be impractical

for V&V of most software because they are too expensive and difficult to apply. Anthony Hall[9] lists seven misconceptions about the use of formal methods (too expensive, too difficult, etc.). There is a lot of confusion about this subject and a lot of controversy. Without getting sidetracked into the esoterics of program proving, an important point to bring up is formality vs. rigor. One dictionary definition of formal is "based on conventional forms and rules" and a definition of rigor is "strict precision - exactness" (both are from Webster's New Collegiate Dictionary). It is possible to construct rigorous informal arguments about programs, ones that strive for precision but are not constructed using the convention and form of symbolic logic notation. An example was given in part 2 concerning unique naming of timers. We were able to precisely show that timer names must be unique at the end of each time cycle but we only constructed an informal natural language argument. However, it is still helpful to understand formal arguments so one can avoid some common logical pitfalls and generally do a better job of constructing rigorous arguments.

Another misconception about formal methods is that they are all about proving that programs are "correct". There are two problems behind this misconception. One problem is in defining correctness; we have to know what "correctness" for a certain program means before we can even begin to think about proving correctness. And as we have seen in part 1 (pieces of the verification puzzle), there are many different forms of correctness and some of them are quite open ended. So we can not even begin to think about proving complete correctness of a program. Instead we are limited to proving certain properties about programs (e.g., unique timer names). Another problem is that some forms of correctness such as user-interface correctness do not lend themselves to rigorous arguments, formal or informal. But once we know what we can and can not justify through the use of formal methods and use formal methods for what they do best, they can be very powerful tools.

Expert Systems/AI in Particular

ES technology is often thought of as being some magical new set of tools for building intelligent software quickly and easily. In reality, one spends a lot of time and effort to construct software that is quite unintelligent, though highly useful. Part of the reason for this misconception is that non-procedural languages can be used to quickly solve some difficult looking problems. Unfortunately, real expert systems must incorporate a lot of knowledge that is often difficult to aquire and must be

developed and tested as any other type of software. Roger Schank, a well known AI researcher who has spent a lot of trying to build real-world intelligent programs has summarized this concisely by saying "AI entails massive software engineering. ... Software engineering is harder than you think: I can not emphasize strongly enough how true this statement is."[4].

Another ES misconception is that they are "all-or-nothing"; something is either an expert system or not an expert system. In part 2 of this worshop, we have discussed some characteristics of expert systems and pointed out that not all expert systems have all of the characteristics. This might lead one into trying to decide whether or not something is an ES. As mentioned earlier in the workshop, it is better to just look for ES characteristics and not worry about whether or not the overall system is really an ES or not. It is important to note that one must look for all characteristics and not just the easily recognized ones. For example, one should not assume a system does not have any ES characteristics just because it is written in a conventional procedural programming language. Likewise, one should not assume that a system has all the ES characteristics simply because it was written in a non-procedural ES language. Finally, it should not be expected that all the parts of a system, especially a large embedded system, consistently share characteristics. Instead there may be parts that have some ES characteristics, other parts that have other ES characteristics, and some parts with no ES characteristics at all.

A misconception about ESs that is important from a V&V point of view is whether or not they can ever be trusted. Although it has been suggested that ESs are inherently unreliable, and even unpredictable, because they are based on heuristic information, it is important to remember that ESs are still computer programs executing in a deterministic computer. Therefore they are predictable; they can be analyzed and predictions made as to whether they will operate as desired or not as desired. The heuristics upon which an ES is based can also be analyzed and certain properties about the heuristics can be made and proven. Most importantly, one can determine, with a small degree of uncertainty, whether or not an ES will be safe to use or not.

An ES is more than just a non-procedural program; it has more characteristics than that. So there is much more to learning about ESs than just learning ES shells (i.e., ES languages and their development environments). And to V&V ESs, it is best to have a good understanding of ESs. To fully understand ESs, one needs

to understand many forms of knowledge representation, many forms of reasoning strategies, how to acquire knowledge from an expert, and how to engineer software. The most appropriate form of representation and reasoning strategy should be used for the problem at hand and using the right form makes analysis of the system much easier. Also, it is good to understand and be prepared for common problems encountered in knowledge acquisition so that bad, incomplete, or poorly organized knowledge is not used as a basis for building and analyzing the system. Finally, since a large part of building an ES is basic software engineering, a good understanding of software engineering is needed to build and understand ESs. Because so much knowledge is needed to build ESs, two types of people are sometimes involved in the development, domain engineers and system engineers. Domain engineers are more familiar with the subject domain and how to acquire and represent knowledge. System engineers are more familiar with the computer aspects of ESs such as non-procedural languages and software engineering.

Implications for Guidelines

Overview

So far, a lot of foundation information has been discussed, including key V&V ideas, conventional V&V techniques, ES characteristics, ES V&V issues, ES V&V techniques, and some common related misconceptions. Most of the ideas have been illustrated using a single problem, the Traffic Light Controller (TLC) problem.

Based on all this information and ideas, a set of conclusions can reached about what one should consider when doing V&V of ESs. And a set of straightforward guidelines can be generated that address the considerations. We will follow this approach in this section by first listing all the key ideas that have been discussed so far. Under each key idea, we will list a set of implications that directly follow from the key idea. These implications could be about some design or requiements information needed for V&V, a development approach that would simplify V&V, when a certain technique would or wouldn't apply, or a general consideration for managing the V&V process. Most of these implications may seem somewhat trivial but they are all very important considerations and should not be forgotten.

Continuing the basic approach, next a set of guidelines will be listed. These guidelines based on the implications. That is, will be suggested approaches to collecting and using information needed for V&V, determining the right set of V&V techniques to use, and planning/managing the V&V process.

Finally, a straightforward step-by-step approach to V&V of ESs is discussed. This high level approach makes use of all the suggestions (i.e., guidelines) previously listed and includes steps for identifying key ES characteristics and tailoring the V&V approach based on the characteristics identified.

Conventional Validation Implications

The first key idea is what is meant by V&V. Validation can be loosely defined as trying to answer the question "am I building the right product". An obvious implication of this definition is that in order to perform validation one must be able to determine when something is the right product or not. That is, one needs to know or at least have a very good idea of what the user/customer wants. And ideally this should be stated early in the

development process. This statement of what is desired, i.e. what the right product is or is not, is called *requirements*. If there is sufficient time to do so, it would be good to read the following poem to the class which will give emphasis to this very key consideration.

Twas the Night before Crisis

'Twas the night before crisis,
and all through the house,
not a program was working,
not even a browse.

The programmers were wrung out,
too mindless to care,
knowing chances of delivery
hadn't a prayer.

The users were nestled
all snug in their beds,
while visions of windows
danced in their heads.

When out in the lobby
there arose such a clatter,
that I sprung out of bed
to see what was the matter.

And what to my wondering
eyes should appear,
but a Super Programmer,
oblivious to fear.

More rapid than eagles,
his programs they came,
and he whistled and shouted
and called them by name.

On Update ! On Add !
On Inquiry ! On Delete !
On Editor ! On Closing !
On Functions Complete !

His eyes were glazed over,
his fingers were lean,
from weekends and nights
in front of a screen.

A wink of his eye,
and a twist of his head,
soon gave me to know
I had nothing to dread.

He spoke not a word,
but when straight to his work,
turning desires into code,
then turned with a jerk,

And laying his finger
on the ENTER key,
the system came up,
and worked perfectly.

The updates, updated;
the deletes, they deleted;
the inquires, inquired;
and the closing completed.

He tested each whistle,
he tested each bell,
with nary a re-boot,

and all had gone well.

The system was finished,
the tests were concluded,
the client's last wishes
were even included !

The user smiled and then gasped
at what he had seen,
"It's just what I asked for,
but it's not what I need."

- Anonymous

The moral to this story is to try not base the system on a loose informal indication of what the user/customer wants. Sufficient time should be spent validating requirements to make sure they are what the customer/user wants and are not just what was asked for.

Recall that there are many different pieces to the V&V puzzle. There are different kinds of correctness that each need to be considered. One must know what types of correctness are most important. At a minimum, one must know whether or not the system will satisfy the user's needs.

Another part of the verification puzzle is completeness and consistency. Once one understands the problem to be solved and what the system is expected to do, they should analyze this understanding, checking for inconsistent statements and signs of missing information.

An important implication of the verification puzzle key idea is that once all the pieces to the puzzle are known, they must all be fit together. That is, once one has identified all the correctness considerations and developed a complete and consistent picture of the problem to be solved, he or she must develop a V&V approach to address the type of problem and correctness considerations.

Recall that validation is based on a black-box view of the system. That is, the tester does not look inside the system but only bases success or failure on observable behavior of the system. This key idea implies that, in order to perform validation, one must know what is correct behavior. A standard way of describing expected behavior is via stimulus/response pairs. That is, one needs to know more than what knowledge to base the system upon; all the stimuli and associated responses must also be identified.

Recall that a natural way of organizing stimulus/response pairs is by creating operations scenarios. Each scenario is a series of stimulus inputs to the system along with expected responses from the system. Thus, if users can describe how they expect to

use the system, the knowledge engineer can extract stimulus/response pairs from these descriptions which can be used to create validation tests. One good way of helping the user generate the scenarios is via prototyping. A knowledge engineer can get operational scenarios by observing users using a prototype as they would expect to use the operational system. Prototypes make it easier for users to visualize how they will use the eventual system and knowledge engineers can better understand the scenarios by observing them than by hearing them described.

All of the validation implications can be summarized in one word - ***requirements***. Requirements should contain information about all the relevant forms of correctness and how the system is intended to be used. Requirements should be thoroughly analyzed for consistency and completeness. Prototyping is a useful method for early analysis of requirements.

Conventional Verification Implications

Recall that comprehensive validation of any complex system is practically impossible. However, by dividing the system into managable pieces and by looking inside the system (i.e., doing verification), one check for correctness less expensively and more comprehensively. The obvious implication of this key idea is that verification greatly reduces the cost testing a system. But verification adds new pieces to the V&V puzzle, namely when and where to check what types of correctness.

Verification can loosely be defined as trying to answer the question "am I building the system right ?" It directly follows from this definition that to do verification, one must understand how the system is being built and how it *should* be built. How the system is being built can be seen by looking inside the system at various points during development. How the system should be built needs to be documented in some type of system design.

Recall that a key aid to verification is modularity. Dividing the system into small relatively independent pieces benefits verification in many ways. So it is obviously advantageous to develop a very modular system.

Many different verification techniques exist but none of them are comprehensive. Each is best at detecting certain types of errors, can best be applied at certain points during development, and requires certain inputs. This imples that a mixture of techniques should be used. Furthermore,

to minimize cost, the techniques should be applied in a certain order (i.e., at a certain point in the development process).

Another key point is that the eariler an error is found, the more cheaply it can be found and fixed. This directly implies that verification should be done as early as possible and that emphasis should be placed on early detection techniques.

The static testing class of techniques can be applied earliest because they do not rely on executing the software. So they should be given particular emphasis. Unfortunately, many of these techniques are difficult (and somewhat painful) to apply. This is not so much because the techniques are complicated but because they require a certain amount of discipline on the part of the programmer. They can also lead to tedious and time consuming work.

Abstraction, refinement and proper documentation (as well as modularity) all make static testing easier. Since a good design can make static testing, which is important, much easier, having a well designed system should be a high priority.

All the verification implications discussed in this section can be summaized in one word - ***design***. How the system is designed has a strong and direct impact on how easily the system can be verified. It is also part of development and thus the cheapest place to find and correct errors.

General Expert System V&V Implications

The key idea that has the most implications for ES V&V is that ESs are software. Being software, the basic V&V implications discussed in the previous section all apply to ESs. It also means that one could start with a convetional V&V approach and modify it as necessary for ES characteristics.

The above implication turns out to be very convenient because as has been pointed out, ESs may satisfy some, but not all of the ES implementation and/or problem characteristics. So if ES V&V and conventional software V&V were radically different, one would have to make the difficult decision of whether the system should be treated as conventional of as ES. Instead, since ES V&V is more of a variant of conventional V&V, one can use a conventional V&V approach as a base and modify it as necessary based on the ES characteristics (and extent of them).

Expert System Validation Implications

Recall that one ES characteristic is that the system mechanically applies the experts heuristics ("rules of thumb") to solve a problem. This occurs when the solution to the problem already exists in a expert and this solution need only be translated into a computer program. An implication of this characteristic is that since the ES is a "clone" of the expert, it should behave closely to the expert. So it must be validated by comparing it to the expert. An implication that is important but so obvious that it might be overlooked is that an expert must be readily available for validation. It is always obvious that experts needs to be available during knowledge aquisition but sometimes forgotten that they also need to be available for validation activities.

Instead of the ES being a clone of ane existing expert, it may be a new solution to a complex problem that has never been adequately solved. Oftentimes in such cases, it is difficult to determine if a solution to the problem is truly correct. For example, in order to check whether a certain schedule is an optimal one, all possible schedules may need to be generated in order to show that none of the other schedules is better. If this is impractical, then one may have to be satisfied with the fact that the generated schedule is reasonable and "looks good" (e.g., it is at least as good as the schedule generated by any previous approach).

It may be the case that solutions can be easily checked for correctness but can really only be checked by an expert. This is because correctness my be vagely defined. It may be possible for a non-expert to tell whether or not some of the ES responses are right but hard to determine the correctness of others. In these cases, the expert will need to be available to help in this analyze test case results.

Expert System Verification Implications

The format of a software system's design and code has a large effect on the verification of that system. Because the format of ES design and code is very different from conventional software, it would seem that there would be some related verification differences and as we have seen, there are some differences. Most of the verification implications for ESs are due to the use of non-procedural languages.

As was discussed in part 2, the internal interactions between parts of an ES can be complex and difficult to follow. Examples of rule interaction were given in the sample solutions to the

TLC problem. Rule, frame, etc. interaction, especially if it is complex and not made explicit can make knowledge bases extremely difficult to analyze. For example, checking the correctness of a single rule potentially requires analyzing possible interaction (both direct and indirect) with every other rule in the knowledge base. A direct implication of this issue is that inspections, which are an important verification tool, can be much more difficult and less effective than with conventional software.

A problem closely related to the internal interaction problem is that non-procedural programs, by their nature, do not make the problem solving method explicit. So when a particular problem solving method needs to be implemented in an ES, it can be difficult to verify.

Internal interaction and lack of explicit problem solving methods create the biggest ES verification problem - difficulty in manually analyzing knowledge bases.

A separate issue is implied by the highly iterative development that is most often used in developing ESs. Each time a program is changed, it must be reverified before it is released. Additionally, if this issue is combined with the need to catch errors as early as possible in the development process, it can be concluded that a program should be reverified each time it is changed, whether the newly changed version is released or not. So the more iterative the development process, the more reverification is done. So it is apparent that ESs will involve a lot of reverification such as regression testing.

Other Implications

Aside from conventional software and ES characteristics and aside from software issues at all, there are a few "common sense" implications that should be clearly made. These are called "common sense" implications because they do not rely upon any analysis of the characteristics of a system and are ones that it would be expected that anyone would immediately recognize.

The first implication is that V&V should be performed and it should be performed as a distinct activity. Any effort to check the correctness of a system is a V&V activity and one can not know if the system solves any problem without checking to see it if is correct. So if correctness is important, V&V must be done. If correctness is in no way important, that is if one either has no expectations about what the system will do or does not care what the system does, then it is questionable why the system is being built at all.

Another common sense point is that V&V will take time and money. A recent survey of ES projects found that, on average, about 20% of the total development cost is spent on V&V. So if one needs an estimate of the total development cost, the cost of V&V should be included in the estimate. Additionally, many of the V&V activities really need to be done by the people developing the system and some of them will take some extra time (e.g., minimum competency testing). Time should be included in the development schedule for V&V activities. When they are not included in the schedule and an attempt is made to skip V&V or do a minimal amount of it, especially early in the development process, there is a great risk of schedule over-runs later in the process when errors begin showing up and are time consuming to correct.

One common sense point mentioned earlier is that the expert is the best person to check the correctness of the system. So, obvously, one should try to involve the expert in some of the V&V activities. This may be difficult to do since often the expert's time is hard to get. But it is important to keep in mind that it is easier to get part of the experts time if one asks far in advance of the final validation of the system. It is much harder to get the expert to drop other things at the last minute to help with validation of the system.

It is part of human nature that after someone looks at something for a long time, they begin to see nothing new or different in it. The also become bored at looking at it. So if someone misses an error the first time they analyze a program, they are less likely to catch it the second time, even less likely to catch it the third time, etc. For this reason, and for many other reasons, the developer of a progam is usually the least qualified to evaluate the correctness of it. Analysis by another person who was not involved in development can often find more errors; this is usually called *independent* V&V. So independent V&V (IV&V) should be done if possible, that is, if one or more people who were not involved in the development of the system can be found to help with V&V.

Guidelines

Overview

From the implications discussed in the previous few sections, some recommendations for V&V of ESs are readily apparent. After having discussed all the implications, the recommendations, or guidelines, can be listed with little discussion. To make the guidelines a little easier to use, they are organized into major categories

What will require a little more discussion is how to approach the V&V of a system "from scratch". That is, it is often much easier to study V&V techniques and discuss general recommendations than it is to develop a complete V&V plan and approach for a project that has no current V&V plan or explicit approach.

Because the development of a completely new V&V plan and approach can be difficult the first time, the students will be given a longer exercise to practice this. That way, they will be much better prepared to apply the material in this workshop to their real ES projects.

Project Management Guidelines

Planning

Many of the implications were concerned with various aspects of preparation and planning for V&V. Various resources such as time, money, and personnel need to be allocated for V&V. It is not so much the case that V&V adds a lot of cost and time to the overall development plan but just that for the plan to be complete and accurate, V&V should be included.

All resources that may be needed for V&V should be included in the plan but one resource that is particularly important is the experts time. It has been discussed in several plances how the expert can be used to assist with V&V but usually the expert's time is difficult to get. Early planning to use the expert will allow the expert to plan ahead and reserve time for helping with V&V.

Also, V&V can get forgotten if explicit plans are made for doing it. This is because much of the verification work needs to be done early during requirements and design time yet sometimes the need for verification is not apparent until much later in the development when, as has been repeated many times, it is much more expensive to do.

There may also be a tendancy to skip the creation of several things that or needed for V&V because they are perceived as not important to the primary job of generating the system. For example, many verification problems can be mitigated by designing the system in certain ways. So planning to spend more time designing the implementation can payoff later in reduced verification cost.

In figuring out a way to manage V&V for a project, there is no need to start from scratch. There do exist life cycle processes along with associated descriptions and documentation formats that cover a lot of the conventional aspects and thus are a reasonable base to use for managing ES V&V. Just make sure that the life cycle includes all three types of testing. Also make sure that the life-cycle used as a base is easily tailored. Most well-documented processes are primarily for large projects so if they are to be used on a small project, they will need to be "downsized" to fit the project. The life-cycle may also need to be tailored to fit some particular ES characteristics. At a minimum, the life-cycle should support an iterative development approach. Finally, the life-cycle should be able to include support for checking all types of correctness and all types of V&V techniques. For example, the life-cycle should be able to accomodate prototyping.

A guideline that does not directly come from the implications but does address several of them indirectly is configuration management. In order to check correctness for a system, one needs to understand exactly what is included in the system. This is especially important if the system is changing a lot, as in a highly iterative development process. Proper configuration management includes keeping track of versions of all parts of the system and keeping track of which part versions went into a given executable system release. In addition to reducing the time spent tracking down problems only to find that there was a misunderstanding of what was included in the system, proper configuration management can help keep track of which parts have been V&V'ed and what type of V&V has been done on them. This is important to get the full benefit of a modular design.

Problem Analysis Guidelines

The better one understands the problem to be solved, the better one can check that a proposed solution does indeed solve the problem fully and completely. So it is a good idea to spend some time in the beginning of a project to understand the problem to be solved. This is especially important with ESs

because of the potential complexity and vagueness of the problems they address.

