|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

**UNIT-1 (Continuation..)**

**ARCHITECTURAL STYLES AND CASE STUDIES:**

Architectural styles; Pipes and filters; Data abstraction

Object-oriented organization; Event-based, implicit invocation; Layered systems; Repositories; Interpreters; Process control;

Other familiar architectures;

Heterogeneous architectures.

**Case Studies:**

**Keyword in Context;**

**Instrumentation software;**

Mobile robotics

Page 16



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

**Architectural Styles and Case Studies:**

|  |  |  |  |
| --- | --- | --- | --- |
| **2.1 Architectural styles;** | |  |  |
| List of common architectural styles: | |  |  |
| Dataflow systems: | |  | Hierarchical layers. |
|  | Batch sequential | Data-centered systems: | |
|  | Pipes and filters |  | Databases |
| Virtual machines: | |  | Hypertext systems |
|  | Interpreters |  | Blackboards. |
|  | Rule-based systems | Independent components: | |
| Call-and-return systems: | |  | Communicating processes |
| Main program and subroutine | |  | Event systems |

OO systems

1. **Pipes and Filters**

Each components has set of inputs and set of outputs

A component reads streams of data on its input and produces streams of data on its output.

By applying local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence, components are termed as filters.

Connectors of this style serve as conducts for the streams transmitting outputs of one filter to inputs of another. Hence, connectors are termed pipes.

Conditions (invariants) of this style are:

Filters must be independent entities.

They should not share state with other filter

Filters do not know the identity of their upstream and downstream filters.

Specification might restrict what appears on input pipes and the result that appears on the output pipes.

Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

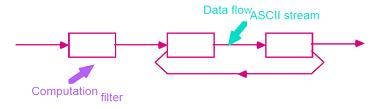


Figure 2.1 Pipes and Filters

Common specialization of this style includes:

Pipelines:

|  |  |  |  |
| --- | --- | --- | --- |
|  | o | Restrict the topologies to linear sequences of filters. |  |
|  | Bounded pipes: | |  |
|  | o | Restrict the amount of data that can reside on pipe. |  |
|  | Typed pipes: | |  |
|  | | | Page 17 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

o Requires that the data passed between two filters have a well-defined type.

A degenerate case of pipeline architecture occurs when each filter processes all of its input data as a single entity. In these systems pipes no longer serve the function of providing a stream of data and are largely vestigial.

|  |  |  |  |
| --- | --- | --- | --- |
| Example 1: | |  |  |
| Best | known | example of pipe-and-filter architecture are programs written | in UNIX-SHELL. |
| Unix | supports this style by providing a notation for connecting components | | [Unix process] and |
| by providing | | run-time mechanisms for implementing pipes. |  |
| Example 2: | |  |  |

Traditionally compilers have been viewed as pipeline systems. Stages in the pipeline include lexical analysis

parsing, semantic analysis and code generation other examples of this type are.

Signal processing domains Parallel processing

Functional processing Distributed systems.

Advantages:

They allow the designer to understand the overall input/output behavior of a system as a simple composition of the behavior of the individual filters.

They support reuse: Any two filters can be hooked together if they agree on data.

Systems are easy to maintain and enhance: New filters can be added to exciting systems. They permit certain kinds of specialized analysis eg: deadlock, throughput

They support concurrent execution.

Disadvantages:

They lead to a batch organization of processing.

Filters are independent even though they process data incrementally. Not good at handling interactive applications

* When incremental display updates are required.
* They may be hampered by having to maintain correspondences between two separate but related streams.
* Lowest common denominator on data transmission.

This can lead to both loss of performance and to increased complexity in writing the filters.

**2.3 Object-Oriented and Data Abstraction**

In this approach, data representation and their associated primitive operations are encapsulated in the abstract data type (ADT) or object. The components of this style are- objects/ADT’s objects interact through function and procedure invocations.

Two important aspects of this style are:

Object is responsible for preserving the integrity of its representation. Representation is hidden from other objects.

