Unit 4
Software Architecture

Data abstraction and object-oriented organization

- In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object.

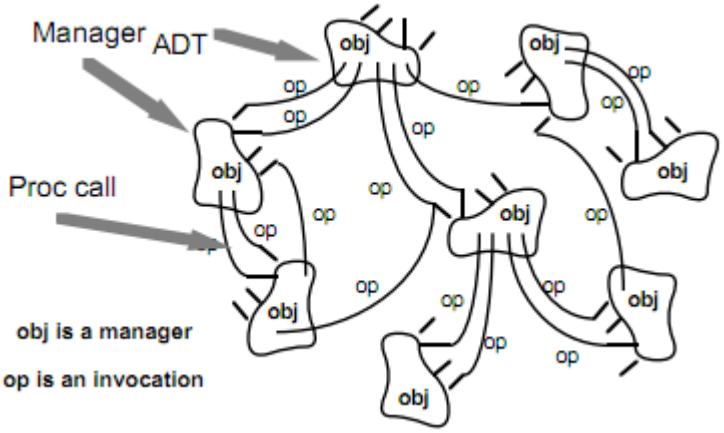
- The components of this style are the objects—or, if you will, instances of the abstract data types.

- Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation).

- Objects interact through function and procedure invocations.

- Two important aspects of this style are

  (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and
  (b) that the representation is hidden from other objects.



- The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread.

- Object-oriented systems have many nice properties, most of which are well known.

- Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients.

- Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

- But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other
- object.

- This is in contrast, for example, to pipe and filter systems, where filters do need

- The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it.

- In a module oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module.

Event-based, implicit invocation

- Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines.

- However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast.

- This style has historical roots in systems based on actors, constraint satisfaction, daemons, and packet-switched networks.

- The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events.
- Other components in the system can register an interest in an event by associating a procedure with the event.

- When the event is announced the system itself invokes all of the procedures that have been registered for the event.

- Thus an event announcement ``implicitly'' causes the invocation of procedures in other modules.

- Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events.
- Procedures may be called in the usual way.

- But in addition, a component can register some of its procedures with events of the system.

- This will cause these procedures to be invoked when those events are announced at run time.

- Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls.

- The main invariant of this style is that announcers of events do not know which components will be affected by those events.

- Thus components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events.

- For this reason, most implicit invocation systems also include explicit invocation (i.e., normal procedure call) as a complementary form of interaction.

- One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system.

- A second benefit is that implicit invocation eases system evolution.

- Components may be replaced by other components without affecting the interfaces of other components in the system.

- In contrast, in a system based on explicit invocation, whenever the identity of a that provides some system function is changed, all other modules that import that module must also be changed.

- The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system.

- When a component announces an event, it has no idea what other components will respond to it.

- Another problem concerns exchange of data. Sometimes data can be passed with the event. But
- in other situations event systems must rely on a shared repository for interaction.

- Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked.

- This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about an invocation of it

Common Architectural Styles

- Software architects use a number of commonly recognized ìstylesî to develop the architecture of a system.

- Architectural style implies a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints that are implemented.

- Components, including encapsulated subsystems, may be distinguished by the nature of their computation (e.g., whether they retain state from one invocation to another, and if so, whether that state is available to other components).

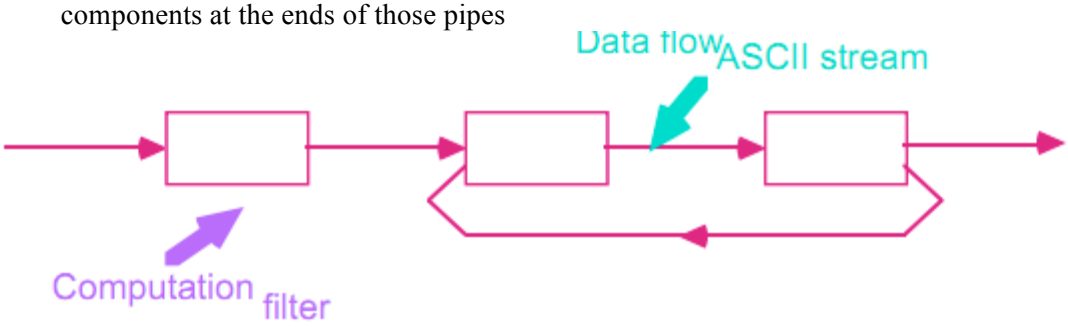
- Component types may also be distinguished by their packagingóthe ways they interact with other components.

- Packaging is usually implicit, which tends to hide important properties of the components. To clarify the abstractions we isolate the definitions of these interaction protocols in connectors (e.g., processes interact via message-passing protocols; unix filters interact via data flow through pipes).

- The connectors play a fundamental role in distinguishing one architectural style from

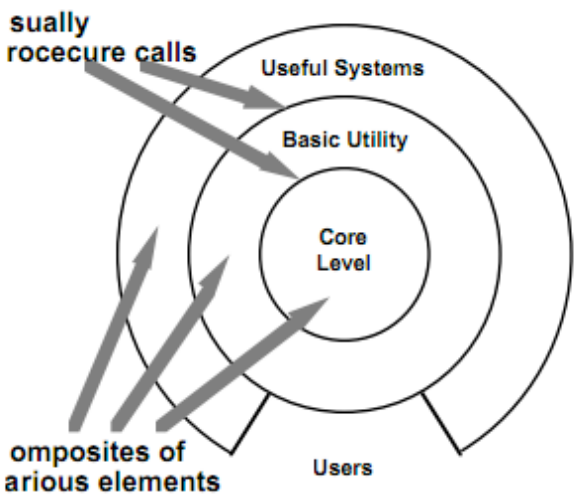
Pipes and Filters

- The style of a specific system is usually established by appeal to common knowledge or intuition. Architectures have traditionally been expressed in box-and-line diagrams and informal prose, so the styles provide drawing conventions, vocabulary, and informal constraints Pipes and Filters
- In a pipe and filter style each component has a set of inputs and a set of outputs.
- A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order.
- This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed.
- Hence components are termed "filters". The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed "pipes".
- Among the important invariants of the style, filters must be independent entities: in particular, they should not share state with other filters.
- Another important invariant is that filters do not know the identity of their upstream and downstream filters.
- Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes



Layered Systems

- A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below.
- In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export.
- Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.)
- The connectors are defined by the protocols that determine how the layers will interact.
- Topological constraints include limiting interactions to adjacent layers.