It may be necessary to try to reduce the size and/or simplify an initial problem description[8]. Often, the first description of the problem to be solved contains extra information that is not really part of the problem. It may also mention related problems that are not intended to be solved. It might include problems that would be nice to solve if possible but are not as important as the main problem to solve. Finally, as was discussed under ES misconceptions, there can be some over optimistic expectations about what can be done with ES technology and this leads to over overly large and complex problem descriptions. It is important to identify, as clearly as possible, the main problem to be solved and realistic expectations about what the system is expected to do to solve or help solve the problem. Once the main problem is solved or at least well-understood, it can be enlarged or enhanced. This approach of first working on a reduced problem and later expanding the problem is consistent with the iterative development and V&V approach.

When analyzing a problem to be solved, it is important to not commit to an implementation approach, especially not to commit to building an ES or not.

Silly as it may sound, one guideline is just to expect the system to work correctly[3]. This may sound silly because it should seem obvious that if one builds a system, one should like for it to work correctly. However, as has been discovered in surveying both ES and conventional software developers, there is typically a very low expectation about the likelihood that a system will work correctly, at least the first time it is executed. Such low expectations should be avoided because they can lead to a lack of confidence in early detection methods (i.e., an attitude of "No matter what I do, I know it is not going to work so let me just execute it and find out what isn't right."). So the low-expectation becomes a self-fulfilling prophecy.

Requirements Guidelines

A good set of requirements is the cornerstone of V&V. So either once the problem is understood or as the problem is being understood, it is important to state as precisely as possible what the system is expected to do. The description of the requirements may be formal or informal but it should be easily undersood by a wide audience of users and developers. User's need to understand the requirements so they can ensure that it describes what they really want. Developers need to understand the

requirements to ensure that they will be able to tell if the system is correct (so developers do not have to constantly ask "does this look right ?"). Prototyping parts of the requirments first can help users and developers come to a common understanding about what the requirements mean. But prototypes are very difficult to V&V against so in addition to the observable prototype behavior, a more precise statement of requirements is needed.

It is important that the requirements be as complete as possible. They should include a complete description of the behavior expected from the system. As appropriate, the expected behavior should be documented in the form of scenarios describing the expected operational use of the system. But unexpected uses and situations should also be considered (though one can never know for sure if all situations and issues are covered by any set of requirements).

In addition to the behavior of the system, all other types of correctness should be considered and addressed in the requirements.

Design Guidelines

An extremely important point of this workshop is that the way a system is developed, especially the way it is designed can either make verification much easier or much harder. The primary way that design can help with verification is by breaking up the system into small manageable pieces. The different pieces, or modules, can then be verified separately and verification of many small systems is much easier than verification of one large system. So modular designs should be developed.

Modular designs, as discussed earlier, are more effective for large systems when they are documented at differing levels of detail. So abstraction and refinement should also be used during design.

There are other ways that system design can make verification easier. To assist the verifier in checking the completeness of design, the design should be cross referenced back to the requirements. Checking of consistency can be made easier by using a common notation across all modules. It can be very confusing to compare two modules for consistency if they are documented in very different ways. It as important that each module be documented in a particular style as it is that the style is consistent. So, for example, although precise mathematical notation is, in general, good for verification, it is not good if all the other modules are consistently documented using structured English.

General Guidelines

Because Independent V&V can be effective, it is advantageous to look for a independent group of people to help in V&V, say, final validation. Sometimes a prospective user who has not been involved in V&V can serve this role. It is not necessary for a developer to spend extra time teaching the user about the system. For if the user can not figure out the system and validate that it does what it should from the available documentation that demonstrates that the available documentation is insufficient.

As has been emphasized several times, one should always try to identify errors and eliminate them as soon as possible. Also, one should use a variety of techniques instead of relying on any one technique.

V&V Technique Guidelines

For all modules that lend themselves to static testing methods, static testing should definitely be used. In other words, static testing should be used whenever possible and to the greatest extent possible. The specific static testing techniques will depend on the design/implementation langauge. Rule consistency checking should be done on rule-based systems while data consistency checking should be performed on frame-based systems. If a classification type of ES is being developed, sensitivity analysis should be used. In all cases, some type of consistent design notation should be used to create specifications for specification-directed analysis.

One class of errors that is difficult to detect using static testing techniques is user-interface testing. These are often best validated by exercising them. So, in order to find these types of errors early, one must find some way to execute user-interface functions early in the system. This can be done by creating simply simulations of lower level routines that have not yet been finished (called "stubbing out a module" or "scaffolding"), say by having them always return a constant value. This is sometimes called "scaffolding". In general, any function that one does not feel comfortable with after static analysis should be integrated into a scaffolded system and executed. In a way, this is like combining prototyping with normal system development.

The above approach can be combined with iterative development by first developing those functions that are the most risky. That way they can be checked out early. Then other modules that are more straightforward can be incrementally added. Each time, the system should be run through a realistic set of test cases. Good checking of user-

interface and utility correctness requires the system to be used the way it is expected to be used in actual operations. It is not good enough to run a few simple contrived test cases. Once a large enough portion of the system has been integrated, stress and performance test cases should begin to be executed to check out resouce-consumption correctness and to check out portions of the system that are only used in unusual situations. As the system grows, it is also good to randomly exercise all portions of the system, whether they are expected to be used often or not.

At various points during development, one should assess the amount of testing that various parts of the system have received. If there are major parts of the system that have received little if any testing then the testing approach (e.g., the test case suite) should be reassessed and enhanced to where more parts of the system are covered. To make test coverage analysis easier, it should be combined with the configuration management system so that one knows which parts of the system have been changed since they were last exercised by a set of test cases.

Recommended Approach

The previous set of guidelines are very straightforward to understand, given the material that has been covered in this workshop. What is not as easy is to see is how to put them all together in a real-world setting to develop and implement a V&V approach for an actual expert system. So in this section, a recommended set of steps will be covered that can walk someone through the process of developing and implementing a V&V approach.

Before discussing the steps, it is important to point out ahead of time that it is not necessary that each step take a lot of time. Each step involves asking oneself a lot of questions. It is not necessary to initiate a study project for each question. It may only necessary to ask oneself the question and spend a minute thinking about the answer. It is not always necessary to even arrive at an answer. An answer of "I just don't know" may be the right answer. The important thing is to not neglect to ask oneself a question ahead of time that could be answered and which would change the V&V approach, preventing a serious problem or issue from arising later.

Asking and answering even trivial questions and documenting the answers can be of great value on projects involving several people. The answer may be obvious to one person but not obvious to another. Or the answer could be obvious to two different people but they each think the answer is different.

Step 1: Analyze the Problem

The first step is to thoroughly analyze the problem to be solved. For ES projects, especially those that solve complex problems, this step may be quite long and involved. It may involve an extensive prototyping activity, including multiple prototypes, to better understand risky and/or complex parts of the problem. And if the total problem to be solved is very large, analysis of the parts of the problem may be done while other parts of the problem, that have already been analyzed, are being implemented.

As part of planning for V&V, it is important to look for critical and non-critical parts of the system. Hazard and fault analysis (see part 1) are useful techniques for identifying critical parts of the problem. If it is possible to redefine the problem to reduce critical parts, that will reduce the V&V needed, as well as making the eventual system much safer.

If parts of the problem will require significant amounts of knowledge to

solve, identify where that knowledge is to come from. That is, does it exist in the head of an expert ? In the form of written documentation ? Or will it need to be created ? This characteristic needs to be identified so it will be known if the knowledge in the eventual system can just be analyzed by an expert or, if the knowledge is created as part of solving the problem, extensive analysis and testing of the new knowledge will be needed.

Without thinking about how to solve the problem, it is important to begin thinking about what a solution to the problem would do and how it would be used. For example, would the system solve the problem and take action on its own ? Or would it make a decision and simply report the decision back to the user ? Or would extensive interaction between a user and the system be required in order to use the system ? This type of information can be used to begin identifying the kinds of correctness that will be most important. It also helps to understand the scope of the problem.

The problem should be compared against the characteristics of an ES problem. This is not so that one can determine whether an ES implementation technique should be used. Instead, it is only to determine the characteristics of the problem to predict whether any ES problem V&V issues will be encountered. For example, does it require human/expert judgement to determine if the problem has been successfully solved (i.e., will the expert be needed to look at test case results) ? Will it be possible to realistically determine if any solution is correct or can one only determine reasonableness of any solution ? Has this problem been solved before (i.e., can test results of the eventual system be compared to solutions generated by other means) ?

Step 2: Do Initial Planning

It is important to emphasize that this step involves only initial planning. Because much is often learned in the process of implementing an ES, comprehensive up-front planning is sometimes not a good idea. And, as with analyzing the problem, it is not the case that this step is only done once during the development of the system; it may be done for each iteration. Or it may be done continuously, as each new problem, issue or characteristic is identified, the plan and approach is adjusted accordingly.

Probably the most important consideration of initial planning is to determine what is to be accomplished (during the first/next iteration of development). That is, is the entire problem to be solved or just a certain

part of it ? Not only does this make it clear what the developers are supposed to do but it is also absolutely mandatory thing for V&V to be performed. Unless one knows exactly what goal one is working toward, there is no way to assess whether the goal has been achieved.

It is also helpful during this step to try to understand how critical each goal is. This is slightly different from analyzing the criticality of different parts of the problem. During initial planning, it is important to understand the criticality of each goal (for the first/next iteration) and how critical it is for that iteration. For example, suppose there is a goal of implementing a certain feature that to see what its effect will be on the rest of the system (i.e., if it will interfere with other features). This goal is noncritical in the sense that whether or not it works correctly is not as important as whether it adversly affects other parts of the system, especially other critical parts.

Estimating size and cost of software to be implemented is always difficult. It may even be much harder for ESs because of the nature of the problem and the implementation approach. However, it is important to make an estimate. Is it three *weeks* of labor or three *years* ? Also, be very sure to include V&V costs in this estimate. For some systems, this can make a very big difference. Consider again the goal of making a change only in order to determine its effects on the rest of the system. That change may be trivial to make but analyzing all its effects on the system may require extensive V&V.

Initial planning should also involve trying to identify appropriate milestones for the first/next increment. For example, if the expert will be needed during validation, it could be helpful to estimate, in advance, when they would be needed for validation. So one good milestone might be "System Testing Begins". Identifying milestones can be much easier if a standard life-cycle model is used. However, no one would suggest identifying a bunch of milestones that are not meaningful, whether they follow a standard life-cycle model or not. Milestones such as "10% complete" can be of little, if any, value to anyone.

From the initial costing and sizing, as well as from the answers to other planning questions, it will become apparent that certain things will be needed at certain times. For example, the expert might be needed again when system testing begins; not is it important to communicate this fact to the expert but the expert should be kept aware of status (i.e., is the "System Testing Begins" milestone expected to occur on

time ?) In general, one should be careful to follow up on actions identified during initial planning, especially if they involve acquiring resources that will be needed later.

The main input to the initial planning step is an understanding of the problem (from step 1). During initial planning, one should double check this information. For example, is the problem still too broad (i.e., is initial planning difficult because the problem is still not well understood ?) Another example is, given the estimated cost, is it worthwhile to try to solve this problem ? Such questions should be asked during initial planning instead of at some later phase of development.

The last but not least consideration of initial planning is to make sure that requirements have been created during problem analysis or else will exist early in development. Because this workshop is not advocating a development life-cycle, it is not said when requirements should be written but only that they are written and used in V&V.

Step 3: Perform Specification-Directed Analysis During Design

Design for the system should be created. It is not necessary that the design be documented separately from the implementation (in fact, verification would be easier if it were not) but only that the design of the system be documented. Also, it should be documented in a form that supports verification. That is, it should support static testing, preferably specification-directed analysis. Any of the specification forms mentioned in part 2 could be used. At reasonable points during design (at a minimum, as each module is completed), the design should be checked for completeness and correctness. This verification should be done by the designer but it should also be done by others also (to get some relatively independent views). The expert can be useful with high level design reviews but will probably not be needed for lower level design reviews.

The design should also be mapped back to some higher level document or information. Very high level design may be mapped back to requirements, a prototype, and information from problem analysis. As the design gets closer to the eventual implementation language (e.g., rules or frames) then a static testing approach suitable to the implementation language should be used (e.g., rule consistency checking).

Step 4: Check Each Completed Increment

In addition to doing static testing during design, dynamic testing should be done to check certain functions as soon as a module can be executed, usually at the end of an iteration of development. Tests to check the overall "health" of the system can check how well verification was done. If verification does not reveal many errors during design but the first initial tests reveal many problems, then that is an indication that verification was not being done effectively. This may require either a change to the design approach (e.g., to simplify it) or to the verification approach (e.g., using different techniques or a different type of specification language). These tests can be according to realistic operating scenarios or they can be random. Stress testing should also be done to check how effective verification was with parts of the system that are used in complicated off-nominal operating situations.

In addition to checking verification effectiveness, dynamic testing should be used at each iteration to check out types of correctness that are difficult to analyze statically such as user-interface correctness. Using realistic scenarios, each display should be brought up and inputs should be made. It is also good to check overall system execution, that system initialization such as opening files and system termination such as closing files are done correctly. That is because static testing is better for testing detailed functions than it is for predicting overall system behavior.

Before testing of the iteration is complete, try to check coverage of the testing. If there are no tools used to measure the coverage of test cases, then this may have to be estimated. But, at a minimum, the coverage at the module level should be determined, i.e., was each module executed at least once by a test ? If certain parts of the sytem were exercised little, if any, then additional tests should be generated and executed.

Either during or after the exercising of new features to the system, the enture system should be exercised to check that the new features did not interfere with the functioning of older features. Often this is done during the initial health testing of the system. But if analysis of test coverage reveals that there are major parts of the system not tested, especially if they are in areas that are related to the newly changed areas, then previous test cases for those areas (i.e., regression tests) should be re-executed.

It is good for developers to perform the initial tests but at some point during testing, others should be involved, especially potential users and experts.

Discussion

There should be no suprises to the student in the previous section. So rather than spending too much time discussing the previous section, it would be best to cover it relatively quickly and discuss how it might be applied to the enhanced TLC problem. This can be used as a "warm up" exercise for the main exercise in the next section.

The class exercise basically involves applying the four steps from the previous section to the TLC problem. This should be relatively easy because much of the work has already been done; most of the examples used to illustrate the material are based on the TLC problem. It is important to point this out so that the students realize that they are not expected to generate new ideas about the TLC problem. Instead they should be encouraged to bring up examples previously given and relate them to the four step approach. For example:

Step 1 : Problem analysis

- Much of the problem is conventional and only the "enhanced" TLC problem has characteristics of an ES.
- The ES parts of the problem are based on an existing solution in the form of an expert traffic cop so the solution will be designed by knowledge acquisition.
- Most parts of the problem are not vague and it appears that testers will be able to determine, objectively if the results of test cases are correct - an exception is judging fairness and for this it would be good for the traffic cop to check cases involving judgement about fairness.

Step2: Initial Planning

- The first iteration should probably involve implementing the conventional part of the TLC problem and a second increment used for the enhanced features.
- Because the first increment is relatively straightforward, no IV&V is needed but some IV&V will be needed for the second increment. The expert will also be needed for final valiation to check the results for fairness.
- It will be good to do some initial design which will result in identifying several modules which can then be sized. For example, the first increment appears to involve one simple, two medium, one one moderately complex module; so we could guess that it will involve about 4 1/2 days (4 1/2 = 1*1/2 + 2*1 + 1*2). Adding about 20% V&V time

still makes it about a week of labor, give or take.

Step 3: Design and Specification-Directed Analysis

- Several examples of specifications involving the conventional TLC problem have been given.

- For the enhanced version of the TLC problem, it still appears that a rule-based approch makes sense given the heuristics offered by the expert traffic cop. These heuristics could be made more formal in the form of preconditions and postconditions.

Step 4: Checking of Each Increment

- Coverage of the conventional TLC problem could be checked by checking that each state and state transition in the state-transition diagram was covered. If this involves firing each rule at least once, then this should be adequate.

- There is no real user-interface in terms of windows or menus. So no special testing for this is needed.

- The enhanced TLC problem could be tested statistically. Different arrival rates of cars and pedistrians can be generated in each direction, creating heavy, medium and light conditions in each direction. Developers can check that traffic seems to flow well in all directions but the traffic cop needs to be brought in to check some of the test cases for fairness. He should probably check those involving combinations of very heavy traffic in some directions with medium and/or light traffic in other directions.

Exercise

After discussing the four step approach as applied to the TLC problem. The class should be divided into several groups. Each group should be given one of the case study problems. For their given problem, they should think through the four step process. It will not be possible for them to ask all of the questions such as test coverage but they should be able to think through the entire process and see which questions they are able to answer. Most importantly they should think about the implications of the answers to each question and what things they would have in their V&V approach th address the implications.

It will be easy for some students to spend a great deal of time on this exercise. It is important to keep each of the groups moving through the process and not spending more that 10-15 min. on any step of the process.

Although this is a course on V&V, the students should now be aware of the imprtance of good design (e.g., whether it has a modular design). Since it is expected that the students also be ES developers (i.e., not just V&V'ers of ESs), they should be able to design solutions to the problems. So this exercise involves them designing very high level solutions to their given problem. They should be encouraged to think about keeping the designs at a high level, designing the solution so it will be easier to V&V, and designing it so that others can easily understand it. The reason for this last encouragement will become apparent to them because they will be asked to give their design, once it is fairly complete, to another team who will then act as an IV&V team to plan the final validation.

Once all the teams have traded designs and done the IV&V part of the exercise, they should discuss their thoughts on the exercise with the rest of the class. This can either be done as a class exercise, by having each group stand up and discuss their thoughts, or by going around the room, soliciting thoughts from each person.

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.

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.

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: Essays on Software Engineering, 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. 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

10. Bundy, Alan. "How to Improve the Reliability of Expert Systems."

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.

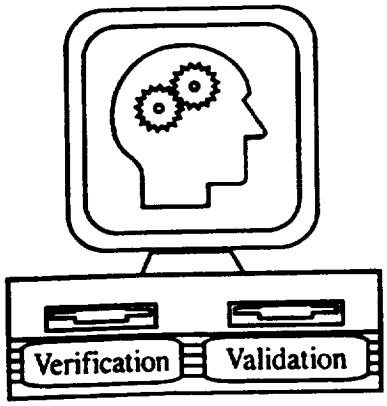
Workshop on Verification and Validation of Expert Systems Introduction

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



Agenda

Day 1 = Part 1: Conventional Software V&V

- Verfication and Validation (V&V)
- Conventional Software (i.e., non expert system) V&V techniques
- Primarily lecture

Day 2 = Part 2: Expert System V&V

- Differences between expert systems and conventional software
- Expert system V&V techniques
- Part lecture, part exercises

Day 3 = Part 3: Guidelines

- Summary of V&V considerations
- Recommended V&V process (guidelines)
- Some lecture, mostly exercise

Student Materials

1. Copy of all presentation slides
   - Part 1 (tab)
   - Part 2 (tab)
   - Part 3 (tab)
2. Handouts (tab)
   - Initially empty
   - Will be handed out periodically during course
   - Contains exercises and some possible solutions

Student Materials ...

3. Case Studies (tab)

- Two complete solutions to TLC problem
- Additional case studies to be used for final class exercise

4. References

- Collection of optional but suggested reading

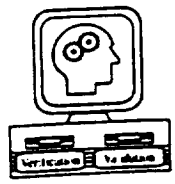
Class Participation

Questions encouraged during lectures

Class discussion questions will be posed (informal roundtable discussion)

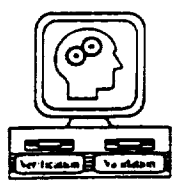
Will be divided into teams for some exercises

- 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.
- Differences between conventional and ES V&V
- Conventional and ES V&V techniques
- Some key V&V rules of thumb
- How to make V&V easier
- A suggested approach to V&V



Workshop On Verification and Validation of Expert Systems Part 1: Conventional Software

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

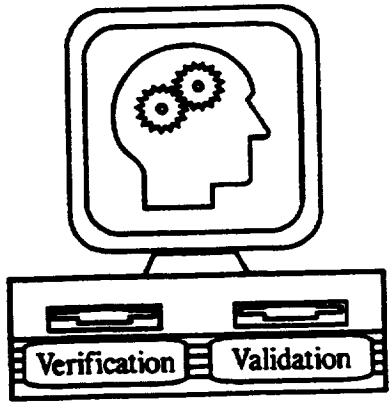
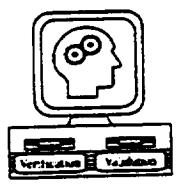


Table of Contents

Table of Contents ...