Page 18



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

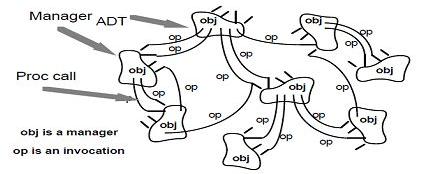


Figure 2.2 Abstract data types and objects

Advantages

It is possible to change the implementation without affecting the clients because an object hides its representation from clients.

The bundling of a set of accessing routines with the data they manipulate allows

designers to decompose problems into collections of interacting agents. Disadvantages

To call a procedure, it must know the identity of the other object.

Whenever the identity of object changes it is necessary to modify all other objects that explicitly invoke it.

1. **Event-Based, Implicit Invocation**

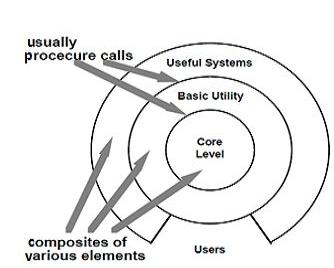
Instead of invoking the procedure directly a component can announce one or more events.

Other components in the system can register an interest in an event by associating a procedure to it.

When the event is announced, the system itself invokes all of the procedure 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 interface provides both a collection of procedures and a set of events.

|  |  |  |
| --- | --- | --- |
|  | Figure 2.3 Layered Systems |  |
| Page 19 |  |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

Advantages:

It provides strong support for reuse

* + Any component can be introduced into the system simply by registering it for the events of that system.

Implicit invocation eases system evolution.

* + Components may be replaced by other components without affecting the

interfaces of other components. Disadvantages:

Components relinquish control over the computation performed by the system. Concerns change of data

Global performance and resource management can become artificial issues.

1. **Layered Systems:**

A layered system is organized hierarchically

Each layer provides service to the layer above it.

Inner layers are hidden from all except the adjacent layers.

Connectors are defined by the protocols that determine how layers interact each other. Goal is to achieve qualities of modifiability portability.

Examples: Layered communication protocol a) Operating systems b) Database systems

Advantages:

They support designs based on increasing levels abstraction.

Allows implementers to partition a complex problem into a sequence of incremental steps. They support enhancement and They support reuse.

Disadvantages:

Not easily all systems can be structures in a layered fashion.

Performance may require closer coupling between logically high-level functions and their lower-level implementations.

Difficulty to mapping existing protocols into the ISO framework as many of those

protocols bridge several layers.

Layer bridging: functions are one layer may talk to other than its immediate neighbor.

1. **Repositories: [data cantered architecture]**

Goal of achieving the quality of integrability of data. In this style, there are two kinds of components.

* + 1. Central data structure- represents current state.
    2. Collection of independent components which operate on central data store. The choice of a control discipline leads to two major sub categories.

Type of transactions is an input stream trigger selection of process to execute

Current state of the central data structure is the main trigger for selecting processes to execute.

Page 20



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

Active repository such as blackboard. Blackboard:

Three major parts: Knowledge sources:

Separate, independent parcels of application – dependents knowledge. Blackboard data structure:

Problem solving state data, organized into an application-dependent hierarchy Control:

Driven entirely by the state of blackboard

Invocation of a knowledge source (ks) is triggered by the state of blackboard. The actual focus of control can be in

* knowledge source
* blackboard
* Separate module or
* combination of these

Blackboard systems have traditionally been used for application requiring complex interpretation of signal processing like speech recognition, pattern recognition.

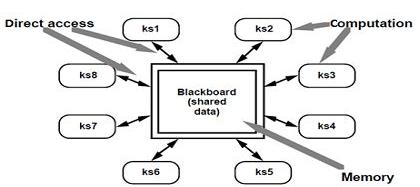


Figure 2.4: The blackboard

1. **Interpreters**

An interpreter includes pseudo program being interpreted and interpretation engine. Pseudo program includes the program and activation record.