Repositories

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems.

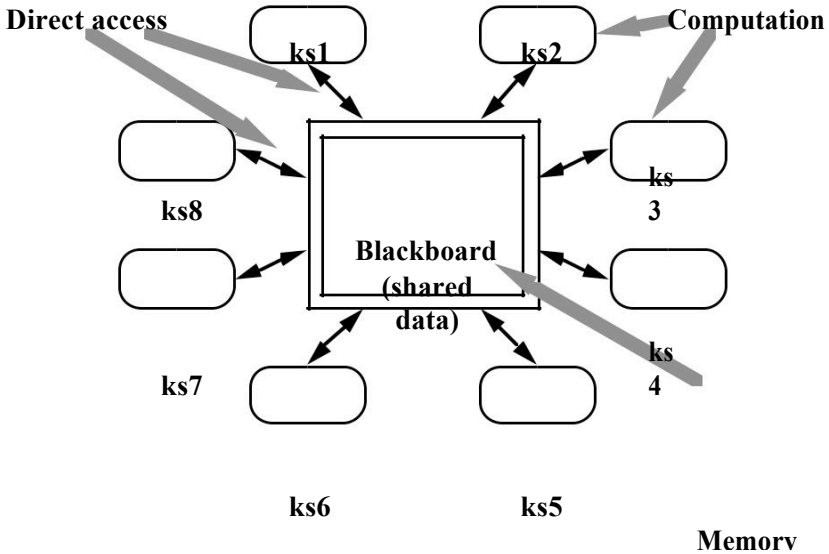
The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard.

Figure 4 illustrates a simple view of a blackboard architecture. (We will examine more detailed models in the case studies.) The blackboard model is usually presented with three major parts:

The knowledge sources: separate, independent parcels of application-dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.

The blackboard data structure: problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

Control: driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.



In the diagram there is no explicit representation of the control component. Invocation of a knowledge source is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these.

There are, of course, many other examples of repository systems. Batch-sequential systems with global databases are a special case. Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments. Even applications that have been traditionally viewed as pipeline architectures, may be more accurately interpreted as repository systems. For example, as we will see later, while a compiler architecture has traditionally been presented as a pipeline, the "phases" of most modern compilers operate on a base of shared information (symbol tables, abstract syntax tree, etc.).

Table Driven Interpreters

In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of *its* execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contain the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated. (See Figure 5.)

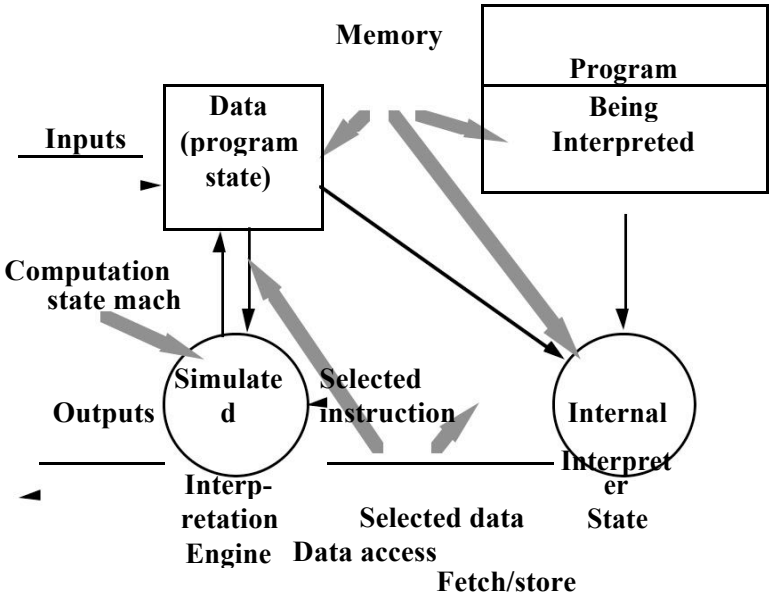


Figure 5: Interpreter

Interpreters are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. We occasionally speak of a programming language as providing, say, a "virtual Pascal machine."

Other Familiar Architectures

There are numerous other architectural styles and patterns. Some are widespread and others are specific to particular domains. While a complete treatment of these is beyond the scope of this paper, we briefly note a few of the important categories.

- **Distributed processes:** Distributed systems have developed a number of common organizations for multi-process systems. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in

terms of the kinds of inter-process protocols that are used for communication (e.g., heartbeat algorithms).

- **Main program/subroutine organizations:** The primary organization of many systems mirrors the programming language in which the system is written. For languages without support for modularization this often results in a system organized around a main program and a set of subroutines. The main program acts as the driver for the subroutines, typically providing a control loop for sequencing through the subroutines in some order.
- **Domain-specific software architectures:** Recently there has been considerable interest in developing "reference" architectures for specific domains . These architectures provide an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle management systems. By specializing the architecture to the domain, it is possible to increase the descriptive power of structures. Indeed, in many cases the architecture is sufficiently constrained that an executable system can be generated automatically or semi-automatically from the architectural description itself.
- **State transition systems:** A common organization for many reactive systems is the state transition system . These systems are defined in terms a set of states and a set of named transitions that move a system from one state to another.
- **Process control systems:** Systems intended to provide dynamic control of a physical environment are often organized as process control systemsthese systems are roughly characterized as a feedback loop in which inputs from sensors are used by the process control system to determine a set of outputs that will produce a new state of the environment.

Heterogeneous Architectures

Thus far we have been speaking primarily of "pure" architectural styles. While it is important to understand the individual nature of each of these styles, most systems typically involve some combination of several styles.

There are different ways in which architectural styles can be combined. One way is through hierarchy. A component of a system organized in one architectural style may have an internal structure that is developed a completely different style. For example, in a Unix pipeline the individual components may be represented internally using virtually any style— including, of course, another pipe and filter, system.

What is perhaps more surprising is that connectors, too, can often be hierarchically decomposed. For example, a pipe connector may be implemented internally as a FIFO queue accessed by insert and remove operations.