Introduction

Overview

Purpose

- Review conventional V&V techniques
- Justify the need for these techniques
- Illustrate techniques on a sample problem

Self-imposed Constraints

- Discuss techniques 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.

- In Part 1, we will discuss software in general, not expert systems per se.

Overview ...

Key Tenants

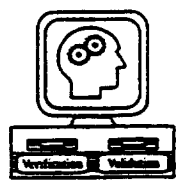
- A full understanding of the problem is never initially possible but must be developed incrementally along with the system.
- Correctness can never be practically proved and a system will always have errors.
- To develop test cases, one needs to understand the problem being solved.
- The earlier an 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 quickly and cheaply.**

- **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 ...

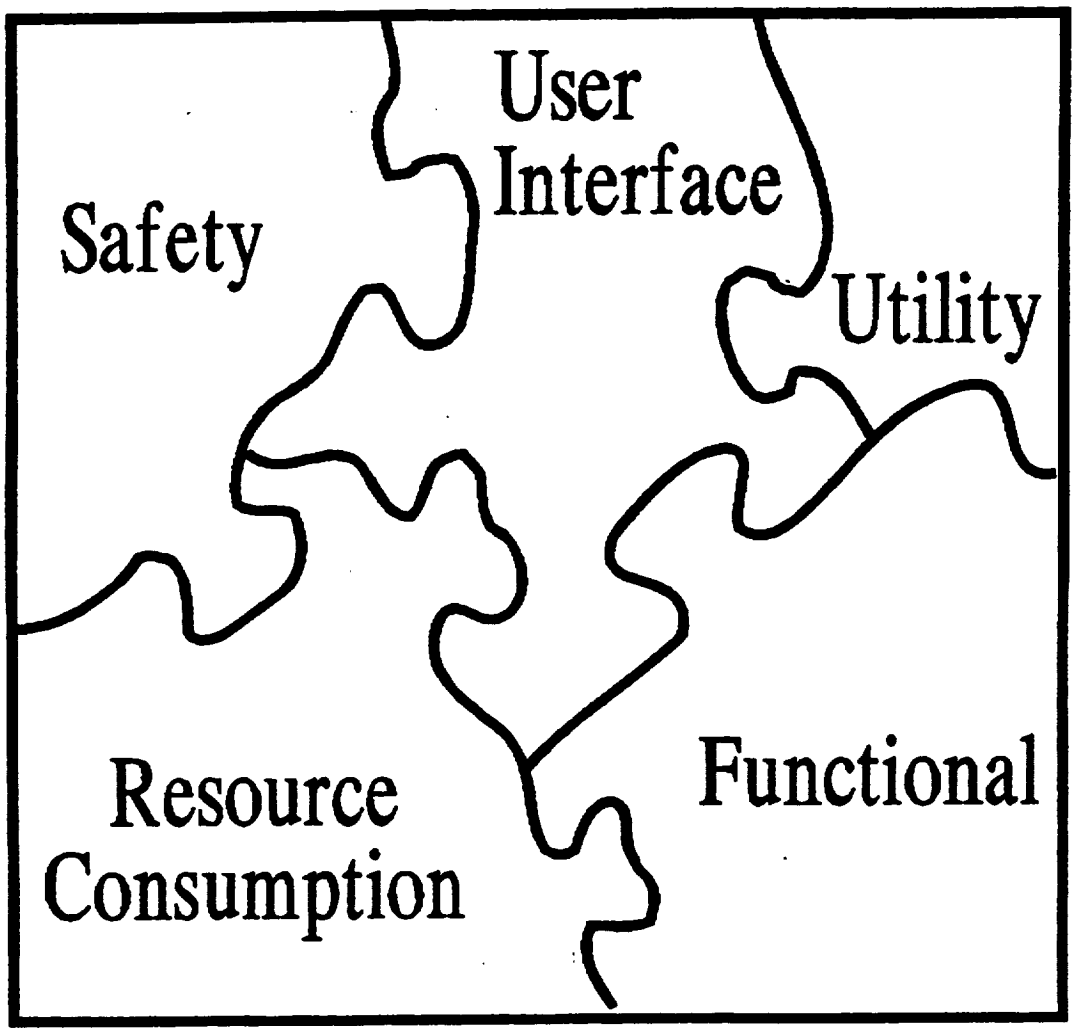
Pieces to *The Verification Puzzle ...*

- ***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



The Verification Puzzle ...

- Three aspects to showing correctness - consistency, completeness and termination.

1. Consistency

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

The Verification Puzzle ...

Aspects to showing 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 ...

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 ...

There are many V&V techniques

- **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 puzzle is to match techniques to situations.

The Verification Puzzle ...

This large puzzle can be divided into smaller puzzles called:

- *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 ...

These smaller puzzles are called *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*.**

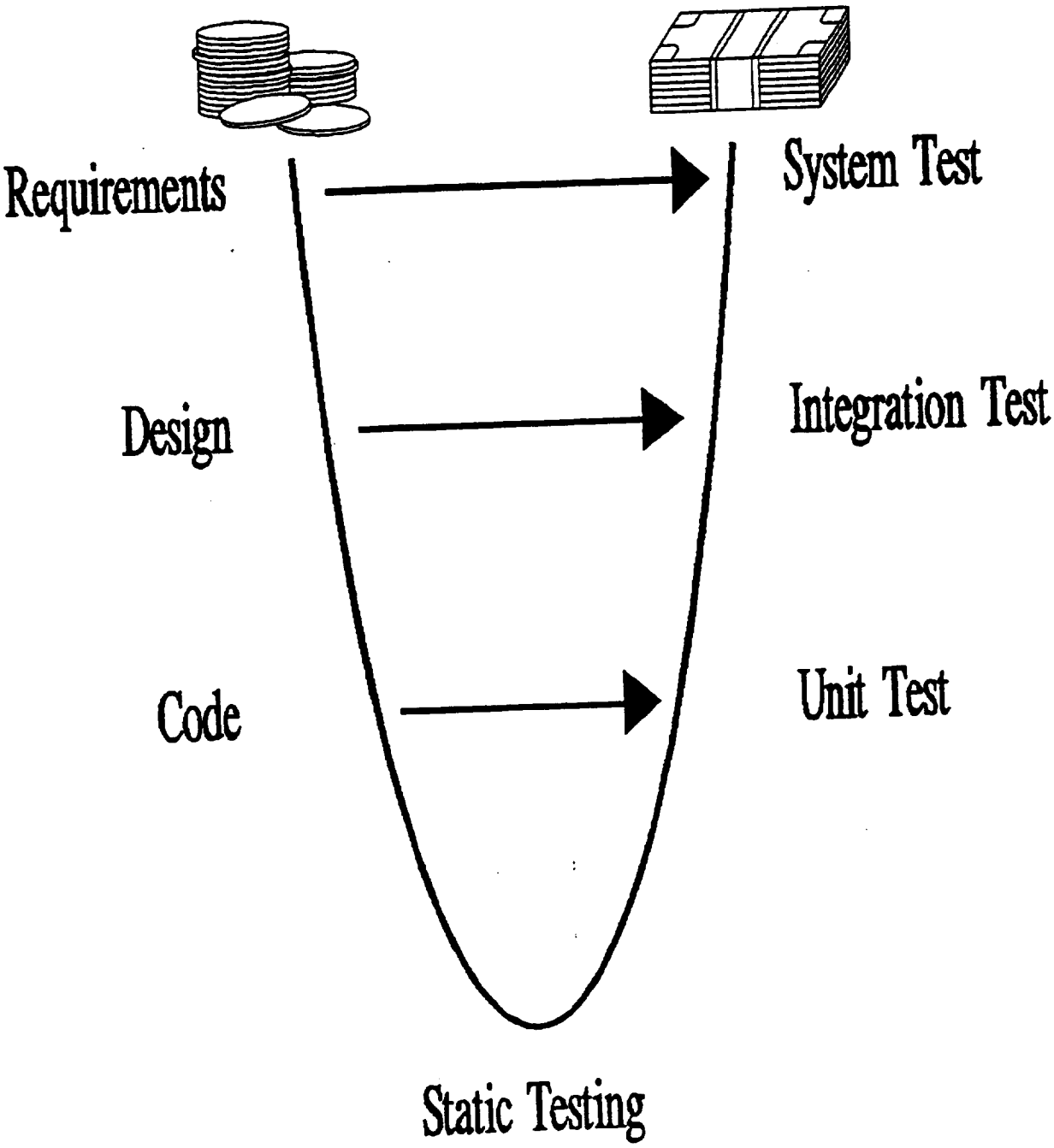
- **We will not restrict our discussion to any specific life-cycle.**

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

First, each phase will be examined, highlighting:

- ***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.**

Second, an example system will be discussed.

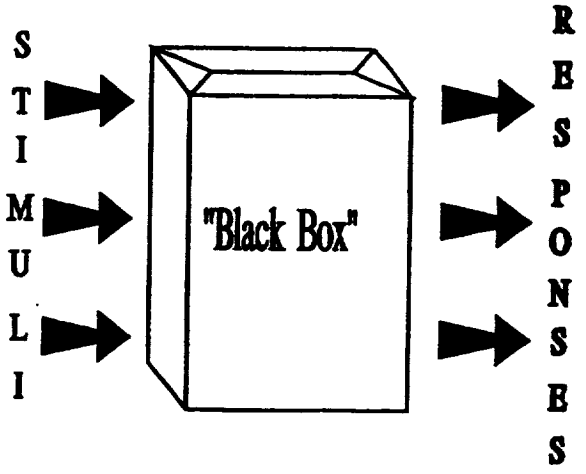
Third, for each phase, specific techniques will be discussed and illustrated using the example

Testing Phases

System Testing

Characteristics

- ***Black box:*** **Does not "look inside the system" to see how it was implemented; only looks at the systems required and observed behavior**



- ***Behavior*****: Can be described in terms of stimulus/response pairs**

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

System Testing ...

Verification vs. 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 ?"

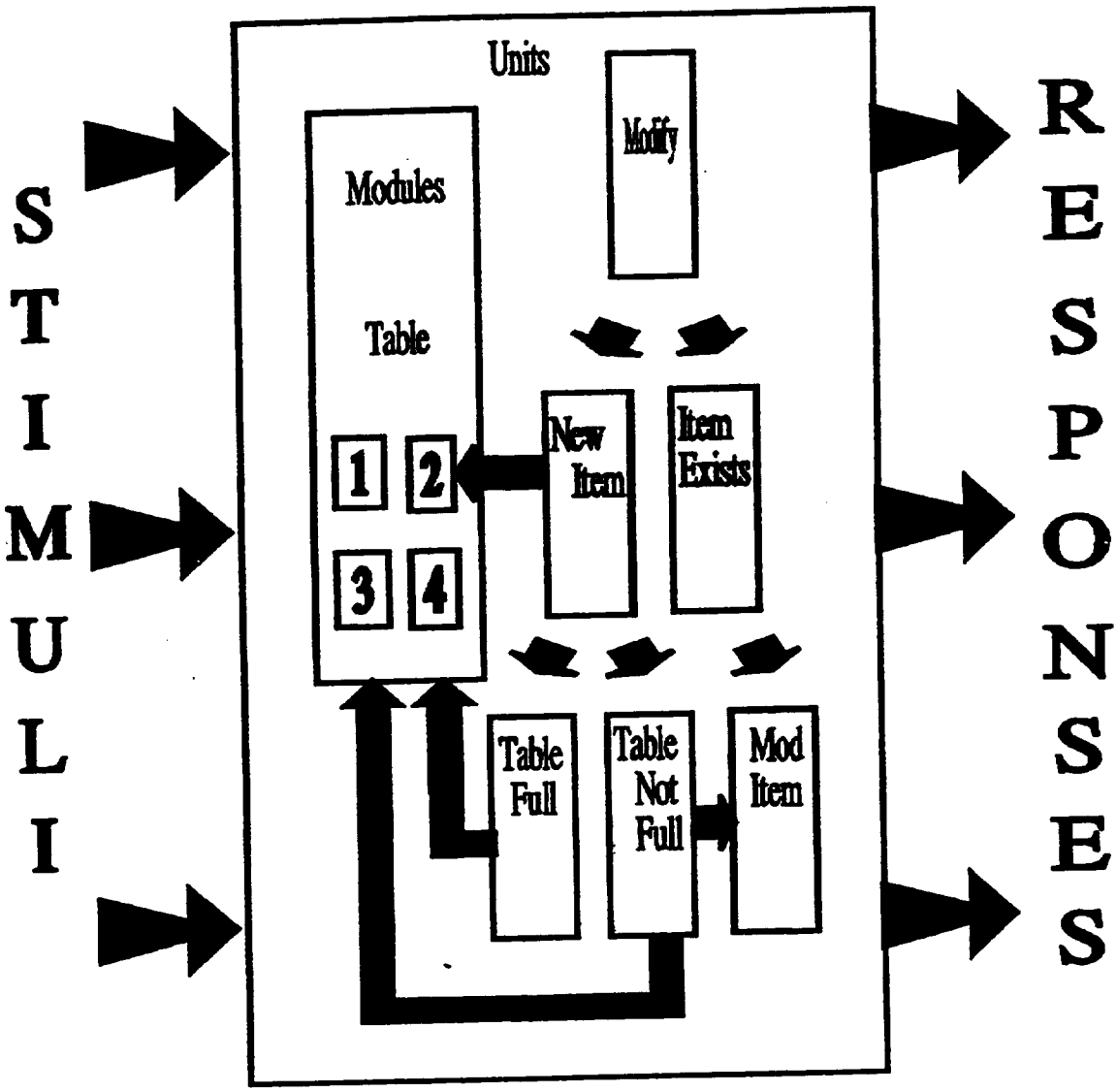
- Best performed when the system is complete
- Can be partially done early via prototyping
- Emphasis is on ensuring the requirements are correct

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

- **Requirements can be specified in terms of operational scenarios and expected system responses**
- **The system can be developed so that the classes of stimulus/response pairs correspond to self-contained units**
  - » **Stimuli tend to fall into *classes* or *groups***
  - » **These groups can be viewed as units[1].**
  - » **These, in turn, may have subunits based on stimulus/response pairs**
- **To overcome the impracticalities of system testing, these self-contained units can be tested separately**

---

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 are created to exercise the internals of the units.
- *Behavior:* Stimulus history can be described in terms of internal software "states" (e.g., sets of variable values) and expected transitions between states.
- *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").

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 (0—∞ modularity has many benefits)

- **Units can be developed and tested separately (if design uses proper encapsulation).**

- **The system can be incrementally integrated and tested until the full system is achieved (build a little, test a little).**

- **Separation of units greatly reduces the re-verification burden by reducing the effects of changes.**

Unit/Integration Testing ...

Implications ...

- **Design bridges the gap between the problem (requirements) and the tested solution.**
- **Individual units and subsystems can be more easily mapped to requirements.**
- **Once tested, the detailed behavior need not be re-tested during system testing.**

***However*, it is still impractical to test exhaustively and many types of errors can be more cheaply found by analyzing the design/code.**

- **This impracticality can be handled by static testing which we will discuss next.**

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, testcases, 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

- **Abstraction**
  - » Simplifying the description of a system by suppressing less important details
  - » More important actions are only considered
  - » Similar objects can be considered identical

Static Testing ...

Refinement

- » Is the incremental use of abstraction
- » Involves creating nested levels of description, each higher level refinement more abstract than lower ones

- Together, abstraction and refinement allow humans to find problems much better than computers can

Static Testing ...

Inputs

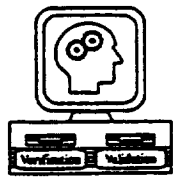
- **Description of the problem to be solved (can be very high level)**
- **Description of requirements (safety, user interface, etc.)**
- **Specifications of the item (e.g., design object) to be statically tested, possibly at different levels of abstraction.**

Static Testing ...

Implications

- **Because static testing can be done on anything at almost any time (does not have to wait for something executable), it can be done *hand-in-hand* with development; this decreases cost.**

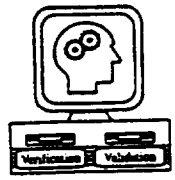
- **Static testing and design are natural precursors activities for unit / integration testing.**