Interpretation engine includes both definition of interpreter and current state of its execution. Interpreter includes 4 components:

1 Interpretation engine: to do the work

2 Memory: that contains pseudo code to be interpreted.

3 Representation of control state of interpretation engine

* Representation of control state of the program being simulated. Ex: JVM or “virtual Pascal machine”

Page 21



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

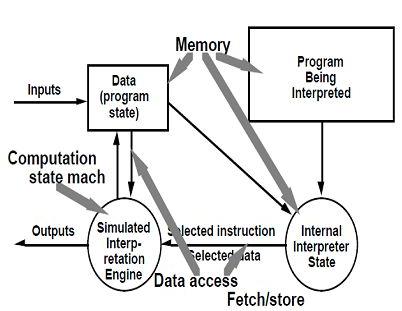


Figure 2.5: Interpreters

Advantages:

Executing program via interpreters adds flexibility through the ability to interrupt and query the program

Disadvantages:

Performance cost because of additional computational involved

|  |  |  |
| --- | --- | --- |
| **2.8 Process control** |  |  |
| **Process control paradigms** |  |  |
| Useful definitions: |  |  |
| Process variables: properties of the process that can be measured | |  |
| Controlled variable :process variable whose value of the system is intended to control | |  |
| Input variable : process variable that measures an input to the process | |  |
| Manipulated variable: process variable whose value can be changed by the controller | |  |
| Set point : the desired value for a controlled variable | |  |
| Open-loop system :system in which information about process variables is not used | | to adjust |
| the system |  |  |
| Closed-loop system :system in which | information about process variables is used to | manipulate |
| a process variable to compensate for | variations in process variables and operating conditions | |

Feedback control system: the controlled variable is measured and the result is used to manipulate one or more of the process variables

Feed forward control system : some of the process variables are measured, and anticipated disturbances are compensated without waiting for changes in the controlled variable to be visible.

The open-loop assumptions are rarely valid for physical processes in the real world. More often, properties such as temperature, pressure and flow rates are monitored, and their values are used

|  |  |
| --- | --- |
| to control the process by changing the settings | of apparatus such as valve, heaters and chillers. |
| Such systems are called closed loop systems. |  |
|  | Page 22 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

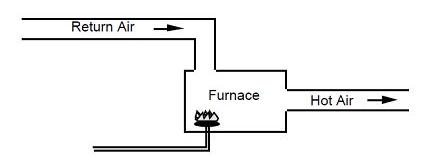


Figure 2.5 open-loop temperature controls

A home thermostat is a common example; the air temperature at the thermostat is measured, and the furnace is turned on and off as necessary to maintain the desired temperature. Figure 2.6 shows the addition of a thermostat to convert figure 2.8 to a closed loop system.

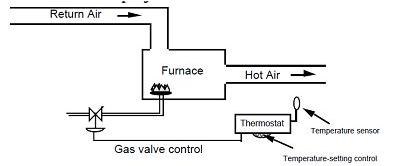


Figure 2.9 closed-loop temperature control

Feedback control:

Figure 2.9 corresponds to figure 2.10 as follows:

The furnace with burner is the process The thermostat is the controller

The return air temperature is the input variable The hot air temperature is the controlled variable The thermostat setting is the set point

Temperature sensor is the sensor

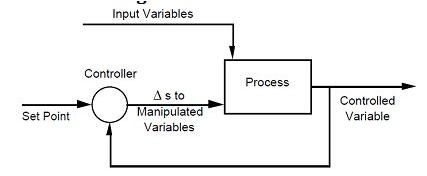


Figure 2.10 feedback control

These are simplified models

They do not deal with complexities - properties of sensors, transmission delays & calibration

|  |  |  |
| --- | --- | --- |
|  | issues |  |
|  | They ignore the response characteristics of the system, such as gain, lag and hysteresis. |  |
|  | They don’t show how combined feedforward and feedback |  |
|  | | Page 23 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