A second way for styles to be combined is to permit a single component to use a mixture of architectural connectors. For example, a component might access a repository through part of its interface, but interact through pipes with other components in a system, and accept control information through another part of its interface. (In fact, Unix pipe and filter systems do this, the file system playing the role of the repository and initialization switches playing the role of control.)

Another example is an "active database". This is a repository which activates external components through implicit invocation. In this organization external components register interest in portions of the database. The database automatically invokes the appropriate tools based on this association. (Blackboards are often constructed this way; knowledge sources are associated with specific kinds of data, and are activated whenever that kind of data is modified.)

A third way for styles to be combined is to completely elaborate one level of architectural description in a completely different architectural style.

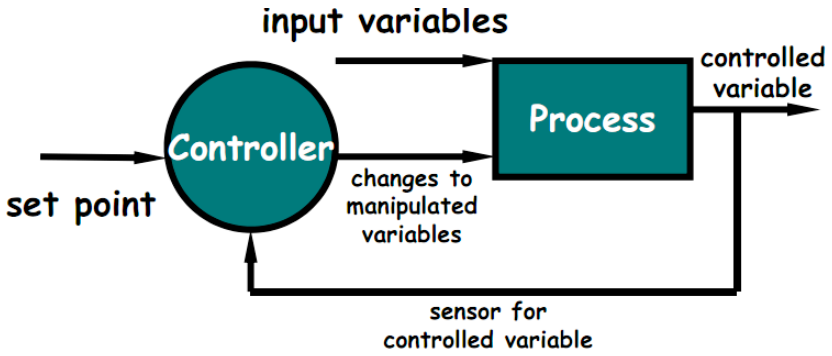
Connectors: are the data flow relations for:

- Process Variables: –Controlled variable whose value the system is intended to control.
- Input variable that measures an input to the process.

- Manipulated variable whose value can be changed by the controller.
- Set Point is the desired value for a controlled variable.
- Sensors to obtain values of process variables pertinent to control

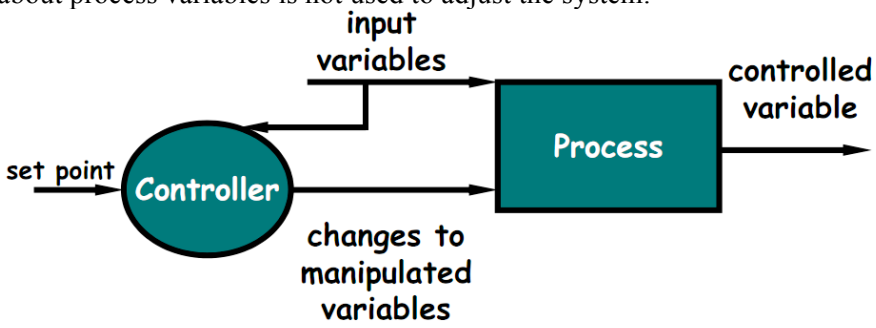
Feed-Back Control System

The controlled variable is measured and the result is used to manipulate one or more of the process variables.



Open-Loop Control System

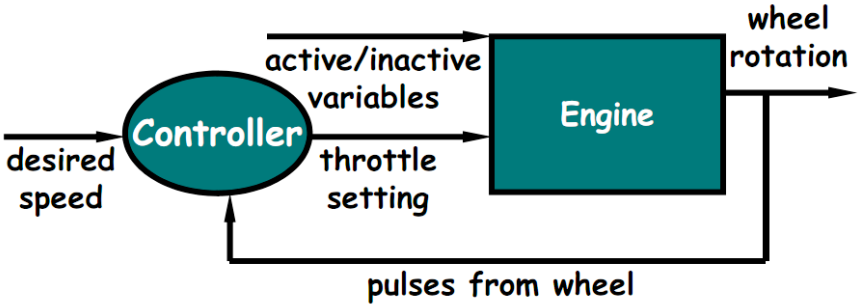
Information about process variables is not used to adjust the system.



Process Control Examples

Real-Time System Software to Control:

- Automobile Anti-Lock Brakes
- NuclearPower Plants
- Automobile Cruise-Control



- The first task is to model the current speed from the wheel pulses; the designer should

- The model could fail if the wheels spin; this could affect control in two ways.

- If the wheel pulses are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving.

- The controller is implemented as a continuously-evaluating function that matches the dataflow character of the inputs and outputs.

- Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines.

- The set point calculation divides naturally into two parts:

  (a) determining whether or not the automatic system is active—in control of the throttle and

  (b) determining the desired speed for use by the controller in automatic mode.

Case Studies

- The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.

- Any line may be ``circularly shifted'' by repeatedly removing the first word and appending it at the end of the line.

- The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

- From the point of view of software architecture, the problem derives its appeal from the fact that it can be used to illustrate the effect of changes on software design.

- Parnas shows that different problem decompositions vary greatly in their ability to withstand design changes.

Among the changes he considers are:

• Changes in processing algorithm: For example, line shifting can be performed on each line as it is read from the input device, on all the lines after they are read, or on demand when the alphabetization requires a new set of shifted lines.

• Changes in data representation: For example, lines can be stored in various ways. Similarly, circular shifts can be stored explicitly or implicitly (as pairs of index and offset).

• Enhancement to system function: For example, modify the system so that shifted lines to eliminate circular shifts that start with certain noise words (such as “a”, “an”, “and”, etc.).

• Performance: Both space and time.

• Reuse: To what extent can the components serve as reusable entities.

Solution 1:

- Main Program/Subroutine with Shared Data The first solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output.

- These computational components are coordinated as subroutines by a main program that sequences through them in turn.

- Data is communicated between the components through shared storage ("core storage"). Communication between the computational components and the shared data is an unconstrained read-
- write protocol.

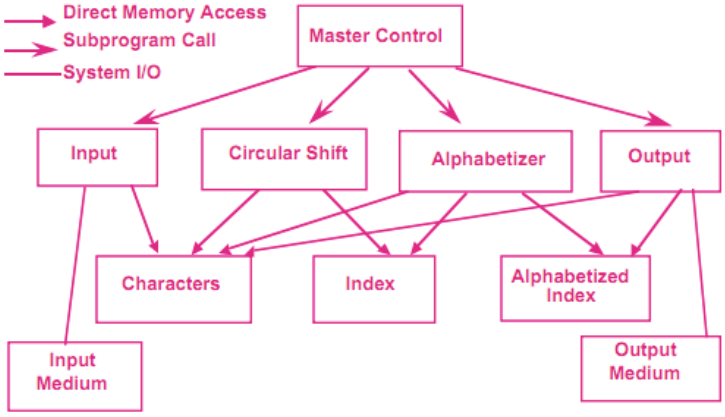
- This is made possible by the fact that the coordinating program guarantees sequential access to the data.