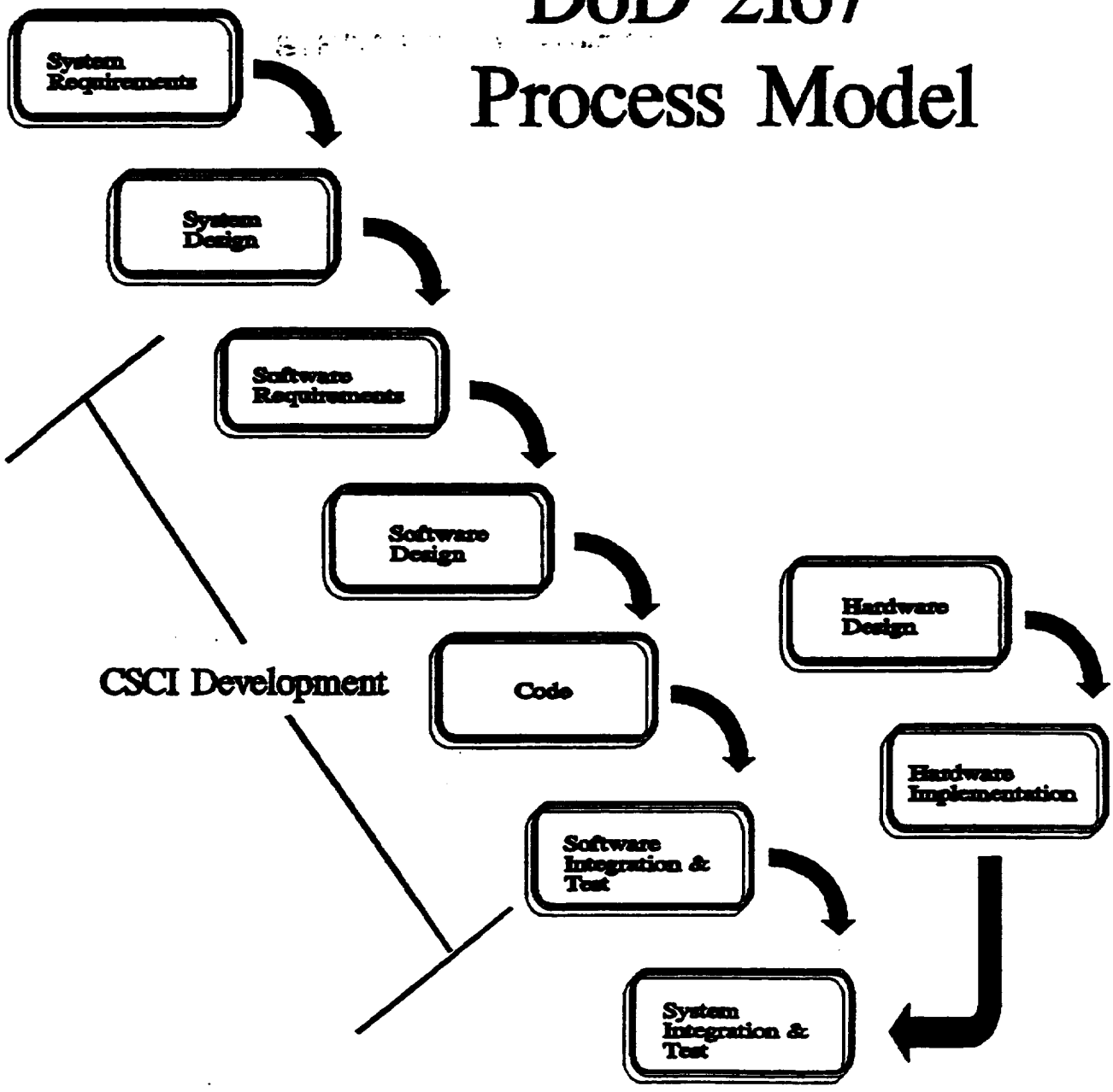
Life-Cycle Models

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

Example model : DoD 2167



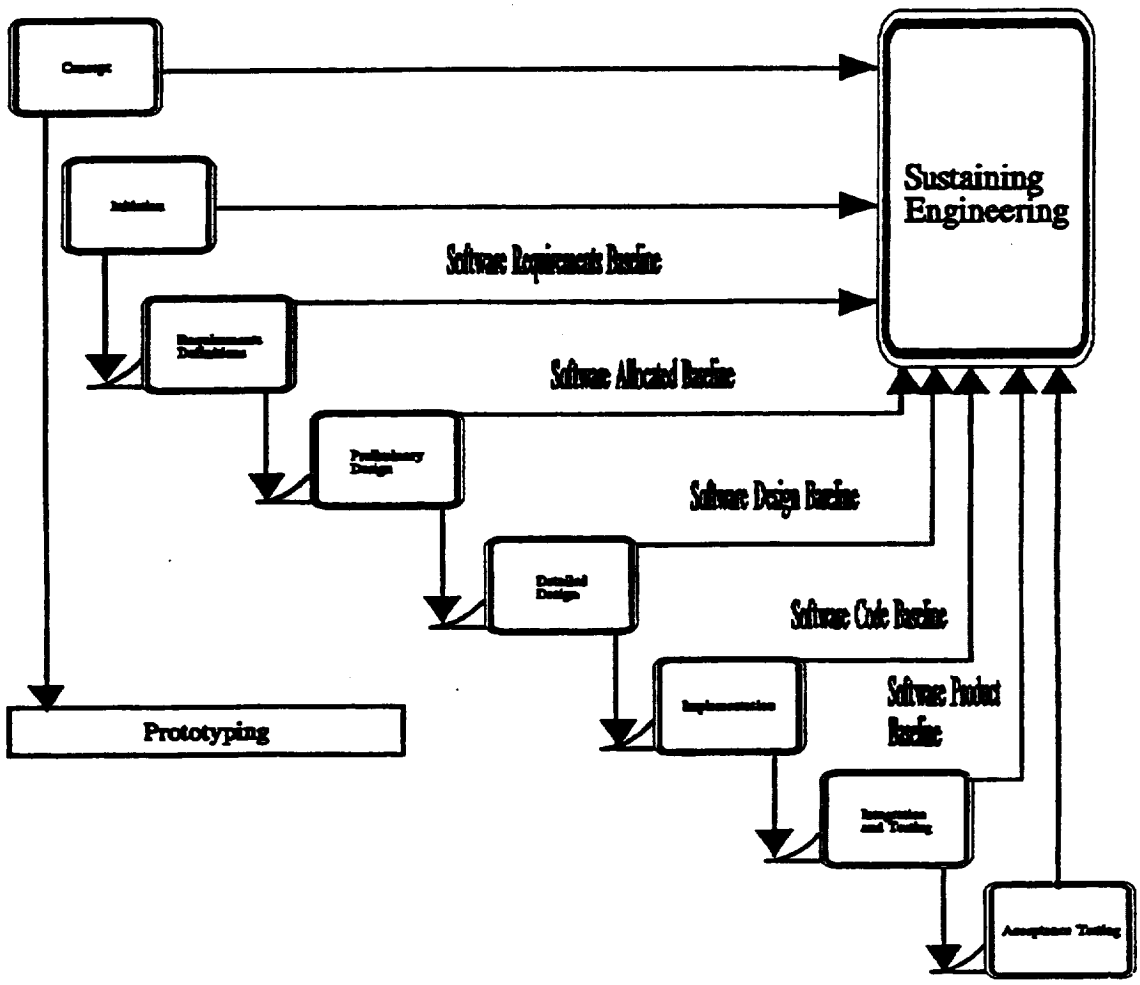
DoD 2167 Process Model



Life Cycle Models ...

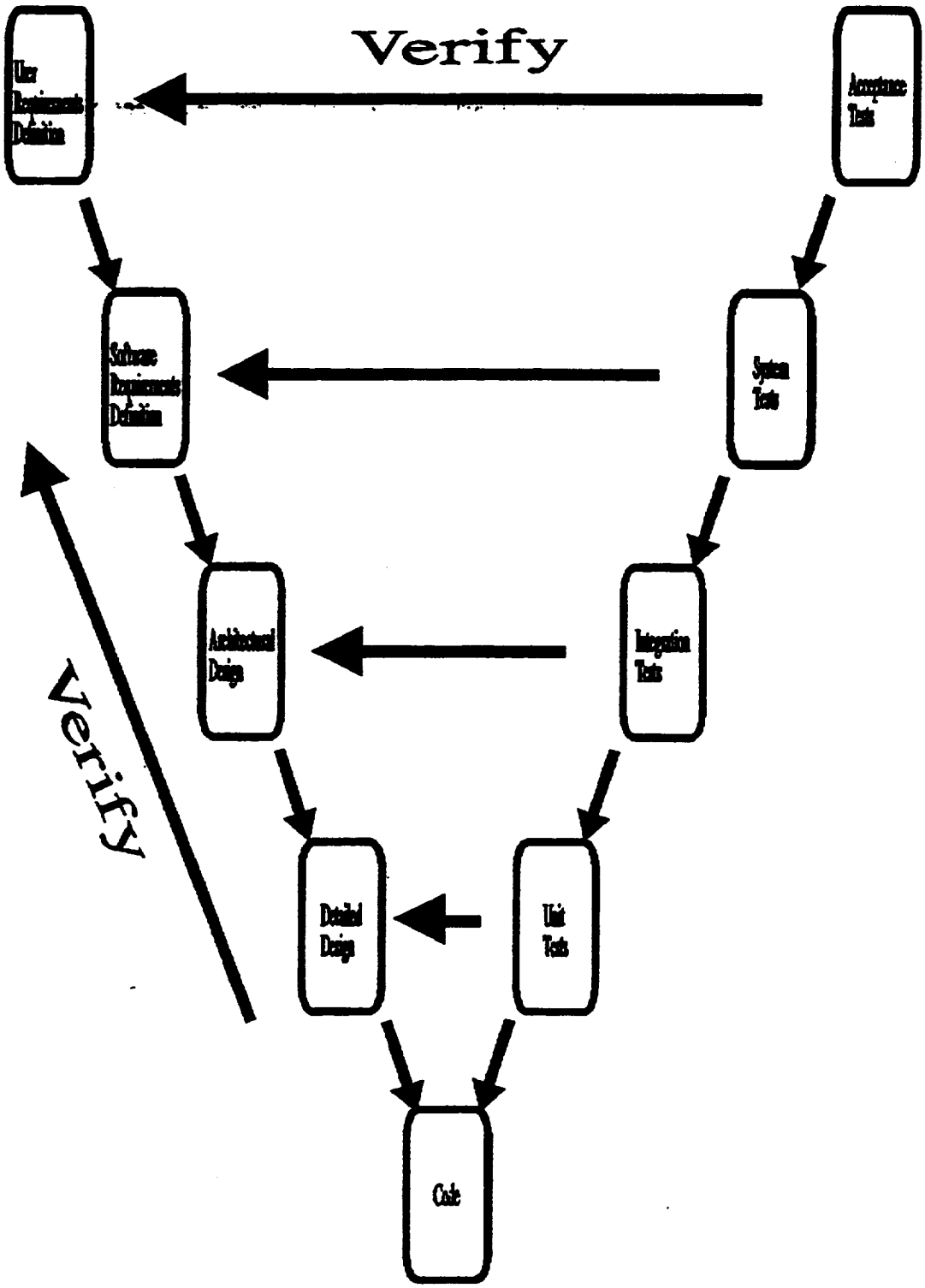
Example model: NASA Model

NASA Life-Cycle Model



Life Cycle Models ...

Example model: European Space Agency Model


Verify
Verify
Code

The Traffic Controller Problem

Problem Description

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.

Problem Description ...

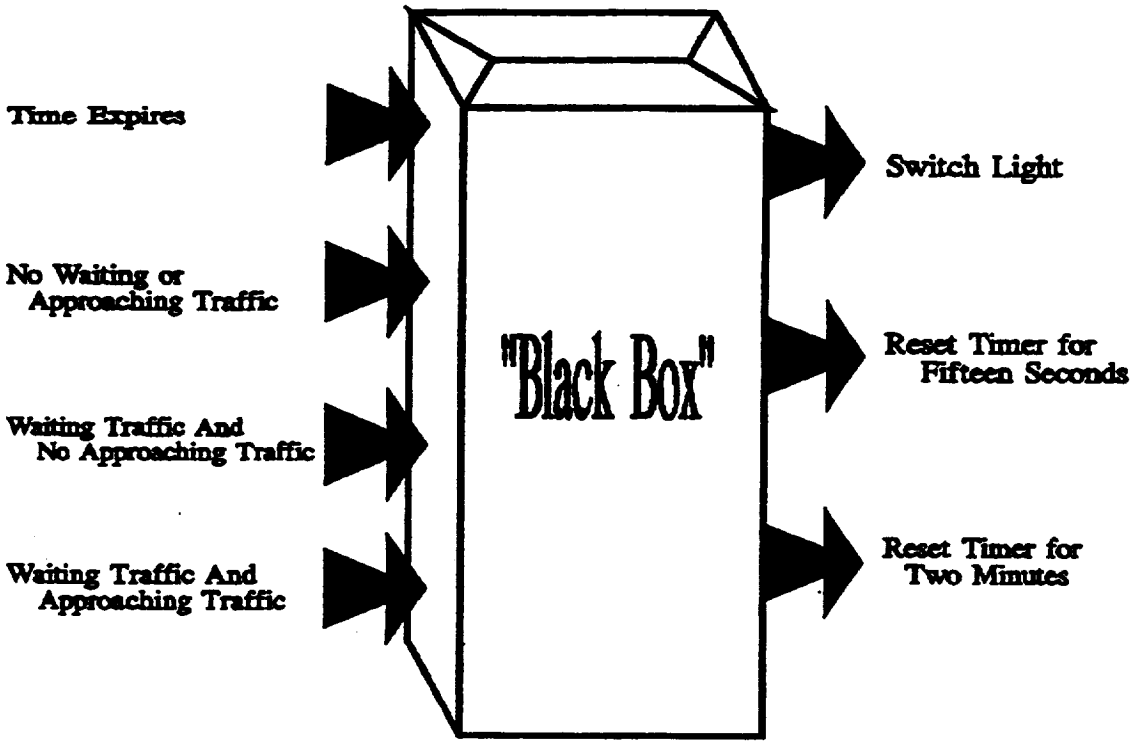
1. Take a few minutes and write down the key tasks the *traffic controller* is to do

2. Exchange your descriptions with a neighbor and then spend a few minutes deciding how well their description fits your understanding of the *traffic controller*

3. Ask yourself "is this a testable description of the system?"

Black Box View

Initial *black box* view of system testing



Refinement

Refine Requirements based on further understanding of the problem

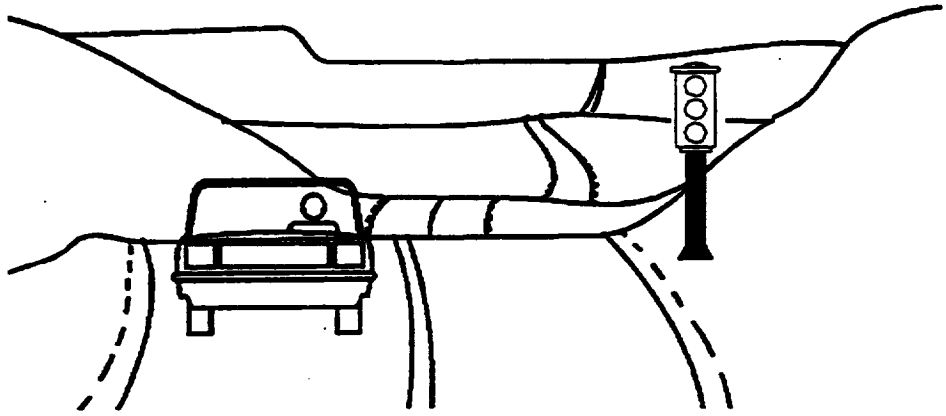
- State becomes evident
  - » What is the color of the light in a given direction?
  - » How long has the controller waited to switch the light?
- State helps identify and classify stimulus/response histories.
- The state remaining constant might imply testing one scenario verifies the other scenario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

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

- Operational scenarios can be constructed/selected.

Refinement ...

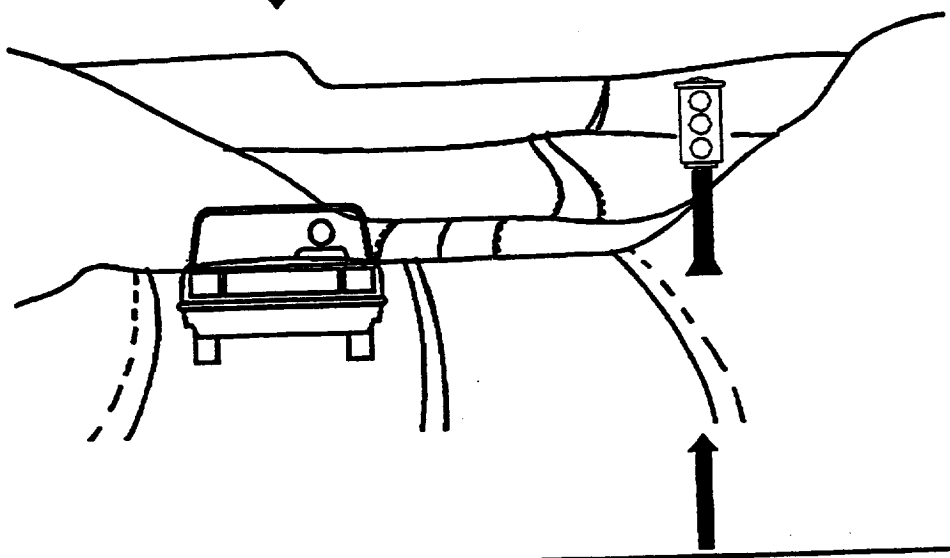
- Car Arrives from the West
- No North-South Traffic for 15 seconds following last signal change



- 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

General Techniques

General Techniques

Regression Testing

- Typically a maintenance activity
- Requires some process for capturing and retrieving test cases
- Assume that the traffic controller is to be changed so that a pedestrian or car may have to wait up to 1.5 minutes at a red light.
- All scenarios involving no pedestrians or cars waiting at a red light during a 2 minute interval should work as before.

General Techniques ...

Prototyping

- Develop a working model to test aspects of requirements or design

- E.g., prototyping of a red/green light system might reveal the need for a yellow light.

General Techniques ...

Competing Designs

- Define mutliple *design* teams
  - » *Design* can mean any particular representation of the system (e.g., requirements, code, etc.)
- Each team designs a solution
- Either select one that is best or merge the differing solutions into one common solution that is best

General Techniques ...

Independent V&V

- **Define a team that will perform V&V on the software**

- ***Must*** **be independent of the development team to avoid any potential bias in analysis of the product**

- **Usually applied at the System Testing level, but can be applied anywhere in the process**

System Testing Techniques

Functional Correctness

Focus:

- **Make sure that all identified scenarios work correctly.**
  - » **e.g., For each controller stimulus, is the correct response generated?**

Specific functional testing methods

- **Realistic Testing**
  - » **Focus on those functions used the most.**
  - » **Realistically, the majority of the time a request to change traffic flow is received from a car instead of a pedestrian.**
  - » **Therefore, select the appropriate majority of cases to exercise this scenario.**

Functional Correctness ...

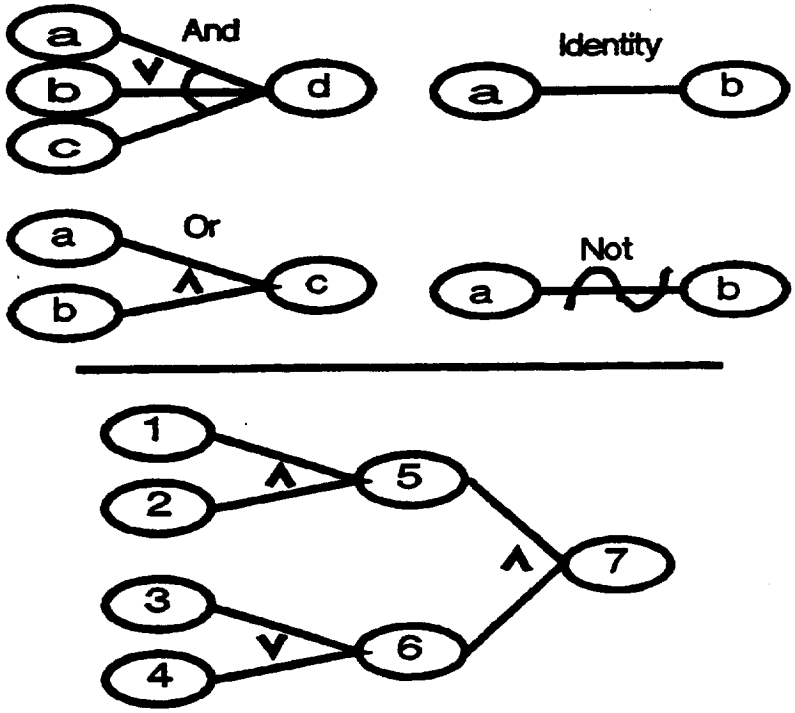
Specific functional testing methods ...

- Attribute-based Test Case Selection
  - » Choose test cases based on an attribute or characteristic such as
    - Complexity, Criticality, Reliability, ...
  - » Tests can be chosen according to
    - Statistical Record-keeping
    - Random
    - Error Guessing
- ***Example:*** Scenarios involving both a request to change traffic flow and an approaching traffic signal appear to be more complicated.

Functional Correctness ...

Specific functional testing methods ...

- **Cause-Effect Graphing**
  - » **Technique for selecting tests that exercise combinations of causes**
  - » **Highlights *interesting* cases**



Functional Correctness ...

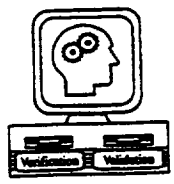
Specific functional testing methods ...

- **Boundary-Value Testing**

  - » Identifies cases at the boundaries of each stimulus/response class

  - » ***Example***, what happens when on-coming traffic is detected at the exact time a timer expires?

    - This exercises the *boundary* value of when the timer should expire so that the light will change



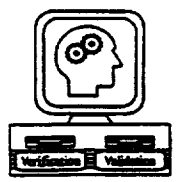
Safety Correctness

Focus:

- Verify that no stimulus generates an unsafe response

Specific Safety Correctness methods

- Stress Testing
  - » Choosing "*off-nominal*" tests that will determine if the system can operate safely in high stress and/or critical situations.
  - » Examples:
    - » What happens when if the pedestrian repeatedly hits the *change signal* button?
    - » What happens if a power surge occurs while a pedestrian repeatedly hits the *change signal* button?



User-Interface Correctness

Focus

- Demonstrate that the human to computer interface is correct

Specific User Interface Correctness methods

- Active Interface Testing

  » Choosing tests that will determine if the interface to an external agent (e.g., a person) works correctly

  » Examples:

  - How heavy does the car have to be to trip a signal to the controller?
  - Does a stuck pedestrian button prevent a signal to change traffic flow from being received?

Resource Consumption Correctness

Focus:

- Show environment resources are used correctly when a response is generated

Specific Resource Consumption Correctness methods

- Performance Testing
  - » Choosing tests that "push the envelope" (speed, accuracy, etc.)
  - » Examples:
    - How will a delay in receiving a request to change traffic flow affect changing the light?