They don’t show how to manipulate process variables. A Software Paradigm for Process Control

An architectural style for software that controls continuous processes can be based on the process-control model, incorporating the essential parts of a process-control loop:

Computational elements: separate the process of interest from the controlled policy Process definition, including mechanisms for manipulating some process variables Control algorithm, for deciding how to manipulate variables

Data element: continuously updated process variables and sensors that collect them

Process variables, including designed input, controlled and manipulated variables and knowledge of which can be sensed

Set point, or reference value for controlled variable

Sensors to obtain values of process variables pertinent to control

The control loop paradigm: establishes the relation that the control algorithm exercises.

Other Familiar Architectures

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. | |  |  |  |
| Domain-specific | | software | architectures: | | 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.

State transition systems: 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.

Heterogeneous Architectures

Architectural styles can be combined in several ways:

One way is through hierarchy. Example: UNIX pipeline

Second way is to combine styles is to permit a single component to use a mixture of architectural connectors. Example: “active database”

Third way is to combine styles is to completely elaborate one level of architectural description in a completely different architectural style. Example: case studies

Page 24



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

**ONLY MOBILE ROBOTICS IS INCLUDED IN SYLLABUS (LAST CASE STUDY)**

**2.9 Case Studies: KEYWORD IN CONTEXT (KWIC)**

This case study shows how different architectural solutions to the same problem provide different benefits.

Parnas proposed the following problems:

KWIC 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 repeated removing the

first word and appending it at the end of the line. KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Parnas used the problem to contrast different criteria for decomposing a system into modules. He describes 2 solutions:

1. Based on functional decomposition with share access to data representation.
2. Based on decomposition that hides design decision.

From the point of view of Software Architecture, the problem is to illustrate the effect of changes on software design. He shows that different problem decomposition vary greatly in their ability to withstand design changes. The changes that are considered by parnas are:

1. The changes in processing algorithm:

Eg: line shifting can be performed on each line as it is read from input device, on all lines after they are read or an demand when alphabetization requires a new set of shifted lines.

1. Changes in data representation:

Eg: Lines, words, characters can be stored in different ways. Circular shifts can be stored explicitly or implicitly,

Garlan, Kaiser and Notkin also use KWIC problem to illustrate modularization schemes based on implicit invocation. They considered the following.

3. Enhancement to system function:

Modify the system to eliminate circular shift that starts with certain noise change the system to interactive.

4. Performance:

Both space and time 5. Reuse:

Extent to which components serve as reusable entities Let’s outline 4 architectural designs for KWIC system.

**Solution 1: Main Program/Subroutine with Shared Data**

Decompose the problem according to 4 basic functions performed.

Input Shift

Alphabetize output

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

Data is communicated between components through shared storage.

Page 25



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

Communication between computational component and shared data is constrained by read-write protocol.

Advantages:

Allows data to be represented efficiently. Since, computation can share the same storage

Disadvantages:

Change in data storage format will affect almost all of the modules.

Changes in the overall processing algorithm and enhancement to system function are

not easily accommodated.

This decomposition is not particularly support reuse.

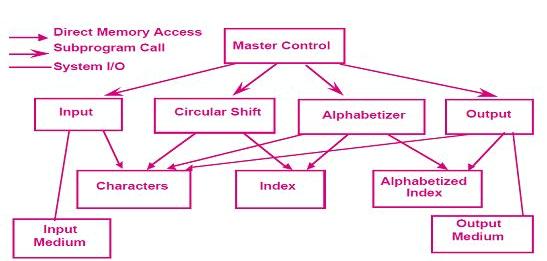


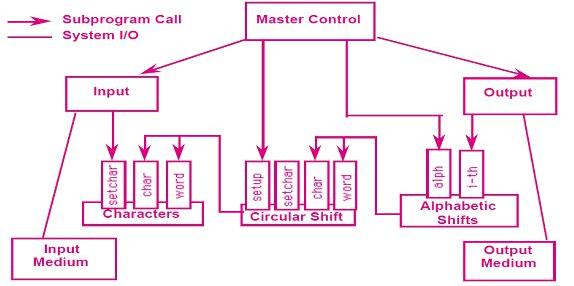
Figure 2.11. KWIC shared data solution

**Solution 2: Abstract Data Types**

Decomposes The System Into A Similar Set Of Five Modules.