- Using this solution data can be represented efficiently, since computations can share the same storage.

- The solution also has a certain intuitive appeal, since distinct computational aspects are isolated in different modules.

- However, as Parnas argues, it has a number of serious drawbacks in terms of its ability to handle changes. In particular, a change in data storage format will affect almost all of the modules. Similarly changes in the overall processing algorithm and enhancements to system function are not easily accomodated.

- Finally, this decom-position is not particularly supportive of reuse.



Solution 2: Abstract Data Types

- The second solution decomposes the system into a similar set of five modules.

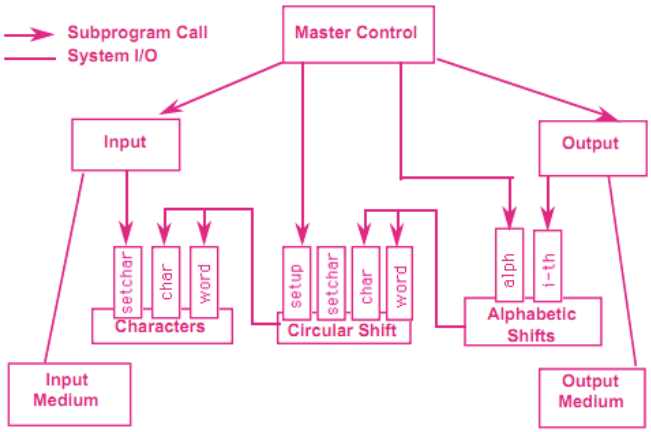
- However, in this case data is no longer directly shared by the computational components.

- Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface.

- This solution provides the same logical decomposition into processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered.

- In particular, both algorithms and data representations can be changed in individual modules without
affecting others.

- Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.



- On the other hand, as discussed by Garlan, Kaiser, and Notkin, the solution is not particularly well-suited to enhancements.

- The main problem is that to add new functions to the system, the implementor must either modify the existing modules—compromising their simplicity and integrity—or add new modules that lead to performance penalties.

Solution 3: Implicit Invocation

- The third solution uses a form of component integration based on shared data similar to the first solution. However, there are two important differences.

- First, the interface to the data is more abstract. Rather than exposing the storage formats to the computing modules, data is accessed abstractly.

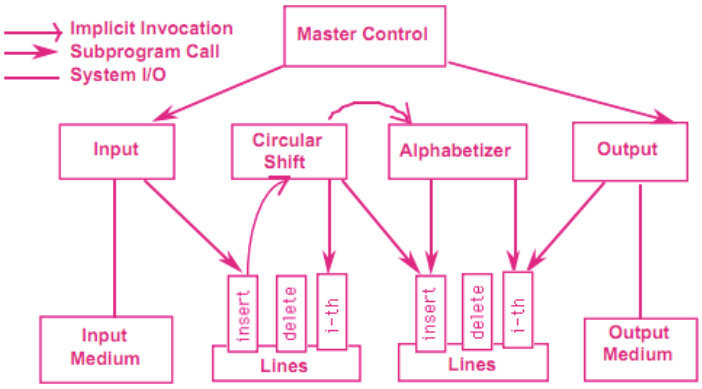
- Second, computations are invoked implicitly as data is modified. Thus interaction is based on an active data model. This in turn causes the alphabetizer to be implicitly invoked so that it can alphabetize the lines.

- This solution easily supports functional enhancements to the system: additional modules can be attached to the system by registering them to be invoked on data-changing events. Because data is accessed abstractly, it also insulates computations from changes in data representation.

- Reuse is also supported, since the implicitly invoked modules only rely on the existence of certain externally triggered events

- However, the solution suffers from the fact that it can be difficult to control the order of processing of the implicitly invoked modules.

- Further, because invocations are data driven, the most natural implementations of this kind of decomposition tend to use more space than the previously considered decompositions.



Solution 4: Pipes and Filters

The fourth solution uses a pipeline solution.

In this case there are four filters: input, shift, alphabetize, and output. Each filter processes the data and sends it to the next filter.

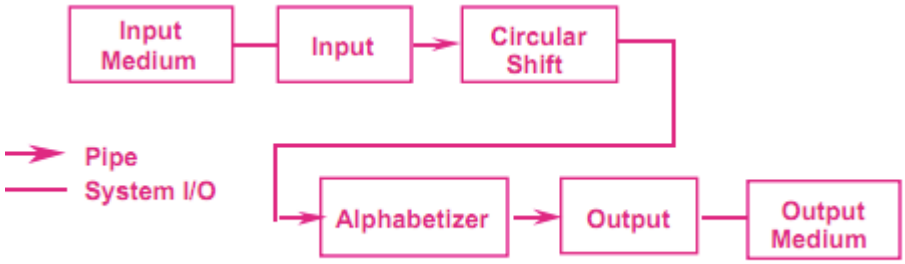
Control is distributed: each filter can run whenever it has data on which to compute. Data sharing between filters is strictly limited to that transmitted on pipes.

This solution has several nice properties.

First, it maintains the intuitive flow of processing.

Second, it supports reuse, since each filter can function in isolation (provided upstream filters produce data in the form it expects). New functions are easily added to the system by inserting filters at the appropriate point in the processing sequence.

Third, it supports ease of modification, since filters are logically independent of other filters.



On the other hand it has a number of drawbacks. First, it is virtually impossible to modify the design to support an interactive system. For example, in order to delete a line, there would have to be some persistent shared storage, violating a basic tenet of this approach.

Second, the solution is inefficient in terms of its use of space, since each filter must copy all of the data to its output ports.

Case Study 2: Instrumentation Software

Our second case study describes the industrial development of a software architecture at Tektronix, Inc. This work was carried out as a collaborative effort between several Tektronix product divisions and the Computer Research Laboratory over a three year period.