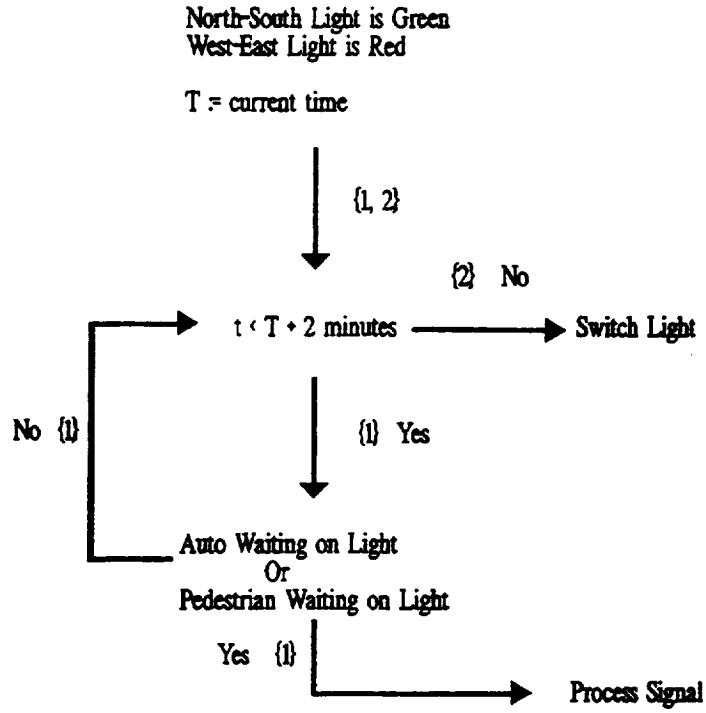
      e.g., What happens at time t_0 +14.999 when a pedestrian signal was received at time t_0 ?

Unit/Integration Testing Techniques

Functional Correctness

Branch Coverage

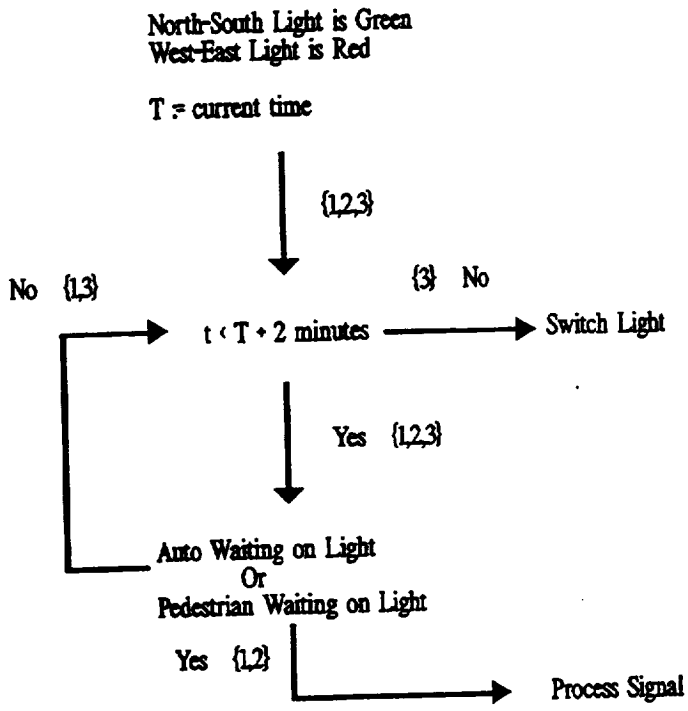
- **Choosing tests that will cover all possible outcomes of each internal logical decision (e.g., if-then-else)**



Functional Correctness ...

Path Coverage

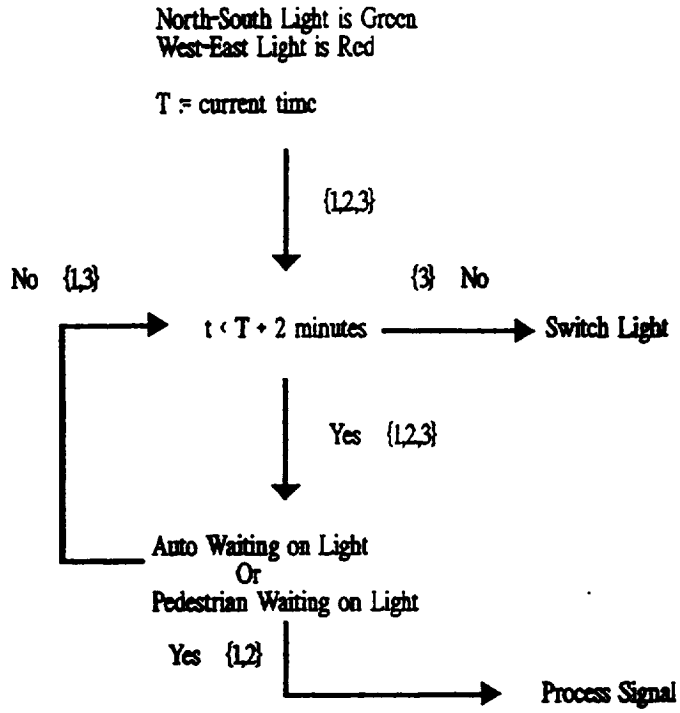
- **Choosing tests that will cover all possible combinations of outcomes of each internal logical decision**



Functional Correctness ...

Condition Coverage

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



Functional Correctness ...

Partition Analysis

- Branch, Path and Condition coverage focus on implementation
  - » This may not be sufficient
- Build input domains by analyzing the specification for a given implementation
- Blend this with the domains for Branch, Path and Condition coverage for more complete test suites

Functional Correctness ...

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

Functional Correctness ...

InterProcedural Dataflow Testing

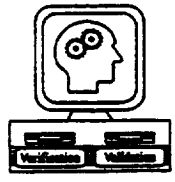
```
Procedure IsMax(I, J: In Integer; Max:
   Out Integer) Is
    Begin If I > J
            Then Max := I;
            Else  Max := J;
            End If;
End IsMax;
```

Definition/Use Table for IsMax		
Variable	Definition	Use
Max	3	6
	4	6

Functional Correctness ...

Flavor Analysis

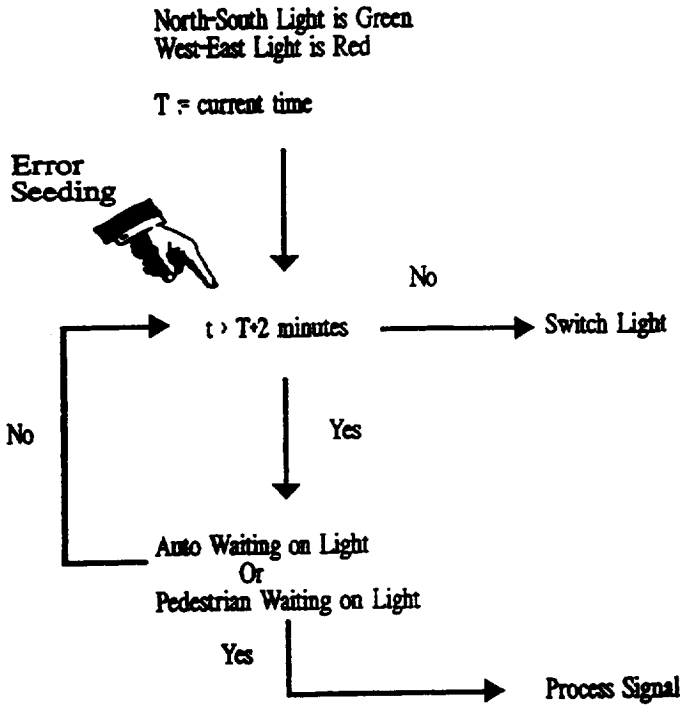
- Attempts to find errors of omission
- Documents:
  - » expected sequences of actions
  - » assertions about the effects of a piece of code
- Methods:
  - » Data Comments: documents abstractions used in program construction
  - » Operator Comments: documents a legal "ordering" of operators
- Goal: Compare actual execution against expectations



Functional Correctness ...

Mutation Testing

- **Changing the software to determine if the current set of test cases are good enough to detect the change.**
- **This technique evaluates the effectiveness of the current set of test cases.**



Safety Correctness

Reliability Testing

- **Identify structures that could adversely affect system reliability if they fail**
  - » **These structures do not necessarily have to be error-prone themselves**
- **For example, most functions rely on *clock*. All major system functions will fail if the clock fails.**

User Interface Correctness

Prototype Evaluation

- Test the user-interface pieces of the system early (before the other subsystems are finished)
- Involves either *stubbing out* some pieces of the system or developing a simulation
- For example: the interfaces to the light and signal hardware could be "stubbed out" and simulated so the traffic light software can be prototyped.

Static Testing

Functional Correctness

Inspections

- Formal/Informal (or walkthrough) inspections follow a set of rules to guide a reader, moderator, author, and several experts through inspecting a work product.
- Continuous inspections involve just an author and a peer. The peer frequently reviews the work of the author.
- The use of inspections is probably the biggest single advancement in the practice of verification.
  - » There is hard psychological evidence that introducing an "active verification frame of mind" significantly reduces errors

Functional Correctness ...

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

Functional Correctness ...

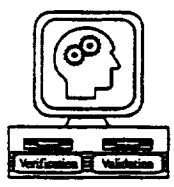
Object-Oriented Analysis

- 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.

Functional Correctness ...

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



Functional Correctness ...

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*

Functional Correctness ...

Stepwise Refinement

- A general technique of separating a unit into equivalent descriptions, each at increasing levels of detail.

- Analysis involves comparing each level of detail to the preceeding one, checking for consistency and completeness.

Functional Correctness ...

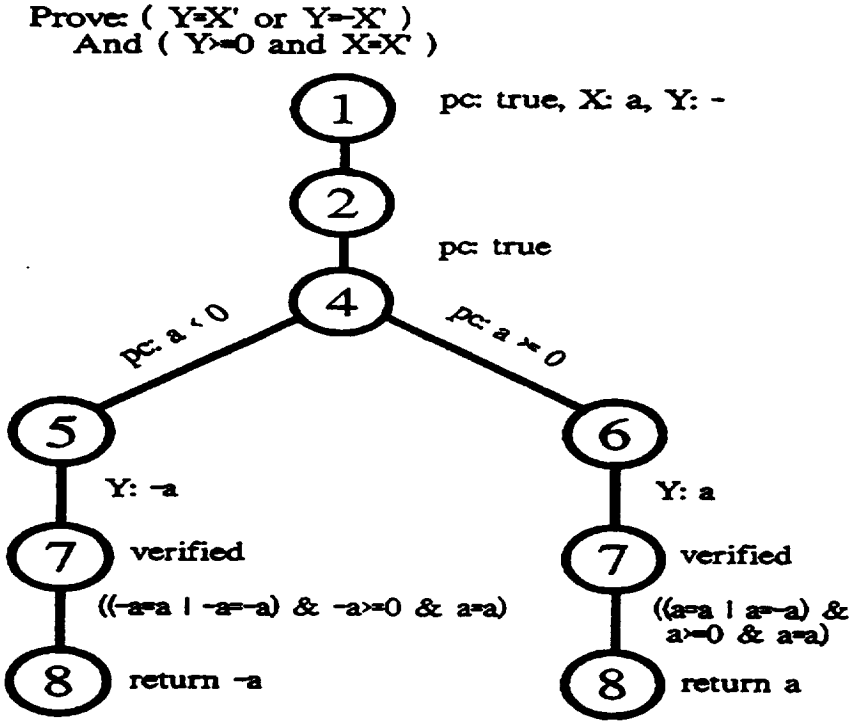
Axiomatic Analysis

- Involves specifying a "precondition" and "postcondition" for each fragment of code
- Each fragment is checked to see if the postcondition is guaranteed to be true after the fragment is executed, assuming the precondition was true before the fragment executed.
- Combined fragments are analyzed to see if preconditions are always satisfied and the end postcondition guarantees the desired result.

Functional Correctness ...

Symbolic Execution

- **Uses pre/post conditions to trace execution of the implementation**
- **Uses mathematical symbols to act as placeholders for real values (similar to classes)**



Safety Correctness

Hazard Analysis

- A hazard is a very undesirable situation (e.g., the light being green in both directions)

- Each hazard is analyzed to determine how it could arise (e.g., a hardware failure results in one light stuck to green)

- The system is analyzed to ensure that a hazardous state can never be reached (e.g., before red light is changed to green, the other light is checked to make sure it is not stuck green)

Safety Correctness ...

Fault Analysis

- A fault is a potential error in the system (e.g., failure of the module that controls a light_timer)

- Analysis is performed to determine the safety effects of potential faults (e.g., failure of light_time means that a light will remain the same color, say green)

Summary

Key Points

1. Verification vs. Validation
   - Verification: building the system right
   - Validation: building the right system

2. Static, Unit/Integration, and System Testing
   - Static: desk checking/code reviews
   - Unit/Integration: testing in pieces
   - System: Overall V&V

Key Points ...

3. Consistency vs. completeness
   - Completeness: Does all it should
   - Consistency: Does it correctly

4. Use of abstraction and refinement
   - Abstraction: Suppress details
   - Refinement: Incremental abstraction

5. Benefits of modularity
   - Divide and Conquer

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)

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 grouped into three types of testing
  1. Static Testing
  2. Unit/Integration Testing
  3. System Testing

Appendix A: References

References

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

2. 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.

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

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

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

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

References ...

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

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

9. 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.

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

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

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

References ...

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

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

15. 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.

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

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

References ...

17. 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.

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

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

21. 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.

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

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

References ...

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

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

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

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

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

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

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

References ...

31. 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.

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

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

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

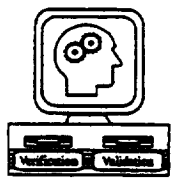
35. Wilson, W.M.. "NASA Life Cycle Model." *Aerospace Software Engineering: A Collection of Concepts.* Ed. Christine Anderson and Merlin Dorfman. Volume 136 pp. 319-336, American Institute of Aeronautics and Astronautics, Publisher. 1991.

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

Appendix B: Techniques Vs. Phases

Techniques Vs. Phases

Techniques	Phases			
	General	System	Unit	Static
Active Interface Testing		✓		
Anomaly Analysis				✓
Attribute-Based Test Case Selection		✓		
Axiomatic Analysis				✓
Boundary Testing		✓		
Branch Coverage			✓	
Cause-Effect Graphing		✓		
Competing Designs	✓			
Compilation Testing			✓	
Condition Coverage			✓	
Defect Analysis		✓		

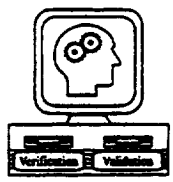


Techniques Vs. Phases ...

Techniques	Phases			
	General	System	Unit	Static
Error Guessing		✓		
Fault Analysis				✓
Flavor Analysis			✓	
Hazard Analysis				✓
Independent V&V	✓			
Inspections				✓
Interprocedural Dataflow Testing			✓	
Mutation Testing			✓	
Object Oriented Analysis				✓
Partition Testing			✓	
Path Coverage			✓	

Techniques Vs. Phases ...

Techniques	Phases			
	General	System	Unit	Static
Performance Testing		✓		
Pre/Post Condition Testing				✓
Prototyping	✓			
Random Testing		✓		
Realistic Testing		✓		
Regression Testing	✓			
Reliability Testing				✓
Stepwise Refinement				✓
Stress Testing		✓		
Symbolic Execution				✓



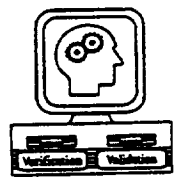
Appendix C: Techniques Vs. Correctness

Techniques Vs. Correctness

Techniques	Kinds of Correctness					
	General	Functional	Safety	UI	RCC	Utility
Active Interface Testing		✓				
Anomaly Analysis		✓				
Attribute-Based Test Case Selection		✓				
Axiomatic Analysis		✓				
Boundary Testing		✓				
Branch Coverage		✓				
Cause-Effect Graphing		✓				
Competing Designs	✓					
Compilation Testing		✓				
Condition Coverage		✓				
Defect Analysis		✓				

Techniques Vs. Correctness ...

Techniques	Kinds of Correctness					
	General	Functional	Safety	UI	RCC	Utility
Error Guessing		✓	✓			
Fault Analysis			✓			
Flavor Analysis	✓					
Hazard Analysis			✓			
Independent V&V	✓					
Inspections	✓					
Interprocedural Dataflow Testing			✓			
Mutation Testing			✓			
Object Oriented Analysis		✓				
Partition Testing		✓				
Path Coverage		✓				



Techniques Vs. Correctness ...

Techniques	Kinds of Correctness					
	General	Functional	Safety	UI	RCC	Utility
Performance Testing					✓	
Pre/Post Condition Testing		✓				
Prototyping	✓					
Random Testing		✓				
Realistic Testing		✓				
Regression Testing	✓					
Reliability Testing			✓			
Stepwise Refinement		✓				
Stress Testing		✓				
Symbolic Execution		✓				

Appendix D: Techniques Vs. References

Techniques Vs. References

Techniques	References
Active Interface Testing	*32*
Anomaly Analysis	*32,2*
Attribute-Based Test Case Selection	*32*
Axiomatic Analysis	*7,14*
Boundary Testing	*26*
Branch Coverage	*26*
Cause-Effect Graphing	*26*
Competing Designs	*16*
Compilation Testing	*32*
Condition Coverage	*26*
Defect Analysis	*32*
Error Guessing	*26*
Fault Analysis	*17,18*
Flavor Analysis	*13*
Hazard Analysis	*17,18*

Techniques Vs. References ...

Techniques	References
Inspections	*5,26*
InterProcedural Dataflow Testing	*9*
Mutation Testing	
Object Oriented Analysis	*36,22,15*
Partition Analysis	*31*
Path Coverage	*26*
Performance Testing	*32,32,2*
Pre/Post Condition Testing	*20*
Prototyping	
Random Testing	*32*
Realistic Testing	*32*
Regression Testing	*32*
Reliability Testing	*32*
Stepwise Refinement	*27-28,23-25*
Stress Testing	*1,26*
Symbolic Execution	*8, 11*

Workshop on Verification and Validation of Expert Systems Part 2: Expert Systems

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

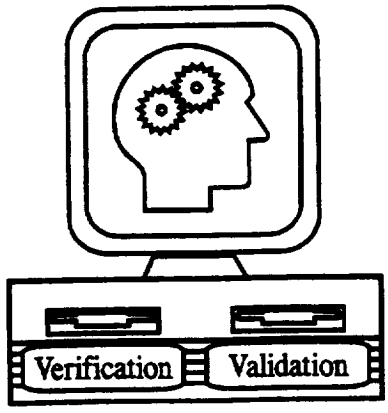


Table of Contents

Table of Contents ...

Introduction

Goals

1. To understand the differences and similarities between Expert Systems and conventional software.
2. To understand how the differences impact verification and validation.
3. To understand applicable analysis methods/techniques to overcome these impacts.

Overview

1. We will discuss how expert systems are software, but a unique type of software.
   - Different implementation languages
   - Different problem types
2. To illustrate language differences, V&V of two solutions to the Traffic Light Controller problem will be discussed.
3. To illustrate problem differences, a new version of the Traffic Controller Problem, that is more like a "true" expert system problem will be discussed.
4. Using the new problem, we will discuss both new expert system V&V techniques and modified "conventional" V&V techniques.

Expert System Differences

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**
    - **execution sequence is not apparent (but is implicit)**
  - » **language semantics unclear or complex (works by "magic")**
    - **e.g., conflict resolution**

Expert System Implementation Differences ...

Often developed iteratively

- **especially if design by knowledge aquisition**
- **especially if it is unclear whether the solution will work satisfactorally**

No explicit algorithm is used, e.g.,

```
t := t0
While t < t0+ 2 minutes Loop
    If Auto or Pedestrian
signal received
    Then Process-Signal
            Exit
    End If
End Loop
```

Expert System Problem Differences

Often solve problems requiring human expertise

- **solution already exists (in someones 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 aquisition" 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 judgement for correctness of solutions (i.e., are "fuzzy")

May replace or just augment human expert

Two Traffic Light Controller Implementations

Overview

TLC problem does not have expert system problem characteristics.

TLC problem could be implemented using either conventional or expert system implementation approach.

After further analysis of the problem, we will look at a conventional and expert system approach to the problem.

Though the problem "looks conventional", an expert system approach to implementation works very well.

Finally, we will look at V&V of each solution and compare the differences.

Scenario Testing

Test case scenarios can be developed by looking at possible combinations of events.