Data is no longer directly shared by the computational components.

Each module provides an interface that permits other components to access data only by invoking procedures in that interface.



|  |  |
| --- | --- |
|  | Figure 2.12. KWIC Abstract data type solution |
|  | Page 26 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

Advantage:

Both Algorithms and data representation can be changed in individual modules without affecting others.

Reuse is better supported because modules make fewer assumption about the others with which they interact.

Disadvantage:

Not well suited for functional enhancements

To add new functions to the system To modify the existing modules.

**Solution 3: Implicit Invocation**

Uses a form of component integration based on shared data Differs from 1stsolution by these two factors

o Interface to the data is abstract

o Computations are invoked implicitly as data is modified. Interactions is based on an active

data model. Advantages:

Supports functional enhancement to the system

Supports reuse. Disadvantages:

Difficult to control the processing order.

Because invocations are data driven, implementation of this kind of decomposition uses more space.

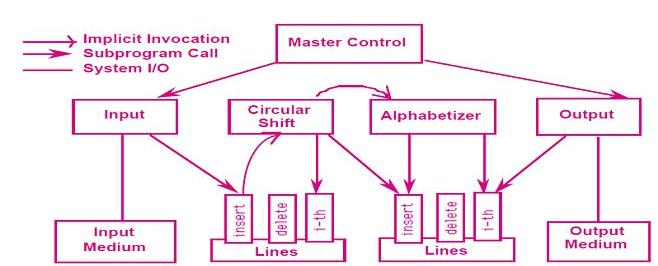


Figure 2.13. KWIC Implicit Invocation solution

**Solution 4: Pipes and Filters:**

Four filters: Input, Output, Shift and alphabetize

Each filter process the data and sends it to the next filter Control is distributed

|  |  |
| --- | --- |
| o Each filter can run whenever it has data on which to compute. |  |
| ata sharing between filters are strictly limited. |  |
|  | Page 27 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

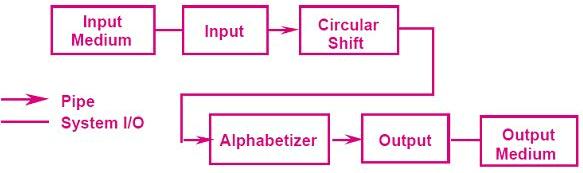


Figure 2.14. KWIC pipes and filters solutions

Advantages

It maintains initiative flow of processing It supports reuse

New functions can be easily added to the system by inserting filters at appropriate level. It is easy to modify.

Disadvantages:

Impossible to modify the design to support an interactive system. Solution uses space inefficiently.

**Comparisons**

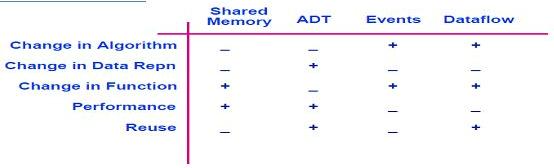


Figure 2.15. Comparison of solution

**2.10. Instrumentation Software:**

Describes the industrial development of software architecture.

The purpose of the project was to develop a reusable system architecture for oscilloscope

Oscilloscope is an instrumentation system that samples electrical signals and displays pictures of them on screen.

Oscilloscope also performs measurements on the signals and displays them on screen.

Modern oscilloscope has to perform dozens of measurements supply megabytes of internal storage.

Support an interface to a network of workstations and other instruments and provide sophisticated user interface, including touch panel screen with menus, built-in help facilities and color displays.

Page 28



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

Problems faced:

* + Little reuse across different oscilloscope products.
  + Performance problems were increasing because the software was not rapidly configurable within the instrument.