The purpose of the project was to develop a reusable system architecture for oscilloscopes. An oscilloscope is an instrumentation system that samples electrical signals and displays pictures (called traces) of them on a screen. Additionally, oscilloscopes perform measurements on the signals, and also display these on the screen. While oscilloscopes were once simple analogue devices involving little software, modern oscilloscopes rely primarily on digital technology and have quite complex software. It is not uncommon for a modern oscilloscope to perform dozens of measurements, supply megabytes of internal storage, interface to a network of workstations and other instruments, and provide sophisticated user interface including a touch panel screen with menus, built-in help facilities, and color displays.

Like many companies that have had to rely increasingly on software to support their products, Tektronix was faced with number of problems. First, there was little reuse across different oscilloscope products. Instead, different oscilloscopes were built by different product divisions, each with their own development conventions, software organization, programming language, and development tools. Moreover, even within a single product division, each new oscilloscope typically required a redesign from scratch to accommodate changes in hardware capability and new requirements on the user interface. This problem was compounded by the fact that both hardware and interface requirements were changing increasingly rapidly. Furthermore, there was a perceived need to address "specialized markets". To do this it would have to be possible to tailor a general-purpose instrument, to a specific set of uses.

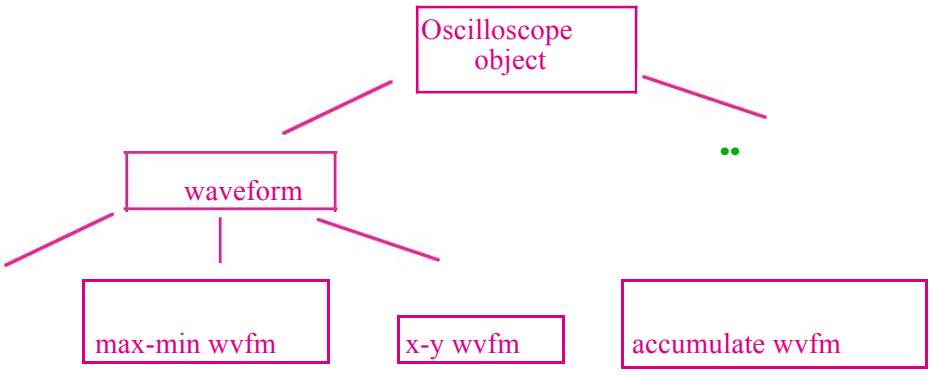
Second, there were increasing performance problems because the software was not rapidly configurable within the instrument. This problem arises because an oscilloscope can be configured in many different modes, depending on the user's task. In old oscilloscopes reconfiguration was handled simply by loading different software to handle the new mode. But as the total size of software was increasing, this was leading to delays between a user's request for a new mode and a reconfigured instrument.

The goal of the project was to develop an architectural framework for oscilloscopes that would address these problems. The result of that work was a domain-specific software architecture that formed the basis of the next Sgeneration of Tektronix oscilloscopes. Since then the framework has been extended and adapted to accommodate a broader class of system, while at the same time being better adapted to the specific needs of instrumentation software.

In the remainder of this section, we outline the stages in this architectural development.

An object-oriented model

The first attempt at developing a reusable architecture focused on producing an object-oriented model of the software domain. This led to a clarification of the data types used in oscilloscopes: waveforms, signals, measurements, trigger modes, etc. (See Figure 11.)



While this was a useful exercise, it fell far short of producing the hoped-for results. Although many types of data were identified, there was no overall model that explained how the types fit together. This led to confusion about the partitioning of functionality. For example, should measurements be associated with the types of data being measured, or represented externally? Which objects should the user interface talk to?

A layered model

The second phase attempted to correct these problems by providing a layered model of an oscilloscope. In this model the core layer represented the signal manipulation functions that filter signals as they enter the oscilloscope. These functions are typically implemented in hardware. The next layer represented waveform acquisition. Within this layer signals are digitized and stored internally for later processing. The third layer consisted of waveform manipulation, including measurement, waveform addition, Fourier transformation, etc. The fourth layer consisted of display functions. This layer was responsible for mapping digitized waveforms and measurements to visual representations. The outermost layer was the user interface. This layer was responsible for interacting with the user and for deciding which data should be shown on the screen.

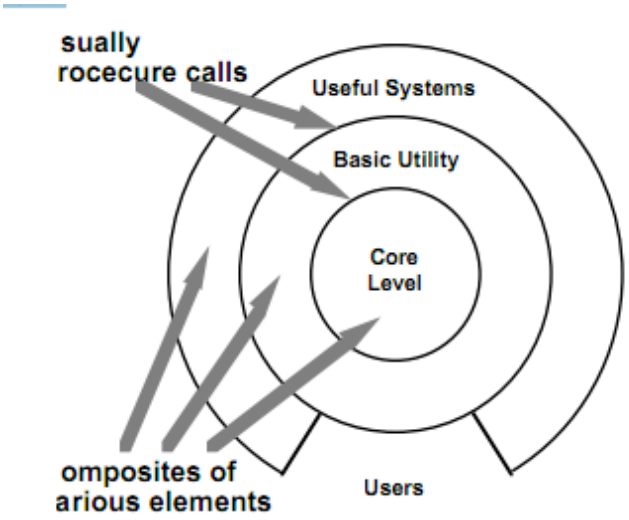


Figure 12: Oscilloscopes – A Layered Model

This layered model was intuitively appealing since it partitioned the functions of an oscilloscope into well-defined groupings. Unfortunately it was the wrong model for the application domain. The main problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction between the various functions. For example, the model suggests that all user interactions with an oscilloscope should be in terms of the visual representations. But in practice real oscilloscope users need to directly affect the functions in all layers, such as setting attenuation in the signal manipulation layer, choosing acquisition mode and parameters in the acquisition layer, or creating derived waveforms in the waveform manipulation layer.

A Pipe and Filter Model

The third attempt yielded a model in which oscilloscope functions were viewed as incremental

transformers derive digitized waveforms from these signals. Display transformers convert these waveforms into visual data. (See Figure 13.)