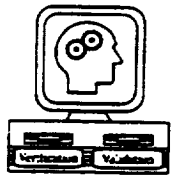
Some event definitions:

switch: *traffic light changes*

approaching: *controller detects an approaching auto or pedestrian in the direction of the current flow of traffic*

waiting: *controller detects an auto or pedestrian waiting for the light to change (in the direction opposite the current flow of traffic)*

(t, event): *ordered pair describing an elapsed time, t, and the event that occurs (should occur)after that elapsed time.*

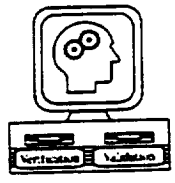


Scenario Testing ...

The following scenarios are generated from our understanding of the problem

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 < 1 minute 45 seconds, waiting) (15 seconds, switch)**
5. **...**

This list is NOT exhaustive; In fact, it is impossible to come up with an exhaustive list since the possibilities are infinite.



Testing State Changes

An alternative approach is to think of the problem as changes in the state of the traffic control system.

Then test that the system makes all the correct state changes.

For example, the system can be said to be in one of the following unique states.

- **S_1: In a 2 minute wait before changing the light**
- **S_2: In a 15 second wait before changing the light and has been in this state for less than 45 sec.**
- **S_3: In the remainder of a 1 minute wait before changing the light**

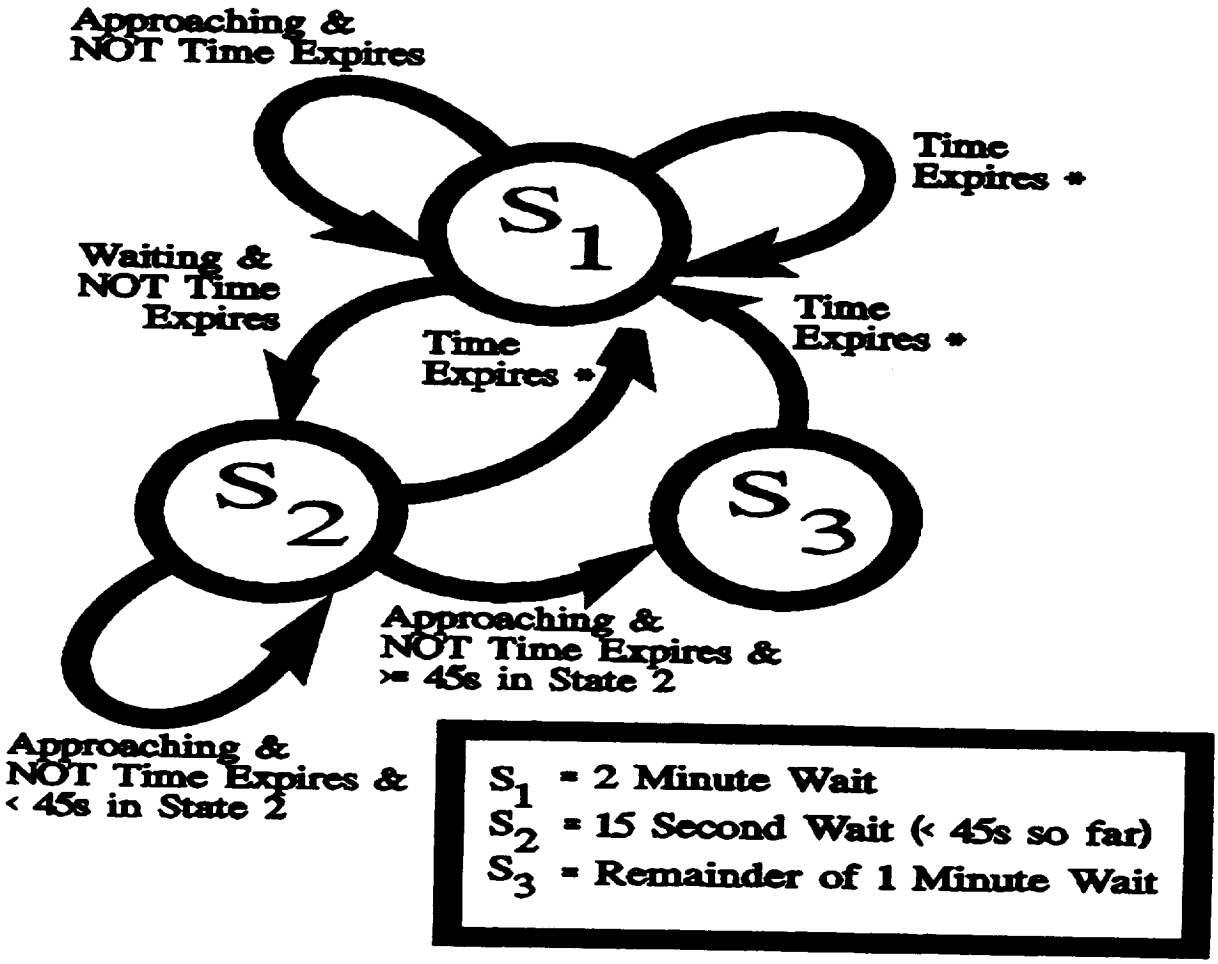
Testing State Changes ...

With the following event possibilities

- **Approaching**
- **Waiting**
- **Time expires**
- **< 45 sec. in S_2**
- **>= 45 sec. in S_2**

Testing State Changes ...

State Diagram



Testing State Changes ...

If

- State diagram is analyzed and shown to be correct
- Implementation can be shown to make all transitions correctly

Then

- It is reasonable to think the implementation is correct

Because there are only 7 transitions, compared to an infinite number of scenarios, testing state changes is easier.

Testing State Changes ...

This type of approach is sometimes called "conformance testing"; the implementation conforms to the abstract solution (the state diagram in our case).

<u>But</u> only transitions involving switching the light are actually visible.

We can either test all ways of getting to visible transformations or "look inside" the implementation.

Handouts (1,2,5) and Class Exercise

1. Divide into 3 teams.
   - Procedural Implemenation Team
   - "Short" CLIPS Implementation Team
   - "Longer" CLIPS Implementation Team
2. Study solution
3. Develop tests to "cover" all parts of the given solution.

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

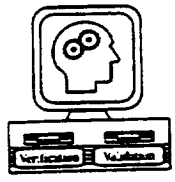
• • •

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 Ada "whens" as CLIPS 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 appoach has fewer and simpler internal interactions.

- **Execution order is very explicit**
  - » ***whens*** **"executed" exactly once per "cycle"**
  - » **priorities are used in CLIPS to control execution**
- ***Pure*** **functions (no side effects)**
  - » **Function "Change_Light" affects several rules**
- **No "garbage collection" concerns**
  - » **CLIPS implementation must retract old facts**

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

Comparison and V&V Implications ...

Both approaches have V&V issues.

Each has different V&V concerns.

Procedural concerns

- **More decisions to test (more code)**
- **Overall control structures to test (e.g., termination of the loop around the large case statement)**

Comparison and V&V Implications ...

Expert System concerns

- Must test correct cleanup of old facts - facts are retracted only if they are no longer needed
- Must test that there are no invalid rule interactions.
- Must test function side-effects.
- Must test that rule patterns are not too broad (i.e., act against too many instances)
- Must test that rules only fire at the right time (e.g., only at the end of a "cycle")

Comparison and V&V Implications ...

These different concerns create the need for different test approaches and different techniques for testing expert systems.

- **Procedural testing focuses on ensuring that the implementation solves all aspects of the problem.**
- **Expert system testing focuses on ensuring that the implementation does not have unexpected interactions.**
- **V&V of both must test that it solves the problem and "does not do funny things" but the emphasis is a little different in each case.**
- **Most of the rest of part 2 will cover new expert system testing techniques that address the new emphasis.**

Comparison and V&V Implications ...

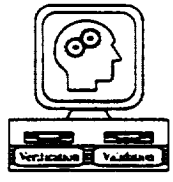
Different view of expert system V&V

- **Would expect expert system failures to be more like errors humans make.**
  - » **Expert system computation model based on how some psychologists believe humans think.**
  - » **Conventional software based on a much different computation model.**

Comparison and V&V Implications ...

Some sample errors humans make:

- **Slips/Lapses: usually caused by interruption in train of thought (overlapping rule sequences)**
- **New exceptions: applying knowledge that always worked in the past to a new situation which turns out to be an exception (rules with LHS too broad)**
- **Erroneous beliefs: (bad rules)**



Handout (3,4,6,7) and Exercise

1. Compare shorter and longer CLIPS versions.

2. Discuss differences in rule interactions

3. Look at handouts.

Testing Good and Bad Rule Based Designs

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

The shorter version:

- **has fewer rules**
- **has more complex rules**
- **is less modular**
- **has 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.

"Expert " Traffic Light Controller Problem

New 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).

New 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)

New Problem ...

Using this information, the system will decide when to allow a street (north/south, east/west) to either go straight, turn left, or wait on another street.

1. **Take a few minutes and write down the key tasks the *traffic controller* is to do**
2. **Exchange your descriptions with a neighbor**
3. **Spend a few minutes deciding how well their description fits your understanding of the *problem.***
   - **Is this a testable description of the system?**
   - **How is it different from the conventional problem?**
   - **Are there any new V&V issues, compared to the conventional problem?**

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

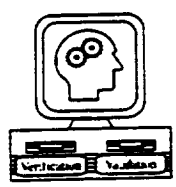
Initial Knowledge Acquisition ...

- **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 judgement?

Will it replace or augment a human expert?

Expert System Implementation V&V Techniques

Overview

This section will summarize some key techniques:

- **specific to expert systems**
- **specific to implementation (e.g., rules, frames)**

Each technique will be discussed in terms of

- **overall description**
- **implementation construct/aspect addressed**
- **error detection capability**
- **tools available**
- **example(s) based on TLC problem**

Overview ...

Will cover the most common/important techniques.

Tools and automation will be discussed, though few tools are commercially available.

More discussion on how to use these techniques will be covered in Part 3 - Guidelines.

Rule Consistency Checking[1,2]

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.

Rule Consistency Checking ...

Reachability anomalies (shorter CLIPS version)

- Dead-end rules
  - » Del_old_changes does not affect any other rule
  - » Is a dead-end rule because the fact to be modified should have been "signal_change" instead of "signal_changes"
- Unreachable rules (shorter CLIPS version)
  - » If signal_changed then ...
  - » Would be unreachable because "signal_changed" is not created by any other rule (should be "signal_change")

Rule Consistency Checking ...

Reachability anomalies ...

- **Cycle Rules**
  - » Update_time is in a cycle
  - » This "anomaly" does not indicate an error in this case
  - » Why?

C-4

Rule Consistency Checking ...

Redundant Rules (longer CLIPS 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
```

- There is an attempt to retract medium timer twice if light_changed

Rule Consistency Checking ...

Conflicting rules (longer CLIPS version)

- set_long_timer:

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

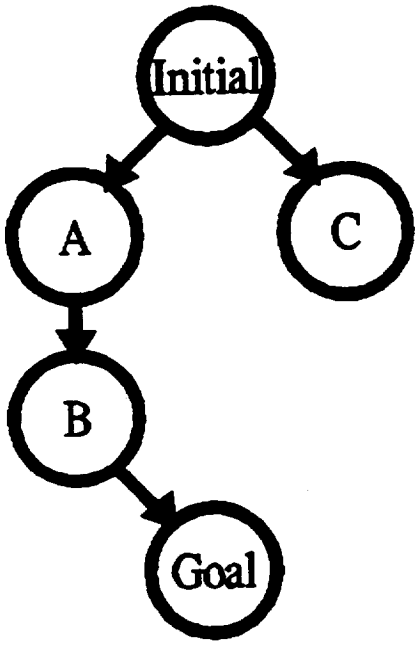
- retract_medium_timer:

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

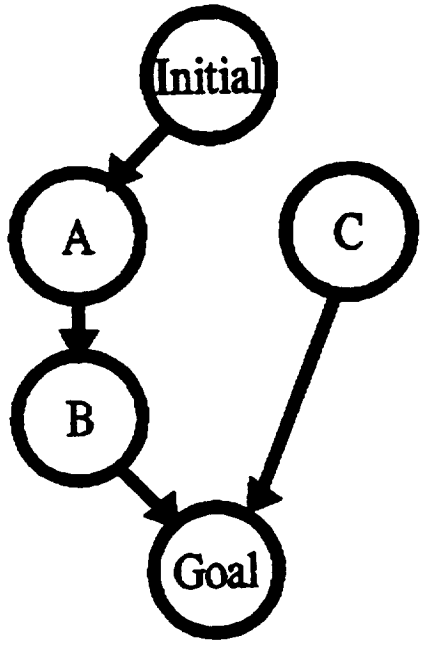
- there are two conflicting actions if light_changed (set and retract timer)

Rule Consistency Checking ...

Dead-End Rule (Rule C)



Unreachable Rule (Rule C)



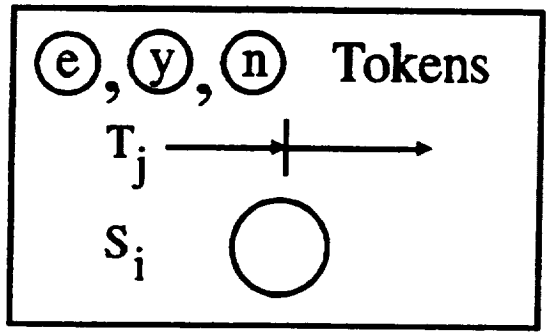
Cycle

Rule Consistency Checking ...

Graphing Techniques

- **Petri-Nets[2,15,17]**
  - » **Useful in describing dynamic behavior of discrete event systems (e.g., rule firings)**
  - » **Similar to other diagramming techniques (e.g., state diagrams, cause-effect diagrams, etc.)**

Basic Elements



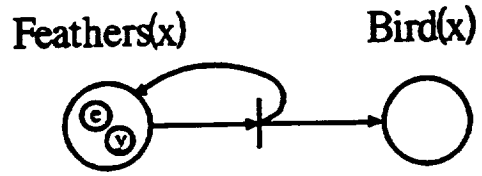
Rule Consistency Checking ...

Graphing Techniques ...

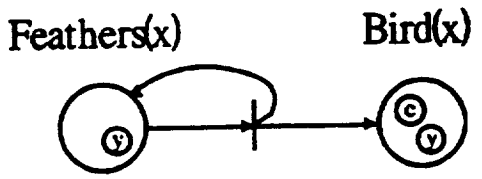
- **Petri Nets ...**
  - » **Builds a network of propositions (e.g., rule antecedents and consequents)**
  - » **"Tokens" are traced through the network**

Rule: If X has feathers
Then X is a bird

Initial:



Result:



Rule Consistency Checking ...

Graphing Techniques ...

- **Petri Nets ...**
  - » **Tracing highlights the kinds of consistency errors just discussed**
  - » **Can be helpful in finding completeness anomalies**
- **Directed Graphs (or Network Flows)[2]**
  - » **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**

Rule Consistency Checking ...

Graphing Techniques ...

- Connectivity Graphs[16]
  - » 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 #8)

Data Consistency Checking[3,4,5]

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., use of *deftemplates* in longer version of TLC can catch errors like the mispelling of "signal_change" in del_old_changes rule (in shorter version)**

Sensitivity Analysis[6]

Determining the sensitivity of one parameter (data item) to changes in other parameters

More a debugging than an error finding technique

Supported only by a research tool

Most directly applicable to classification problems

Sensitivity Analysis ...

E.g., Suppose, the goal was to output system state based on system variables

- **Given:**
  - » **S_1 = long_timer only**
  - » **S_2 = short timer and medium timer not within 15 sec of expiring**
  - » **S_3 = by short timer and medium timer withing 15 sec of expiring**
- **S_1 is least sensitive because it does not depend on the value of the medium timer**

Structural Testing[7]

Attempting to execute all parts of a knowledge base

Can be adapted to cover any type of KB construct

Does not detect any errors, just tries to ensure comprehensive testing

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

Structural Testing ...

E.g., generate test cases for longer CLIPS 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**

The creation of coverage tests can help one find errors.

Specification-Directed Analysis[8,9]

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

Specification-Directed Analysis

E.g.,

- in Timer module of the longer CLIPS version
- assertion is that timer names are unique
- by analyzing timer_name-conflict rule, it can be verified that the assertion is true (at the end of each cycle)

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

Specification-Directed Analysis ...

Some 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"

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'

Decision Tables[10]

Very popular in the early and mid '70s

Once thought of as a complete development methodology

Really is a specification approach

Very similar to rule-based progamming

- » 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

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.**

Decision Tables ...

Example: Complete TLC solution ($2^5*6=192$ rules)

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	3	0
...	...	...	...	...	...	...	...

Example: Abstract TLC solution ($2^4*3=48$ rules)

Approaching Vehicle	Waiting Vehicle	Time Expires	In state < 45 Sec	Current State	New State	Change Light
0	0	0	0	1	1	0
1	0	0	0	1	1	0
0	1	0	0	1	2	0
...	...	...	...	...	...	...

Decision Tables ...

Number of rules can get very large

Is 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 actually do?**

Expert System Problem V&V Techniques

Overview

Techniques in this section will be problem oriented

- **will treat solution as a black box**

We will not care how the solution is "coded".

- **could be rules, frames, procedural, mixture**

- **could be nonmonotonic, case-based, or just a big decision table**

Overview ...

We will be concerned with

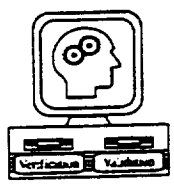
1. Is the system based on correct knowledge ?

2. Does the system adequately solve the problem ?

3. Does the system satisfy all correctness objectives such as

   » safety

   » user interface



Knowledge Aquisition Correctness Checking[11]

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

Similar to analyzing system requirements.

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

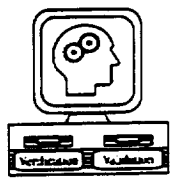
Example: How does the expert traffic controller know when to stop and go back to conventional mode?

Minimum Competency Testing[12]

Certifying the competency of an expert system by giving it the same test as would be given to a human expert

Certification exams exist for many types of human experts.

- **CPA**
- **MD**
- **PE**



Minimum Competency Testing ...

The approach may assume the expert system will make errors as an expert would (i.e., not expected to get 100% correct on the exam)

Expert can be asked to identify what he would expect a novice, advanced beginner, etc. to be able to do.

Similar to statistical testing (exam is a representative sample)

Discussion: What things would be in a certification test for a new human traffic controller.

Disaster Testing

Involves identifying scenarios that indicate potential disaster (during knowledge aquisition) and guarding against them.

- **experts are often good at recognizing potential disasters**

- **for most experts, many disaster situations are "common sense" (this knowledge must be drawn out)**

Disaster Testing ...

Tests can specifically be generated to check that the system recognizes potential disasters and prevents them from occuring.

- **Can also be used in conjuction with specification-directed verification (disaster forms the specification/constraint).**

- **Example TLC disaster: two intersecting directions not allowed to go at the same time**

Expert Review[13]

The expert is the expert.

Some answers can only be judged correct by the expert.

Expert can check:

- **test scenarios**
- **test results**

Expert may not understand implementation details

Expert Review ...

With minimal training, an expert can also check

- **Acquired knowledge**
  - » **Did you hear what he thought he said ?**
  - » **Are there any "holes" or deficiencies in what he told you ?**
- **Knowledge base design**
  - » **Is overall problem solving approach correct ?**
  - » **Was any aquired knowledge misinterpreted ?**

The key to expert review is formatting the review material so the expert can easily understand it.

Explicit Modelling[14]

Different kinds of models:

- **set of equations**
- **a small scale replica (e.g., toy airplane model)**
- **a 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;"[14]

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[18]

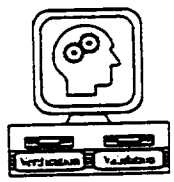
Explicit Modelling ...

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



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.

Appendix A: References

References

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

2. 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)

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

   Section 2 is the description of the capabilities of CRSV

4. 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 ...

5. 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.

6. 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

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

8. "Designing a Solution for the Traffic Light Problem Using Terms, Operators, and Productions" - This is the first case study in the Case Studies section.

References ...

9. 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)

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

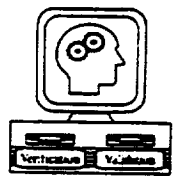
11. Marcus, S., "SALT, A Knowledge Acquisition Tool That Checks and Helps Test a Knowledge Base", *1988 AAAI Workshop on Verification, Validation, and Testing of Knowledge-Based Systems*

12. "Quality Measures and Assurance for AI Software", This is the last reference in the references section of this workshop

    pp.74-79 includes a discussion of minimum competency testing

13. 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



References ...

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

15. 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.

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

17. 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.

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

Appendix B: Techniques Vs. References

Techniques vs. References

Techniques	References
Rule Consistency Checking	1,2
Petri-Nets	2,15,17
Network Flows	2
Connectivity Graphs	16
Data Consistency Checking	3,4,5
Sensitivity Analysis	6
Structural Testing	7
Specification-Directed Analysis	8,9
Decision Tables	10
Knowledge-Acquisition Correctness Checking	11
Minimum Competency Testing	12
Expert Review	13
Explicit Modeling	14

Workshop on Verification and Validation of Expert Systems Part 3: 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

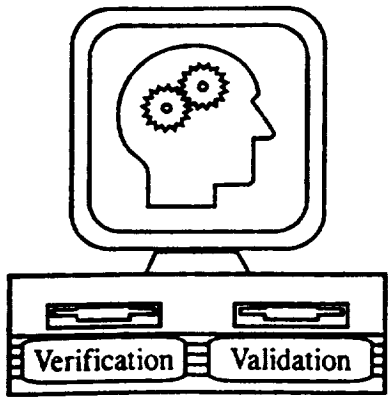


Table of contents

Introduction

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

Overview

1. Discuss some common misceptions about software (including expert systems)

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

3. Discuss a set of V&V guidelines

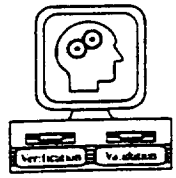
4. Discuss tailoring of guidelines

Common Misperceptions

Software in general:

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

- **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 saved[1]**



Software in general ...

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

- **"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"2**

- **"Building large programs is NOT like building small ones and software engineering is different from most other engineering disciplines."3**

Software in general ...

Methodical examination of software is too costly.

- **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"[3]**

Software can be proved correct

- **One can prove certain properties about software (e.g., the algorithm never results in deadlock)**

- **One can not prove *all* aspects of correctness.**

Expert Systems/AI in particular:

Expert Systems are Magic.

Expert Systems are quick and easy to build

- **"AI entails massive software engineering."[4]**

- **"Software engineering is harder than you think: I can not emphasize strongely enough how true this statement is."[4]**

Expert Systems/AI in particular ...

All "expert systems" are expert systems

- **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**

Expert Systems are all "Expert" Systems.

- **Most Expert Systems have a significant amount of conventional code/function (survey results indicate at least 45% of the developed system is conventional[5]).**

Expert Systems/AI in particular ...

The hueristic nature of Expert Systems make them inherently unreliable.

- **They are still predictable.**

- **They should be as effective as the hueristic**

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

Expert Systems/AI in particular ...

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

- **Knowledge representation (i.e., language) is key to expert systems and V&V of them**

- **Knowledge aquistition, reasoning paradigms, and software engineering are also needed skills**

  - » **Domain engineer: knowledge centered**

  - » **System engineer: computer centered**

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)**

Overview ...

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**

Note: 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

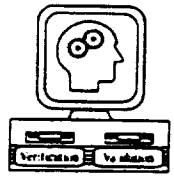
- **Manual analysis may be very difficult (i.e., inspections)**

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 expert5**
- **"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.

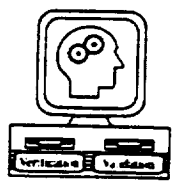
1. **Functional**

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

Discussion

As a class discussion exercise, develop a V&V plan and approach for the Traffic Light Controller problem.

1. **Discuss initial planning issues**

2. **Discuss additional requirements that are needed**

3. **Discuss additional testing that is needed**

Exercise

For your case study problem (5-10 min. for each step):

1. **Analyze the problem**

2. **Do initial planning**

3. **Pick a general implementation approach.**

4. **Develop a very high level design (including specifications) and trace back to problem statement (may be only for a piece of the total problem)**

5. **Generate an argument for the completeness and correctness of the very high level design.**

Exercise ...

Then, trade problems with another group and for your new problem, continue to:

6. **Generate a realistic set of test scenarios**

7. **Describe additional types of testing that are needed**

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.

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.

References ...

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.

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.

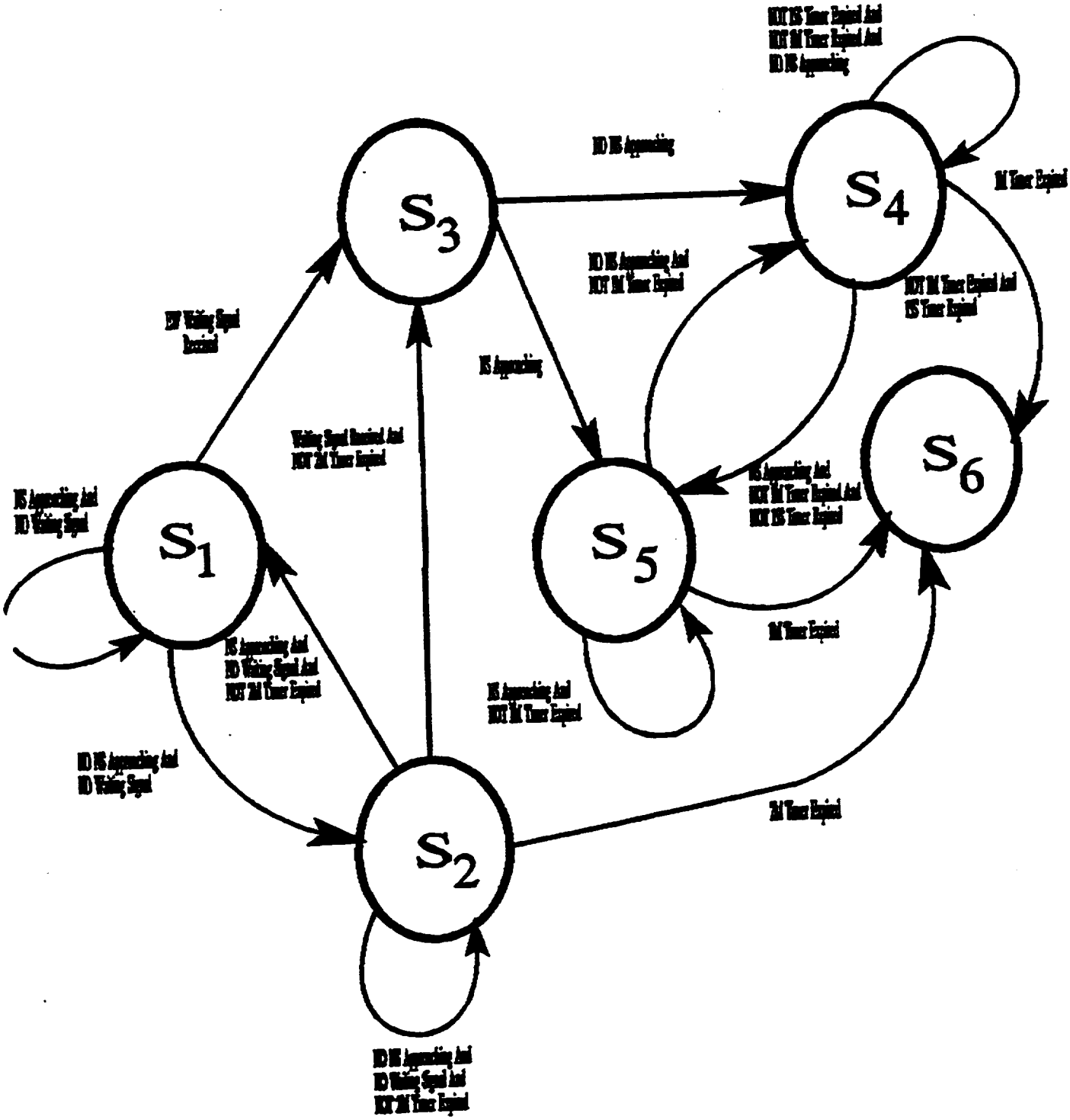
11. Culbert, Chris. "Knowledge-Based Systems Verification and Validation." *The Verification and Validation of Expert Systems Workshop.* Austin, TX, June 18, 1991.

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.

Workshop Handout #1



Workshop Handout #1