Goal of the project was to develop an architectural framework for oscilloscope.

Result of that was domain specific software architecture that formed the basis of the next generation of oscilloscopes.

**Solution 1: Object Oriented Model**

Different data types used in oscilloscope are:

|  |  |  |
| --- | --- | --- |
|  | Waveforms |  |
|  | Signals |  |
|  | Measurements |  |
| Trigger modes so on | |  |
| There | was no overall model that explained | how the types fit together. This led to confusion |
| about | the partitioning of functionality. Ex: it is not clearly defined that measurements to be | |
| associated with types of data being measured | | or represented externally. |

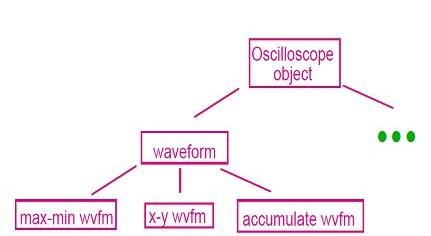


Figure 2.16. Oscilloscope – object oriented model.

**Solution 2: Layered Model**

To correct the problems by providing a layered model of an oscilloscope.

Core-layer: implemented in hardware represents signal manipulation functions that filter signals as they enter the oscilloscope.

Initially the layered model was appealing since it partitioned the functions of an oscilloscope into well defined groups.

But, it was a wrong model for the application domain. Because, the problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction among various functions.

Page 29



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

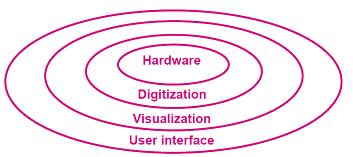


Figure 2.16. Oscilloscope a layered model

**Solution 3: Pipe-And-Filter Model**:

In this approach oscilloscope functions were viewed as incremental transformers of data. o Signal transformer: to condition external signal.

o Acquisition transformer: to derive digitized waveforms

o Display transformers: to convert waveforms into visual data.

It is improvement over layered model as it did not isolate the functions in separate partition. Main problem with this model is that

o It is not clear how the user should interact with it.

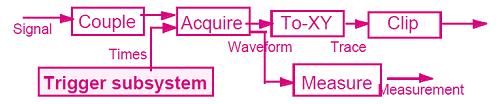


Figure 2.17, Oscilloscope a pipe and filter model

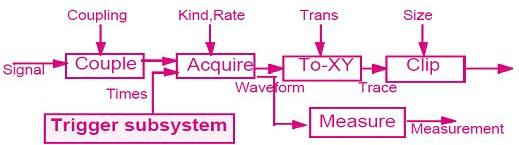
**Solution 4: Modified Pipe-And-Filter Model:**

To overcome the above said problem, associate control interface with each filter that allowed external entity to set parameters of operation for the filter.

Introduction of control interface solves a large part of the user interface problem

It provides collection of setting that determines what aspect of the oscilloscope can be modified dynamically by the user.

It explains how user can change functions by incremental adjustments to the software.



|  |  |
| --- | --- |
|  | Figure 2.18, Oscilloscope a modified pipe and filter model |
|  | Page 30 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

**Further Specialization**

The above described model is greater improvement over the past. But, the main problem with this is the performance.

a. Because waveform occupy large amount of internal storage

It is not practical for each filter to copy waveforms every time they process them.

b. Different filters run at different speeds

It is unacceptable to slow one filter down because another filter is still processing its data. To overcome the above discussed problems the model is further specialized. Instead of using same kind of pipe. We use different “colors” of pipe. To allow data to be processed without copying, slow filters to ignore incoming data. These additional pipes increased the stylistic vocabulary and allowed pipe/filter computations to be tailored more specifically to the performance needs of the product.