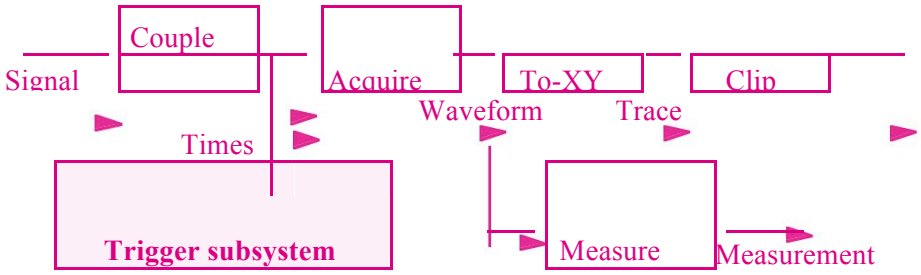


Figure 13: Oscilloscopes – A Pipe and Filter Model

This architectural model was a significant improvement over the layered model in that it did not isolate the functions in separate partitions. For example, nothing in this model would prevent signal data directly feeding into display filters. Further, the model corresponded well to the engineers' view of signal processing as a dataflow problem. The main problem with the model was that it was not clear how the user should interact with it. If the user were simply at one end of the system, then this would represent an even worse decomposition than the layered system.

A Modified Pipe and Filter Model

The fourth solution accounted for user inputs by associating with each filter a control interface that allows an external entity to set parameters of operation for the filter. For example, the acquisition filter might have parameters that determine sample rate and waveform duration. These inputs serve as configuration parameters for the oscilloscope. Formally, the filters can be modelled as "higher-order" functions, for which the configuration parameters determine what data transformation the filter will perform. Figure 14 illustrates this architecture.

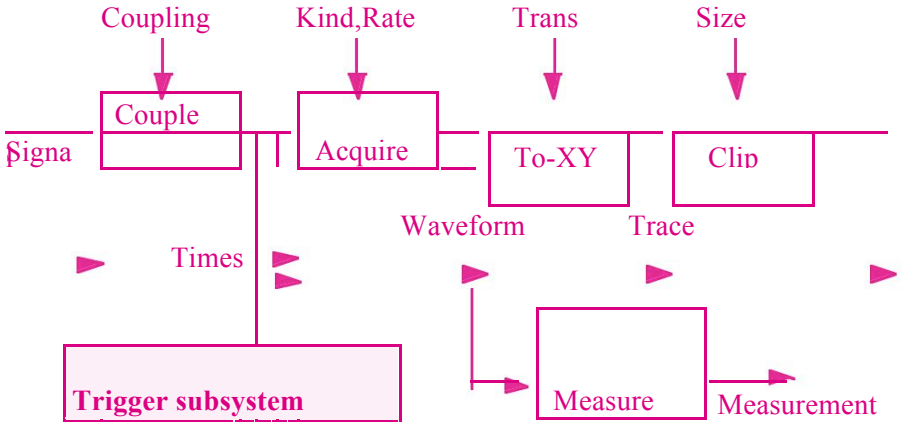


Figure 14: Oscilloscopes – A Modified Pipe and Filter Model

The introduction of a control interface solves a large part of the user interface problem. First, it provides a collection of settings that determine what aspects of the oscilloscope can be modified dynamically by the user. It also explains how changes to oscilloscope function can be accomplished by incremental adjustments to the software. Second it decouples the signal processing functions of the oscilloscope from the actual user interface: the signal processing software makes no assumptions about how the user actually communicates changes to its control parameters. Conversely, the actual user interface can treat the signal processing functions solely in terms of the control parameters. This allowed the designers to change the implementation of the signal processing software and hardware without impacting an interface, provided the control interface remained unchanged.

Controls manned/partially manned vehicles Space exploration, hazardous waste disposal, underwater exploration

The software must deal with:

- Real-time responsiveness
- Acquire sensor I/P, control motion and plan

Future paths

- Many issues from imperfect inputs to unexpected/unpredictable obstacles, and events

Mobile Robots

Four basic requirements

Accommodate deliberate and reactive behavior

• Coordinate actions it must undertake to achieve its designated objective (collect rock sample, avoid obstacles)

Allow for uncertainty

• Framework for actions even when faced with incomplete or unreliable information (contradictory sensor readings)

Account for dangers

• Must be fault tolerant, safe and with high performance (e.g., cope with reduced power, dangerous vapors, etc.)

Give design flexibility

• Development requires frequent experimentation and reconfiguration

Four Architectural Designs

- Lozano's Control Loops
- Elfes's Layered Organization
- Simmons's Task Control Architecture
- Shafer's Application of Blackboards

Solution 1: Control Loop

- Industrial robots need only handle minimally unpredictable events
- Tasks are fully defined (no need for a planer) and has no responsibility wrt its environment

• Open loop paradigm applies

• Robot initiates actions without caring about

Consequences

Lets add feedback for closed loop

• Robot adjusts the future plans based on monitored information

Requirements Trade-Off Analysis

Req1– Advantage: simplicity

- Simplicity is a drawback in more unpredictable environments
- Robots mostly confronted with disparate discrete events that require them to switch between very different behavior modes
- For complex tasks, gives no leverage for decomposition into cooperating components

Req2– Advantage: reducing unknowns through iteration

- Is biased toward one method (only).
- Trial and error process of action-reaction to eliminate possibilities at each turn.
- No framework for integrating these with the basic loop or for delegating them to separate entities.

Req3– Advantage: supports fault tolerance and safety

- Simplicity makes duplication easy
- Reduces the chance of errors creeping into the system

Req4 – Advantage: clearly partition-able into supervisor, sensors and motors that are independent and replaceable

- More refined tuning is however not really supported (inside the modules)

Solution 2: Layered Architecture

Influenced the sonar and navigational systems design used on the Terregator and Neptune mobile Robots
Level 1 (core) control routines (motors, joints,..),
Level 2-3 real world I/P (sensor interpretation and integration (analysis of combined I/Ps)
Level 4 maintains the real world model for robot
Level 5 manage navigation
Level 6-7 Schedule & plan robot actions (including exception handling and replanning)

Top level deals with UI and overall superviosory functions

Requirements Trade-Off Analysis
Req1

- Avoids some problems encountered in the control loop style by defining more components to delegate tasks.
- Defines abstraction levels (robot control versus navigation) to guide the design
- Does not however fit the actual data / control flow patterns!!
- Information exchange is less straightforward because the layers suggest that services and requests be
  passed between layers