```
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.
   --*>
   2_Minute_Timer, 1_Minute_Timer, 15_Second_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);
```

Workshop Handout #1

State : Current State of the Traffic Controller

Possible states the Traffic Controller can be in are:

State_1 : 2M_Timer := Clock+2 Minutes
NS-Light := Green

State_2 : 2M_Timer Is Unchanged
NS-Light := Green
Clock Updated by 1 second

State_3 : 1M_Timer := Clock+1 Minute
15S_Timer := Clock+15 Seconds
NS-Light := Green

State_4 : NS-Light := Green
1M_Timer Unchanged
15S_Timer Unchanged
Clock Updated by 1 second

State_5 : NS-Light := Green
1M_Timer Unchanged
15S_Timer := Clock+15 Seconds

State_6 : NS-Light := Red
2M_Timer := Clock+2 Minutes

```
State := State_1;
Loop
  Case State Is
    When in state_1 => perform state_1 transitions
    When in state_2 => perform  state_2 transitions

                . . .

    When in state_n => perform state_n transitions
  End Case;

  Update Clock;
End Loop;

End Traffic_Controller;
```

Workshop Handout #1

```
        --<*
        --  State_1 Transitions
        --
        --  Truth Table:
        --
        --      Waiting_Traffic   Approaching_Traffic  Satisfied By:
        --           T                 T                1.3
        --           T                 F                1.3
        --           F                 T                1.1
        --           F                 F                1.2
        --
        --*>
<* 1.1 *> When State_1 And (Approaching_Traffic And
                  NOT Waiting_Traffic)      =>
          State := State_1;
<* 1.2 *> When State_1 And (NOT Approaching_Traffic And
                  NOT Waiting_Traffic And   =>
          State := State_2;
<* 1.3 *> When State_1 And (Waiting_Traffic)        =>
          State := State_3;
        --<*
        --  End State_1 Transitions
        --*>
```

Workshop Handout #1

```
--<*
-- State_2 Transitions
--
-- Assumptions : Once waiting traffic is detected detection of
--               oncoming traffic is irrelevant
--
-- Truth Table:
--
--      Waiting_Traffic   Approaching_Traffic   Expired
-- 2.3        T                 T                T
-- 2.3        T                 F                T
-- 2.3        F                 T                T
-- 2.3        F                 F                T
-- 2.4        T                 T                F
-- 2.4        T                 F                F
-- 2.2        F                 T                F
-- 2.1        F                 F                F
--
-- What happens when oncoming traffic is detected at the exact
--   same time the timer expires?
--*>
<* 2.1 *> When State_2 And (NOT Approaching_Traffic And
                NOT Waiting_Traffic And
                NOT Expired(2M_Timer))    =>
        State := State_2;
<* 2.2 *> When State_2 And (NOT Waiting_Traffic And
                NOT Expired(2M_Timer) And
                Approaching_Traffic))     =>
        State := State_1;
<* 2.3 *> When State_2 And (Expired(2M_Timer))       =>
        State := State_6;
<* 2.4 *> When State_2 And (Waiting_Traffic And
                NOT Expired(2M_Timer))    =>
        State := State_3;
        --<*
        -- End State_2 Transitions
        --*>
```

```
        --<*
        --  State_3 Transitions
        --
        --  Assumptions : Detecting additional waiting traffic does
        --                not effect state transition
        --
        --  Truth Table:
        --
        --      Approaching_Traffic
        --  3.1         T
        --  3.2         F
        --*>
<* 3.1 *> When State_3 And (Approaching_Traffic)      =>
            State := State_5;
<* 3.2 *> When State_3 And (NOT Approaching_Traffic)  =>
            State := State_4;
        --<*
        --  End State_3 Transitions
        --*>
```

Workshop Handout #1

```
--<*
-- State_4 Transitions
--
-- Assumptions : Once waiting traffic is detected detection of
--               oncoming traffic is irrelevant
--
-- Truth Table:
--
--       Approaching_Traffic  15S_Timer Expired  1M_Timer Expired
-- 4.4         T                    T                  T
-- 4.4         T                    F                  T
-- 4.4         F                    T                  T
-- 4.4         F                    F                  T
-- 4.2         T                    T                  F
-- 4.3         T                    F                  F
-- 4.2         F                    T                  F
-- 4.1         F                    F                  F
--
-- What happens when the oncoming traffic is detected at the
--   exact same time that the timer expires?
--*>
<* 4.1 *> When State_4 And (NOT Expired(15S_Timer) And
                NOT Expired(1M_Timer) And
                NOT Approaching_Traffic))  =>
        State := State_4;
<* 4.2 *> When State_4 And (NOT Expired(1M_Timer) And
                Expired(15S_Timer))        =>
        State := State_6;
<* 4.3 *> When State_4 And (Approaching_Traffic And
                NOT Expired(1M_Timer) And
                NOT Expired(15S_Timer))    =>
        State := State_5;
<* 4.4 *> When State_4 And (Expired(1M_Timer))        =>
        State := State_6;
        --<*
        -- End State_4 Transitions
        --*>
```

```
        --<*
        --  State_5 Transitions
        --
        --  Assumptions : Physically impossible for the 15S_Timer to
        --                expire at the same time it is set
        --
        --  Truth Table:
        --
        --       Approaching_Traffic  1M_Timer Expired
        --  5.1        T                   T
        --  5.2        T                   F
        --  5.1        F                   T
        --  5.3        F                   F
        --*>
<* 5.1 *> When State_5 And (Expired(1M_Timer))          =>
            State := State_6;
<* 5.2 *> When State_5 And (Approaching_Traffic And
                    NOT Expired(1M_Timer))      =>
            State := State_5;
<* 5.3 *> When State_5 And (NOT Approaching_Traffic And
                    NOT Expired(1M_Timer))      =>
            State := State_4;
        --<*
        --  End State_5 Transitions
        --*>
```

Workshop Handout #2

```
(deffacts initial-facts

  (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)

)

(defrule update-time (declare (salience -1))
  ?f1 <- (time ?t)
=>
  (retract ?f1)
  (assert (time =(+ ?t 1)))
)

(defrule trigger-signal-change
   (green ?direction ?)
   (time ?t)
   (signal ?other_direction ? ?t) (test (neq ?direction ?other_direction))
=>
   (assert (signal-change ?t))
)

(defrule del-old-changes
  ?f1 <- (signal-changes ?dt)
  (time ?t)
  (test (> (- ?t ?dt) 120))
=>
  (retract ?f1)
)
```

Workshop Handout #2

```
(defrule trigger-signal-delay
  (green ?direction ?)
  (time ?t)
  (signal ?direction ? ?t)
=>
  (assert (signal-delay ?t))
)

(defrule del-old-delays
  ?f1 <- (signal-delay ?dt)
  (time ?t)
  (test (> (- ?t ?dt) 15))
=>
  (retract ?f1)
)

(defrule change-no-signal
  ?f1 <- (green ?direction ?last_changed)
  (time ?t)
  (test (>= ?t (+ ?last_changed 120)))
  (not (signal-delay ?))
  (not (signal-change ?))
=>
  (retract ?f1)
  (if (eq ?direction NS) then (bind ?other_direction EW)
                         else (bind ?other_direction NS))
  (assert (green ?other_direction ?t))
  (fprintout t "green " ?other_direction " (no signal) at " ?t crlf)
)
```

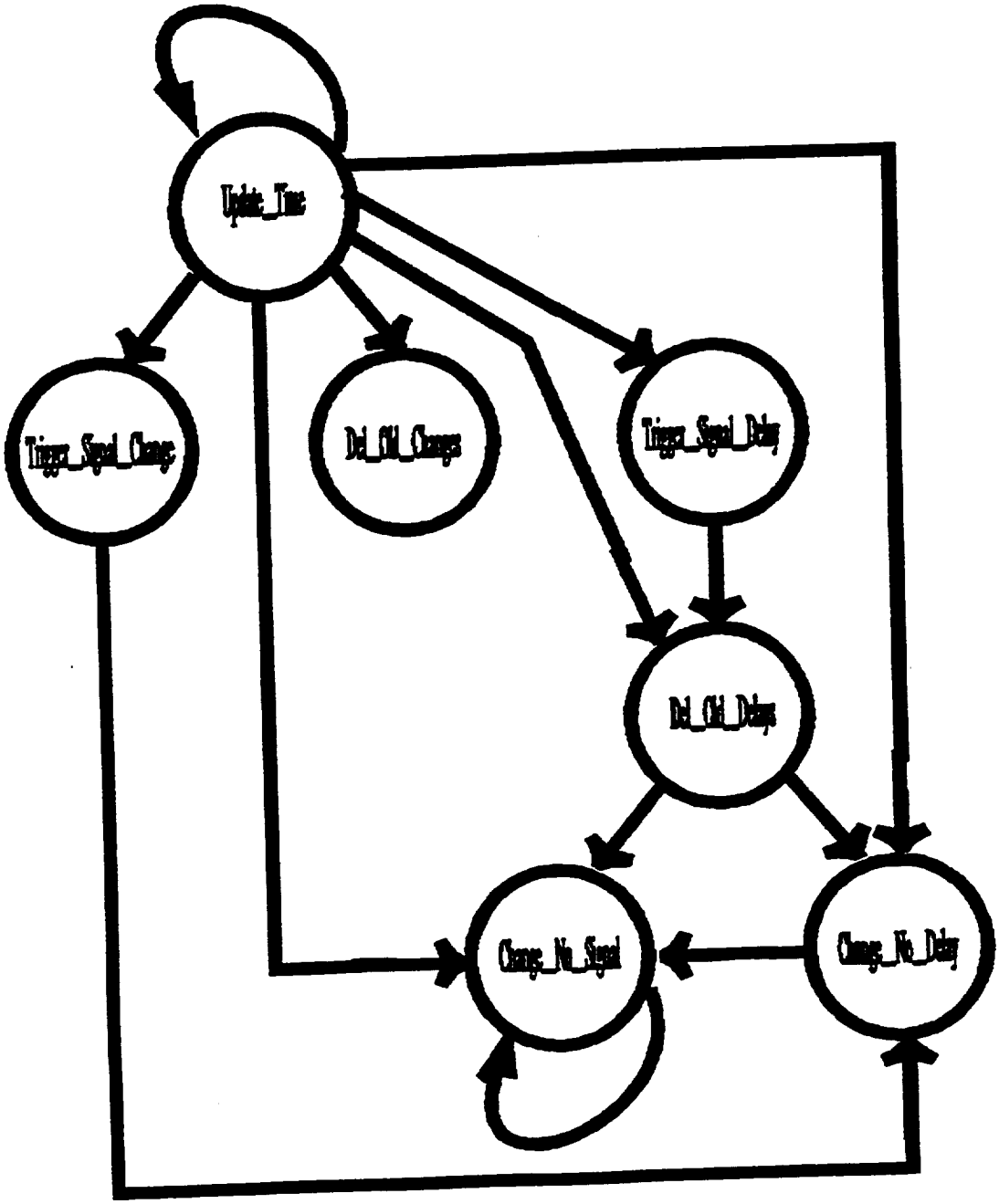
Workshop Handout #2

```
(defrule change-no-delay
  ?f1 <- (green ?direction ?last_changed)
  (time ?t)
  ?f2 <- (signal-change ?sg)
  (not (signal-delay ?))
  (test( >= ?t (+ ?sg 15)))
=>
  (retract ?f1 ?f2)
  (if (eq ?direction NS) then (bind ?other_direction EW)
                  else (bind ?other_direction NS))
  (assert (green ?other_direction ?t))
  (fprintout t "green " ?other_direction " (no delay) at " ?t crlf)
)

(defrule change-delay
  ?f1 <- (green ?direction ?last_changed)
  (time ?t)
  ?f2 <- (signal-change ?sg)
  ?f3 <- (signal-delay ?sd)
  (test (>= ?t (+ ?sg 60)))
=>
  (retract ?f1 ?f2 ?f3)
  (if (eq ?direction NS) then (bind ?other_direction EW)
                  else (bind ?other_direction NS))
  (assert (green ?other_direction ?t))
  (fprintout t "green " ?other_direction " (delay) at " ?t crlf)
)