**CASE STUDY of Mobile robotics with all solution**

**2.11 Mobile Robotics**

Mobile Robotic systems

Controls a manned or semi-manned vehicle

input from various sensors

Controlling the motion and movement of robots

|  |  |  |  |
| --- | --- | --- | --- |
| E.g., car, space vehicle, etc | |  | Planning its future path/move |
| Used in space exploration missions | |  | � Unpredictability of environment |
|  | Hazardous waste disposal |  | Obstacles blocking robot path |
|  | Underwater exploration |  | Sensor may be imperfect |
| The system is complex | |  | Power consumption |
|  | Real Time respond | Respond to hazardous material and | |
|  |  |  | situations |

**Design Considerations**

REQ1: Supports deliberate and reactive behaviour. Robot must coordinate the actions to accomplish its mission and reactions to unexpected situations

REQ2: Allows uncertainty and unpredictability of environment. The situations are not fully defined and/or predicable. The design should handle incomplete and unreliable information

REQ3: System must consider possible dangerous operations by Robot and environment

REQ4: The system must give the designer flexibility (mission’s change/requirement changes)

Page 31



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

**Solution 1: Control Loop**

Req1: an advantage of the closed loop paradigm is its simplicity. It captures the basic interaction between the robot and the outside.

Req2: control loop paradigm is biased towards one method, reducing the unknowns through iteration

Req3: fault tolerance and safety are supported which makes duplication easy and reduces the chances of errors

Req4: the major components of a robot architecture are separated from each other and can be replaced independently

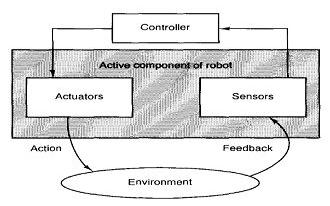


Figure 2.19. A control loop solution for mobile robots

**Solution 2: Layered Architecture**

Figure shows Alberto Elfes’s definition of the layered architecture. 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 re-planning)

Top level deals with UI and overall supervisory functions

* Req1: it overcomes the limitations of control loop and it defines abstraction levels to guide the design
* Req2: uncertainty is managed by abstraction layers
* Req3: fault tolerance and passive safety are also served
* Req4: the interlayer dependencies are an obstacle to easy replacement and addition of components.

**Solution 3: Implicit Invocation**

The third solution is based on the form of implicit invocation, as embodied in the Task-Control-Architecture (TCA). The TCA design is based on hierarchies of tasks or task trees

|  |  |  |
| --- | --- | --- |
|  | Parent tasks initiate child task |  |
|  | Temporal dependencies between pairs of tasks can be defined |  |
| A must complete A must complete before B starts (selective concurrency) | |  |
|  | | Page 32 |



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

Allows dynamic reconfiguration of task tree at run time in response to sudden change(robot and environment)

Uses implicit invocation to coordinate tasks

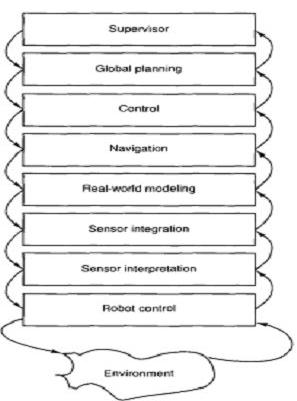
Tasks communicate using multicasting message (message server) to tasks that are registered for these events

TCA’s implicit invocation mechanisms support three functions:

Exceptions: Certain conditions cause the execution of an associated exception handling routines i.e., exception override the currently executing task in the sub-tree (e.g., abort or retry) tasks

Wiretapping: Message can be intercepted by tasks superimposed on an existing task tree. E.g., a safety-check component utilizes this to validate outgoing motion commands

Monitors: Monitors read information and execute some action if the data satisfy certain condition. E.g. battery check

.  Figure 2.20 Task-Control-Architecture (TCA)