• Fast reaction times drives the need to bypass layers to go directly to the problem-handling agent at level 7 skip layers to improve response time!
Two separate abstractions are needed that are not supported
• Data hierarchy
• Control hierarchy
Req2 Abstraction layers address the need to manage uncertainty
What is uncertain at the lower layers may become clear with added knowledge available from the higher layers For Example
• The context embodied in the world model can provide the clues to disambiguate conflicting sensor data
Req3 Fault tolerance and passive safety (strive not to do something)

- Thumbs up data and commands are analyzed from different perspectives
- Possible to incorporate many checks and balances
- Performance and active safety may require that layers be short circuited

Req4 Flexibility in replacement and addition of components

- Interlayer dependencies are an obstacle
- Complex relationships between layers can become more difficult to decipher with each change
- Success because the layers provide precision in defining the roles of each layer

Solution 3: Implicit Invocation
Basis and Specifics

- Based on various hierarchies of tasks
- Utilizes dynamic task trees
- Run-time configurable
- Permits selective concurrency

Supports 3 different functions
Exceptions
• Suited to handle spontaneous events
• Manipulate task trees
Wiretapping

Monitors
• Read info and execute some actions if data fulfills a criterion

Req1 Advantage: clear cut separation of action
- Explicit incorporation of concurrent agents in its model

Req2 Disadvantage: uncertainty not well addressed
- Task tree could be built by exception handler

Req3 Advantage: accounts for performance, safety, & fault tolerances
- Redundant fault handlers
- Multiple requests handled concurrently

Req4 advantage: Incremental development & replacement straightforward
- Possible to use wiretaps, monitors, or new handlers without affecting existing components
- Based on CODGER system used in NAVLAB project (known as whiteboard arch)
- Relies on abstractions similar to those found in the layered architecture example
- Utilizes a shared repository for communication between components

Blackboard Arch.
- Based on CODGER system used in NAVLAB project (known as whiteboard arch)
- Relies on abstractions similar to those found in the layered architecture example
- Utilizes a shared repository for communication between components
- Components register interest in certain types of data
- This info is returned immediately or when it is inserted onto blackboard by some other module

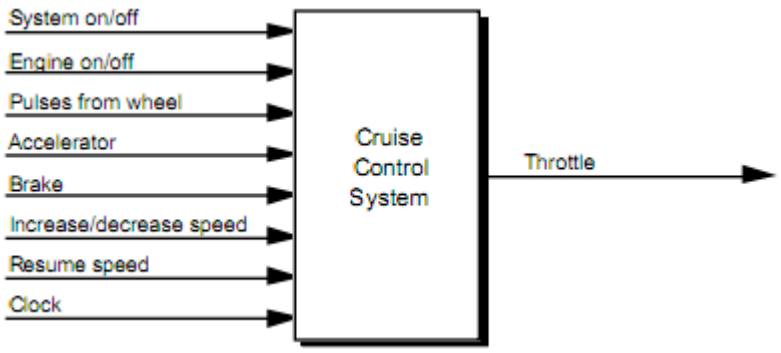
Components of CODGER architecture are:
Captain: overall supervisor
Map navigator: high-level path planner
Lookout: monitors environment for landmarks
Pilot: low-level path planner and motor controller
Perception subsystems: accept sensor input and integrate it into a coherent situation interpretation

Req1 Deliberative and reactive
- Components register for the type of information they are interested in and receive it as it becomes available
- This shared communication mechanism supports both deliberative and reactive behavior requirements
- However, the control flow must be worked around the database mechanism; rather than communication between components

Req2 Allow for uncertainty
- provides means for resolving conflicts or uncertainties as all data is in database (from all components)
- modules responsible for resolution simply register for required data and process it accordingly

Cruise Control
A cruise control system exists to maintain the speed of a car, even over varying terrain.

The block diagram of the hardware for such a system.



There are several inputs:

• System on/off If on, denotes that the cruise-control system should maintain the car speed.
• Engine on/off If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
• Pulses from wheel A pulse is sent for every revolution of the wheel.
• Accelerator Indication of how far the accelerator has been pressed.
• Brake On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
• Increase/Decrease Speed Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
• Resume: Resume the last maintained speed; only applicable if the cruise control system is on.
• Clock Timing pulse every millisecond.

There is one output from the system:

• Throttle Digital value for the engine throttle setting.

- The problem does not clearly state the rules for deriving the output from the set of inputs.
- Booch provides a certain amount of elaboration in the form of a data flow diagram, but some questions remain unanswered.
- In the design below, missing details are supplied to match the apparent behavior of the cruise control on the author's car.
- Moreover, the inputs provide two kinds of information: whether the cruise control is active, and if so what speed it should maintain.
- The problem statement says the output is a value for the engine throttle setting.
- In classical process control the corresponding signal would be a change in the throttle setting; this avoids calibration and wear problems with the sensors and engine.
- A more conventional cruise control requirement would thus specify control of the current speed of the vehicle.
- However, current speed is not explicit in the problem statement, though it does appear implicitly as “maintained speed” in the descriptions of some of the inputs.
- If the requirement addresses current speed, throttle setting remains an appropriate output from the control algorithm.

- To avoid unnecessary changes in the problem we assume accurately calibrated digital control and achieve the effect of incremental signals by retaining the previous throttle value in the controller.

- The problem statement also specifies a millisecond c lock.

- In the object-oriented solution, the clock is used only in combination with the wheel pulses to determine the current speed.

- Presumably the process that computes the speed will count the number of clock pulses between wheel pulses.

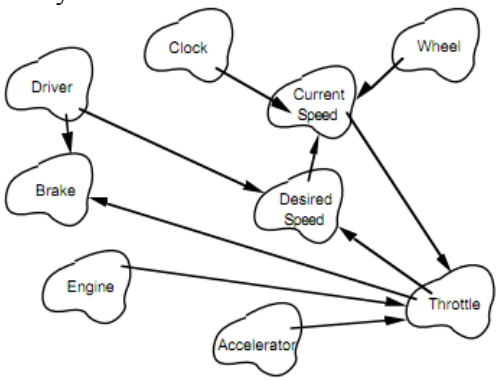
- A typical automobile tire has a circumference of about 6 feet, so at 60 mph (88 ft/sec) there will be about 15 wheel pulses per second.
- The problem is over specified in this respect: a slower clock or one that delivered current time on demand with sufficient precision would also work and would require less computing.