(defrule stopit
  (time ?t)
  (end ?t2)
  (test (>= ?t ?t2))
=>
  (halt)
)
```

Workshop Handout #3

update_time	time=t	time=t+1
trigger_signal_change	signal.time=time signal.direction=green_direction	signal_change
del_old_changes	time> signal_changes + 120	^signal_changes
trigger_signal_delay	signal=green_direction	signal_delay=?t time=?t
del_old_delays	time>signal_delay+15	^signal_delay
change_no_signal	^signal_delay	last_changed=time ^signal_change time>=last_changed+120
change_no_delay	^signal_delay	last_changed=time time> signal_change+15
update_time	trigger_signal_change	time=t+1 del_old_changes trigger_signal_delay del_old_delays change_no_signal change_no_delay update_time
trigger_signal_change	change_no_delay	
del_old_changes	____	
trigger_signal_delay	del_old_delays	
del_old_delays	change_no_signal change_no_delay	
change_no_signal	change_no_signal	
change_no_delay	change_no_signal	

Workshop Handout #4



» **Can you detect any errors in the structure shown above?**

Workshop Handout #5

```
;=====================================================================
;=                                             =
;= Simulated Solution to Traffic Light Controller Problem      =
;=                                             =
;=====================================================================

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Problem Solving Method
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 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
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Workshop Handout #5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; TIME module
;;;
;;; Update time count at end of each cycle
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; State Data

    (deftemplate time
      (field is (type NUMBER))
    )

  (deffacts time_facts

    (time (is 0))
    (stop-time 600)
  )

; Transitions

  ;< update time at the end of each cycle >
  ;<* time := time + 1 *>
   (defrule count_time (declare (salience -2))
     ?f1 <- (time (is ?t))
   =>
     (modify ?f1 (is =(+ ?t 1)))
   )

  ;< halt when stop time reached >
   (defrule stopit
     (stop-time ?t)
     (time (is ?t))
   =>
     (halt)
   )
```

Workshop Handout #5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 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

    (deftemplate timer
      (field called (type ?VARIABLE))
      (field set_for (type NUMBER))
      (field has_expired (allowed-words TRUE FALSE) (default FALSE))
       ; private
        (field expires_at (type NUMBER) (default 99999))
    )

  ; Constraint: set_for > 0

    (defrule timer_error
      (timer (called ?name) (set_for ?sf))
      (test (<= ?sf 0))
    =>
      (fprintout t "TIMER_ERROR: " ?name crlf)
    )
```

Workshop Handout #5

```
; Constraint: only one timer of a given name
;           This is resolved be deleting oldest timer.

  (defrule timer_name-conflict
   ?f1 <- (timer (called ?name) (expires_at ?ea-1))
          (timer (called ?name) (expires_at ?ea-2))
          (test (< ?ea-1 ?ea-2))
  =>        (retract ?f1)
   )

 ; Initial: expires_at := time + set_for

; Transitions

 ;< initialize expires_at >
  (defrule initialize_expires_at
   ?f1 <- ( timer (expires_at 99999) (set_for ?sf) )
   (time (is ?t))
  =>
   (modify ?f1 (expires_at =(+ ?sf ?t)))
  )

 ;< indicate timer has expired >
  (defrule timer_expired
   ?f1 <- ( timer (expires_at ?ea) (has_expired FALSE) )
   (time (is ?t))
   (test (<= ?ea ?t))
  =>
   (modify ?f1 (has_expired TRUE))
  )
```

Workshop Handout #5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Signal Controller
;;;
;;; Simulate car and pedestrian arrival sensors.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; State Data

  (deffacts signal_facts

    (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)

 )

; Model: Signal_data is a list of signal_names and times;
;       the time indicates when the signal will be simulated

  (deftemplate signal
    (field in_direction (allowed-words NS EW))
    (field signalled_by (allowed-words car pedestrian))
  )

; Constraint: none

; Initial: none
```

Workshop Handout #5

```
; Transitions

  ;< assert signal >
   (defrule assert_signal
    (signal_data ?direction ?type ?time)
    (time (is ?time))
   =>
    (assert (signal (in_direction ?direction)
                    (signalled_by ?type)
   ) )     )

  ;< retract signal at end of cycle >
   (defrule retract_signal (declare (salience -1))
    ?f1 <- ( signal (in_direction ?direction) (signalled_by ?type))
   =>
    (retract ?f1)
   )
```

Workshop Handout #5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Traffic Light State
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; State Data

  ; Initial:

    (defglobal
      ?*green-light* = NS
      ?*red-light* = EW
    )

; Transitions

  (deffunction change-light ()
   (assert (light-changed))
   (if (eq ?*green-light* NS) then
     (bind ?*green-light* EW)
     (bind ?*red-light*   NS)
   else
     (bind ?*green-light* NS)
     (bind ?*red-light*   EW)
  ) )

 ;< reset light-changed fact at end of cycle >
 (defrule retract-light-changed (declare (salience -1))
  ?f1 <- (light-changed)
 =>
  (retract ?f1)
 )
```

Workshop Handout #5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Traffic Light Controller
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;;
    ;;; 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.
    ;;;
    ;;;
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Workshop Handout #5

```
; Constants

   (defglobal
     ?*long-time*   = 120
     ?*medium-time* = 60
     ?*short-time*  = 15
   )

; Initial

(deffacts traffic_light_controller-facts

  (timer (called long) (set_for ?*long-time*))

)

; Transitions

 ;< light-changed or approaching traffic -> set long timer >
 (defrule set-long-timer
   (or (light-changed)
       (signal (in_direction ?direction&:(eq ?direction ?*green-light*)))
   )
 =>
   (assert (timer (called long) (set_for ?*long-time*)))
 )

 ;< signal to change the light -> set medium and short timers >
(defrule set-medium-timer
  (signal (in_direction ?direction&:(eq ?direction ?*red-light*)))
=>
  (assert (timer (called short)  (set_for ?*short-time*)))
  (assert (timer (called medium) (set_for ?*medium-time*)))
)
```

Workshop Handout #5

```
;< approaching traffic detected and medium timer exists
;    -> reset short timer >
(defrule reset-short-timer
 (signal (in_direction ?direction&:(eq ?direction ?*green-light*)))
 (timer (called medium))
=>
 (assert (timer (called short)  (set_for ?*short-time*)))
)

;< timer expires -> change light >
(defrule timer_expires
 (timer (has_expired TRUE))
 (time (is ?t))
=>
 (change-light)
 (fprintout t "change light at " ?t  " " ?*green-light* crlf)
)

;< light changed -> retract medium and short timers >
(defrule retract-medium-timer
 (light-changed)
 ?f1 <- (timer (called medium))
 ?f2 <- (timer (called short))
=>
 (retract ?f1 ?f2)
)
```

Workshop Handout #6

count_time stop_it	time=t time=stop_time	time=t=1 (halt)
–		
timer_expired	timer.expires_at=time	timer.has_expired=TRUE
–		
assert_signal	signal_data.at_time=time	signal=?direction signal._data=?direction
retract_signal	signal	^signal
–		
retract_light_changed	light_changed	^light_changed
–		
set_long_timer	ight_changed or signal=green_light	long_timer.expires_at =time+120
set_mediumm_timer	signal=red_light	short_timer.expires_at =time+15
	mediumm_timer.expres_at	=timer+60
reset_short_timer	signal=green_light and	short_timer.expires_at
retract_mediumm_timer	light_changed	^mediumm_timer, ^short_timer
timer_expires	timer.has_expired=TRUE stop_it	(change_light) light_changedcount_time timer_expired assert_signal

Workshop Handout #6

stop_it

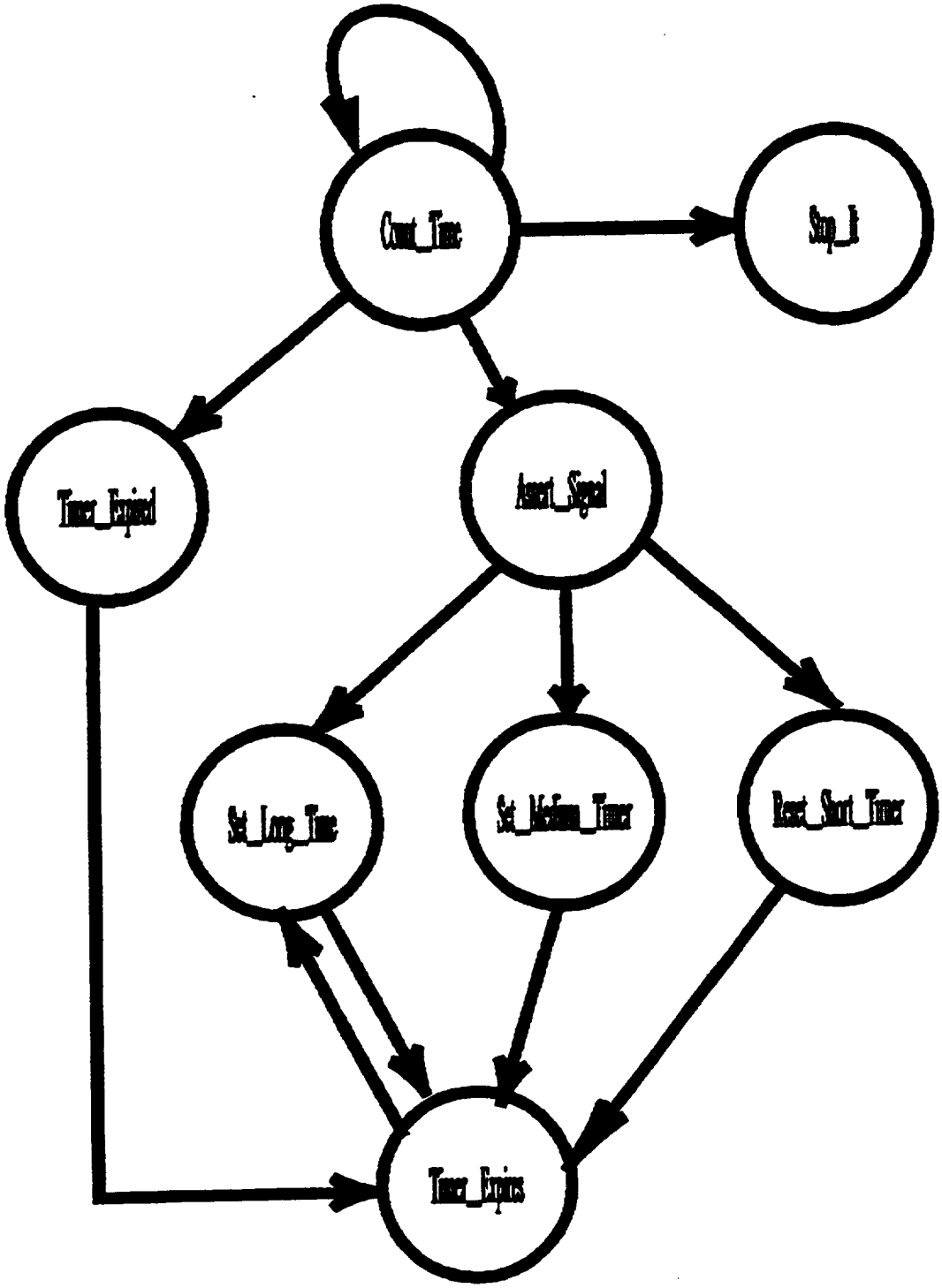
timer_expired

timer_expires

Workshop Handout #6

assert_signal	set_long_timer set_medium_timer reset_short_timer
retract_signal	———
retract_light_changed	———
—	
set_long_timer	timer_expires
set_medium_timer	timer_expires
reset_short_timer	timer_expires
retract_medium_timer	———
timer_expires	retract_light_changed set_long_timer medium_timer

Workshop Handout #7



Workshop Handout #8

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 number 16 in Part 2 of the Presentation Material) for more complete descriptions of this approach. Nazareth (reference number 2 in Part 2 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 on pages 2 and 3 show these lists from the first CLIPS implementation of the traffic controller problem.

Tables 3 and 4 on pages 4 and 5 show the lists of rules and facts from the second CLIPS 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 Traffic Controller CLIPS 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 Traffic Controller CLIPS 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 Traffic Controller 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 Traffic Controller 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 on pages 7 and 8 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 ad-hoc CLIPS implementation (see Table 7 on page 9). 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 CLIPS implementation? The matrix should show that this implementation is easier to analyze. In fact, the matrix of Table 8 on page 10 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 CLIPS 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 CLIPS 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 Ad-Hoc CLIPS 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 Ad-Hoc CLIPS 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 CLIPS 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 on pages 15 and 16 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 ad-hoc CLIPS implementation. Tables 11 and 12 on pages 17 and 18 show the analagous matrices for the modular CLIPS 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 on pages 19 and 25). The second equation matches rules against other rules (see Tables 14 and 20 on pages 20 and 26).

- $(Rd^{TR}) * (Wr)$

- $(Wr) * (Rd^{TR})$

where **(Rd)** is the initial Rule\Fact read matrix and (Rd^{TR}) is the transpose of that matrix

Identifying Anamolies

Tables 13 and 19 on pages 19 and 25 show the fact to fact connectivity relationships for the ad-hoc CLIPS and modular CLIPS 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 on pages 20 and 26. 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 #2. 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 on pages 21 through 24 show the results of applying this equation to the ad-hoc CLIPS implementation. Tables 15, 16 and 17 show successive impiementations while Table 19 shows the cumulative results of applying this equation to A^9. Tables 21 through 24 on pages 26 and 30 show the results as applied to the modular CLIPS 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 Ad-Hoc CLIPS 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 Ad-Hoc CLIPS 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 CLIPS 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 CLIPS 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 Ad-Hoc CLIPS 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 Ad-Hoc CLIPS 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 CLIPS 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 CLIPS 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 21: 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 ($A+A^2+ \ldots +A^5$)

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 that this restriction is incorrect because it would prevent the monkey from climbing the ladder with the key (to unlock the 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.

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

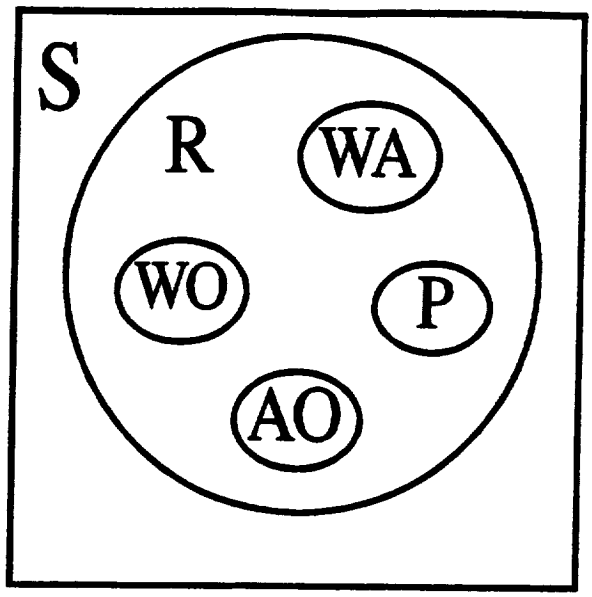
units. These are the easiest concepts to identify. Further understanding of the problem reveals refinements to these high level concepts, such as *Open_Sensor*, *Received_Signal*, etc.. Each of these refinements serve to clarify the primary idea captured by the knowledge unit and therefore belong in 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.

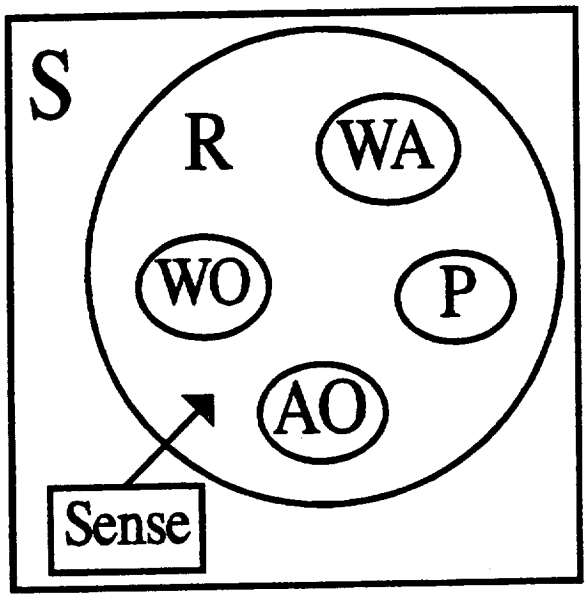
Verification and Validation Approach:

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

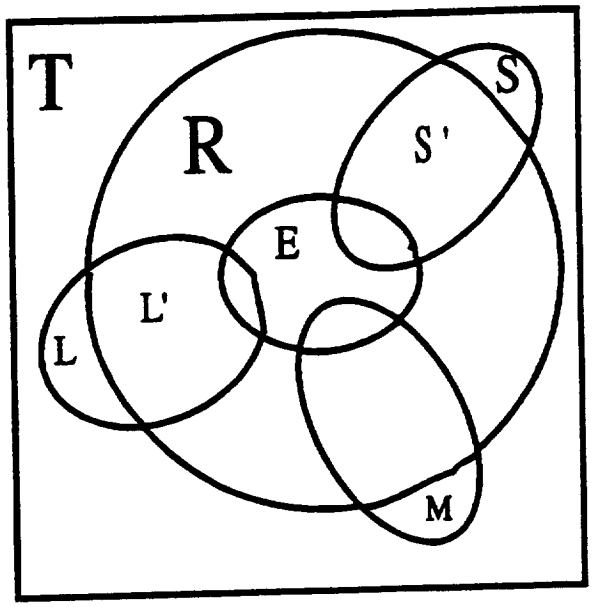
behavior by abstracting the specifics of what the system does to a general approach. For example, using the Traffic Light Problem definition, an abstracted problem solving approach might be as follows.

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}

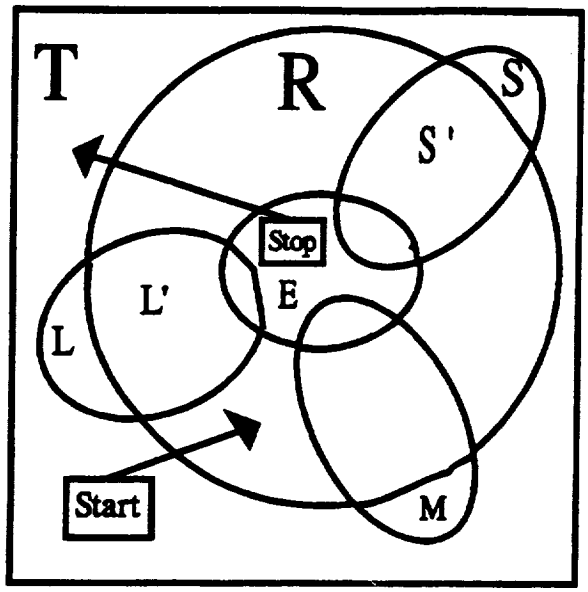
S' = {Set of all unexpired short timers}

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 I 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 SEGMENT**s that may interact with other **KB SEGMENT**s 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. **WHENEVER**s 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 forms 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 normal 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** form 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 **WHENEVER**s. The language allows **WHENEVER**s 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. When Statement Initialize implements it's Intended Function:

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. When For Statement implements it's Intended Function:

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 Wait_time:
Type: Integer
end

Parameter Flow_direction:
Type:
(EASTWEST,NORTHSOUTH)
end

Parameter Traffic_waiting:
Type: Boolean
end

Parameter Time:
Type: Integer
end

Parameter State:
Type: (UPDATE_WAIT_TIME,
SWITCH_TRAFFIC,NULL)
end

Frame Type Flow_sensor:
Direction: Type:
(EASTWEST,NORTHSOUTH);
Traffic_detected: Boolean;
end

Frame Eastwest_lane:
Direction: EASTWEST
end

Frame Northsouth_lane:
Direction: NORTHSOUTH
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
```

Copies of this publication have been deposited with the Texas State Library in compliance with the State Depository Law.