Page 33



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

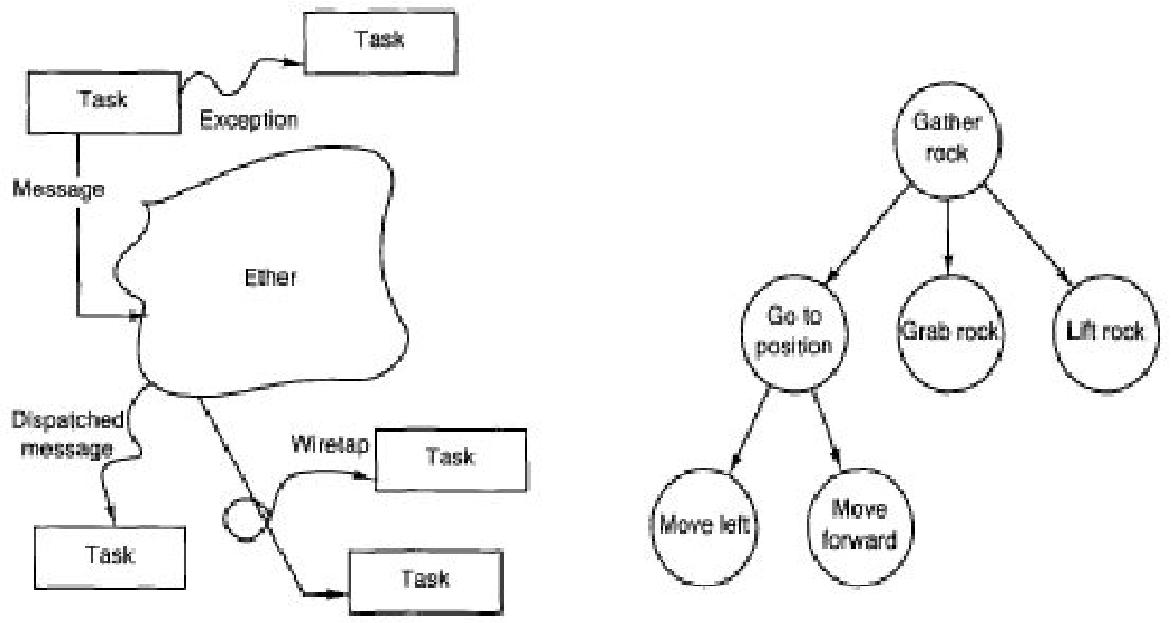


Figure 2.21 An implicit invocation architecture Figure2.22 Task tree for mobile robots

model for mobile robotos

Req1: permits clear cut separation of action and reaction

Req2: a tentative task tree can be built to handle uncertainty Req3: performance, safety and fault tolerance are served

Req4: makes incremental development and replacement of components straight forward

**Solution 4: Blackboard Architecture**

The components of CODGER are the following: 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

The requirements are as follows:

Req1: the components communicate via shared repository of the blackboard system.

Req2: the blackboard is also the means for resolving conflicts or uncertainties in the robot’s world view

Req3: speed, safety and reliability is guaranteed

Req4: supports concurrency and decouples senders from receivers, thus facilitating maintenance.

Page 34



|  |  |
| --- | --- |
| Software Architecture |  |
|  |  |
|  |  |

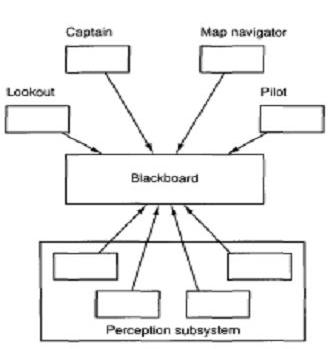


Figure 2.22: blackboard solution for mobile robots

**Comparisons**

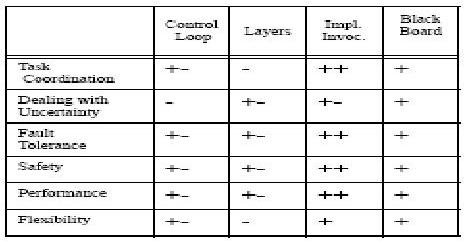


Table 1: Comparison of strength and weakness of robot architecture