- Further, a single system clock is not required by the problem, though it might be convenient for other reasons.

- These considerations lead to a restatement of the problem: Whenever the system is active, determine the desired speed and control the engine throttle setting to maintain that speed.

Object view of cruise control

- Booch structures an object-oriented decomposition of the system around objects that exist in the task description.
- This yields a decomposition whose elements correspond to important quantities and physical entities in the system.



Process control view of cruise control

- The selection of a control loop architecture when the software is embedded in a physical system that involves continuing behavior, especially when the system is subject to external perturbations.

- These conditions hold in the case of cruise control: the system is supposed to maintain constant speed in an automobile despite variations in terrain, vehicle load, air resistance, fuel quality, etc.

Computational elements

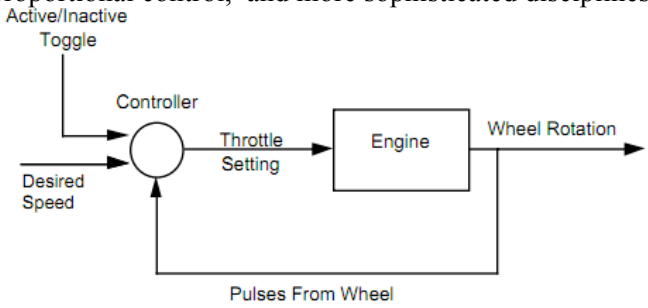
• Control algorithm: This algorithm models the current speed based on the wheel pulses, compares it to the desired speed, and changes the throttle setting.

Data elements

• Controlled variable: For the cruise control, this is the current speed of the vehicle.
• Manipulated variable: For the cruise control, this is the throttle setting.
• Set point: The desired speed is set and modified by the accelerator input and the increase/decrease speed input, respectively.
• Sensor for controlled variable: For cruise control, the current state is the current speed, which is modeled on data from a sensor that delivers wheel pulses using the clock.

The first task is to model the current speed from the wheel pulses; the designer should validate this model carefully. The model could fail if the wheels spin; this could affect control in two ways. If the wheel pulses are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving.

The controller is implemented as a continuously-evaluating function that matches the dataflow character of the inputs and outputs. Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines.



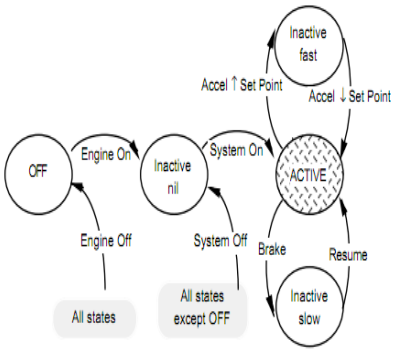
The set point calculation divides naturally into two parts: (a) determining whether or not the automatic system is active—in control of the throttle and (b) determining the desired speed for use by the controller in automatic mode.

- The active/inactive toggle is triggered by a variety of events, so a state transition design is natural.

- The system is completely off whenever the engine is off. Otherwise there are three inactive and one
- active states.

- In the first inactive state no set point has been established.

- The active/inactive toggle input of the control system is set to active exactly when this state machine is in state Active.



Three Vignettes in Mixed Style

Purpose is to review three systems with mixed styles of architecture

PROVOX process control system

Hayes-Roth Rule Based system

HEARSAY II speech recognition system

PROVOX

The Fisher Controls PROVOX system offers distributed process control for chemical production processes:
simple control loops to control pressure, flow, levels
complex strategies involving interrelated control loops

provisions for integration with plant management and information systems

PROVOX Hierarchy

Integrates process control with plant management and other corporate information systems

Level 1: Process measurement and control

direct adjustment of final control elements

Level 2: Process supervision

operations console for monitoring and controlling Level 1

Level 3: Process management
computer based plant automation; including management,reports, optimization strategies, and guidance to the operations console

Level 4 & 5: Plant and corporate management
higher level functions such as cost accounting, inventory control, and order processing/scheduling

Different computation and response times are required at the different levels of the system

Therefore different computation models are used to achieve these results

Levels 1 - 3: object-oriented

Levels 4 - 5: Largely based on conventional data processing repository (database) models

Expanded Hayes-Roth Rule Based system

Rule based systems heavily use pattern matching and context (currently relevant rules)

Added special mechanisms to facilitate these features complicate the original simple interpreter design

Knowledge base is a relatively simple structure; yet is able to distinguish between active and inactive components

Rule interpreter is implemented as a table driven interpreter

With control procedures for pseudocode and

Execution stack modeling the current program state

Expanded Hayes-Roth Rule Based System

Rule and data element selection" is implemented as a pipeline that progressively transforms active rules and facts to prioritized activations the third filter ("nominators") uses a fixed database of meta-rules Working memory is not further elaborated

Hayes-Roth Conclusions

In a sophisticated rule-based system, elements of the simple rule-based system are elaborated in response to the execution characteristics of the particular class of languages being interpreted

Retains the original concept to guide understanding and ….Ease later maintenance of the system As the design is elaborated, different components can be elaborated with different idioms

Rule-based model can itself be thought of as a design structure:

Set of rules whose control relations are determined during execution by computation state

A rule-based system provides a virtual machine (rule extractor) to support this model

HEARSAY-II

segmenting the raw signal

identifying phenomes

generating word candidates

hypothesizing syntactic segments

proposing semantic interpretations

Knowledge sources contain:

Condition Part:

Action part

Control component is realized as a blackboard monitor and scheduler scheduler monitors blackboard and calculates priorities for applying knowledge source to various blackboard elements

PDP-10 was not directly capable of condition-triggered control

HEARSAY-II implementation compensates by providing mechanisms of a virtual machine to realize implicit invocation semantics

this addition complicates Fig 3.27

Blackboard model can be recovered by

suppressing the control mechanism and

regrouping the conditions and action into knowledge sources

HEARSAY-II

Function assignment facilitates the virtual machine in the form of an interpreter

Blackboard corresponds cleanly to the current state of the recognition task

Collection of knowledge sources roughly supply the pseudocode of the interpreter

actions also contribute

Interpretation engine includes:

blackboard monitor

focus-of-control database

scheduler

actions and knowledge sources